Проектиране и Тестиране на Софтуер
ТУ, кат. КС, летен семестър 2012

Лекция 22

Тема:

ЖЦПС

Етапите преди Програмиране

Съдържание:

- Концепцията ЖЦПС
- Част 1
  - Етапите преди програмиране
  - Методи за проектиране
- Част 2
  - Техники на проектиране

Жизнен цикъл на ПС

От гледна точка на концепцията за преобразуване на данни (информация) от едно представяне в друго жизненият цикъл на една програмна система от идеята за нейното създаване през реализацията и експлоатацията й до изхвърлянето й от употреба се описва със следните етапи:

ЖЦПС - 1

1. Дефиниране (специфициране) на проблем за решаване;
2. Изготвяне на задание;
3. Проектиране;
4. Програмиране;
5. Тестване и настройка;
6. Демонстрация и предаване на клиента (краен потребител);
7. Експлоатация и съпровождане.

ЖЦПС - 2

1. Системен анализ (Systems Analysis);
2. Системно проектиране (Systems Design);
3. Системна реализация (Systems Implementation);
4. Ситемна поддръжка (Systems Support).

ЖЦПС - 3

1. Формулиране на изисквания (Requirements phase) – 3%;
2. Описание на проблема (Specification phase) – 4%;
3. Планиране (Planning phase) – 2%;
4. Проектиране (Design phase) – 6%;
5. Реализация (Implementation phase) – 12%;
6. Интегриране (Integration phase) – 6%;
7. Поддържане/съпровождане (Maintenance phase) – 67%.

ЖЦПС - 4

- Software Development Method (SDM)
  - Specify the problem requirements
  - Analyze the problem
  - Design the algorithm to solve the problem
  - Implement the algorithm
  - Test and verify the completed program
  - Maintain and update the program

ЖЦПС - 5

- Software Development Method (SDM)
  - Requirements analysis and definition
  - System design
  - Program design
  - Program implementation
  - Unit testing
  - System delivery
  - Maintenance

Опростен ЖЦПС - 6

- SDM (според E.Petroutsos, pp28):
  - Decide what the application will do and how it will interact with the user.
  - Design the application's user interface according to requirements of step 1.
  - Write the actual code behind the events you want to handle.

Част 1

Етап проектиране – методи

Съдържание:

- ЖЦПС Етапи преди програмиране
- Методи за проектиране

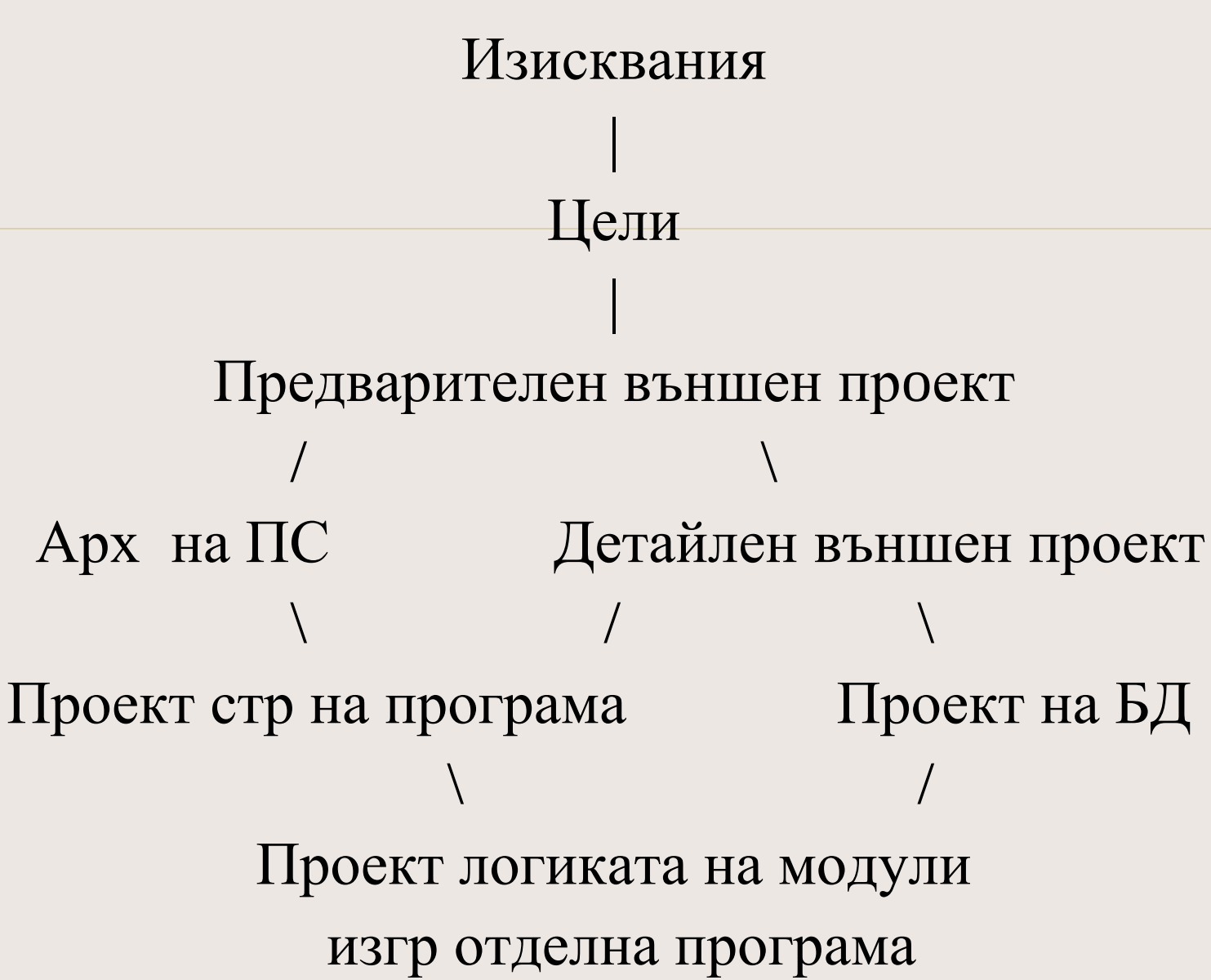
Етапи преди програмиране

Три етапа предшестват програмирането в ЖЦПС

- дефиниране на проблем за решаване;
- изготвяне на задание за проект;
- проектиране.

Тези три етапа приключват със завършен проект за програмиране.

Основните процеси и дейности, които се изпълняват при проектирането на голяма програмна система, включват:


Изисквания
Цели
Предварителен външен проект
Арх на ПС
Детайлен външен проект
Проект стр на програма
Проект на БД
Проект логиката на модули
изгр отделна програма

Детайли

Изисквания: Какво очаква възложителят и/или крайният потребител от готовия програмен продукт.

Цели: Какви цели/задачи се поставят за решаване с оглед удовлетворяване на изискванията, формулирани преди това.

Детайли

Предварителен външен прект:

Взаимодействие (диалог) с потребителя се изгражда в най-общ вид без подробности (формат на диалога).

Архитектура на Програмната Система:

Съставя се структурата (фасадата) на ПС. Фиксира се множеството програми, изграждащи програмния продукт.

Детайлен външен проект:

Уточняване на подробности относно взаимодействието с потребителя.

Детайли

Проект структурата на отделна програма:

Структурата на всяка една програма от ПС се проектира. Специално внимание се отделя за избор на алгоритъм.

Проектиране на БД:

Описание на всички външни за ПС структури данни.

Проект логика на модули, изгр отд програма:

Формиране структурата на всеки модул.

Избор на инструментален ПЕ

Изброените етапи са примерни. Възможно е някое действие да липсва: например, ако ПС не работи с външни данни или структурата й е от единствен програмен модул.

Обратно, възможно е определено действие да се натовари с повече от описаните дейности: например изборът на инструментален ПЕ може да се направи на етап

Предварителен външен прект или

Арх на ПС или

Проект стр на програма

Избор на алгоритъм

Изборът на сходящ и ефективен алгоритъм е важен фактор за успеха на проекта.

За даден проблем са възможни множество алгоритмични решения. Недобра практика е при проектирането да се избере първият дошъл на ум алгоритъм, който може би не винаги е най-доброто възможно решение.

Един час повече, използван при проектиране за избор на удачен алгоритъм, може да спести много часове за програмиране, тестване и настройка.

Избор на алгоритъм

Пример 1: $y = ax^3 + bx^2 + cx + d$

Решение 1: y = a*x**3 + b*x**2 + c*x + d

Решение 2: y = a*x*x*x + b*x*x + c*x + d

Решение 3: y = (a*x + b) * x + c) * x + d

полином по Хорнер

Избор на алгоритъм

Пример 2: Да се определи дали *n* е просто число (дели се на 1 и на себе си)

Решение 1: цикъл от 2 до *n-1*, стъпка 1

Решение 2: проверка за 2,
цикъл от 3 до *n-1*, стъпка 2

Решение 2: проверка за 2,
цикъл от 3 до sqrt(*n-1)*, стъпка 2

Избор на алгоритъм

Пример 3: Подобрение на рекурсивния алгоритъм за редицата на Фибоначи

```
int fibr(int n)
    {
       if (n==0 || n==1) return n;
         return fibr(n-1) + fibr(n-2);
    }
```

Подобрено бързодействие при Фибоначи

Таблица съхранява вече пресметнати стойности

```
struct val { double value; bool filled; };
extern val tab[];
int fibr(int n) {
   if (n==0 || n==1) return n;
   if (tab[n].filled == false) {
        tab[n].value = fibr(n-1) + fibr(n-2);
        tab[n].filled  =  true; }
    return tab[n].value;
                }
```

Useful Mathematic Summations
Liang, ppt file 23, slide 9

Методи за проектиране

Недостатъците, които се проявяват на етап изготвяне на задание (непълнота, изменчивост, противоречивост)

Необходимостта от систематичност при провеждането на етап проектиране

Налагат развитието на различни методи и техники за описание на заданието и формулиране на резултатите от процеса на проектиране:

Методи за проектиране

- Словесен метод
- Схеми на потока данни
- Йерархични диаграми
- Таблици на решение
- Блок схеми
- CRC карти
- Езикът UML (отделна лекция)

Методи за проектиране

- Предимства:
  - пълнота, непротиворечивост, строгост
  - възможност за формален преход към програма (автоматизация)
- Недостатъци:
  - всеки нов метод изисква усилия за усвояване

Словесен метод

Касае се за прилагане на ограничен, структурен естествен език, с други думи псевдоезик, който се комбинира с традиционни синтактични изразни средства на ПЕВН като оператори за цикъл и разклонение. Така се получава хибрид, който след това се уточнява и прецизира във формата на конкретна програма. Пример на този подход може да се види и в книгата на Kernighan/Ritchie “The C Programming Language”.

Словесен метод

Проектирането и реализацията на програма, филтрираща най-дългия текстов ред от входния поток, въведен от клавиатурата, се описва така:

Словесен метод

Прочети първия ред и го приеми за най-дълъг ред

while (има данни във вх.поток)

{

if(текущият ред е по-дълъг от най-дългия обработен досега)

{

запомни текущия ред като най-дълъг ред

}

}

Изведи най-дългия ред

Словесен метод

Разновидност на словесния метод с ориентация към ООПП е дадена от R.Abbott в статията “Program Design by Informal English Descriptions” (Comm of the ACM, vol26, no11). Проблемът се описва в изречение на естествен език. Съществителните имена се маркират като кандидати за означаване на обекти и/или шаблони на класове, а глаголите се маркират като кандидати, с които се именоват методите на избраните класове.

Словесен метод

Например, от изречението

“Стекът записва и извежда данни.”

по естествен начин се съставя описание на клас **`Стек`** с два метода:

- за запис на данни в стек **`push()`** и
- за извеждане на данни от стек **`pop()`**.

Словесен метод

```
      //
      class Stack
      {
// метод за съхранение (запис) на данни
   в стек
      void  push(int  ..)
        {  ...  }
// метод за извеждане (четене) на данни
   от стек
      int  pop()
        {  ...  }
      };
```

Схеми на потока данни

Data Flow Diagrams (DFD). За разлика от процедурните блоксхеми, схемите на потока данни не показват потока на управлението при изпълнение. Те не задават проверки на условия и циклични обработки. Тези схеми показват само потока на данните (data flow), които се обработват – откъде идват, къде се съхраняват, къде се извеждат резултати, кои са процесите, обработващи данните.
Физически сх на потока данни (physical DFD).
Логически сх на потока данни (logical DFD).

Йерархични диаграми

Съставя се йерархична структурна и функционална диаграма за проектираната ПС. Значими компоненти (функции) от системата се маркират. За всяка функция се съставя диаграма на обработките по класическия модел на изчислителен процес (Input, Process, Output). С тях се описват функционалните изисквания на потребителя и те му дават нагледна представа за бъдещата програмна система без да съдържат описания на конкретни структури от данни или първични текстове на конкретен програмен език.

Таблици на решение

(Decision Tables). Те са средство за описание на проблем, което позволява систематична формулировка на всички възможни варианти на входни условия във формата на правила. За всяко от записаните правила се задава потребителска реакция, наречена действие. ТР са средство за общуване между проектант, краен потребител и програмист. Съставят се от проектанта. Те са информативни за потребителя и служат като отправна точка за програмиста.

Таблици на решение

Условия	Правила							
C1. Стаж>10 г	Y	Y	Y	Y	N	N	N	N
C2. Пол-М	Y	Y	N	N	Y	Y	N	N
C3. Възраст>50	Y	N	Y	N	Y	N	Y	N
Действия								
A1. Премия	900	700	800	600	0	0	0	0

Таблици на решение

Стих: древноперсийски епос:

Този, който не знае и не знае, че не знае,
Той е глупак, бягай от него!
Този, който не знае и знае, че не знае,
Той може да се изучи, помогни му!
Този, който знае и не знае, че знае,
Той спи, събуди го!
Този, който знае и знае, че знае,
Той е пророк(мъдрец), учи се от него!

Таблицы Решений

В древнем персидском стихотворении сказано:

Тот, кто не знает и не знает, что он не знает,

- глупец, избегай его.

Тот, кто не знает и знает, что он не знает,

может научиться, научи его.

Тот, кто знает и не знает, что он знает,

спит, разбуди его,

Тот, кто знает и знает, что он знает,

- пророк, учись у него.

Decision Tables

Version in English presented as Jewish proverb:

He who knows not and knows not he knows not,
he is a fool, shun him.
He who knows not and knows he knows not,
he is ignorant, teach him.
He who knows and knows not he knows,
he is asleep, awaken him.
He who knows and knows he knows,
he is wise, follow him.

Таблици на решение

Условия	Правила			
C1. Човек знае	N	N	Y	Y
C2. Човек знае, че C1	N	Y	N	Y
Действия				
A1. Бягай от него!	X	-	-	-
A2. Помогни му!	-	X	-	-
A3. Събуди го!	-	-	X	-
A4. Учи се от него!	-	-	-	X

Блок схеми (flowcharts)

Те са средство, приложимо както в проектирането (процесът завършва с продукт блоксхема), така и в програмирането (процесът започва с готова блоксхема).

За разлика от DFD, блоксхемите показват потока на управлението. Те описват какви действия се извършват в зависимост от едни или други условия при изпълнение с набор входни данни. В този смисъл за блоксхема се ползва и терминът control flow diagram (CFD).

Блок схеми (flowcharts)

От гледна точка на практиката всяка блоксхема се състои от следните конструктивни елементи:

1. Блок действие с един вход и един изход;
2. Блок разклонение с един вход и два или повече изхода;
3. Елемент начало само изход(и);
4. Елемент край само вход(ове);
5. Елемент сливане с два или повече входа и един изход.

Блок схеми (flowcharts)

От гледна точка на теорията блоксхемата е абстрактна структура – насочен (ориентиран) ацикличен граф, който указва реда на изпълнение на операторите в една програма.

Всеки оператор от програмата се представя като възел в графа.

Всяко възможно предаване на управлението се представя като дъга (линия, свързваща два възела) в графа.

Блок схеми (flowcharts)

Ако един възел има една входяща и една изходяща дъга, той описва оператор за действие от програмата и се нарича *функционален възел*.

Ако един възел има една входяща и две или повече изходящи дъги, той описва управляващ оператор от програмата и се нарича *предикатен възел*.

Третият възможен тип възли имат две или повече входящи дъги и една изходяща дъга и се нарича *възел за сливане*. Възлите за сливане не влияят по никакъв начин на обработваните данни.

Блок схеми (flowcharts)

Управляващата структура на всяка една блок схема се конституира чрез формиране на комбинации от функционални възли, предикатни възли и възли за сливане.

Изпълнение на една блок схема се нарича последователността от действията, написани във функционалните блокове, през които минава пътят. Дървото на изпълненията представя множеството от възможните последователности от действия през всички възможни клонове на блок схемата.

CRC карти

Това е техника с приложение в ООПП. Съкращението означава Class (Клас), Responsibilities (Отговорности, Действия), Collaborators (Сътрудници).

Клас	
Отговорности	Сътрудници
. . .	. . .

CRC карти

Отляво в колона се изброяват отговорностите на класа. Това са действията (един или няколко метода), които проектираният клас ще изпълнява.

Отдясно срещу всяка отговорност се изброяват класове (обекти), които ще се активират при изпълнение на съответното действие.

Това е начин за изясняване на връзките между класовете в общия модел.

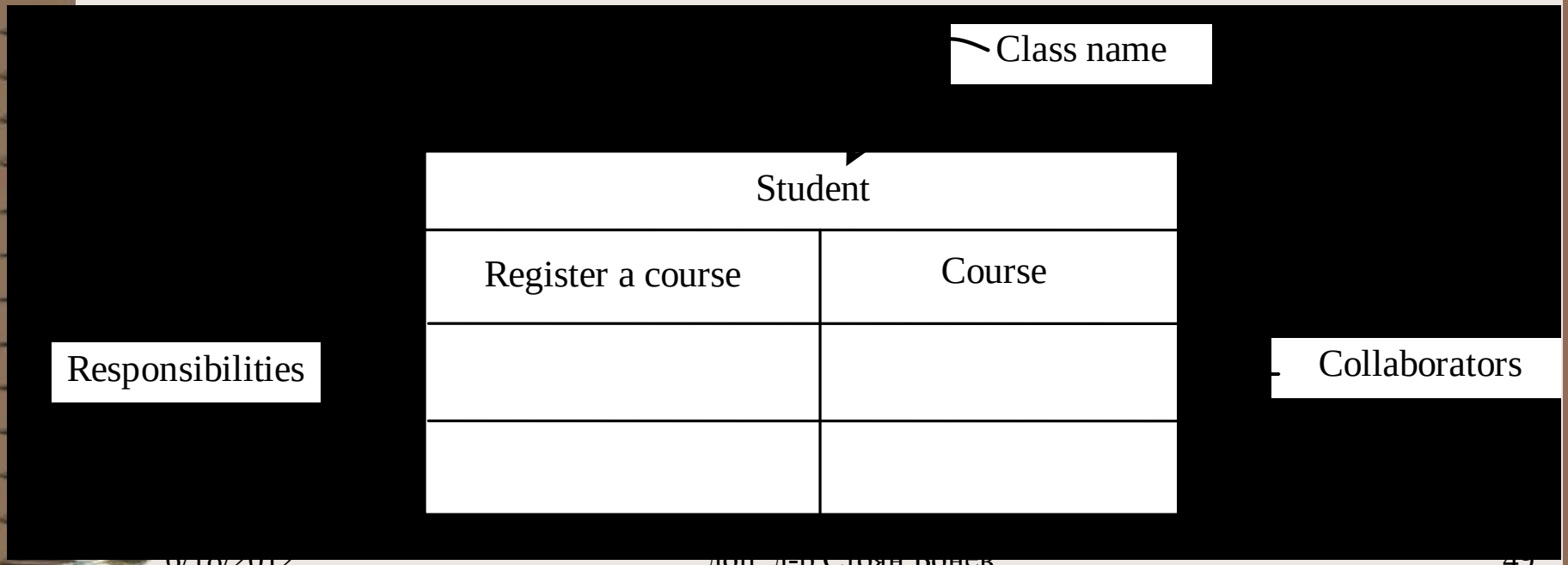
CRC карти

Ниска степен на обвързване (low coupling) между класовете е характерна за добрите обектно ориентирани модели.

Обратно, признак за некачествено създаден обектно ориентиран модел в съответната предметна област е случаят на CRC карта с отговорност, на която съответстват много сътрудници.

Discovering Classes

One popular way for facilitating the discovery process is by creating CRC cards. CRC stands for *classes*, *responsibilities*, and *collaborators*. Use an index card for each class, as shown in the Figure below.



Езикът UML

- Виж раздела ООП/П (цялата следваща лекция)

Част 2

Етап проектиране – техники

Съдържание:

- Техники на проектиране
- Техники на програмиране

Техники (методики)

Изпълнението на етапите проектиране и програмиране се реализира по определена техника (методика). Възможни са следните подходи:

Тотално проектиране/програмиране;

Модулно проектиране/програмиране;

Възходящо проектиране/програмиране;

Низходящо проектиране/програмиране.

Тотално П/П

Тук няма никакава специална идея или замисъл.

Целият проект се замисля като един модул, който се свежда до една работеща програма.

Намира ограничено приложение за малки по обем учебни и/или демонстрационни програми.

Модулно П/П

Проектите/програмите се разделят на логически части и последователно се проектират/програмират. Предимството се състои в разделянето на голям проект или програма на по-малки логически обособени части, решаването на всяка от които е по-лека задача от решаването на проекта като цяло.

Top-Down Design method

Stepwise refinement

Модулно П/П

Изисквания, които се поставят при обособяване на модулите, имат за цел да осигурят:

1.Независимост. Всеки модул е самостоятелна единица и не зависи от останалите. Може да се замени с друг без това да влияе на останалите.

2.Приемлив размер. Цитират данни от компютърни компании и софтуерни фирми, които налагат различни практически нормативи:

- Модул е част от програма, който се пише, тества и настройва за 1 месец.
- Модул съдържа не повече от 100 оператора на ЕВН.

Възходящо П/П

Отделните функционални модули на дадена програмна система се разработват (проектират/програмират) отначало отделно и след това се обединяват в единно цяло. Основният недостатък на този подход се крие във факта, че грешки в проектирането често се откриват едва при настройката (обединяването) на модулите в едно цяло, след като много модули са написани и се оказват безполезни.

Низходящо П/П

Основната идея е в противовес на възходящото проектиране. Управляващите програми се съставят отначало (проектират, програмират, тестват и настройват), а функционалните изпълнителни модули се добавят в процеса на разработката на цялата система. Обичайно низходящото проектиране започва с логическо проектиране. Така се обезпечава съгласуваност на работата между отделните програмни модули.

Низходящо П/П

При възходящото проектиране програмните модули не се проверяват като съставна част на цялата система до края на разработката им.

При низходящото проектиране програмните модули се проверяват веднага за съвместимост към системата. Тяхното място в системата се заема от специални модули, празни програмни единици (stubs) първоначално преди да са съставени.

Низходящо П/П

Низходящата методика допуска ранното откриване и отстраняване на допуснати грешки при П/П. Това става в периода, когато програмите са все още при разработчика.

Обратно, възходящата методика предпоставя откриването на грешките да става на по-късен етап при комплексната настройка на ПС. Това е недостатък, защото разработчикът на конкретните модули е приключил работа и е възможно да работи по друг проект.

Структурно П/П и еволюция

Описаните методи и техники за проектиране (словесен, схеми на потока данни, йерархични диаграми, таблици на решение, блоксхеми), както и методиките за тяхното провеждане (тотално, модулно, възходящо, низходящо П/П произтичат от принципите на структурния анализ и проектиране на системи (SSAD – Structured Systems Analysis and Design) и са свързани със структурното програмиране.

Структурно Програмиране

Какво е структурното програмиране (СП)? То е и метод, и техника, и методика. Обобщено казано, *структурното програмиране е концепция* със значително влияние върху принципите на разработка на софтуер. СП се счита като принос в технологията на програмиране, равностоен по значимост с концепцията подпрограма и работата с ПЕВН.

Структурно Програмиране

Общоприето строго и ясно определение на структурно програмиране няма. Най-широко известната, но тясна и повърхностна представа за СП е като програмиране без използване на оператор за безусловен преход goto. По тази причина СП е популярно с термина **`goto-less programming`**.

Структурно Програмиране

Две публикации на Дийкстра:

1965 доклад “Programming Considered a Human Activity”: квалификацията на един програмист е обратно пропорционална на броя на операторите goto в неговите програми.

Структурно Програмиране

Две публикации на Дийкстра:

1965 доклад “Programming Considered a Human Activity”: квалификацията на един програмист е обратно пропорционална на броя на операторите goto в неговите програми.

1968 Comm of the ACM, дописка до редактора “GOTO Statement Considered Harmful”. Обявява оператора за безусловен преход като вреден и развива някои идеи на низходящото проектиране/програмиране.

Структурно Програмиране

Дийкстра посочва, че м/у текста на програма и потока на нейното управление (реда на изпълнение на операторите й) трябва да съществува просто съответствие. Програмата следва да е така структурирана, че да се чете и възприема по низходящ (top-down) принцип отгоре надолу. Неограниченото и неконтролирано използване на оператори goto нарушава това съответствие. По тази причина СП е известно като програмиране без оператор за безусловен преход.

Структурно Програмиране

Това е само една частична и непълна представа, тъй като винаги има програми-изключения:

1. Програми без goto, но структурирани така, че са трудни за възприемане или са напълно непонятни;
2. Програми с goto, но с достатъчно ясна логика, така че присъствието на оператор goto не пречи на тяхната четливост.

По този повод през 1974 и Д. Кнут статия: “Structured Programming with GOTO Statements”

Структурно Програмиране

Ето защо добива гражданственост следното определение за структурно програмиране: *Структурното програмиране е ориентирано към общуване на програми с хора, а не към общуване между програми и компютри.*

В същата работа изискванията към една структурна програма са формулирани:

Структурно Програмиране

1. Програмният текст се съставя от три основни елемента. Това са последователност (следване, sequence), избор на условие (разклонение, selection) и повторение (цикъл, repetition);
2. Всеки програмен модул се организира така, че да има един вход;
3. Да се избягва употребата на оператор goto, където е възможно;
4. Програмният текст да е написан в добър и приемлив стил;
5. Програмният текст да се структурира с изместване и вмъкване на празни позиции в редовете с цел леко да изпъкват операторите от тялото на оператор за цикъл, както и ясно да се открояват групата оператори от клаузите then и else в условните оператори;
6. Програмният текст физически да се разбива на части така, че изпълнимите оператори от един модул да заемат обем една печатна страница;
7. Програмата сама по себе си представлява просто и ясно решение на конкретната задача.

Структурно Програмиране

Точка 1: Изискването програмният текст да се съставя от елементите *последователност, избор и повторение* следва от теоретични изследвания, които се основават на доказаната теорема от Bohm и Jacopini и публикувана през 1965 на италиански и през 1966 на английски език в статия Flow Diagrams, Turing Machines and Languages with Only Two Formulation Rules. Теоремата гласи. *Всяка реална програма може да се състави като се използват само две управляващи структури – действие и двоична проверка (разклонение)*. Не е трудно да се обобщи следното:

Структурно Програмиране

Елемент *последователност* е верига от две или повече действия.

Елемент *избор* е познат с трите варианта **if … then …, if … then … else, switch … case … default …,** но реализацията му се базира на двоична проверка.

Елемент *повторение* е известан във варианти с предусловие и с постусловие. Реализацията му се основава на управляващи структури действие и двоична проверка.

Очевидна е рекурсивната природа на структурата действие. В съдържанието на едно действие могат да бъдат включени/вложени/ елементи и от трите вида.

Структурно Програмиране

Точка 2: Изискването всеки програмен модул да има един вход и евентуално един изход се свързва с понятието функционален блок, представен по-долу.

Структурно Програмиране

Функционалният блок има един вход и един изход. Той може да представлява отделен изчислителен оператор (дори команда на машинен език), или друга последователност от изчисления с един вход и един изход.

Описаните елементи последователност, избор и повторение се представят като единствен функционален блок.

По пътя на обобщение се прави следният извод: Всяка проста програма, съставена от функционални блокове (последователност, проверка и повторение), се поддава на трансформация до единствен функционален блок.

Структурно Програмиране

Проста програма: *Това е програма, чиято блок схема или управляваща структура има първо един вход и един изход и второ през всеки възел минава път (поток на управление) от входа към изхода.*

Второто условие по теоретичен път определя като недопустими управляващи структури, които са проектирани да съдържат недостижими при изпълнение възли или безкрайни повторения (цикли). Това не означава, че в програмата като краен продукт няма да се получат неблагогополучия като изброените, но е гаранция, че по време на проектиране те няма да се заложат като задача за програмиране.

Разглеждат се още проста подпрограма, елементарна програма, неелемнетарна програма, съставна програма, структурирана програма.

Структурно Програмиране

Точка 3: Изискването да се работи без оператор goto не бива да бъде абсолютизирано като задължително правило. То е препоръка и насока да се следи безконтролното използване на този оператор. Тук се отнасят и операторите break и continue в езиците C/C++. Те не са структурни оператори, но употребата им е оправдана и не влияе на яснотата на програмните текстове.

Структурно Програмиране

Изискването за добър и приемлив стил на програмиране също е централно в СП. Под стил се разбира маниерът на програмиста, с който той ползва ПЕ. Програмистът следва да възприеме като правило практически препоръки и принципи, спазването на които прави ясни текстовете, които той създава. Вече бе изтъкнато, че в СП задачата на програмиста е да съставя програми, предназначени за хора, а не за компютри. По тази причина се налага доминиращото изискване за *яснота в програмата, простота, достъпност и прозрачност* на програмните текстове пред оценяваните като по-маловажни критерии за краткост или машинна ефективност.

Структурно Програмиране

Програми не отговарят на условието за структурни програми. Въпрос: Могат ли неструктурни програми да се трансформират в стуктурни програми? В общия случай отговорът е негативен: *Произволна програма не може да се преобразува в структурирана програма, която реализира същия алгоритъм, съставена е от същите програмни елементи и не използва допълнителни променливи*. Изводът е, че привеждането на една програма в структуриран вид изисква усилия. Известни подходи за привеждане в структурен вид на програмни текстове са:

метод с дублиране на кододвете;

метод с въвеждане променлива на състоянието;

метод на булевия признак и др.

Структурно Програмиране

Идеите на структурното програмиране са заложени и вътрешно присъщи на съвременните ПЕВН като VBasic, C/C++/C#, Java. В езиците са налице оператори, чрез които в различни варианти е възможно да се програмират разклонения

VBasic:

If <израз> Then <оператор> Endif

If <израз> Then <оператор> Else <оператор> Endif

C/C++/C#, Java:

if (<израз>) <оператор>

if (<израз>) <оператор> else <оператор>

Структурно Програмиране

и циклични повторения

VBasic:

For <ид> = <израз> To <израз> <оператор> Next

Do While <израз> <оператор> Loop

Do <оператор> Loop Until <израз>

Do <оператор> Loop

C/C++/C#, Java:

for (<израз1>;<израз2>;<израз3>) <оператор>

while (<израз>) <оператор>

do <оператор> while (<израз>)

Структурно Програмиране

Наред с цитирания синтаксис на оператори за разклонение и цикъл друга предпоставка за лека реализация на идеите на структурното програмиране се свързва с концепцията съставен оператор и блокова структура на програмния език. Чрез тях програмистът групира няколко оператора в едно логическо цяло и след това към него се обръщат като към отделен оператор. Обединението на група оператори в един съставен оператор се постига чрез включване на операторите в операторни скоби от следния вид:

при C/C++/C#, Java: { <оператори> }

ООП/П

Еволюцията на СП доведе ООП/П. Водещият принцип от структурния анализ и синтез на системи *мислене чрез функции* (Thinking in Functions) остава класически, но се счита остарял и неактуален. Той се измества от наложилия се съвременен принцип *мислене чрез обекти* (Thinking in Objects), който е валиден за обектно ориентирания анализ и синтез на системи. Развиха се множество методи за ООА, ООС, ООП/П. Най-известни са:

ООП/П

1. Метод за обектно ориентирано проектиране OOD (Object Oriented Design) от Гр. Буч (Grady Booch);
2. Метод за обектно моделиране OMT (Object Modeling Technique) от Дж. Ръмбау (James Rumbaugh);
3. Метод за обектно ориентирано софтуерно инженерство OOSE (Object Oriented Software Engineering) от А. Якобсон (Ivar Jacobson).

ООП/П

Тримата цитирани автори започват работа за фирма Rational Software (www.rational.com) през 1994 г. Там те обединяват усилия за разработка на унифициран метод за обектно ориентирано проектиране (Unified Method). В резултат се ражда езикът за описание и моделиране на обектно ориентирани системи, обявен като UML (Unified Modeling Language).

За подробности виж следващата лекция.

Благодаря
За
Вниманието