МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Запорізький національний технічний університет

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

«Компонентні технології в САПР»

для студентів спеціальності

122 «Комп'ютерні науки та інформаційні технології»

освітня програма «Інформаційні технології проектування» денне відділення

2016

Конспект лекцій з дисципліни «Компонентні технології в САПР» для студентів спеціальності 122 «Комп'ютерні науки та інформаційні технології» освітня програма «Інформаційні технології проектування»денне відділення /Укл.: І.В. Левада – Запоріжжя: ЗНТУ, 2016. – 90с.

Укладач: ст. викладач І. В. Левада

Рецензент: доц., к. т .н. А.В. Пархоменко

Відповідальний за випуск: проф. С. О. Субботін

Затверджено на
засіданні НМК ФКНТ
Протокол №1
від 30 вересня

Затверджено на
засіданні кафедри ПЗ
Протокол № 2
від 11 вересня 2016 р.

ЗМІСТ

ПЕРЕДМОВА 6

1 КОНЦЕПТУАЛЬНЕ ПРОЕКТУВАННЯ СКЛАДНИХ СИСТЕМ СТРУКТУРНИМ МЕТОДОМ 8

1.1 Поняття складної системи 8

1.2 Життєвий цикл програмної системи 10
1.2.1 Каскадна модель 11
1.2.2 Спіральна модель 14

1.3 Концептуальне проектування програмної системи 18

1.4 Методології концептуального проектування IDEF 20

1.5 Методологія IDEF0 23

1.6 Основні поняття функціональної моделі IDEF0 25
1.6.1 Функціональний блок і дуги 25
1.6.2 Ієрархічна структура з діаграм 27

1.7 Зв’язок функцій дугами на діаграмі декомпозиції 31

1.8 Порядок виконання функцій на діаграмі декомпозиції 34

1.9 ICOM-коди прикордонних дуг та тунелі 36

1.10 Активації функцій 39

1.11 Допоміжні компоненти IDEF0-моделі 41

2 КРИТЕРІЇ ЯКОСТІ ФУНКЦІОНАЛЬНОЇ МОДЕЛІ 43

2.1 Синтаксичні критерії графічного подавання функціональної моделі 43
2.1.1 Критерії локальних конструкцій 43
2.1.2 Критерії глобальних конструкцій - діаграм 45
2.1.3 Критерії конструкції – модель 46

2.2 Семантичні критерії якості моделі 47

2.3 Контроль якості моделі засобами методології IDEF0 48
2.3.1 Зчеплення та зв’язність, їх роль у розумінні змісту моделі 49
2.3.2 Зчеплення функцій діаграми за природою зв'язку 49
2.3.3 Зчеплення функцій за структурою зв'язку 51
2.3.4 Зв’язність функцій діаграми за змістом 53

3 ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМ ЗАСОБАМИ МЕТОДОЛОГІЇ IDEF0 60

3.1 Властивості якісного програмного забезпечення та принципи розробки, що забезпечують якість 60
3.1.1 Принцип абстракції 61
3.1.2 Принцип модульності ..... **Ошибка! Закладка не определена.**
3.1.3 Принципи локалізації, утаєння інформації, узгодженості, повноти та підтверджування 64

3.2 Зчеплення та зв’язність модулів програми 66
3.2.1 Зчеплення модулів за природою зв’язку 66
3.2.2 Зчеплення модулів за структурою зв’язку 66
3.2.3 Зв’язність елементів модуля за змістом 67

3.3 Створення функціональної моделі програмної системи 71

4 МОДУЛЬНЕ ПРОГРАМУВАННЯ СИСТЕМ МОВОЮ VISUAL PROLOG 77

4.1 Структура модульної програми 77

4.2 Основні погодження написання модульної програми 80

4.3 Об’ява глобальних предикатів 81

4.4 Створення модульної програми експертом застосувань..... 82
4.4.1 Чому проект? 83
4.4.2 Каталоги проекту та їх призначення 84
4.4.3 Реалізація модулів програми 85

4.5 Компіляція і побудова модульної програми 87

ПЕРЕДМОВА

Системи автоматизації процесу проектування нових видів продукції (САПР) відносять до складних промислових систем. Вони розв'язують задачі підвищення ефективності труда інженера на стадіях проектування і підготовки виробництва.

Як будь-яка складна система, САПР складається із підсистем. Підсистеми бувають двох типів: проектування і обслуговування. Підсистеми проектування виконують функції проектування. Обслуговуючі системи – це середовище, яке підтримує функціонування підсистем проектування.

Складність структур систем автоматизованого проектування та їх функціонування, обумовлені складністю реальних задач на підприємстві, тобто складністю предметної області.

Метою викладання дисципліни «Компонентні технології в САПР» – є засвоєння студентами знань та одержання практичних навичок з структурного функціонального моделювання складних програмних систем за методологією IDEF0 та створення якісного програмного забезпечення на базі цих моделей мовою Visual Prolog.

Функціональні моделі застосовують для міркування над функціями програм (модулів) при колективній розробці системи. Готову функціональну модель застосовують для реалізації програмного комплексу або модульної програми.

Функціональні моделі створюються на етапі концептуального проектування програмних систем. Програмування за функціональною моделлю дозволяє створювати якісне забезпечення програмне забезпечення.

Методологія IDEF0 містить правила побудови функціональних моделей, їх корегування та оцінку якості моделей. Вміст правил побудови моделей та контролю їх якості базуються на системному аналізі, що забезпечує якість програм, створених за моделями.

Для реалізації програмного забезпечення нами була обрана мова Visual Prolog. Створення модульної програми або програмного комплексу мовою Visual Prolog також дозволяє прискорити розробку програмного забезпечення. Програми мовою Prolog значно менші за

об’ємом ніж програми написані на С++, бо являють собою опис завдання твердженнями, а не реалізують алгоритм.

Крім того, швидкість роботи програми мовою Visual Prolog не гірше, ніж швидкість роботи програм на С++. Програма мовою Visual Prolog компілюється відразу у оптимізований код.

Студентам що вивчають дисципліну, пропонується реалізувати функціональну модель системи за завданням, і реалізувати програму логічною парадигмою, застосовуючи функціональну модель, яку вони створили.

Студенти повинні осмислити, що принципи розробки якісної системи вже закладені в саму методологію створення функціональної моделі. Тобто, якщо розроблювати програмну систему точно за моделлю, то програмна система буде якісною.

1 КОНЦЕПТУАЛЬНЕ ПРОЕКТУВАННЯ СКЛАДНИХ СИСТЕМ СТРУКТУРНИМ МЕТОДОМ

1.1 Поняття складної системи

Існують різні визначення системи залежно від предметної та проблемної області. Основою для кожного сформованого визначення є визначення системи у системному аналізі. Система – це об'єкт або процес, елементи якого пов'язані у єдине ціле різними зв'язками та відношеннями.

У дисципліні ми застосуємо визначення системи, що базується на тому, що елементи системи пов'язані між собою одною *функцією та метою*.

Система – будь-яка комбінація апаратних, програмних засобів і людей взаємопов'язаних, взаємозалежних, які *працюють разом при виконанні конкретної функції для досягнення спільної мети.*

У системі зберігаються основні властивості при певних зовнішніх та внутрішніх змінах. Структурою системи називають *сукупність стійких зв'язків елементів системи,* що забезпечують її цілісність і *збереження основних властивостей, при певних зовнішніх та внутрішніх змінах.*

Існує два типи систем, які важко створювати слабо структуровані та складні системи.

Слабо структурованою системою називають таку систему, в якій зв'язки елементів не стійки, і тому їх описують тільки з врахуванням умов, при яких зв'язки стійки.

Слабо структуровані системи – це побільшому природні системи. Прикладом слабо структурованої системи є вода. Вода стає льодом при зміні зовнішніх умов - температури. Більшість природних систем слабо структуровані.

Розглядають два типа систем: відкриті та замкнені.

У замкнених систем середовища немає. Реально замкнених систем немає, всі системи відкриті. Замкнена системи абстракція, яку використовують у випадку, якщо середовищем не цікавляться.

Відкриті системи мають середовище. Середовище системи – коло, в якому працює система. Середовище системи містить інші системи, технології, виробничі структури, організації, які будуть працювати разом із системою для досягнення *спільної мети, проте виконують свої функції.*

Всі системи виробництва взаємодіють одна з одною. Кожна система взаємодіє з середовищем через входи та виходи. Входами називають множину контактів, через які вплив середовища передається системі. Виходами називають множину контактів, через які система впливає на середовище. До складних програмних систем відносять системи автоматизованого проектування нових видів продукції (САПР).

Складні системи. Виробничі системи складні. До складних програмних систем відносять системи автоматизованого проектування нових видів продукції (САПР).

Систему називають складною, якщо вона виконує велику кількість функцій, використовує велику кількість даних різного характеру, складається з великої кількості елементів, які пов’язані між собою зв’язками різного типу. Складність системи залежить також від кількості та складності зв’язків з системами середовища.

Для складних систем характерна: відсутність систем – аналогів, що не дозволяє використовувати типові проектні рішення; наявність підсистем, що тісно взаємодіють. Кожна підсистема розв’язує свою локальну задачу і має своє призначення. Підсистеми мають нерегламентовані запити до даних великого обсягу.

Створення складних програмних комплексів вимагає застосування модулів програм створених за різними технологіями різних компаній. Останні роки стали з’являться інтегровані середовища, що забезпечують кросплатформене програмування.

Інтегровані середовища підтримують не тільки програмування, а і реалізують керування проектом. До таких середовищ можна віднести середовища Borland C++, Eclipse. Вони забезпечують роботу над проектом впритул до створення дистрибутиву системи та оформлення документації.

Інтеграція інструментів різних компаній реалізується через експорт та імпорт, а їх взаємодія через єдиний репозиторій.

Кожна виробнича підсистема може функціонувати на своїй апаратній платформі та використовувати свої інструментальні засоби. За системним аналізом підсистеми складних систем також являють собою систему.

Проектування складних та слабо структурованих систем складна задача. Розробка таких систем завжди вимагає роботи колективу спеціалістів різного фаху. Для вирішення поставлених задач необхідно інтенсивне спілкування, як усередині команди розроблювачів системи, так і спілкування із замовниками. Ясно, що при розробці систем витрачається багато часу та коштів, а якість розробки не завжди задовільна.

Щоб уникнути вказаних недоліків колективи розробників застосовують певну технологію розробки систем.

Назвемо т*ехнологією розробки системи* конкретний опис технологічних процесів і порядку їх виконання. А процес колективної розробки системи і результат проектування *проектом*.

Створення складних програмних комплексів вимагає застосування модулів програм створених за різними технологіями різних компаній. Останні роки стали з’являться інтегровані середовища, що забезпечують кросплатформене програмування.

Інтегровані середовища підтримують не тільки програмування, а і реалізують керування проектом. До таких середовищ можна віднести середовища Borland C++, Eclipse. Вони забезпечують роботу над проектом впритул до створення дистрибутиву системи та оформлення документації.

Інтеграція інструментів різних компаній реалізується через експорт та імпорт, а їх взаємодія через єдиний репозиторій.

1.2 Життєвий цикл програмної системи

Під життєвим циклом програмної системи (ЖЦ) розуміють безперервний процес від прийняття рішення про необхідність

створення системи до повної її утилізації. Нас будуть цікавити етапи життєвого циклу програмної системи, що присвячені її розробці.

Структура життєвого циклу програмної системи визначається міжнародною організацією зі стандартизації International Organization of Standardization за стандартом ISO/IEC 12207. Моделі життєвого циклу системи визначають вміст і послідовність етапів розробки та супроводження системи, тобто визначають технології розробки систем.

Існують такі стратегії конструювання програмного забезпечення:

- водоспадна (лінійна послідовність етапів розробки ПЗ);
- інкрементна (циклічна послідовність етапів конструювання ПЗ);
- еволюційна (циклічна послідовність етапів конструювання ПЗ).

Ми розглянемо водоспадну та еволюційну стратегії на прикладі каскадної та спіральної моделі життєвого циклу.

1.2.1 Каскадна модель

Каскадна модель складається з таких етапів розробки системи: системний аналіз, аналіз вимог до програмного забезпечення, проектування, кодування, тестування, супроводження.

Системний аналіз. На етапі визначають роль кожного елементу системи: програм, користувачів, БД, апаратури, виявляють взаємодію елементів.

Визначають вимоги до кожного системного елементу. Кожну підмножину системних вимог призначають певному елементу системи. На етапі системного аналізу планують об’єми проектних робіт. Утворюють графік робіт. Визначають ступінь ризику, що система не відповідатиме вимогам.

Аналіз вимог до програмного забезпечення. Деталізують функції кожного програмного елементу, його характеристики, інтерфейс.

Результат: специфікація для кожного програмного елементу та завершений план робіт.

Проектування. За специфікацією визначають архітектуру програмного забезпечення; модульну, алгоритмічну структуру ПЗ; структури даних; вхідних і вихідних форм.

Система може являти собою програмний комплекс, у якого кожна окрема програма працює на своїй ділянці виробництва.

Кодування (реалізація). Утворюють програмний код обраними мовами програмування. Вибір мов програмування залежить від вимог до ПЗ.

Тестування. Під час тестування ПЗ визначають дефекти у функціях системи, реалізованій логіці, обраних алгоритмах, структурі системи тощо.

Супроводження. Виправляють помилки, адаптують ПЗ до середовища системи; удосконалюють ПЗ. При виконанні вказаних дій використовують етапи, які виконували при створенні нового програмного продукту, але виконують етапи для існуючої системи.

При реалізації каскадної моделі етапи розробки системи виконують послідовно. Кожний етап закінчують повним набором узгодженої документації. Після чого виконують наступний етап.

Перевагою каскадної моделі є можливість планувати строки всіх робіт і витрат. Каскадний підхід добре застосовувати при побудові автоматизованих інформаційних систем (АІС), для яких на початку розробки можна достатньо точно та повно сформулювати всі вимоги. До таких систем відносять системи реального часу, складних розрахункових систем. Це дозволяє авторам системи вільно реалізовувати вимоги, як найкраще з технічної точки зору.

Каскадну модель зображено на рисунку 1.1. Але реально відбувається процес на рисунку 1.2.

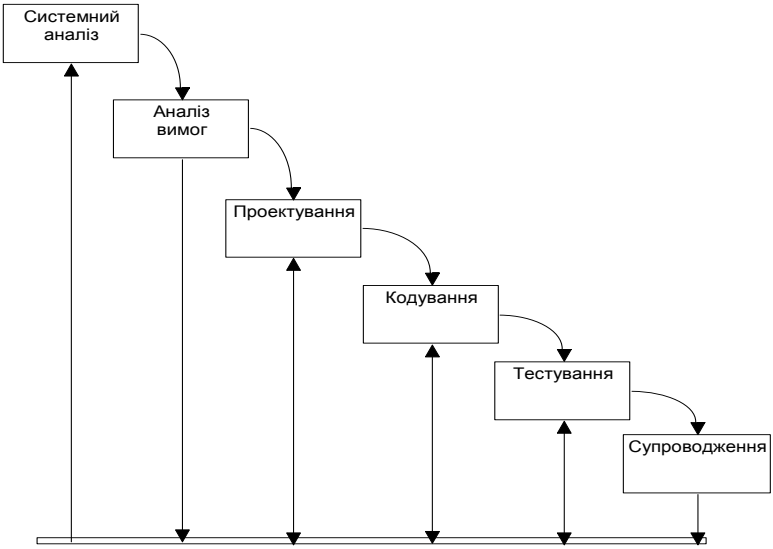


Рисунок 1.1 - Каскадна модель розробки ПЗ

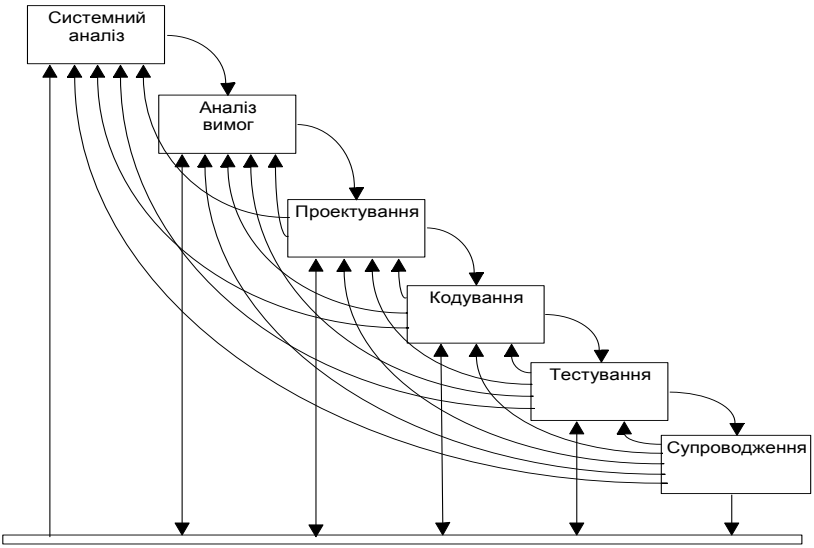


Рисунок 1.2 - Реальна каскадна модель розробки ПЗ

Недоліки каскадної моделі

У процесі використання каскадної моделі виявилось ряд недоліків, породжених тим, що реально процес проектування ПЗ ніколи не виконується за схемою з рисунку 1.1. В процесі роботи постійно виникала потреба в поверненні до попередніх етапів і уточненні або зміні прийнятих рішень. В результаті відбувається істотне запізнення з отриманням результатів.

Розробка за каскадною моделлю вимагає точного формулювання вимог, але замовник не завжди їх може сформулювати на початку. Результати показують замовнику тільки після закінчення кожного етапу. Якщо результати не задовольняють користувача, то відбувається повернення на попередні етапи, що значно затримує розробку.

У випадку неточно сформульованих вимог користувач після довгого часу чекання одержує результати, що не задовольняють його вимогам. Функціональні та інформаційні моделі системи можуть постаріти одночасно з їх затвердженням.

1.2.2 Спіральна модель

Щоб уникнути перелічених вище недоліків каскадної моделі була запропонована спіральна модель.

Спіральна модель базується на понятті прототипу. Прототип – це версія системи. Розробку системи виконують за витками спіралі, де на кожному наступному витку кожен прототип заміщується прототипом, в якому більше конкретизуються деталі проекту. Таким чином прототип поступово наближається до готової системи. Процес закінчується створенням системи.

Головна задача розробки за спіральною моделлю, якомога швидше показати користувачу прототип працюючої системи, тому на кожному вітку створений прототип дивиться замовник. Такий підхід дає можливість отримати від користувача зауваження та доповнення під час розробки.

Наприклад, нехай прототип показує користувачу фрагмент програми з меню системи. Користувач може доповнити його або

погодитися з ним. На наступному вітку замовнику можна показати виконання певних пунктів меню без розрахунків і т. д..

Перед початком роботи над проектом за спіральною моделлю менеджер проекту *складає план розробки системи.*

Менеджер проекту – керівник, який має організаційні знання з методології проектування систем і досвід застосування методології на практиці. Він аналізує попит ринку, планує проект, аналізує можливості інженерів і розподіляє роботи між ними, організує взаємодію розроблювачів, залучає експертів для розв'язку проблем, управляє проектом і контролює його виконання.

Технічний керівник проекту – спеціаліст, що має технічні знання та досвід. Технічний керівник слідкує за відповідністю проекту вимогам замовника та повідомляє замовнику про хід виконання проекту. Часто обидві функції виконує одна людина.

Якість системи, що розроблюється, залежить від досвіду розроблювачів, *координації їх роботи*. Результат створення системи залежить від вірно поставлених цілей; організації слідкування за ходом робіт у відповідності цим цілям. Нижче подано примірний план проекту.

План проекту

1 Утворення списку задач, що розв'язуватимуться у проекті.
2 Утворення графіку виконання робіт.
3 Оцінка витрат.
4 Визначення ресурсів: людських, організаційних, обчислювальних.
5 Розподіл відповідальності розроблювачів за задачами.
6 Визначення залежності між задачами.
7 Розподіл працівників за задачами і визначення їх ролей.
8 Визначення часу виконання кожної задачі.

Спіральна модель поєднує в собі проектування системи з поступовим створенням прототипів системи. Створення прототипу1 починається з аналізу ризиків створення системи за планом. Спіральну модель подано на рисунку 1.3.

Рисунок 1.3 - Спіральна модель розробки ПЗ

Основна проблема спірального циклу - визначення моменту переходу на наступний етап витку. Для рішення проблеми необхідно дотримуватися часових обмежень за планом на кожний з етапів витку. Перехід здійснюється у відповідності з планом, навіть в тому випадку, якщо не вся запланована робота закінчена. Закінчити роботу можна на наступному вітку спіралі.

Для кожного витку спіралі виконують такі етапи: планування; аналіз ризику; конструювання; оцінювання замовником.

Планування. Уточнюють цілі, вимоги та характеристики системи.

Аналіз ризику. Виконують аналіз результатів планування та ступеня ризику. Аналіз ризику прогнозує подальші події в розробці. Баррі Боем сформулював десять ризиків за пріоритетами, до яких в процесі розробки системи можна прийти.

1. Відсутність фахівців необхідного рівня.
2. Недостатні бюджет та строки розробки системи.
3. Функціональність системи не відповідає вимогам.
4. Невірний інтерфейс користувача.
5. Непотрібні намагання до удосконалення програми.

6. Зміни поступають безперервно.
7. Недостатність інформації про середовище системи.
8. Недоліки в роботі ресурсів системи, що повинні поступати із середовища.
9. Недостатня продуктивність системи.
10. Кваліфікація фахівців різних областей різних рівнів.

Аналіз ризиків дозволяє удосконалювати плани та цілі системи відповідно середовищу її функціонування, щоб одержати діючу систему згідно вимогам замовника.

Конструювання. На першому витку *удосконалюється* концепція висунутих вимог і функціоналу. Формулюється список вимог. На другому вітку спіралі список вимог підлягає верифікації та узаконюванню. План розвивається. На третьому вітку створюється функціонал прототипу: реалізують концептуальне проектування і кодування. Створення функціоналу за витками розподіляється на: створення окремих програм, інтеграцію комплексу, тестування, виконання в рідному середовищі, публікацію.

Оцінювання. Замовник оцінює поточні результати конструювання на витку, робить зауваження, уточнює вимоги.

Після цього по результатам оцінювання повторюється етап планування.

Перевагою спіральної моделі є тісний контакт розроблювачів із замовником на кожному витку спіралі. При реалізації системи роблять акцент на аналізі ризиків та концептуальному моделюванні системи.

Спіральна модель розробки системи природно відображує розробку програмного забезпечення. Вона орієнтована на розробку великих складних систем, до яких висуваються вимоги по стійкості роботи в умовах обробки великих об’ємів даних і високою продуктивністю.

Недоліками спіральної моделі є підвищені вимоги до замовника, важкість контролю та керування часом розробки.

Інкрементна модель відрізняється від спіральної виробкою вимог. При інкрементній стратегії спочатку визначають всі вимоги до системи - системні вимоги та вимоги до програмного забезпечення.

Потім розроблюють базову систему з використанням базових вимог. Далі утворюють нову версію системи, що реалізує додаткові можливості.

Версії системи змінюються, доки не будуть реалізовані всі можливості. При реалізації інкрементної стратегії кожна версія системи, продукт, що працює.

Вибір моделі ЖЦ системи залежить від можливостей розроблювача та замовника.

1.3 Концептуальне проектування програмної системи

Для успішної реалізації проекту треба спочатку описати об’єкт проектування адекватно реальному об’єкту. Опис застосовують на стадіях системного аналізу системи, при створенні специфікацій програмного забезпечення, концептуального проектування системи.

Опис системи може створюватись на різних мовах: графічній, математичній, природній. Для маніпулювання описом (аналізу опису, його перетворюванням) існують програмні засоби підтримки. Тому опис системи повинен подаватись у формалізованому вигляді.

Для опису виробничої системи (наприклад, функціонування певного цеху підприємства) або програмної системи застосовують методи візуального подавання інформації у вигляді структурних або об’єктно-орієнтованих діаграм. Графічні засоби подавання майбутньої системи дозволяють розроблювачам у наочному вигляді вивчати її, перетворювати представлення системи у відповідності з поставленими цілями, вимогами та обмеженнями.

Будемо називати моделлю системи представлення системи, яке описує суть системи.

Для різних типів моделі суть системи різна. Наприклад, модель може описувати поведінку системи, функції системи, її інформацію. Суть системи подають у поняттях. Слово «концепт» означає поняття.

Концептуальна модель – це абстрактний, достатньо повний для досягнення мети, опис об’єктів системи, їх властивостей, відношень між ними на понятійному рівні.

Вибір базових об’єктів системи для відображення, залежить від типу моделі. Для функціональних моделей системи основним об’єктом є функція, а допоміжні об’єкти дані. Для інформаційних моделей системи основним об’єктом є дане.

Графічні мови дозволяють подавати модель у формалізованому вигляді. Вони будують об’єкти за суворими правилами на основі графічних примітивів. Примітив – це графічний елемент моделі, який не можна розкласти на складові. З примітивів будують конструкції моделі.

Одна система може подаватися множиною моделей, кожна з яких відбиває якусь грань її сутності. Кожна модель дозволяє досліджувати одну властивість або зв'язок між властивостями системи. Утворити повну групу моделей, що описують всі властивості системи, не має сенсу і не можливо. По Тюрінгу повна група моделей - сама система.

Архітектура виробництва – це динамічна концептуальна модель, яка дозволяє досліджувати, як взаємодіють між собою важливі процеси виробництва у часі та зі зміною істотних для виробництва параметрів. Під істотними параметрами розуміють параметри, що визначають хід процесів виробництва.

Використовуючи архітектуру виробництва, можна досліджувати, як зміна значень параметрів буде впливати на об’єми та якість продукції. Архітектура дозволяє керувати виробництвом, моделюючи різні виробничі ситуації.

Для моделювання архітектури виробництва важливо своєчасно одержувати інформацію про нього, а саме мати інтегровану виробничу базу даних. Модель дозволяє керувати виробництвом через керування даними. Тому дані про виробництво вважають ресурсом виробництва.

Управління підприємством у сучасних умовах це - управління інформацією. Успіх на ринці збуту залежить від того, наскільки ефективно та чи інша організація використовує інформаційні ресурси, які технології використовує вона для управління інформацією.

Слова Бенджаміна Дизраєлі (1804-1881), лорда Біконсфілда, прем'єр-міністра та міністра фінансів, видатного державного діяча Великобританії XIX сторіччя, не втрачають своєї актуальності і сьогодні: "Найбільшого успіху домагається той, хто володіє кращою інформацією".

Для моделювання системи або утворення архітектури виробництва треба мати мову моделювання, процедури та правила утворення моделей, критерії оцінки моделей, а саме методологію.

1.4 Методології концептуального проектування IDEF

Методологія – вчення про структуру, логічну організацію, методи та засоби діяльності. Під діяльністю ми будемо розуміти концептуальне проектування системи.

Під структурою діяльності концептуальне проектування систем ми будемо розуміти етапи роботи над моделлю, та зв'язки між етапами. Під логічною організацією діяльності розуміють: хто реалізує кожний з етапів, які функції вони виконують і як взаємодіють між собою розроблювачі.

До засобів розробки моделі відносять графічну мову створення моделі та інструментальні програмні засоби для підтримки процесу створення моделі, а до методів відносять процедури створення та контролю моделі.

Складні системи утворюють великі колективи фахівців різних професій. Сумісна робота вимагала *стандартизації засобів* створення моделей для точного і однозначно зрозумілого змісту моделей всіма фахівцями. Стандартизація дозволила б забезпечувати інтеграцію моделей; можливість застосовувати їх на схожих підприємствах та розроблювати інструментальні програмні засоби для підтримки процесів створення моделей. Також на підприємствах працювали спеціалісти з різним досвідом. Необхідні були стандартні засоби передачі досвіду фахівців молодим розроблювачам.

Сучасна стандартна методологія концептуального проектування містить стандартну графічну мову створення моднлі, вчить як одержувати інформацію про систему, описує ролі та функції

розроблювачів, етапи розробки моделі, процедури взаємодії розроблювачів, процедури оцінки якості моделі, які дозволяють створювати моделі, що відповідають меті та точці зору колективу.

Існує два підходи до моделювання систем: структурний та об'єктно-орієнтований. Ми вивчатимемо структурний підхід реалізований у методології функціонального моделювання IDEF0, яка відноситься до групи стандартних методологій IDEF.

Розглянемо причини створення групи стандартних методологій IDEF. У 1981 році в галузі ВПС США була впроваджена «Програма інтегрованої комп'ютеризації для автоматизації виробництва - ICAM». Метою було збільшення продуктивності виробництва шляхом застосування комп'ютерних технологій.

Програма передбачала розробку систем загального призначення типових для всіх підприємств галузі: диспетчерські служби; складування матеріалів; бухгалтерський облік, планування, тощо. Але цього було мало.

На початку робіт з'ясувалося, що не можна розглядати окремі ділянки підприємства без врахування їх зв'язків з середовищем. Для аналізу процесів всього виробництва, необхідно було мати сучасні засоби, які б дозволяли створювати модель підприємства – його архітектуру.

Модель виробництва дозволила б аналізувати процеси з метою збільшення продуктивності; можливості модернізації виробництва; дозволила б швидко перебудовувати підприємство на новий вид продукції.

Так була створена стандартна методологія розробки складних систем IDEF(Integrated Definition).

Сьогодні, методології групи IDEF - група взаємопов'язаних і взаємозалежних методологій. Кожна методологія призначена для створення моделі, що визначає певний аспект системи, на різних етапах її проектування.

В IDEF існують такі методології концептуального проектування:

– IDEF0 створення функціональної моделі системи, середовища системи;

– IDEF1X створення інформаційної моделі системи, середовища системи;

– IDEF2 створення моделі поведінки системи. Модель описує зміну станів системи у часі залежно від параметрів;

– IDEF3 моделювання безперервних процесів системи;

– IDEF4 створення об’єктно-орієнтованих моделей системи;

– IDEF5 систематизації об’єктів виробництва. Моделювання загальних категорій та закономірностей об’єктів виробництва (онтологія);

– IDEF6 зберігання раціонального досвіду розробки систем;

– IDEF8 моделювання діалогу людини з технічною системою з метою створення оптимальних умов праці, щоб збільшити продуктивність;

– IDEF9 аналізу умов і обмежень майбутньої системи, їх впливу на рішення, які приймаються в процесі реінжінірінга;

– IDEF14 моделювання мереж з описом конфігурацій, черг, мереживних компонент, вимог до надійності.

Ручна розробка проекту завжди породжувала проблеми на всіх етапах життєвого циклу системи: неадекватна специфікація вимог; нездатність виявляти помилки в проектних рішеннях; низька якість документації, що знижує експлуатаційну якість; затяжний цикл розробки та незадовільні результати тестування.

Для підтримки методологій концептуального проектування систем і прискорення побудови моделей існує спеціальні програмні системи.

Такі системи називають CASE-системами. Абревіатуру CASE розуміють у поширеному та вузькому сенсі. У вузькому сенсі Computer Aided Software Engineering означає автоматизоване проектування програмного забезпечення. У поширеному сенсі Computer Aided System Engineering означає концептуальне проектування складних систем.

Широко відомі наступні програмні продукти, що підтримують методології групи IDEF.

1 BPWIN – застосовують для розробки функціональних моделей. Методологія IDEF0.

2 ERWIN - застосовують для розробки інформаційних моделей. Методологія IDEF1X.

3 OOWIN - застосовують для об’єктно-орієнтованого проектування інформаційних моделей. Методологія IDEF4.

4 DESIGN/IDEF - застосовують для розробки функціональних та інформаційних моделей методологій IDEF0, IDEF1, IDEF1X. Інформація, що міститься в IDEF-моделях, може експортуватися в бази даних і електроні таблиці для подальшого аналізу, а самі моделі можуть експортуватись у пакети Design/CPN або Workflow Analyzer - пакети динамічного моделювання та аналізу складних систем.

Design/IDEF версії 3.5 можна настроювати на стандартний набір із 13 найбільш розповсюджених СУБД і на СУБД користувача. Design/IDEF дозволяє автоматично генерувати структури бази даних на мові SQL.

5 Workflow Analyzer - застосовують для імітаційного моделювання та динамічного аналізу систем. Методологія IDEF2.

У дисципліні ми будемо вивчати оболонку DESIGN/IDEF для підтримки методології IDEF0. Оболонка контролює виконання правил графічних мов при побудові функціональних моделей; полегшує їх побудову, автоматично створюючи за правилами необхідні об’єкти. Оболонка також дозволяє будувати моделі типу ER (сутність - зв’язок).

1.5 Методологія IDEF0

Методологію IDEF0 призначено для побудови функціональних моделей. Методологія містить провідний документ про графічну мову подавання функціональної моделі (IDEF0 - моделі).

Графічне подавання IDEF0-моделі має ряд переваг перед природною мовою.

Графічна мова подавання функціональної моделі

Зміст функцій системи подається у природній для людини формі, поступово деталізуючи основну функцію системи. Основна функція системи деталізується функціями модулів програми, а функція кожного модуля деталізується функціями процедур модулів. Така деталізація функцій системи за рівнями дозволяє людини просто розуміти функціональну модель.

У кожній предметній області є своя термінологія. Графічне подавання функціональної моделі та визначення понять в глосарію моделі робить зрозумілим зміст моделі фахівцям будь-якої предметної області (аналітикам, програмістам, експертам, адміністраторам).

Конструкції моделі будуються за суворими правилами мови на базі графічних примітивів, що дозволяє програмному забезпеченню аналізувати та перетворювати модель, а людині однозначно розуміти модель.

Формулювання назв функцій та необхідних для розуміння роботи функцій об’єктів за правилами мови, дозволяють точно і стисло передавати зміст функцій системи.

IDEF0-модель застосовується фахівцями для поглибленого розуміння і аналізу функцій системи, середовища функціонування системи, взаємодії її компонентів. Особливо це стосується таких складних систем, як САПР.

Методи збору інформації

До провідних документів методології відносять також методи збору інформації для створення функціональної моделі. Серед існуючих методів (анкетування робітників, знайомство з літературою та документацією, знайомство з аналогами системи) методологія виділяє опит експертів.

Методологія навчає як проводити опит експерта. У документах подаються: типи опитів, стадії опиту експерта; правила проведення опиту, правила оформлення результатів опиту.

Методологія вчить, як за короткий строк від експерта одержати максимальний об’єм якісної інформації.

Читання та побудова IDEF0 моделі

Методологія вчить, як вірно читати модель за її змістом для загальної уяви або для детального знайомства з частиною моделі. В документах методології подаються основні етапи розробки моделі та які дії виконуються на кожному етапі.

Колективна розробка IDEF0 моделі

При розробці програмних систем важливо вміти організовувати сумісну роботу колективу розроблювачів і управляти ходом виконання проекту. Управління проектом вимагає знань про управління та вмінь застосовувати їх на практиці.

Методологія IDEF0 визначає: структуру колективу розроблювачів; вчить, як розподіляти ролі та функції розроблювачів під час сумісної розробки або переробки функціональної моделі.

Методологія визначає правила рецензування моделі, відповідальності за невірні результати, обговорення розбіжностей між автором і рецензентом, ведення та оформлення документації.

Контроль якості IDEF0 - моделі

У документах контролю якості моделі подаються синтаксичні та семантичні критерії якості моделі. До синтаксичних правил відносять правила побудови конструкцій моделей. Семантичні критерії вказують на вірність, зрозумілість та повноту моделі; а також узгодженість між собою компонент моделей та стислість формулювань.

1.6 Основні поняття функціональної моделі IDEF0

1.6.1 Функціональний блок і дуги

Базовим об’єктом діаграм функціональної моделі є *функція.* Функцією являється будь-яка *виробнича операція*. На діаграмі назву функції розташовують у середині блоку та записують дієсловом у інфінітиві або групою дієслова. Блок називають функціональним. У правому нижньому куту блоку розташований його ідентифікатор. Дивись рисунок 1.4.

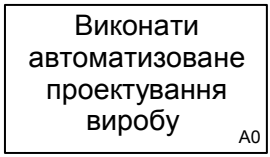


Рисунок 1.4 - Приклад функціонального блоку

На діаграмі функціональний блок зображують разом з допоміжними об‘єктами, що використовуються функцією у роботі, керують нею, виконують її або виробляються нею. Такі об’єкти зображують дугами (стрілками). Кожна дуга має мітку, що визначає її зміст. Мітки записують іменником або групою іменника. Дивись рисунок 1.5.

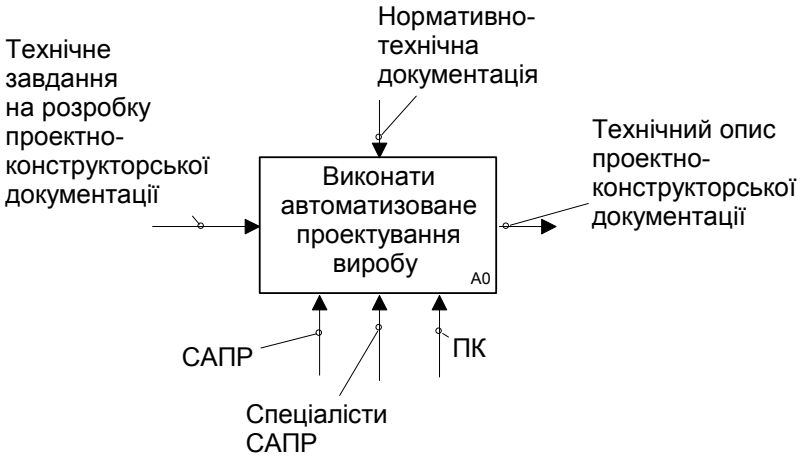


Рисунок 1.5 - Приклад функціонального блоку з дугами

Дугу проведену до: лівої сторони блоку називають «вхідною», вона вказує, що перероблюється функцією. Дугу від правої сторони блоку називають «вихідною», вона вказує на результати роботи функції. Дугу проведену до нижньої сторони блоку називають «механізмом», вона виконує функцію. Дугу проведену до верхньої сторони блоку називають «керуючою», вона визначає коли та як виконується функція. Вхідні, вихідні та керуючі дуги *називають обмеженнями виконання функції*. Функція виконується лише при

наявності цих дуг. Обмеженнями можуть бути інформація, люди, механізми, інші матеріальні об’єкти.

Функція включається в роботу та контролюється керуючою дугою. Дуга керування повинна бути хоча б одна для кожної функції. Кількість дуг проведених до кожної грані може бути не більше 4. Вхідні дуги перероблюються функцією у вихідні дуги. Вихідна дуга повинна бути хоча б одна.

1.6.2 Ієрархічна структура з діаграм

Модель IDEF0 - це ієрархічна структура з IDEF0-діаграм, які визначають, що робить система. Структура подається разом з пояснюючим текстом, глосарієм і діаграмами FEO (For Expression Only). Модель побудована за суворими синтаксичними правилами, що пов’язують діаграми в єдине ціле.

Для прикладу розглянемо функціональну модель процесу створення САПР, подану у книзі Норенкова І.П. «Основи автоматизації проектування».

Вершиною ієрархічної структури з діаграм є діаграма А-0. На діаграмі А-0 зображують одну функцію, яка визначає *призначення системи.* Функціональний блок з призначенням системи завжди має ідентифікатор А0. Дуги на діаграмі А-0 показують інтерфейс між основною функцією системи і середовищем системи. Тому діаграму А-0 називають інтерфейсною, а дуги функції інтерфейсом системи.

Мету утворення системи, мету утворення функціональної моделі та точку зору на утворення моделі розташовують на діаграмі А-0 під функціональним блоком.

Зміст моделі залежить від мети утворення системи та моделі, від точки зору розроблювачів або замовників. При розробці моделі колективом певного підприємства модель буде мати функції, які виконують на ньому. Якщо мета розробки моделі аналіз системних вимог, то в моделі знаходяться функціональні вимоги до системи.

Дивись рисунок 1.6.

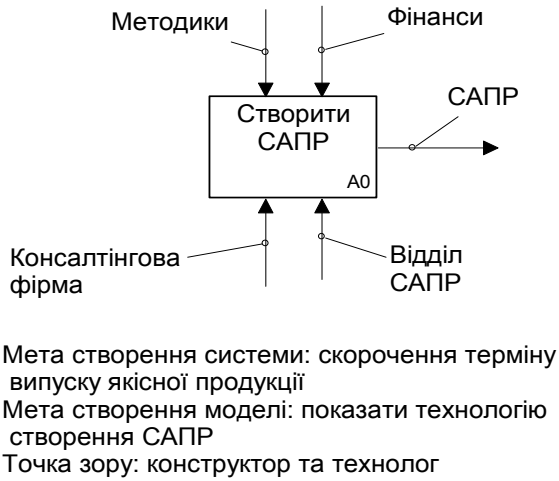


Рисунок 1.6 - Діаграма А-0 функціональної моделі «Процес створення САПР»

Існує п’ять точок зору на зміст програмного продукту і, як наслідок, на модель системи: договірна (замовника та постачальника); експлуатаційна (оператора та користувача); інженерна (розроблювачів системи та спеціалістів по супроводженню); підтримки (виконавців допоміжних процесів); управлінська (менеджерів).

Функціональне моделювання різних точок зору дозволяє реалізувати різні моделі, що відповідають кожній точці зору. Після чого всі точки зору враховують у одній моделі. Такий підхід дозволить реалізувати якісний програмний продукт.

На діаграмі А-0 функція має дуги тільки для відкритих систем. Дуги на діаграмі А-0 називають зовнішніми дугами, вони являються інтерфейсом між середовищем і системою. Замкнена система не має середовища і тому не має дуг для спілкування.

На діаграмі рівня нижче ніж діаграма А-0 розташовують функції, що деталізують функцію діаграми А-0.

Функція, що деталізується, називається вузлом. Діаграма, що деталізує вузол, називається діаграмою декомпозиції, а діаграма, що містить вузол, називається батьківською діаграмою. Діаграма

декомпозиції має ідентифікатором ідентифікатор свого вузла, а назва функції стає назвою діаграми декомпозиції.

Зв'язок діаграм моделі за ідентифікаторами вузлів та назвами функцій дозволяє розглядати модель як єдине ціле не тільки за змістом, а і формально.

На кожній *діаграмі декомпозиції* можна розмістити від 3 до 6 блоків. Кожен блок має свій ідентифікатор, який складається з ідентифікатору діаграми та номера блоку на діаграмі. Виняток - ідентифікатори блоків на діаграмі А0, вони записуються без 0. Наприклад, А15 – блок номер 5 на діаграмі А1, але А2 блок номер 2 на діаграмі А0. Приклад діаграми декомпозиції, яка деталізує призначення системи (функцію з блоку А0) дивися на рисунку 1.7.

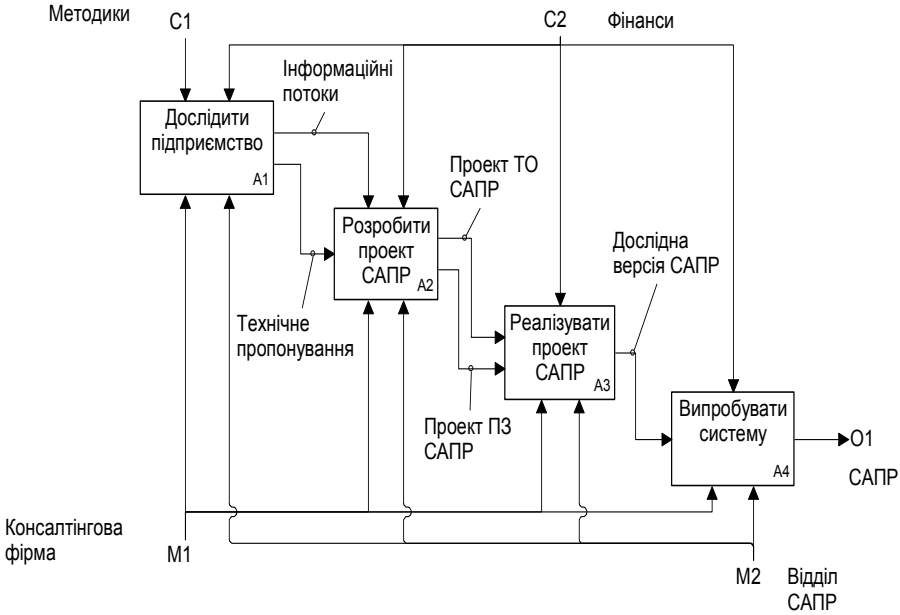


Рисунок 1.7 - Діаграма декомпозиції А0 «Створити САПР»

Формулювання функцій на рисунку 1.7 загального характеру. Вони характеризують різні етапи роботи над створенням САПР. Кожна функція має свої інтерфейсні дуги, через які вона спілкується з

іншими функціями. Вихідні дуги однієї функції є вхідними або керуючими дугами іншої функції.

Вміст діаграми А0 не повинен бути ширше вмісту її вузла. На діаграмі А0 не можна розмістити функцію «Експлуатувати САПР». В той же час вміст діаграми повинен містити всі етапи створення САПР. Рівень узагальнення назв функцій та міток дуг діаграми декомпозиції А0 має бути однаковим.

Функції діаграми декомпозиції можуть також деталізуватися на нижчому рівні (не обов'язково всі). Вибір функції на діаграмі для деталізації залежить від мети створення моделі, тобто що треба показати детально. На рисунку 1.8 показана деталізація функції «Дослідити підприємство» А1 діаграми А0.

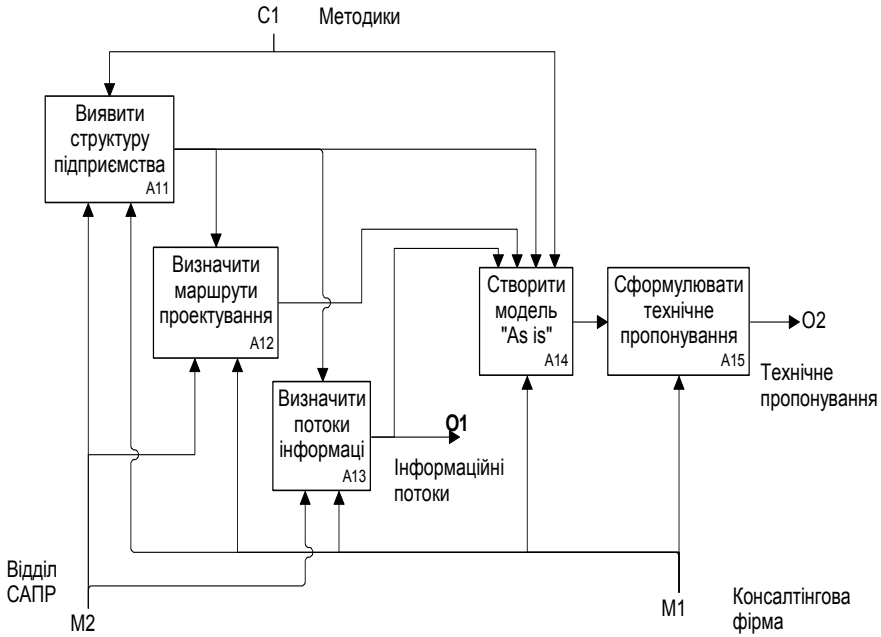


Рисунок 1.8 - Діаграма декомпозиції А1 «Обстежити підприємство»

На зміст діаграми декомпозиції А1 впливає *контекст її вузла* на батьківській діаграмі А0. Контекстом являються функції та дуги, які

розташовані разом з вузлом «Дослідити підприємство» на батьківській діаграмі.

Перелічені правила створення діаграм декомпозиції А0 та А1 застосовують при створенні будь-якої діаграми декомпозиції.

Створення діаграм декомпозиції продовжується до тих пір, поки не буде подана суть моделі. Кажуть, що мета, з якою утворюють модель, досягається.

Ієрархічна структура з діаграм спрощує розуміння змісту моделі. Кожний рівень ієрархії призначено для свого читача. Верхні діаграми моделі містять функції загального рівня і призначені для керівництва. Діаграми нижчих рівнів призначені для спеціалістів, що працюють на робочих місцях. Діаграми містять функції, які визначають виробничі операції. Тому назви функцій та міток дуг на нижчих рівнях мають бути більш конкретизованими.

Діаграми моделі зображують на стандартній майстер-сторінці. Верхні поля сторінки – робоча інформація, яку використовують тільки під час розробки діаграми, а потім відрізають. Нижні поля сторінки використовують для ідентифікації діаграми та розробника. На середній частині сторінки зображують саму діаграму: функціональні блоки та їх дуги.

На початку конспекту ми давали означення системи, в якому стверджували, що система – це колектив фахівців, який виконує певну діяльність (загальну функцію) для досягнення спільної мети і застосовує для цього апаратуру та програми.

Функціональна модель, яку ми розглядали на рисунку 1.6 – 1.8 показує функції такої системи.

1.7 Зв’язок функцій дугами на діаграмі декомпозиції

Функції діаграми декомпозиції завжди деталізують свій вузол, розкривають його зміст. Тобто функції, що розташовані на одній діаграмі декомпозиції пов’язуються змістом.

Зв'язок між функціями встановлюється дугами. Вихідна дуга однієї функції може стати керуючою або вхідною дугою інших функцій.

Кілька функцій діаграми можуть застосовувати або створювати одну дугу. Щоб показати застосування дуги кожною функцією, дуга гілкується і містить свою назву на спільній частині. Дивися керуючу дугу «директива» та вхідну дугу «гроші» на рисунку 1.9.

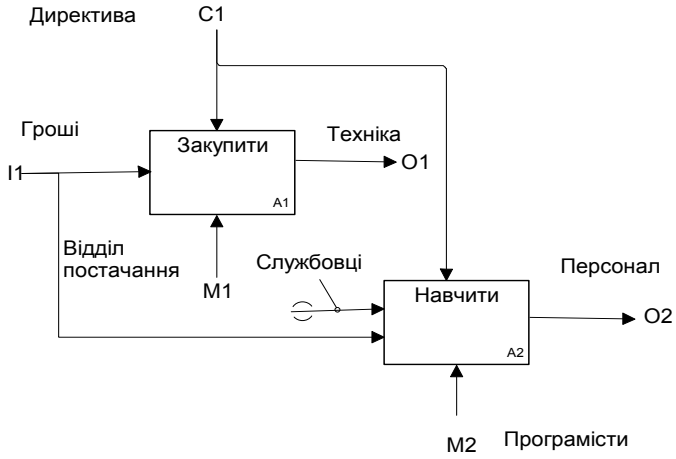


Рисунок 1.9 - Приклад використання дуги різними функціями

Якщо кілька функцій діаграми виробляють однотипні дуги, то їх можна злити в одну вихідну дугу. Така дуга також містить свою назву на спільній частині. Дуга загального характеру при застосуванні або створенні кількома функціями може деталізуватися.

Загальна мітка такої дуги розміщується на спільній частині. Спільну частину дуги називають трубопроводом. Щоб показати яку частину пояснювальної записку застосовує кожна функція, на гілках дуги вказують мітки частин Дивися на рисунку 1.10 дуги: «реферат», «інструкція системному програмісту» деталізують дугу «пояснювальна записка».

Загальна мітка такої дуги розміщується на спільній частині. Спільну частину дуги називають трубопроводом. Щоб показати яку

частину пояснювальної записку застосовує кожна функція, на гілках дуги вказують мітки частин.

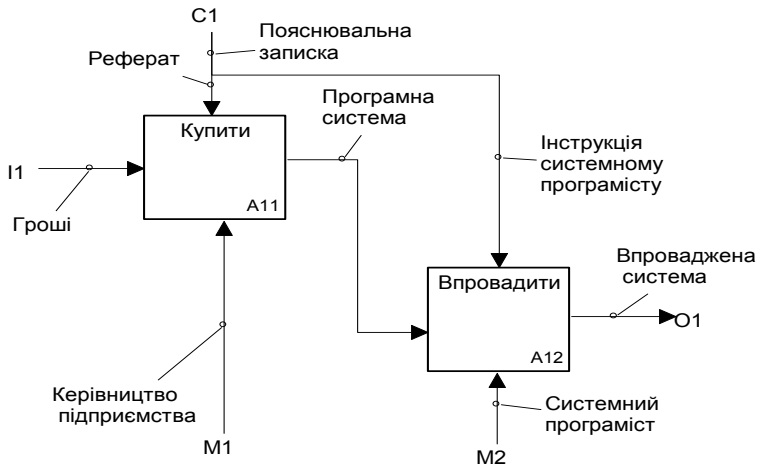


Рисунок 1.10 - Приклад деталізації дуги

Дуги, які породжуються на діаграмі декомпозиції, та мають два кінця на сторонах блоків називають внутрішніми дугами. На рисунку 1.10 дуга «програмна система» внутрішня дуга.

Дуги діаграми декомпозиції, що породжуються вузлом діаграми, та мають один вільний кінець називають прикордонними дугами.

Прикордонні дуги повинні відповідати дугам вузла, завдяки чому діаграми моделі пов'язуються в єдине ціле і модель буде несуперечливою.

На рисунку 1.10 прикордонними дугами є: «гроші», «керівництво підприємства», «системний програміст», «пояснювальна записка», «впроваджена система».

1.8 Порядок виконання функцій на діаграмі декомпозиції

Ідентифікатори функціональних блоків не вказують на порядок виконання функцій діаграми. Графічна мова подавання моделей взагалі не має спеціальних засобів для вказівки порядку виконання функцій. Але дуги можуть диктувати порядок їх виконання.

Функція може виконуватися тільки при наявності всіх необхідних дуг. Вихідна дуга однієї функції може стати керуючою або вхідною дугою іншої функції. Таким чином встановлюється порядок виконання функцій на діаграмі (але при наявності всіх інших дуг). Взагалі, функції на діаграмі можуть виконуватися одночасно або про порядок їх виконання навіть не можна нічого сказати. Тому кажуть, що *на діаграмі не зображується алгоритм.*

Про порядок виконання функцій «Закупити» і «Навчити» на рисунку 1.9 не можна нічого сказати. Функції можуть виконуватися одночасно або послідовно, бо невідомо, коли поступить дуга «службовці».

З рисунку 1.10 видно, що функція діаграми «Впровадити» не виконається раніше, ніж виконається функція «Купити». Поки не з’явиться дуга «Програмна система», функція «Впровадити» не може працювати. Діаграма на рисунку 1.10 зображує послідовне виконання функцій.

Якщо на діаграмі вилучити дугу «службовці», то функції виконаються паралельно. Для них дуги «директива» та «гроші» поступають одночасно.

На рисунку 1.11 зображено конкуренцію функцій за матеріал, що користується попитом. У цьому разі невідомо, яка функція буде працювати першою., якщо матеріалу недостатньо. При наявності матеріалу для обох функцій вони можуть працювати одночасно.

Вказаний приклад показує, що на функціональних діаграмах немає алгоритму, а тільки показують усі можливості порядку виконання функцій.

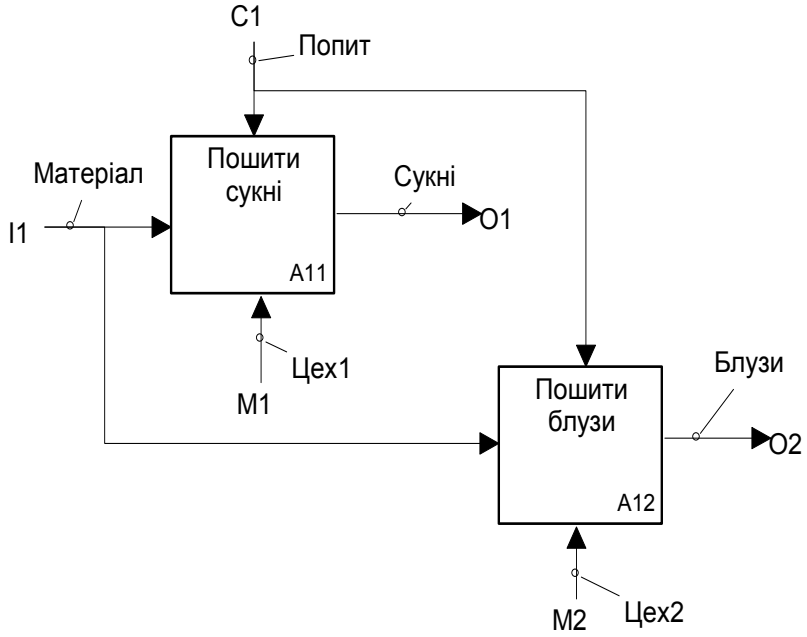


Рисунок 1.11 - Приклад конкуренції функцій за матеріальні об’єкти

Діаграми можуть містити зворотні зв’язки при помилках у роботі функцій та для зображення ітерацій. При ітераціях функції виконуються повторно.

На рисунку 1.12 показано приклад зворотного керуючого зв’язку за помилкою компіляції програми. Зворотом керує наявність помилки. За правилами графічної мови зворотний керуючий зв’язок подається на діаграмі *керуючою дугою зверху*. Зворотній зв’язок для накопичення даних подається *вхідною дугою низом*.

Наприклад, нехай треба ввести завдання тесту і накопичити їх у базі даних з тестами. Дуга, що показує зворотний зв'язок на увід наступного завдання тесту показує накопичення даних. Таку зворотну дугу подають низом.

Зворотний зв’язок майже завжди з’являється на діаграмах декомпозиції виробничих систем. Звортним зв’язком можна показати накопичення продукції цеху або браковану продукцію, що повертають на переробку, якщо це можливо.

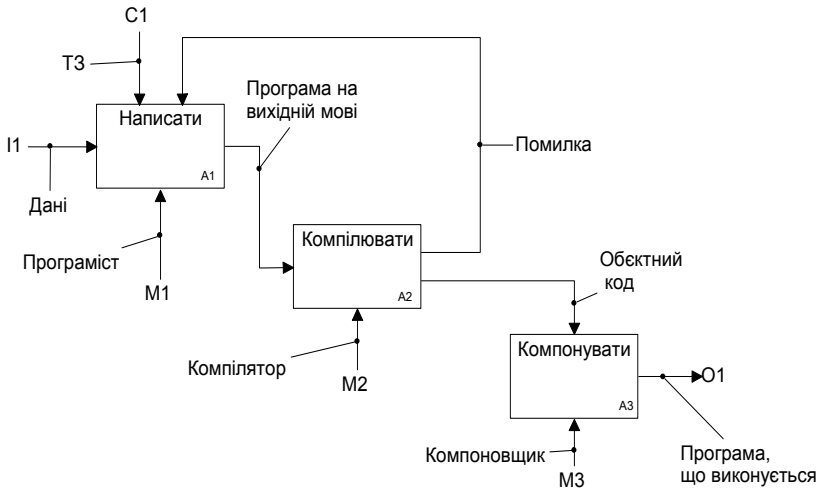


Рисунок 1.12 - Приклад діаграми із зворотним зв’язком

1.9 ICOM-коди прикордонних дуг та тунелі

Зв’язок між вузлом батьківської діаграми та діаграмою декомпозиції встановлюється через дуги вузла. Дуги вузла переносяться на діаграму декомпозиції, якщо це не заперечується тунелем ().

Тунель на кінці дуги, який лежить на стороні функціонального блоку вузла, заперечує перенос дуги на діаграму декомпозиції. Тунель на вільному кінці дуги, показує, що дуга не перенесена від вузла.

Застосовуючи тунелі можна показати семантичний зв'язок між класом і його елементами; цілим і його частинами тощо. Наприклад, клас медпрацівники, а елементи класу лікар, медсестра; ціле програмна система, а її частини модулі.

На вузлі батьківської діаграми зображують дугу «клас» і тунель на стороні блоку, а на діаграмі декомпозиції зображують дуги «елементи класу» і застосовують тунелі на вільних кінцях дуг.

Такі зв'язки між дугами «клас-елементи» «ціле-частини» тощо називаються абстрактними. Дуга «клас» не переходить на діаграму декомпозиції, а дуги «елементи» не приходять від батьківської діаграми.

Звичайно дуги загального характеру не переносять на діаграму декомпозиції, бо діаграми декомпозиції деталізують не тільки функцію а і дуги.

На рисунку 1.13.а подані дуги загального характеру, які не будуть використовуватись на діаграмі нижнього рівня. На рисунку 1.13.б показані дуги, що не прийшли від вузла.

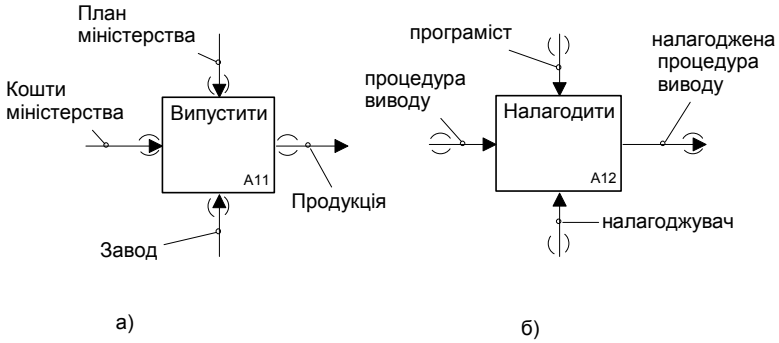


Рисунок 1.13 - Приклади використання тунелів

Дуги вузла, що переносяться на діаграму декомпозиції, називають *прикордонними дугами*.

Прикордонні дуги на діаграмі декомпозиції позначають мітками, які називають ICOM-кодами. ICOM-код вказує, *де у вузла розташована дуга, що помічена ICOM-кодом на діаграмі декомпозиції.*

Вхідні прикордонні дуги вузла позначають буквою I (input), вихідні прикордонні дуги позначають буквою O (output), керуючі

прикордонні дуги позначають буквою C (control), прикордонні дуги механізми позначають буквою M (mechanism).

Керуючі дуги та дуги механізмів вузла нумерують зліва направо. Вхідні дуги та вихідні дуги вузла нумерують зверху вниз. ICOM-коди дозволяють швидко знайти на вузлі прикордонні дуги діаграми декомпозиції.

Розглянемо приклад вузла і його діаграми декомпозиції для системи «Створення програми». Діаграми подано на рисунках 1.14 – 1.15.

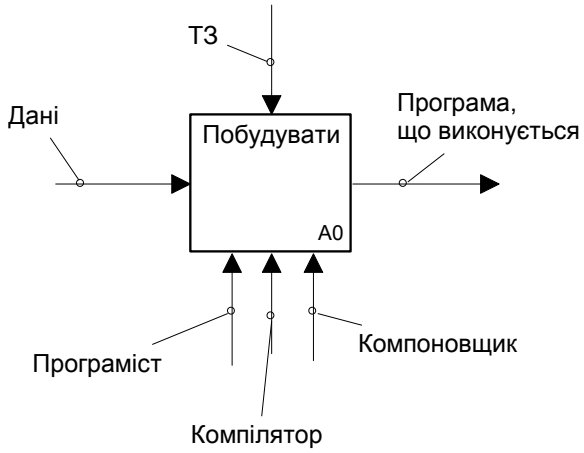


Рисунок 1.14 - Приклад вузла для системи «Створення програми»

Всі дуги, що використовуються функцією «Побудувати» переносяться на діаграму декомпозиції. Це не порушування правила про деталізацію функції, бо вузол використовує тіж самі дуги, що використовуються на діаграмі декомпозиції. Деталізується тільки функція.

Для прикордонних дуг діаграми декомпозиції створюються ICOM-коди відповідно їх розташуванню на вузлі. Дуги діаграми декомпозиції, що мають на вільному кінці тунель не мають ICOM-коду.

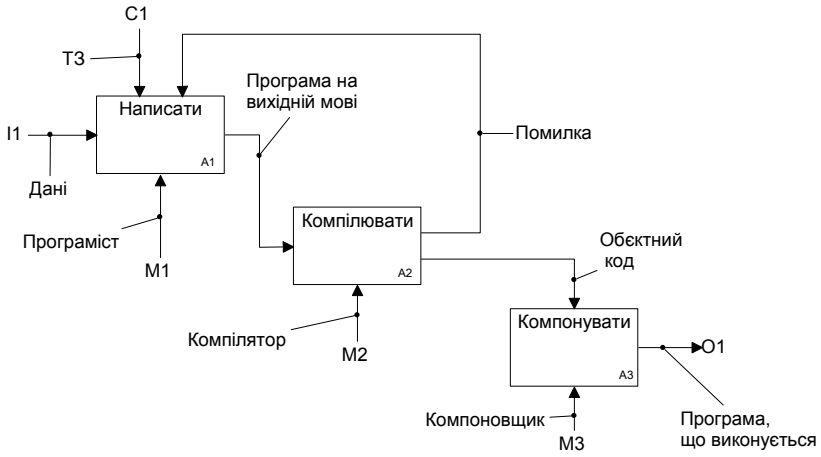


Рисунок 1.15 - Приклад діаграми декомпозиції для системи «Створення програми»

Якщо на діаграмі декомпозиції в свою чергу деталізується функція, то для її дуг ICOM-коди призначають заново. ICOM-коди не проставляють на зовнішніх дугах діаграми А-0.

1.10 Активації функцій

Функція може працювати в різних режимах роботи. Кожен режим вимагає різних комбінацій вхідних, керуючих дуг і механізмів. Комбінації дуг, що необхідні для роботи функції у певному режимі, називають активаціями. Між активаціями та результатами роботи функції встановлюється відповідність. Кожному режиму роботи функції-вузла на діаграмі декомпозиції відповідає певна функція та її активація і результат роботи.

Якщо функція не вузол, то не завжди можна визначити активації для її режимів роботи. В цьому разі треба обов’язково деталізувати функцію на діаграмі декомпозиції. Кожен режим функції треба показувати окремою функцією з відповідною активацію і результатами. Приклад функції-вузла, що працює в кількох режимах, подано на рисунку 1.16. За діаграмою неможливо виявити режими роботи функції. На рисунку 1.17 подана діаграма декомпозиції

функції-вузла. Кожна функція діаграми визначає певний режим роботи.

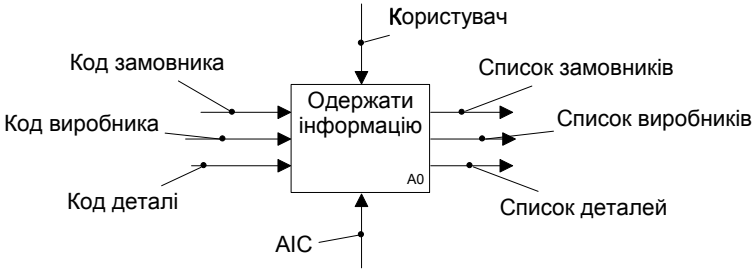


Рисунок 1.16 - Приклад функції-вузла, що працює в кількох режимах

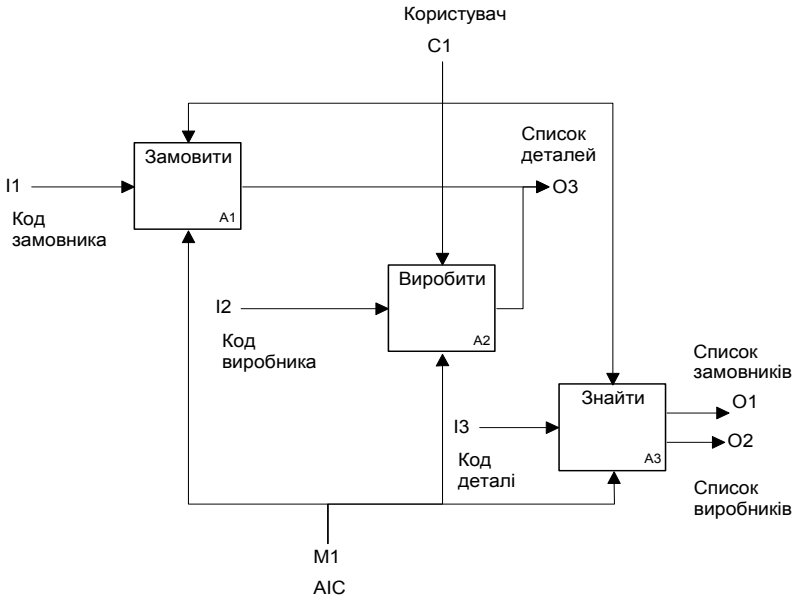


Рисунок 1.17 - Приклад діаграми декомпозиції, на якій кожна функція показує окремий режим

Якщо функція має більше 6 режимів (кількість функцій на діаграмі декомпозиції обмежена 6), то треба режими розподілити на групи, узагальнити для кожної групи назву функції і показати режими групами функцій. Після чого треба зробити додаткові діаграми декомпозиції, деталізувавши кожну групу режимів.

1.11 Допоміжні компоненти IDEF0-моделі

Функціональна модель є засобом передачі інформації між фахівцями. Для читання ієрархічної структури з діаграм необхідні знання термінології, що застосовується в моделі; графічна та текстова інформація, яку не можна подати конструкціями діаграм. Тому структура доповнюється глосарієм, текстом, графічними діаграмами, що містять рисунки, креслення тощо.

Глосарій - словник, в якому визначають спеціальні терміни і абревіатури у назвах функцій та мітках. Визначення необхідні для розуміння змісту незнайомої термінології; однозначного розуміння змісту текстів.

Наприклад, слово «мережа» може означати множину пов’язаних між собою комп’ютерів або тип моделі бази знань. Поняття визначають через атрибути або текстом.

Текст. Кожна діаграма IDEF0-моделі може доповнюватися однією текстовою сторінкою. Всі текстові сторінки, структуровані так, як структурована модель.

У тексті розповідають про те, що не можливо відобразити графічно. Текстова сторінка не повторює зміст діаграми, а доповнює його. Текст записується у формі змістовного оповідання. Текст до діаграми А-0 відносять до всієї моделі, він дає загальну характеристику всієї моделі. Текст роз’яснює, як досягається мета утворення моделі та точку зору розроблювача.

Текст утворювати необов’язково. Але, якщо його утворюють, то після закінчення розробки діаграми (моделі). В тексті можна застосовувати спеціальну мову посилань на моделі, діаграми, функції, дуги. Посилання повинні однозначно вказувати на об’єкт.

Для посилання на дугу використовують її код. Для цього дугу ідентифікують, як дуги вузла. Наприклад, перша керуюча дуга позначається C1. Ідентифікатор 4I2 означає другу вхідну дугу до блоку A4. Першу вихідну дугу блоку A2, яка стає першою вхідною дугою блоку A3 ідентифікують 2O1 до 3I1. Посилання на діаграму іншої моделі проекту може записуватися M01/A12, де M01 ідентифікатор моделі.

Діаграми FEO (For Exposition Only). Діаграми FEO використовують тільки для ілюстрації деяких моментів, зображених на діаграмах IDEF0. На діаграмі FEO можна подавати графіки, креслення, малюнки, тощо.

Кожна IDEF0-діаграма може мати кілька діаграм FEO. В DESIGN/IDEF є спеціальна команда ATTACH, яка прив’язує діаграму FEO до діаграми IDEF0.

2 КРИТЕРІЇ ЯКОСТІ ФУНКЦІОНАЛЬНОЇ МОДЕЛІ

2.1 Синтаксичні критерії графічного подавання функціональної моделі

Модель IDEF0 – формалізована конструкція, що побудована на базі примітивів та синтаксичних критеріїв вірності моделі (правил). Примітиви – елементарні компоненти діаграми, які не можна розкласти на складові. До примітивів відносять точки, мітки дуг, ICOM-коди, тунелі, блоки, ідентифікатори блоків, назви функцій.

Конструкція модель будуються за суворими синтаксичними правилами. Існують синтаксичні правила трьох рівнів побудови: локальних конструкцій, глобальних конструкцій (діаграм) і конструкції-модель.

Синтаксичні правила локальних, глобальних конструкцій та конструкції модель розглядають, як синтаксичні критерії вірності моделі. При створенні моделі програма DESIGN/IDEF контролює модель за синтаксичними правилами, для чого виконується компіляція моделі.

2.1.1 Критерії локальних конструкцій

Функціональний блок

Локальна конструкція - функціональний блок призначена для подавання на діаграмі функції системи у блоці. Назва функції повинна точно та однозначно відображати призначення функції. Формулювання назви повинно бути стислим.

Синтаксис функціонального блоку

1 Назва функції записується дієсловом у інфінітиві або групою дієслова.

2 Назва розміщується у середині блоку.

3 Блок має ідентифікатор. Ідентифікатор розміщується в нижньому правому куту блока.

4 Ідентифікатор блоку складається з ідентифікатору діаграми та номеру блока на діаграмі.

5 Ідентифікатор блоку починається з букви А, що означає функцію – Activity. Наприклад, ідентифікатор блоку А12 означає, що функціональний блок з номером 2 розташовується на діаграмі А1.

Дуги

Матеріальні об’єкти та інформація, що використовуються при роботі функції або виробляються нею, позначають стрілками. Стрілки прийнято називати дугами. Дуги класифікуються за призначенням: вхідні, вихідні, керуючі, механізми.

Дуги ідентифікуються мітками. Мітки вказують на зміст дуги. Мітка та дуга утворюють разом локальну конструкцію. Мітки повинні точно та однозначно відображують зміст дуг.

Синтаксис дуг

1 Мітка ідентифікується іменником у початковій формі або групою іменника.

2 Вхідні дуги перероблюються функцією у вихідні дуги.

3 Не існує функцій, які б не мали результату роботи – вихідних дуг.

4 Якщо вихідна дуга має мітку однакову з вхідною дугою, тобто не змінює своєї якості, то така дуга непотрібна функції.

5 На діаграмі мітки прив’язуються до дуги для наочності. У базі даних, що описує модель, дуга подається тільки міткою. А відношення встановлюються між об’єктами: функція – мітка дуги. Причому кожна мітка має властивості.

6 Дуга, що ініціює виконання функції, або керує нею називається керуючою дугою. Функція обов’язково повинна мати хоча б одну керуючу дугу.

7 Якщо дуга містить вхідні дані та керуючі, то її роблять керуючою дугою.

8 Не проводьте до сторони блока більше 4 дуг. Якщо потрібно розмістити більше дуг, об’єднуйте їх за змістом.

9 Дуги бувають одно сегментні та багато сегментні. Сегмент будується на основі двох точок і має один напрям.

10 Не можливо побудувати дугу, у якій обидві точки - вхідна та вихідна, розташовані не на стороні блоку.

2.1.2 **Критерії глобальних конструкцій - діаграм**

Існує два типа діаграм: діаграма А-0 і діаграма декомпозиції. Діаграма А-0 містить одну функцію, що є призначенням системи. Дуги функції являють собою інтерфейс між моделлю системи і моделлю середовища системи. На діаграмі зображують мету утворення моделі та точку зору на модель.

Діаграма декомпозиції розкриває зміст функції – вузла. Зміст вузла передається через семантичний зв'язок функцій діаграми декомпозиції. Зв'язок функцій діаграми декомпозиції зображується дугами.

Синтаксис діаграм

1 На діаграмі декомпозиції розташовують від 3 до 6 функціональних блоків. Діаграма, що містить менше 3 блоків, вважається не інформативною, а більше 6 блоків, складною для розуміння. Винятком є діаграма А-0. На ній розташований один функціональний блок.

2 *Кожен* функціональний блок має назву функції.

3 *Кожен* функціональний блок має унікальний номер.

4 Нумерація блоків починається з 1 і збільшується для кожного наступного блоку на 1. Винятком є блок на діаграмі А-0, він має номер 0.

5 *Кожен* функціональний блок має ідентифікатор.

6 Ідентифікатори *кожного* блоку діаграми складаються з ідентифікатору діаграми та номеру блоку. Винятками є ідентифікатори блоків на діаграмі А0, вони записуються А1, А2, а не А01, А02.

7 *Кожен* функціональний блок має хоча б одну керуючу та одну вихідну дугу.

8 До *кожної* сторони всіх блоків діаграми не може проводитися більше 4 дуг.

9 Всі вільні кінці дуг мають ICOM-коди або тунелі. Виняток зовнішні дуги діаграми А-0.

10 На діаграмі блоки розташовують по діагоналі. Проте, це необов'язково. Розміщення блоків диктується простотою читання діаграми.

11 Залишайте великі відстані між паралельними довгими дугами.

12 Для розгалужених дуг, що входять в кілька блоків, зображуйте одну й ту ж гілку до різних блоків на відповідному місці.

13 Показуйте накопичення даних на діаграмі дугою, що йде низом, а керуючий зворотний зв'язок дугою, що йде верхом.

14 Дуги розміщують на вертикальних та горизонтальних лініях. Мінімізуйте перетини дуг. Не доводьте дуги до межі діаграми. Чим менше перетинів дуг, тим зрозуміліша діаграма.

15 Мітки, перетини дуг, кути розміщують на деякій відстані від блоків.

2.1.3 Критерії конструкції – модель

Модель подається ієрархічною структурою з діаграм і супроводжуючою інформацією: глосарієм, текстовою структурою, діаграмами FEO.

Синтаксис моделі

1 Кожен вузол діаграми IDEF0 деталізується на одній діаграмі IDEF0 і пояснюється кількома діаграмами FEO. Верхній вузол моделі А0.

2 При декомпозиції вузла назва функції стає назвою діаграми декомпозиції, а ідентифікатор вузла стає ідентифікатором діаграми декомпозиції.

3 Існують діаграми рівня вище ніж А-0: А-1, А-2, тощо.

4 Інтерфейсом вузла є дуги вузла, тому дуги вузла переносяться на діаграму декомпозиції, як прикордонні дуги.

5 Взаємозв'язок між дугами вузла та прикордонними дугами діаграми декомпозиції встановлюється через ICOM-коди.

6 Дуги, що не прийшли від вузла, містять тунелі на вільних кінцях.

2.2 Семантичні критерії якості моделі

Методологія IDEF0 має критерії, які дозволяють ще на стадії концептуального проектування оцінити семантичну якість функцій майбутньої системи і відкоригувати її у разі потреби.

Якість системи залежить від етапу концептуального проектування. Розроблювачі системи не мають права робити систему ширшою ніж функції моделі або відкидати її функції. Розробка та оцінювання концептуальних моделей за семантичними критеріями складний процес, який вимагає колективу фахівців: авторів, рецензентів, експертів, читачів. Такий підхід дозволяє значно скоротити час розробки програми.

Типи семантичних критеріїв

Критерії – «повнота» та «вірність» застосовують для оцінки якості змісту моделі (діаграми). Критерії «стислість», «узгодженість» та «складність/зрозумілість» допомагають виявити зрозумілість моделі (діаграми) для читача.

Повнота моделі – це достатність інформації в моделі для досягнення мети, з якою створюється модель. Повнота діаграми декомпозиції – це достатність інформації на діаграмі для *контексту її вузла*. При застосуванні критерію повноти розробники оцінюють на скільки повно функції моделі відповідають уяві замовника про майбутню систему.

Вірність – ступінь, з яким інформація в діаграмі або моделі точно описує систему, що моделюється. Вірність найбільше суб’єктивний критерій.

Стислість – формулювання функцій та міток стислі, якщо вони не мають другорядної інформації. В той же час стислість тексту не повинна заважати точно передати зміст. Терміни з тексту повинні відповідати суті предмету, аудиторії, що читатиме модель.

Узгодженість – буває внутрішньою та зовнішньою. Зовнішня узгодженість – ступінь, з яким контекст діаграми або моделі узгоджується з їх змістом. Внутрішня узгодженість – одноманітність символіки та термінології на діаграмі та в моделі.

Складність/зрозумілість – між складністю та зрозумілістю існує зворотний зв'язок. Чим більш складна модель, тим менше її розуміють.

Зрозумілість – ступінь, з яким ясно, чи досягається мета діаграми (моделі). Або ступінь, з яким можна дати відповідь на питання "про що розповідає діаграма (модель)?".

2.3 Контроль якості діаграм моделі за критеріями Складність/Зрозумілість

Показниками *зрозумілості* діаграм декомпозиції моделі є: легкість знаходження головного шляху діаграми.

Шляхи виявляється, якщо рухатися за функціями, які пов’язані дугами. Серед шляхів діаграми є головний шлях, що веде до розв’язку основної задачі. Побічні шляхи можуть оброблювати збійні ситуації і т.п. .

До показників зрозумілості також відносять: число активацій функцій, число побічних шляхів; легкість ідентифікації активацій та шляхів; можливість покращати розміщення об’єктів на діаграмі.

Важливими показниками зрозумілості діаграми декомпозиції є зв’язність та зчеплення.

2.3.1 Зв’язність та зчеплення, їх роль у розумінні змісту діаграм декомпозиції моделі

1. Діаграма декомпозиції моделі подається у формалізованому вигляді, але зміст діаграми можна описати текстом природною мовою. Текст буде зрозумілим, якщо його речення пов’язані змістом, тобто разом являють собою оповідання. Кожне оповідання має свою мету.

Людина, що не вміє складати оповідання, напише текст, який буде погано пов'язаний за змістом. Такий текст буде незрозумілим.

Аналогічно сказаному про текст зрозумілість діаграми декомпозицій залежить від «зв’язності за змістом її функцій». *Чим більше функції діаграми пов’язані змістом, тим краще діаграма.*

2. Якщо описати дві функції діаграми детально, двома текстами, і в одному тексті посилатися на другий текст, який за змістом є частиною першого, то перший текст читатиметься важче і буде більше складним. Про такі тексти кажуть, що вони *зчеплені за змістом*.

Аналогічно сказаному про текст зрозумілість діаграми декомпозицій залежить від «зчеплення вмісту функцій діаграми». Чим менше функції діаграми *зчепленні за своїм вмістом*, тим краще діаграма.

Вивід. «Складність/Зрозумілість» діаграми декомпозиції залежить від двох характеристик: зчеплення та зв'язності.

Зчеплення – ступінь залежності вмісту функцій діаграми. Зчеплення між функціями бувають двох типів: за природою зв'язку, за структурою зв'язку.

Зв'язність – ступень зв’язності функцій на діаграмі декомпозиції *за змістом*.

2.3.2 Зчеплення функцій діаграми за природою зв’язку

1. Існує два типи зчеплень: паталогічне зчеплення і управління.

Якщо одна функція посилається на вміст іншої функції, яка за сенсом є вмістом першої, то між функціями встановлюється *патологічне зчеплення*, а дугу, що пов'язує функції називають "*патологічною дугою*".

На рисунку 2.1 показано дві функції. Насправді, функція «Вилучити відомості про книгу» належить до декомпозицій функції «Списати загублену книгу».

При паталогічному зчепленні порушується принцип створення діаграми декомпозиції: «Всі функції і дуги діаграми декомпозиції повинні бути одного рівня узагальненості». Такий тип діаграми не зрозумілий для реалізації, а зчеплення функцій найгірше.

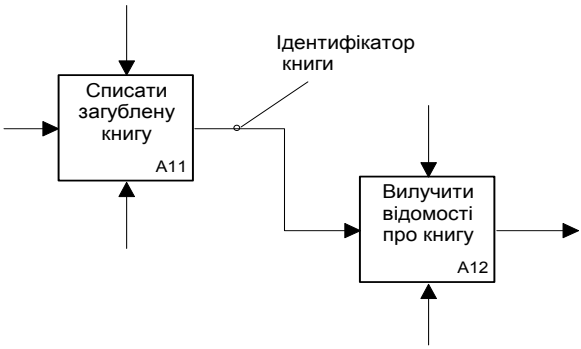


Рисунок 2.1 - Приклад патологічного зчеплення між функціями

2. Якщо вихідні дані однієї функції є потоком керування іншою функцію, то функції зчеплені «*за керуванням*». Потік керування треба відрізняти від керуючої дуги. Потік керування містить не дані, а інформацію, що керує роботою функції. Дивися рисунок 2.2.

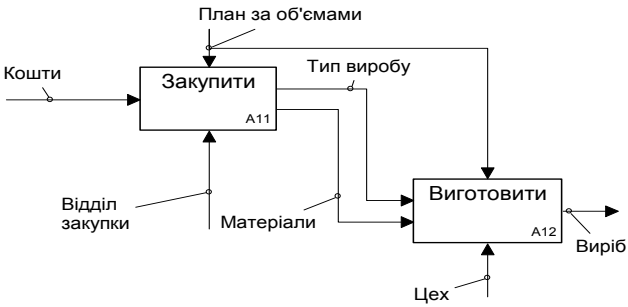


Рисунок 2.2 - Приклад зчеплення функцій за керуванням

Функція «Закупити» керує вибором режиму роботи функції «Виготовити». Керування виконується потоком керування представленим на діаграмі дугою «Тип виробу». Для керування потоком функція «Закупити» повинна детально «знати», які типи виробів функція може виготовити. *При керуванні потоком у функцій велике зчеплення за вмістом*. Тип зчеплення «за керуванням» краще ніж «патологічне зчеплення».

3 Дві функції "*нормально зчепленні*", якщо відсутні типи зчеплень 1 і 2.

2.3.3 Зчеплення функцій за структурою зв'язку

1. *Функції зчеплені за середовищем*, якщо вони всі мають доступ до середовища даних, без вказівки відповідності «функція – дані». Функції зчепленні за середовищем зображені на рисунку 2.3.

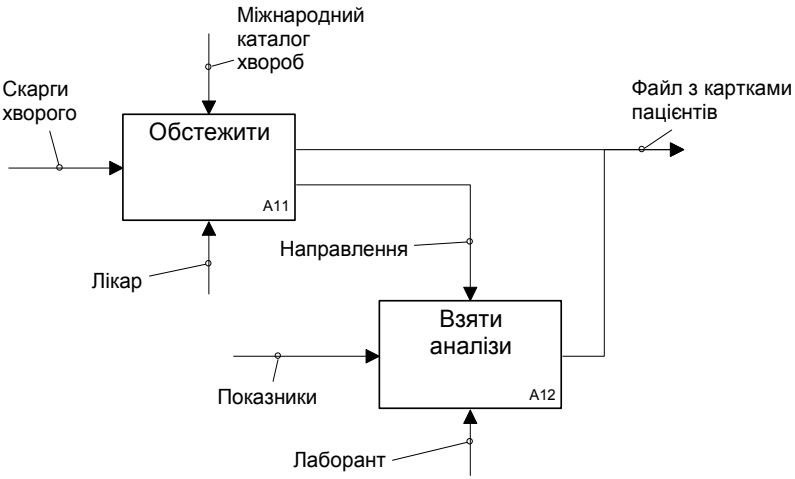


Рисунок 2.3 - Приклад зв’язку функцій за середовищем

Зчеплення за середовищем порушує принцип інкапсуляції. У функцій спільне середовище для даних – файл з картками пацієнтів. У прикладі, що розглядається, не вказано, з якою карткою працює кожна

функція. Лікарю та лаборанту доступні всі картки. В результаті можливі помилки записів і невірний діагноз.

2. *Функції зчеплені записом*, якщо вони мають доступ до не глобального набору даних (запису).

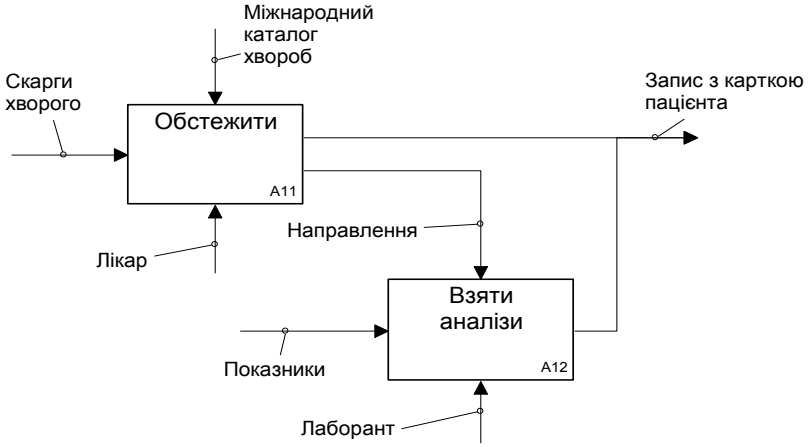


Рисунок 2.4 - Приклад зв'язку функцій через запис

Зчеплення функцій через запис краще, ніж через середовище. Функціям доступний тільки картка певного хворого. У функцій, що спілкуються через запис менше можливості зробити помилку.

Наприклад, запис скарг у місце для аналізів, дасть помилку при комп'ютерній діагностиці, але лікар помітить помилку. Зчеплення функцій через запис показано на рисунку 2.4.

3. *Два об'єкти IDEF0- моделі мають абстрактне зчеплення*, якщо між елементом і його компонентами точно зазначено відношення за змістом. До абстрактного зчеплення відносять наступні відношення між дугами «клас - елемент», «ціле - частина».

Приклад на рисунку 2.5 показує абстрактне зчеплення дуг за семантикою. Запис с карткою хворого містить результати обстеження лікарем і результати з аналізами, між ними має місце відношення «частина - ціле».

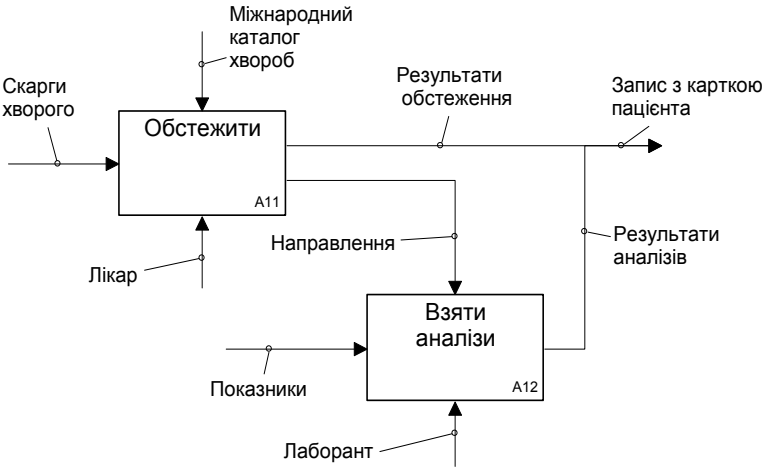


Рисунок 2.5 - Приклад абстрактного зв’язку за семантикою

2.4 Зв’язність функцій діаграми за змістом

Ступінь змістовної зв’язності функцій діаграми декомпозиції буває різного типу. Але часто функції збирають на діаграмі не за змістом. Такі діаграми найгірші.

Існують такі типи зв'язності.

0 Випадкова
1 Логічна
2 Часова
3 Процедурна
4 Комунікативна
5 Послідовна
6 Функціональна

За номерами змістовна зв’язність функцій збільшується, а якість діаграми поліпшується.

Випадкова зв'язність

При випадковій зв'язності функції діаграми мало пов'язані між собою змістом або зовсім не пов'язані. Дивись рисунок 2.6.

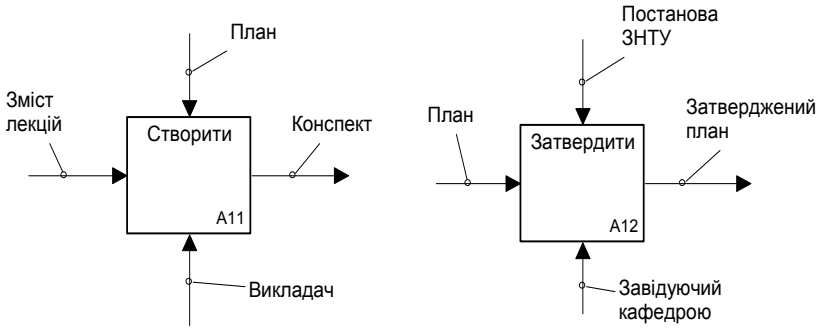


Рисунок 2.6 - Приклад випадкової зв’язності функцій

Функції «Створити» та «Затвердити» зовсім не пов’язані дугами. Вони розташовані на діаграмі тільки за темою «Кафедра». Такий тип зв’язності з’являється на початку створення діаграми, коли зв’язки між функціями тільки починають розроблюватися.

Логічна зв'язність

Функції розташовані разом на діаграмі не за змістом, а за спільним класом функцій або даних, за ознакою подібності.

На рисунку 2.7 функції зібрані разом, бо дуги відносяться одного класу «специфікація» і «програми».

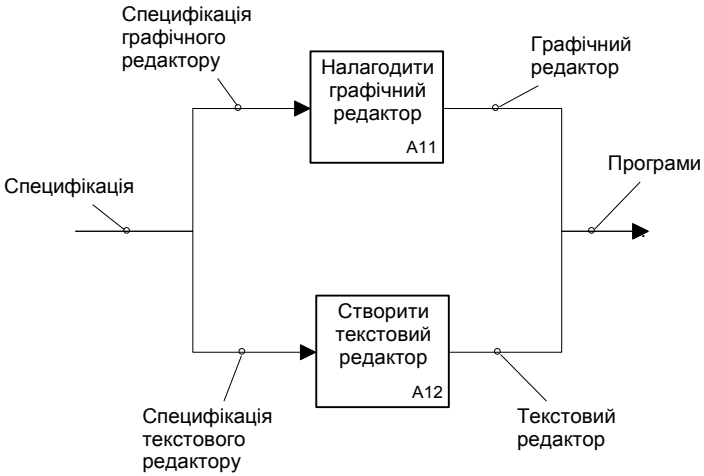


Рисунок 2.7 - Приклад логічного зв’язку функцій

Часова зв'язність

Функції діаграми пов’язані часом виконання. Але самі функції не пов’язані за змістом.

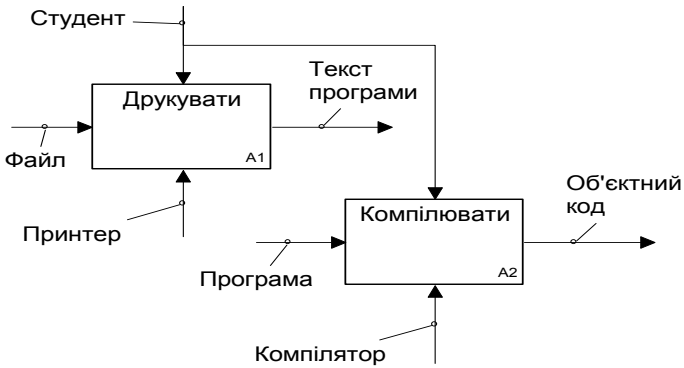


Рисунок 2.8 - Приклад часового зв’язку між функціями

На рисунку 2.8 студент одночасно виконує два завдання: компілює одну програму і друкує текст вже налагодженої програми.

Процедурна зв'язність

Кілька функцій не пов’язаних змістом використовують одну функцію формально. Функціональні блоки розташовуються на одній діаграмі тільки тому, що у різних за змістом процесах або циклах є однакова функція.

На рисунку 2.9 показано два різних за змістом процеси: створення текстового файлу і розв’язок рівняння. У процесах спільна функція «Надрукувати».

Процедурна зв’язність є найкращою із незмістовних типів зв’язностей між функціями, але все ж таки це незмістовна зв’язність. Людина краще розуміє діаграму, якщо її функції пов’язані змістом.

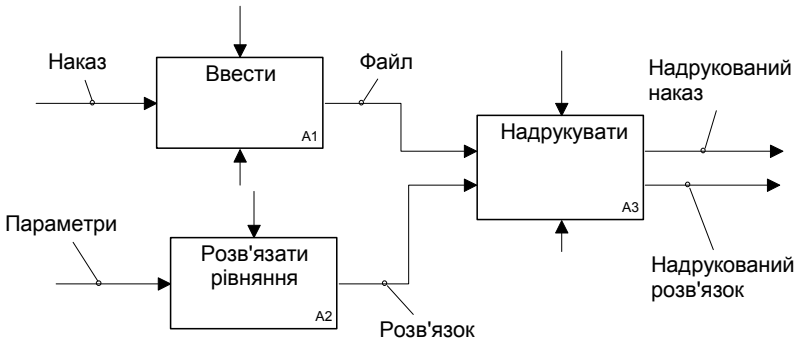


Рисунок 2.9 - Приклад процедурної зв’язності функцій

Комунікативна (інформаційна) зв'язність

Функції діаграми з комунікативною зв’язністю оброблюють, застосовують або виробляють одне дане.

Наприклад, нехай на діаграмі є наступні функції: роботи з лексемою, об’єднання рядків, виділення у рядку підрядку, пошуку символу у рядку.

Комунікативна зв’язність – перший тип зв’язності, де функції пов’язані змістом – обробкою одного даного. При комунікативній зв’язності виникає проблема, яку функцію треба обрати, щоб реалізувати певні дії.

Наприклад, мовою Visual Prolog можна одержати із рядку слово предикатом fronttoken або знайти пропуск предикатом searchchar, після чого одержати слово предикатом substring.

На рисунку 2.10 зображено функції мови програмування MuLisp, що оброблюють рядки.

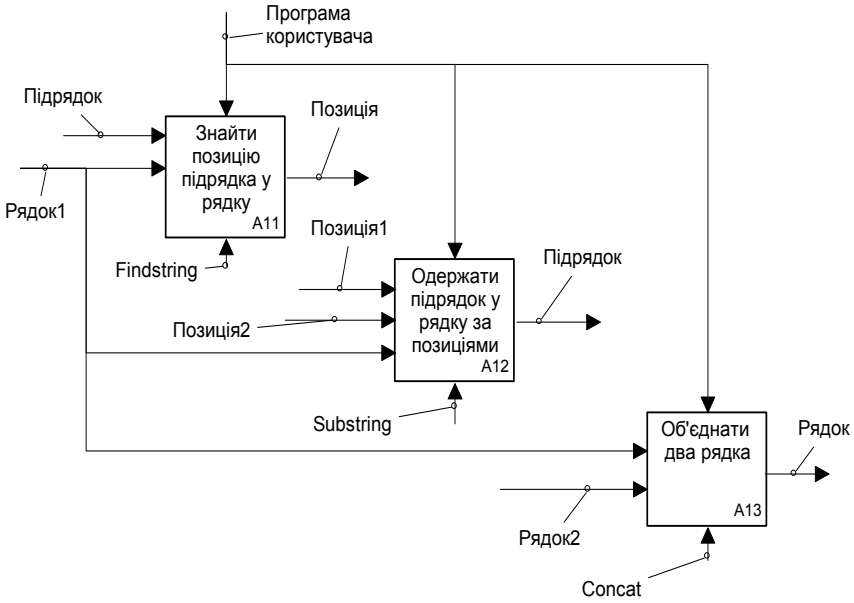


Рисунок 2.10 - Приклад комунікаційної зв’язності між функції

Послідовна зв'язність

При послідовній зв’язності вихідні дані однієї функції використовуються, як вхідні дані іншої функції. Так виникає між функціями змістовний зв’язок «причина - наслідок».

Змістовний послідовний зв’язок виникає у випадку, коли функції діаграми розв’язують одну задачу,

На рисунку 2.11 зображено послідовну зв’язність функцій, що розв’язують задачу – отримати інформацію від експерта.

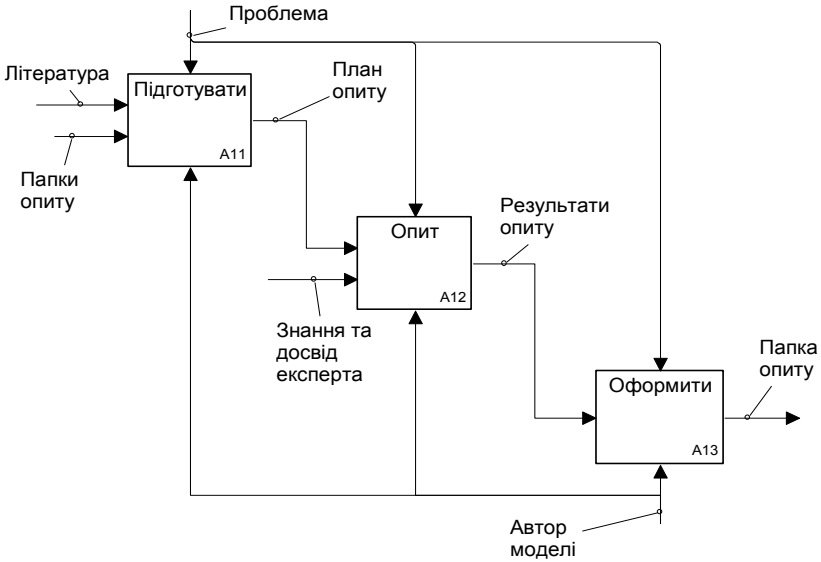


Рисунок 2.11 - Приклад послідовної зв’язності функцій

Функціональна зв'язність

Діаграма подає функціональну зв'язність, якщо всі функції діаграми працюють для виконання певної функції діаграми, розв’язують одну задачу, але виконання функцій не послідовне. Такий тип функціональної зв’язності називають повною функціональною зв’язністю.

Змістовний функціональний зв’язок між функціями найкращий. Діаграма з повною функціональною зв’язністю не може подаватися, як дві діаграми з функціональним зв’язком.

Хоча порядок виконання функцій діаграми не відомий, всі вони підготовляють дані для виконання функції діаграми, що повертає розв’язок задачі.

На діаграмі може бути неповна функціональна зв’язність. На рисунку 2.12 функції пов’язані повною функціональною зв’язністю.

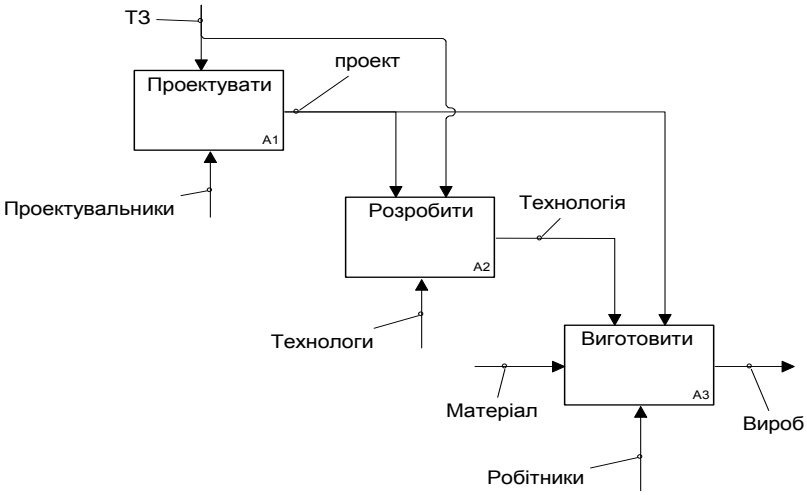


Рисунок 2.12 - Приклад функціональної зв’язності

Повну функціональну зв’язність можна описати текстом, який не можна розірвати на два логічно завершених тексти.

3 ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМ ЗАСОБАМИ МЕТОДОЛОГІЇ IDEF0

3.1 Властивості якісного програмного забезпечення та принципи розробки, що забезпечують якість

Важливим технологічним процесом, необхідним при розробці програмної системи, є контроль якості програмного продукту.

Для контролю IDEF0-моделі програмної системи застосовують семантичні критерії, розглянуті у розділі 2. Розглянемо, чому такий контроль дозволяє утворювати якісний продукт.

Якісний програмний продукт повинен мати наступні чотири властивості.

1 *Модифікуємість системи*. Властивість необхідна для виправлення помилок і для подальшого удосконалення системи.

2 *Ефективність системи.* Властивість означає оптимальне використання алгоритмів, часу роботи програми, пам’яті, пристроїв, засобів програмування.

3 *Надійність системи.* Властивість означає контроль і захист даних на всіх етапах функціонування системи; детальну обробку збійних ситуацій, сталість роботи системи. Надійність результатів обчислень залежить від вибору алгоритмів та їх реалізації.

4 Зрозумілість системи. Властивість означає прозорість структури системи та її функціонування.

Для забезпечення вказаних властивостей у роботі над програмою реалізують наступні принципи.

1 Абстракції
2 Модульності
3 Локалізації
4 Утаєння інформації
5 Узгодженості
6 Повноти
7 Підтверджування

3.1.1 Принцип абстракції

Абстракція - абстрагування від неістотних у даний момент властивостей.

Методологія IDEF застосовує принцип абстракції при створенні групи методологій, кожна з яких присвячена побудові моделей для однієї властивості системи, абстрагуючись від інших її властивостей.

Кожний тип моделі дозволяє аналізувати свій тип об’єктів та їх властивості. Причому, всі моделі взаємопов’язані та взаємозалежні. Поєднання групи моделей дозволяє розроблювати різні аспекти системи, досліджувати ефективність роботи ще на концептуальному рівні. Такий підхід робить роботу елементів системи зрозумілими, а тому надійно працюючими, модифікуємими.

Принцип абстракції лежить також в основі побудови ієрархічної структури діаграм методології IDEF0, що дозволяє розроблювати системи за спіральною моделлю життєвого циклу на концептуальному рівні.

Реалізація системи за витками спіральної моделі у вигляді ієрархічної структури з діаграм дозволяє абстрагуватися на кожному витку від дрібних деталей, виділяти істотні властивості системи для кожного рівня. Кожний наступний рівень моделі – діаграми, що деталізують свої вузли, являє собою новий прототип.

При реалізації діаграми А-0 розроблювач абстрагується від функцій програм програмного комплексу, показуючи тільки його призначення. На діаграмі декомпозиції А0 для комплексу програм показують функції, що відображують призначення кожної програми комплексу, абстрагуючись від їх внутрішніх функцій. Наступний рівень діаграм моделі – показує для кожного модуля програми його основну функцію. На нижчому рівні основна функція кожного модуля деталізується функціями його процедур. Якщо процедури модуля великі, то їх функції можна декомпозувати ще на нижчому рівні. Ясно, така декомпозиція процедур необхідна тільки для міркування над вмістом процедур.

Такий підхід дозволяє точно, прозоро і поступово передавати зміст програмної системи замовнику; робить систему зрозумілою при розробці, надійно працюючою, її можна легко модифікувати.

3.1.2 Принцип модульності

Принцип модульності передбачає розділення системи на програми або модулі з певною метою.

Мета1: спрощення та прискорення рішення проблеми

„Складну проблему легше та швидше вирішити, якщо розділити проблему на 2 простіші проблеми”. Сформульований Г. Майерсом принцип модульності, забезпечує можливість утворювати програми будь-якої складності.

Покажемо, що принцип модульності дійсно дозволяє легше рішити та швидше розв’язати проблему будь-якої складності.

Нехай X1 та X2 проблеми, які треба розв’язати.

C(X1) та C(X2)– функції складності рішення проблем X1 та X2.

T(X1) та T(X2)– функції витрат часу на рішення проблем X1 та X2.

Відомо, що складність першої проблеми менше або дорівнює складності другої проблеми:

$$C(X1) <= C(X2)$$

Ясно, що: $T(X1) <= T(X2)$.

Чим складніша проблема, тим більше часу треба для її рішення.

З досвіду ясно, що:

$$C(X1+X2) > C(X1) + C(X2)$$

$$T(X1+X2) > T(X1) + T(X2).$$

Таким чином, працює принцип: щоб вирішити складу проблему, треба розділити її на частини.

Мета 2: розділення програми за організаційним принципом

Комплекс програм ділять на програми в залежності від призначення ПЗ виробничим ділянкам. Наприклад, ділянки виробництва бухгалтерія, відділ кадрів, складування застосовують у роботі свою програму. Програму можна розподілити на модулі за функціями, де кожна функція має свого користувача. Наприклад, адміністратор, користувач.

Мета3: зменшити розмір програми за обмеженням компілятора

Часто компілятори систем програмування мають обмеження на розмір вхідної програми. Розділення програми на модулі дозволяє компілювати великі програми по частинам.

Мета 4: забезпечити колективну розробку програми.

При колективній розробці програмного забезпечення кожному розроблювачу видають свою частину роботи – модуль. Як ми бачили з попереднього матеріалу, хороший модуль слабо пов’язаний з іншими модулями. Така властивість сприяє колективній розробці.

Мета 5: прискорити розробку прототипу.

При розробці програмної системи можна виділити у модуль складну частину програми, щоб її можна було легко модифікувати надалі. Можна також виділити у модуль не зовсім ясну частину розробки для подальшого окремого налагодження.

У всіх мовах програмування модуль синтаксично відособлена одиниця. Модуль складається з розділу інтерфейсу та розділу реалізації. Інтерфейсом можна вважати формальні параметри програми, що дозволяє розглядати програму, як окремий модуль.

Оптимальний розмір модуля 7 ± 2 процедури. Оптимальний розмір модуля диктується таким міркуванням. Як ми бачили, чим менше модуль, тим простіше він реалізується. Для стиковки двох модулів необхідно мати між модульний інтерфейс. Зі зменшенням розміру модулів, зростає їх кількість, а тому зростають витрати на

інтерфейси. Отож, повинна існувати оптимальна кількість модулів достатньо малих за об'ємом, розробка яких дає мінімальну вартість.

Подавання системи функціональною моделлю, дозволяє розроблювати її модульну структуру, де кожний модуль – функція діаграми. Діаграми декомпозиції таких функцій показують структуру модулів з процедур і функцій. Подавання вмісту модулів у вигляді діаграми, дозволяє зробити структуру системи зрозумілою і дає можливість модифікувати її.

Наочність графічного подавання процедур і функцій дозволяє аналізувати вміст модулів та одержувати якісні модулі. Для аналізу використовують семантичні критерії якості функціональних моделей. Такий аналіз забезпечує надійність, ефективність роботи.

Модульність дозволяє прозоро передавати зміст програмної системи; робить систему зрозумілою, надійно працюючою. Таку систему легко модифікувати.

3.1.3 Принципи локалізації, утаєння інформації, узгодженості, повноти та підтверджування

Принцип локалізації

Принцип локалізації стверджує про необхідність розміщувати всі елементи модуля, що пов'язані змістом, разом. Це *зменшує інтерфейс модуля, впливає на ефективність системи за часом, економить пам'ять.*

Реалізація принципу локалізації при розробці модуля робить *модуль міцним за змістом*. У методології принцип локалізації закладено у семантичний критерій «складність/зрозумілість».

Принцип приховування інформації

Принцип приховування інформації (чорна скриня) робить неприступними деталі модуля для іншого модуля. Деталі модуля розташовують на діаграмі декомпозиції функції модуля.

Приховування інформації модуля на діаграмах декомпозиції забезпечують внутрішні дуги та дуги з тунелями. На діаграмі

декомпозиції подаються локальні функції модуля, а внутрішні дуги діаграми це локальні дані цих функцій. Дуги з тунелями на вільних кінцях вказують на локальні дані модуля, що вводяться з клавіатури або файлу.

Наявність своїх даних у модуля забезпечує *рутинність модуля* (незалежність значень змінних від попередніх звертань до модуля).

Доступ до локальних даних модуля та до їх функцій обмежено інтерфейсом. Прикордонні дуги діаграми декомпозиції це формальні параметри процедур інтерфейсу, а відповідні їм дуги вузла фактичні параметри процедур інтерфейсу.

Хороший модуль зовні повинен бути простішим, ніж усередині. Модулі спілкуються через прикордонні дуги. Вони обмінюються інформацією необхідною для їх зв'язку. Такий підхід робить *модулі слабко залежними, а програму прозорою і надійною в роботі*.

Принцип узгодженості

Узгодженість – це одноманітність символіки та термінології у модулі та по всій системі. *Принцип забезпечує розуміння і як наслідок вірність системи.*

На функціональних діаграмах імена утворюються за правилами мови моделювання. Опис імен у глосарії дозволяє визначити зміст імен.

Принцип повноти

Принцип повноти ПЗ перевіряється досягненням мети розробки ПЗ, що наочно показує функціональна модель. Принцип повноти забезпечується семантичним критерієм «Повнота» контролю якості моделі. *Цей принцип забезпечує ефективність системи (одержання результату)*.

Принцип підтвердження

Принцип підтвердження – вказує на необхідність перевірки результатів роботи системи на практиці, що забезпечують *надійність роботи системи.*

3.2 Зчеплення та зв'язність модулів програми

Особливо важливу роль для створення якісного програмного продукту грають характеристики «Зчеплення» та «Зв'язність» критерію «Зрозумілість».

Чим більше зчеплені за змістом різні модулі, тим гірше розуміється система і *важче корегування модулів*. Як наслідок, тим більше помилок буде зроблено при розробці системи, її модифікації. Треба намагатися утворювати *слабко зчеплені* модулі. Такі модулі можна перероблювати майже не торкаючи інших модулів.

Типи зчеплень модулів при розробці концептуальної моделі програми мають *своєрідне трактування*.

3.2.1 Зчеплення модулів за природою зв'язку

Патологічне зчеплення. Модуль1 частково «знає» структуру Модуля2. Це означає що в Модулі1 частково запрограмований вміст Модуля2. При змінах у одному з модулів розроблювач може забути змінити вміст другого модуля.

Керування. Процедура першого модуля керує функціонуванням процедури другого модуля, надсилаючи керуючі дані. Керування може відбуватися через прапорці, перемикачі. Для цього перший модуль повинен добре «знати» структуру процедури другого модуля. Зміна вмісту процедури другого модуля може привести до помилки. Щоб уникнути такої ситуації треба спілкуватися з процедурою тільки через виклик.

Нормальне зчеплення. Відсутність патологічного зчеплення модулів та керування.

3.2.2 Зчеплення модулів за структурою зв'язку

Зчеплення за середовищем. Назвемо середовищем спільну область даних для модулів. Такою областю може бути область типу Common, велика структура з інтерфейсу або файл. Працюючи з однією спільною областю, один модуль може псувати дані іншого модуля, якщо обидва модулі пишуть у середовище.

Зчеплення за записом. Обидва модулі використовують спільну структуру даних - запис. Помилок буде менше, бо сполучання не через всю структуру даних, а через її частину.

Абстрактне зчеплення. Обидва модулі майже незалежні один від одного. Повну незалежність модулів неможна зробити. Передача даних між модулями встановлюється через інтерфейс стандартної СУБД або користувача; глобальну процедуру. Аргументами процедур інтерфейсу мають бути прості дані або виклик процедури без параметрів.

Реалізація модулів програми за нормальним та абстрактними зчепленнями робить модулі слабко зв'язаними.

3.2.3 Зв'язність процедур модуля за змістом

Зв'язність процедур модуля за змістом називають *міцністю модуля.* Чим більше зв'язність елементів модуля за змістом, тим краще якість ПЗ. Міцність модуля впливає на його розуміння, надійність, на можливість вірної модифікації системи.

Міцність модуля і інтерфейс

Міцність модуля впливає також на обсяг його інтерфейсу. Якщо функції модуля не пов'язані або слабо зв'язані змістом, інтерфейс модуля великий, бо майже всі функції увійдуть до інтерфейсу. Дивися рисунок 3.1.

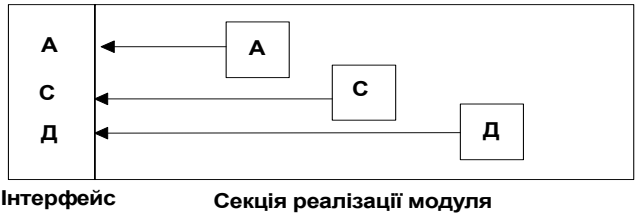


Рисунок 3.1 – Структура модуля для незмістовного зв'язку

Обсяг інтерфейсу впливає на витрати пам'яті під програму. Але, для створення програми мовою Visual Prolog, обсяг інтерфейсу важливий ще за іншою причиною.

Компілятор Visual Prolog має обмеження на кількість предикатів у модулі. Головний модуль програми повинен мати крім своїх предикатів (локальних і глобальних) глобальні предикати всіх модулів програми. Тому при наявності великих інтерфейсів у модулів, максимальна кількість предикатів може бути перевищена.

При змістовному зв'язку функцій (процедур) модуля інтерфейс модуля невеликий, бо більшість функцій локальні. Дивися рисунок 3.2.

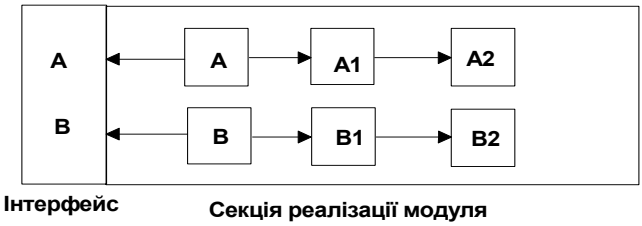


Рисунок 3.2 – Структура модуля для змістовного зв'язку

Незмістовні зв'язки

До незмістовних зв'язків функцій (процедур) модуля відносять: випадкову, логічну, часову, процедурну зв'язності. Якість незмістовних зв'язків поліпшуються за їх переліком.

Випадкова зв'язність

У модулі між процедурами немає явних змістовних зв'язків, тому інтерфейс модуля містить майже всі процедури модуля. Порушено принцип «чорного ящика». При роботі такого модуля буде багато помилок, модифікування його важке.

Наприклад, програміст виявив ряд процедур різного призначення загального характеру і вирішив зробити з них окремий модуль. Процедури такого модуля не пов'язані змістом.

Логічна зв'язність

Модуль складається з процедур, які пов'язані за класом функцій; за класом вхідних або вихідних даних, а не за їх змістом. Наприклад, до модуля відносять процедури, що дозволяє обрати за

характеристиками диск та флешку. Вихідні дуги відносяться до одного класу «пристрої читання-запису».

Помилково до цього типу зв’язності відносять стандартні модулі мов програмування, що орієнтовані на обробку певного типу даного. Наприклад, модуль обробки рядків. Це невірно, бо стандартний модуль не цікавиться вмістом рядка, для нього рядок дане, а не тип даного. Стандартний модуль абстрагується від вмісту рядка.

Процедури стандартних модулів мають комунікативний тип зв’язності.

Часова зв’язність

Процедури модуля виконуються в один конкретний період роботи системи. Між процедурами ніякого зв’язку за змістом немає. В той же час, процедури тісно пов’язані з іншими модулями системи.

Процедури з часовою зв’язністю у модулю мають патологічне зчеплення із своїми модулями.

Наприклад. Нехай створено модуль, що має процедури ініціалізації інших модулів: інтерфейсу; адміністратора; розуміння тексту природною мовою. Ясно, що всі початкові установки для модулів свої. Процедури модуля ініціалізації пов’язані тільки часом виконання системи – входом у систему.

Такий модуль може мати дуже великий інтерфейс. Це порушує принцип захованості інформації.

Процедурна зв’язність

Функції модуля різні за змістом групуються в модулі тому, що формально на якомусь етапі беруть участь в одному процесі або циклі. Програміст, що бажає використати спільний код для кількох різних за змістом процедур, робить структуру модуля заплутаною.

Змістовні зв’язки

До змістовних зв’язків відносять:комунікативні, послідовні, функціональні зв’язки. Якість змістовних зв’язків поліпшуються за їх переліком.

Комунікативна зв’язність

Процедури модуля з комунікативною зв’язністю працюють з одним даним. Прикладом комунікативної зв’язності може бути модуль обробки рядків. Програміст розв’язує проблему, яку використати процедуру, щоб обробити свій рядок. Модулі з комунікативною зв’язністю використовуються в усіх мовах програмування.

Послідовна зв’язність

Процедури модуля утворюють конвеєр, на якому послідовно оброблюються дані. Результат роботи першої процедури стає вхідним даним другої процедури. Далі дії повторюються. Між функціями зв'язок причина – наслідок. Наприклад: прочитати компоненти файлу в список; знайти мінімальний елементу списку; вивести мінімальний елемент на екран.

Повна функціональна зв’язність

Всі процедури модуля розв’язують одну задачу. Всі процедури модуля готують дані для виконання основної функції модуля, що повертає розв’язок задачі. Модулі з функціональною залежністю найкращі.

Рутинність модуля

Розглянемо ще одну важливу властивість модуля. Рутинністю модуля називають незалежність процедур модуля від попередніх звертань до них. Рутинність – достоїнство модуля. Робота процедур модуля повинна залежати тільки від їх параметрів, а не від попередніх викликів. Рутинність модуля забезпечують локальні дані модуля.

Будемо вважати функцію, що написана мовою Common Lisp малим модулем. Розглянемо приклад функції з відсутністю рутинності. Функція рахує кількість елементів списку. При першому виклику функції відповідь вірна. Починаючи з другого виклику функції відповідь невірна.

```
CL-USER> (setq S 0)
          0
CL-USER> (Defun K (L)
(cond ((Null L) S)
```

```
  (T (setq S (+ 1 (K (cdr L) )  )   )    )))
            K
CL-USER> (K ‘(23 45 67 9) )
            4
CL-USER> (K ‘(2 4 6) )
            7
```

Функція не рутинна. Причина: змінна S глобальна, її значення зберігається від виклику до виклику. Щоб функція стала рутинною треба не привласнювати змінній S значення під час роботи функції.

Іноді рутинність заважає. Ситуація може виникнути, якщо кожний раз треба оброблювати не початкові дані, а результат попередньої роботи функції. Наприклад, при кожному виклику функції додавати до попередньої суми наступну.

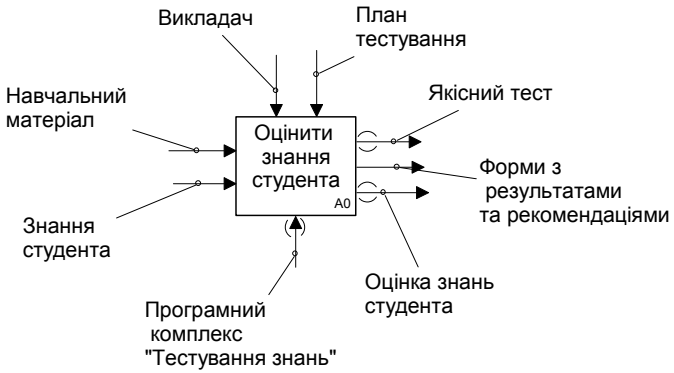
Реалізація модулів з комунікативною, послідовною або функціональною зв’язністю робить програму зрозумілою, ефективною і надійною. Використовуйте рутинність модуля там, де необхідно. Це зробить програму ефективною.

3.3 Створення функціональної моделі програмної системи

Нехай треба розробити програмний комплекс «Тестування знань студента». Вірна оцінка знань студента залежить від якості тесту. Тести можуть бути різної форми залежно від їх призначення. Програмний комплекс складатиметься з трьох програм, які повинні: допомагати викладачу розроблювати тести для контролю знань; проводити заплановане або індивідуальне тестування знань студентів; оброблювати результати тестування і давати рекомендації для повторного вивчення матеріалу.

Програми можуть викликатися як окремо кожна, так і послідовно. Режим роботи залежить від обраної користувачем опції меню головної програми.

Розробка програмного комплексу починається з діаграми А-0. На діаграмі зображується функція загального призначення програмного комплексу. Дуги функції зображують загальний інтерфейс комплексу із його середовищем. Дивись рисунок 3.3.



Мета створення програмного комплексу: поліпшити якість тесту та оцінки знань студента

Мета створення моделі: Розробка функцій програмного комплексу

Точка зору: Викладача

Рисунок 3.3 - Інтерфейсна діаграма А-0 програмного комплексу «Тестувати знання»

Функція «Оцінити знання студента» є призначенням системи. Функція працює в 3-х режимах: поліпшення якості тесту, тестування студентів, одержання результатів тестування студента з рекомендаціями. Всі режими роботи керуються планом тестування і викладачем, а виконуються комплексом програм «Тестування знань». Для виявлення активацій кожного режиму комплексу необхідно деомпозувати основну функцію.

На діаграмі декомпозиції А0 зображуються функції програм комплексу і зв’язки між ними. Функції діаграми А0 розкривають зміст функції «Оцінити знання студента». Важливо, що кожна функція є окремою програмою. Тому дуги кожної функції вказують на своє середовище: людей, інші програми, СКБД, матеріали дисципліни, звітні форми.

Дивись рисунок 3.4.

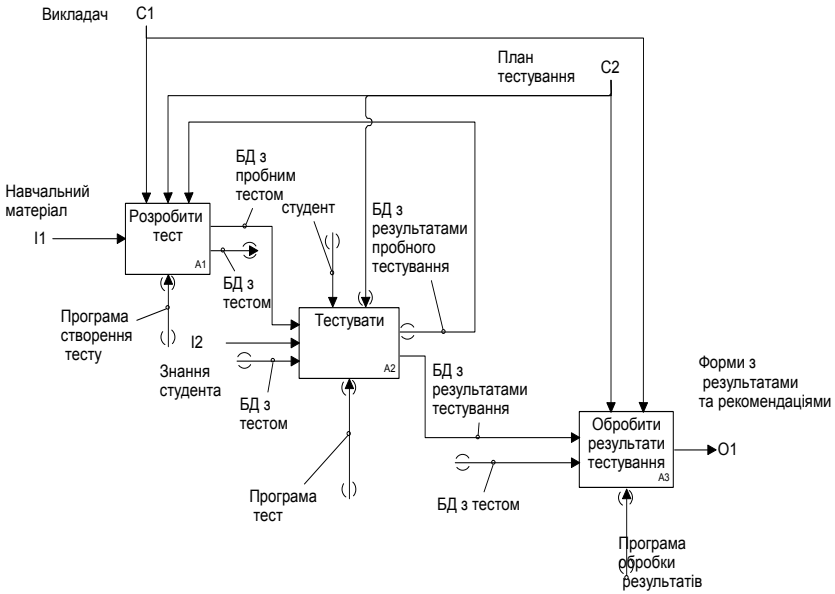


Рисунок 3.4 – Діаграма декомпозиції А0 показує зв’язок між програмами комплексу

На діаграмі А0 є кілька шляхів з послідовним типом зв’язності між програмами комплексу. Шляхи вказують на режими роботи комплексу.

Режим поліпшення якості тесту використовує функції А1 і А2 в циклі, поки якість тесту не стане задовільною. Але виклики програм відповідних функціям розірвані у часі. Завдання тесту та відповідні їм результати тестування аналізуються програмно, осмислюються та корегуються викладачем. Після одержання відповідності тестів вимогам, проводиться тестування групи студентів. Результати тестування будуть оброблені та виведені на екран функцією А3 разом з рекомендаціями студенту.

У *режимі тестування* студентів працюють функції А2 і А3. Після тестування можна автоматично одержати результати у обраній формі з рекомендаціями кожному студенту. Програми комплексу можуть виконуватися і окремо. Можливо запустити тільки одну програму А3 для повторного одержання результатів тестування та

рекомендацій студенту, які теми треба повторно вивчити, які завдання зробити.

Розглянемо докладніше вміст кожної програми комплексу.

Програма створення тесту виконує наступні функції: допомагає структурувати матеріал підказками (розділ, тема, питання); розподіляє питання за поняттями, алгоритмами, стандартними засобами мови, механізмами; пропонує скласти завдання у різних формах і допомагає у діалозі обрати форму відносно теми та призначення тесту. Далі програма допомагає: скласти план тесту, розподіляючи завдання за розділами дисципліни, вміннями. Після чого викликає процедури програми «Тест» для перевірки тесту на студентах; оцінює якість завдань тесту за результатами тестування (важкість, надійність, досягнення цілі навчання).

Програма «Тест» складається з трьох модулів. Вона ідентифікує студента, який обирає тему, рівень складності тесту. Програма тестує студента і повертає йому загальну кількість завдань, кількість вірних відповідей, процент вірних відповідей від загальної кількості завдань. Процент впливає на оцінку, але вона виставляється після детального аналізу тем завдань.

Програма обробки результату аналізує теми завдань, які поняття, механізми не знає студент, або не вміє їх застосувати; *виявляє* систематичність та повноту знань; визначає важкість завдань для студента і складає для нього індивідуальні завдання для поступового оволодінням дисципліни.

Аналіз діаграми декомпозиції

Аналізуючи діаграму декомпозиції А0 можна сказати, що на діаграмі є основна функція «Тестувати», а інші функції допоміжні. Зв'язок між функціями змістовий послідовний.

На діаграмі А0 між функціями програм комплексу «Розробити тест» і «Тестувати» не буде існувати зчеплення через середовище, бо дві різні програми не знають структуру БД, а спілкуються з нею через СКБД.

Для спілкування програм через стандартну БД або БД користувача є мова запиту. Доступ до даних залежить від семантики ключів доступу та семантики даних, а не від структури зберігання даних. Форма зберігання даних не суттєва для зовнішньої до БД програми.

Динамічна БД Visual Prolog заснована на фактах. Ключами такої БД може бути будь-яке дане або їх комбінація. Але для роботи із фактами необхідно знати порядок аргументів з даними у факті.

Вивід: Повністю незалежні програми комплексу, не можливо зробити.

Програми комплексу в свою чергу можуть складатися з модулів. На рисунку 3.5 зображено функції модулів програми «Тест».

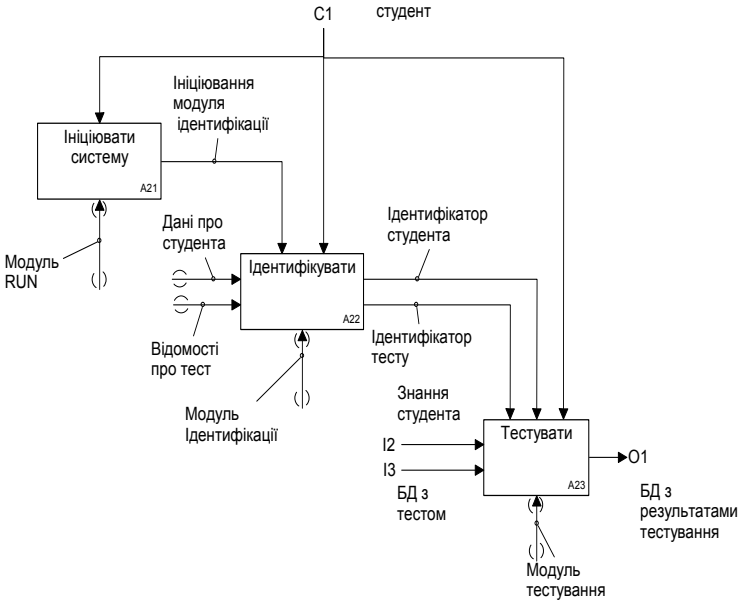


Рисунок 3.5 – Діаграма декомпозиції А2 показує модульну структуру програми «Тест»

Модуль RUN створюється, щоб не перевищити максимальну кількість предикатів компілятора Visual Prolog. Головний модуль програми повинен мати крім своїх предикатів (локальних і

глобальних) глобальні предикати всіх модулів програми. Модуль RUN не має своїх предикатів, тому обмеження не буде перевищене.

Між функціями модулів RUN, «Модуль ідентифікації», «Модуль тестування» показаний послідовний тип зв'язності.

Дуги, що поєднують модулі зображують інтерфейс модулів. У модулі ідентифікації обирається ідентифікатор тесту і передається модулю тестування для завантаження тесту. Ідентифікатор тесту простий тип даного, що забезпечує нормальне зчеплення між модулями. Аналогічне можна сказати про ідентифікатор студента. Якщо передавати сам тест, а не БД буде породжуватися зчеплення за середовищем. Тоді можливо корегування фактів тесту обома модулями, що є джерелом помилок.

На рисунку 3.6 подається вміст функцій модуля тестувати. Між функціями діаграми послідовний тип зв'язності і є цикл. На діаграмі показано два режиму роботи: перше тестування і друге тестування.

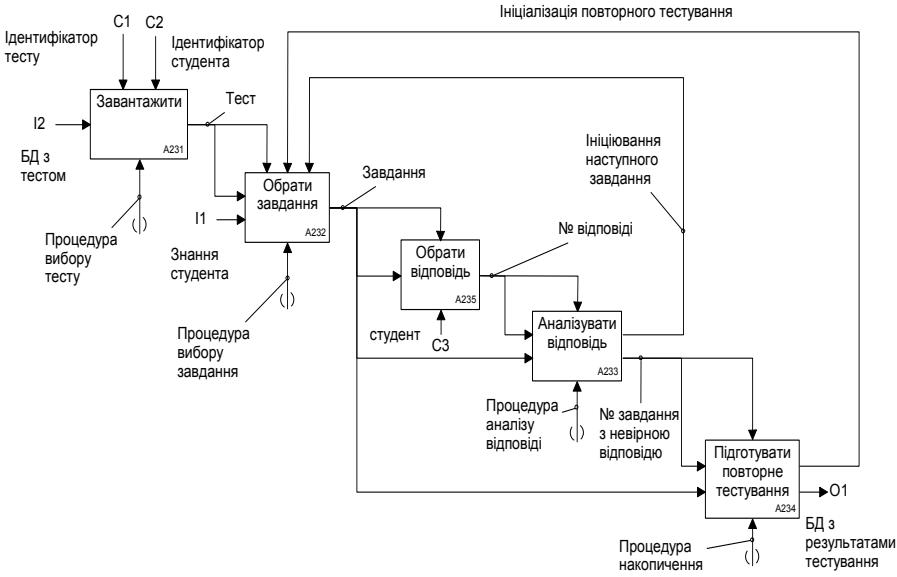


Рисунок 3.6 – Діаграма декомпозиції A23 показує вміст модуля «Тестувати»

Для розробки процедур модуля можна зробити ще один рівень діаграм. У моделі розроблено не всі необхідні діаграми, бо це тільки приклад. При створенні моделі треба, щоб було ясно що робить система і досягалася мета моделі.

4 МОДУЛЬНЕ ПРОГРАМУВАННЯ СИСТЕМ МОВОЮ VISUAL PROLOG

4.1 Структура модульної програми

Програмне застосування, що розв'язує реальну задачу, не складається з одного модуля навіть тоді, коли програміст створив один модуль. Програми завжди застосовують стандартні інструменти мови, які за змістом розподіляються на модулі. Особливо це стосується графіки.

Програма з кількох модулів має один головний модуль і кілька допоміжних. Модулі програми взаємодіють через виклики глобальних процедур або файли. Нагадаємо, що під процедурою в Пролозі розуміють множину тверджень, які записані тим самим предикатом. Взаємодія модулів через виклики групи глобальних фактів (доступних для запису кільком модулям) небезпечна, як ми бачили у попередньому розділі 3.

Всі модулі програми оформлюються подібно програмі з одного модуля, містять секції опису доменів, предикатів, секції з твердженнями.

Відміна в тому, що при оголошенні предикатів та доменів, секції створюють глобальними або локальними. Процедури, твердження яких записані глобальними предикатами, застосовують для взаємодії модулів, а локальні процедури доступні тільки в їх модулі. Треба зауважити, що самі процедури (локальні і глобальні) подаються в секції Clauses. Локальність описів предикатів дозволяє в різних модулях застосовувати однакові ідентифікатори.

Важливо те, що програма завжди має тільки одну ціль, її розташовують у головному модулі.

Розглянемо приклад модульної програми.

Завдання: Даний набір фактів у файлі, який містить відомості про студентів групи: прізвище, групу. Написати програму, яка вилучає студента з групи за прізвищем та № групи, або вставляє факт про нового студента в певну групу.

Хоча за змістом всі процедури пов’язані між собою і повинні знаходитися у одному модулі, ми їх розмістимо у трьох модулях, щоб зрозуміти структуру модульної програми.

Додамо до програми головний модуль RUN для ініціювання програми. Цим покажемо, як можна зменшити кількість предикатів у головному модулю великої програми. Зверніть увагу, що головний модуль RUN містить всі глобальні предикати програми. У програмах середнього розміру можна всі глобальні предикати включати в кожний модуль програми – це зручно. Для великих програм кожен модуль містить ті глобальні предикати, які він застосовує та об’являє для інших модулів.

По замовчанню директива INCLUDE містить тип даних, який застосовується браузером виводу дерева проекту та типи даних для стандартних потоків вводу-виводу.

У головному модулі є три глобальні предикати для звертання до інших модулів і його локальний предикат RUN.

```
Include "pruklad.inc"

Global Predicates
Menu ()
Delete ()
Add ()

Predicates

Run()

Clauses
Run ():- Menu (),!.

Goal
Run ().
```

```
Include "pruklad.inc"

Global predicates
Menu ()
Delete ()
Add ()

Predicates
Go (integer)

Clauses

Menu ():- Write ("1. Вилучити студента з групи "), nl,
          Write ("2. Включити студента в групу "), nl,
          Write ("3. Закінчити роботу"), nl, readint (N), go(N).
Go (1):- Delete ().
Go (2):- Add ().
Go (3):- Exit ().
Go (4):- Menu ()./*за помилкою вводу № дії*/
```

```
Include "pruklad.inc"

Global Predicates
Menu ()
Delete ()
Add ()

Domains /* Об’ява локальних типів даних користувача*/
priz, n_gr = string

Facts-delete /* Іменована локальна область Facts */
Stud (priz, n_gr)

Clauses
Delete ():- Consult ("Stud.txt", delete), Write ("Прізвище? "), nl,
Readln (F), Write ("№ групи? "), nl, Readln (N), Retract (Stud(F, N),
delete), Save ("Stud.txt", delete).
```

```
Include "pruklad.inc"

Global Predicates
```

```
Menu ()
Delete ()
Add ()

Domains
priz, n_gr = string

Facts-add
Stud (priz, n_gr)

Clauses

Add ():-Consult ("Stud.txt", add), Write ("Прізвище?"), nl, Readln (Pr), Write("№ групи "), nl, Readln (N), assert(Stud(Pr, N),add), Save("Stud.txt", add).

Global Domains
Db_selector = browselist_db
File= fileselector1; fileselector2
```

У програмі показано, як можна створювати меню і встановлювати зв’язок між пунктами меню і їх реалізацією. За рекомендацією у критеріях якості програмного забезпечення області Facts обрано локальними.

4.2 Основні погодження написання модульної програми

Написання програми мовою Visual Prolog вимагає виконання наступних погоджень.

1.У модулі глобальними об’являють ті предикати, які будуть застосовуватися іншими модулями. Предикати, що застосовуються з іншого модуля, також об’являють глобальними.

2. Треба пам’ятати, що в програмі може бути лише одна неіменована секція FACTS в головному модулі. В інших модулях секції FACTS треба іменувати.

3. Глобальний факт з характеристикою SINGLE повинен розміщуватися тільки в головному модулі. Причому секція GOAL повинна знаходитися перед секцією CLAUSES, у якій є цей факт.

4. Глобальні предикати можуть мати глобальні або локальні типи даних.

5. У разі необхідності можна об’явити глобальні факти, але окремим модулем.

6. Порядок розміщення предикатів у глобальних секціях різних модулів може бути різним, що дозволяє створювати бібліотеки з процедурами в об’єктних кодах, які може застосовувати вся програма.

7. Порядок розміщення локальних і глобальних секцій у програмі не важливий.

4.3 Об’ява глобальних предикатів

Глобальний предикат повинен мати опис всіх можливих конкретизацій їх аргументів. Кожна конкретизація відповідна певному режиму його роботи. Конкретизації позначаються як для стандартних предикатів: i – input; o – output.

Якщо користувач написав би свій предикат STR_LEN, то він би його описав.

Global Predicates

Nondeterm str_len (string, integer)-(i, o) (o, i) (i, i)

По замовчанню, якщо конкретизація не вказана, всі аргументи вважаються вхідними.

Кожен аргумент предиката передає в процедуру дані або повертає результат. При різних викликах процедури в неї поступають різні дані і відповідно їм повертаються результати. Тобто в процедуру поступає вхідний потік даних, а вона повертає вихідний потік даних. Вивчаючи файли, ми говорили про вхідні і вихідні потоки даних між програмою і файлом. Таким чином можна сказати, що *Visual Prolog базується на потоковій організації.*

Глобальною може бути предикат-функція. Кожна функція має певний тип, який задається окремо. Дивися приклад програми нижче пункт (4.1). При об’яві типу функції перед функтором вказують тип результату. Після чого функції приписують цей тип (4.2).

При виклику функція містить тільки вхідні дані, а результат привласнюється змінній (4.3). У тілі вона має аргументи для вхідних даних і для результату (4.4).

Розглянемо приклад програми з функцією.

```
Global Domains
pr = nondeterm integer (integer)-(i)                    (4.1)
Global Predicates
Kvadrat: pr                                             (4.2)
Predicates
nondeterm Do (integer, integer)
Clauses
Do (N, R):- R = Kvadrat (N).                            (4.3)
Kvadrat (N, Rez):- Rez = N*N.                           (4.4)

Goal
Do (5, R).
```

Іноді необхідно написати програму на іншій мові програмування. Для виклику такої програми з програми мовою Visual Prolog її необхідно об’явити як зовнішній глобальний предикат.

Об’ява предикату повинна мати назву мови. Наприклад, для використання програми на мові С, треба об’явити:

```
Global Predicates
pro_c (integer, integer) - (i, o) language c
```

Від конкретизації аргументів залежить мова подання вхідних даних.

4.4 Створення модульної програми експертом застосувань

Створити модульну програму не можна без експерта застосувань. Для збирання модульної програми треба мати опис структури програми з модулів. Цей опис виконує експерт застосування, який має *засоби опису структури програми з модулів.*

Структуру програми можна розробити після визначення характеристик проекту за опцією Project/New Project головного меню.

4.4.1 Чому проект?

Застосування опції Project при розробці програми навіть з одного модуля дає ряд переваг.

При створенні нового проекту (опція Project/New Project) на екран виводиться діалогове вікно програми «Експерт застосувань» (Application Expert). Експерт застосувань дозволяє програмісту *автоматично реалізовувати ряд дій, реалізація яких вимагає глибинних знань* про використання ресурсів системи програмування Visual Prolog.

Програма може реалізовуватися для роботи в різних ОС (DOS, DOS розширений, WINDOWS, OS/2). Експерт застосувань автоматично враховує особливості створення програм, що виконуються в цих операційних системах.

Система програмування Visual Prolog дозволяє створювати інтерфейс між програмою на Visual Prolog і програмами написаними на інших мовах, при умові, що компілятори цих мов виробляють об’єктний код і задовольняють певним погодженням. Це дозволяє реалізовувати деякі процедури на інших мовах: assembler, C++, Java, Delphi, Visual Basic.

Процеси компіляції і побудови програм, бібліотек у різних мовах значно відрізняються. Експерт застосування зробить все автоматично застосовуючи погодження.

Експерт застосувань може сам відібрати і включати в програму інструментальні засоби роботи з візуальним інтерфейсом, інтерфейсом роботи в текстовому режимі, роботи з не візуальним інтерфейсом під WINDOWS.

При роботі над програмою і при її функціонуванні необхідно мати ряд файлів: зовнішньої бази даних, що описує проект; модулів програми на вихідній мові; ресурси візуального інтерфейсу(вікна, меню, панелі інструментів, піктограми, тощо). Для роботи з файлами потрібно пам’ятати, де розташовані файли та їх імена; шукати їх.

Експерт застосувань інтегрує файли автоматично, створює структуру каталогів - головний каталог і його підкаталоги, при створенні проекту.

Головний каталог проекту може існувати або створюватися під час створення проекту за вибором користувача. Каталог і підкаталоги спеціалізовані, тобто кожен з них містить по замовчанню свій тип файлів.

Всі файли, що містять інтегровану інформацію для всього проекту, мають ім’я проекту і розрізняються тільки розширенням.

4.4.2 Каталоги проекту та їх призначення

Після створення проекту експерт застосувань автоматично створює головний модуль проекту, структуру каталогів проекту; базу даних проекту, що містить всі параметри проекту, обрані при його створенні.

В каталозі з ім’ям проекту автоматично створюються два каталоги: OBJ, EXE і файли:

– <ім’я головного модуля>.pro;

– <ім’я головного модуля >.inc;

– <ім’я головного модуля >.pre.

Файл з розширенням «PRO» містить шаблон типової програми на Пролозі. Шаблон має назви типових секцій і шаблони основних типів тверджень.

Файл з розширенням INC містить по замовчанню глобальні об’яви (домени та предикати), необхідні для перегляду дерева структури програми та глобальні символічні імена файлів для доступу до файлів. У файл також можна вставити фрагменти програми, які розроблено попередньо.

Файл з розширенням PRE буде мати глобальні предикати модулів програми.

Каталог OBJ призначено файлам, які є результатами компіляції. Каталог EXE призначено для програми, що виконується, і файлів необхідних для роботи програми. Каталог EXE містить також файл помилок виконання.

На початку роботи експерта застосувань у його вікні відображується структура програми. На цей час вона складається

тільки з шаблону головного модуля. Програміст може розкрити шаблон у вікні текстового редактору і створити головний модуль.

По замовчанню ім’я головного модуля співпадає з ім’ям проекту.

4.4.3 Реалізація модулів програми

Створити новий модуль і додати його до структури програми можна за опцією NEW.

Опція відкриває каталог проекту. Пропонується ввести ім’я модуля для відкриття файлу у каталозі. В файлі буде зберігатися модуль програми на вихідній мові.

У вікні експерту застосування до структури проекту додається модуль з вказаним ім’ям. На цей час файл і модуль порожні. Написання тексту програми (головного і допоміжних модулів) можна відкласти до завершення створення структури програми або створювати текст програми одночасно із створенням структури. Дивися вікно експерту застосувань на рисунку 4.1.

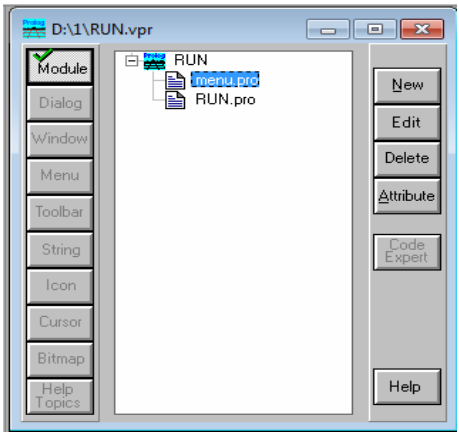


Рисунок 4.1 – Створення модуля Menu експертом застосування

4.4.4 Створення файлу з глобальними предикатами та доменами

По замовчанню кожен модуль містить директиву Include “<ім’я головного модуля>.inc”. Файл з розширенням INC називають заготовочним файлом. За замовчанням у файл

Розглянемо, як експертом застосувань можна створити файл, що містить всі глобальні предикати та глобальні домени. При цьому глобальні описи всіх модулів зливаються в один файл, який вставляється у кожний модуль програми.

При створенні проекту файл порожній. Для заповнення файлу необхідно виконати наступні дії:

Додати всі глобальні секції всіх модулів у файл “<ім’я головного модуля>.inc”.

Такі дії легко виконати вручну, якщо програма мала. Після додавання модуля до структури програми автоматично відкривається вікно експерта для автоматичного об’єднання глобальних об’яв - File Inclusions for Module. При ручному виконанні дій від вікна треба відмовитися.

Для великих програм треба вживати експерт застосувань.

Якщо користувач ще не знає які предикати і домени глобальні, то можна відмовитися від створення файлів. До діалогового вікна можна повернутися за опцією ATTRIBUTES у вікні експерта застосування.

Розглянемо роботу у вікні експерта застосувань File Inclusions for Module.

Виконаємо наступні дії.

1. Відмітимо «галочкою» у розділі «*Create Header Files»* опції «*Create “<ім’я модуля>.pre» і* «*Create “<ім’я модуля>.dom».*

Створюються відповідні порожні файли, їх треба заповнити глобальними об’явами.

2. Відмітимо «галочкою» у розділі «*Include Header Files in»* опції «*Include “<ім’я модуля>.pre» і* «*Include “<ім’я модуля>.dom».*

Створені файли додаються за директивою Include до файлу *“<ім’я головного модуля>.inc”*.

Дивися вікно File Inclusions for Module на рисунку 4.2.

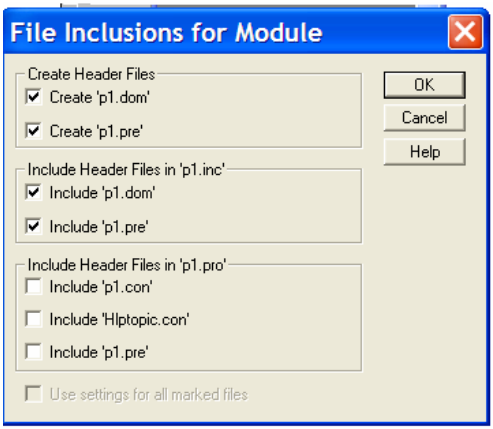


Рисунок 4.2 – Вікно File Inclusions for Module

Якщо обраний файл вже існує, то він буде заміщений. Можна відмовитися від заміни або погодитися із нею. Підтвердження про заміну виконується клацнувши двічі лівою кнопкою миші на відповіді.

У поточний модуль за директивою Include можливо включити файли: з об’явами констант для графічних ресурсів або довідки. Під константою ресурсу розуміють ім’я ресурсу, за яким до нього звертаються у програмі. Символьна константа являють собою ідентифікатори розділів довідки. Довідку генерує генератор довідки.

Опції EDIT та DELETE застосовуються для корегування модулю або вилучання відповідно.

4.5 Компіляція і побудова модульної програми

Модульна структура програми дозволяє заощаджувати час при компіляції програми. Можна змінювати тільки необхідний модуль, окремо компілювати, налагоджувати.

Команда PROJECT/COMPILE MODULE (Ctrl+F9). Команда компілює перший знайдений файл у каталогу проекту з розширенням .pro, якщо відповідний модуль є у структурі проекту.

Якщо перший знайдений файл з розширенням .pro не відноситься до структури проекту, то шукається модуль проекту, що включає за директивою Include цей файл. Файл включається в модуль і модуль компілюється. Файли, що мають розширення: .pre, .dom, .con, .inc включаються в програму директивою Include (перед процесорна обробка).

Якщо у поточному вікні проекту знаходиться модуль або файл з вказаним розширенням, то компілюється модуль або виконуються дії вказані вище для файлів.

Під час компіляції в каталозі OBJ для головного модуля створюються файли з розширенням: obj, sym, map, bro, deb.

Для кожного допоміжного модуля програми створюються тільки файли з розширеннями OBJ і DEB. Крім того в каталог OBJ розміщується файл з головним модулем програми і файл помилок. Вони необхідні для компіляції модулів.

Файл з розширенням OBJ містить об’єктний код головного модуля. Файл з розширенням DEB містить інформацію для налагоджування модуля програмою DEBUGER.

Файл з розширенням MAP - карта пам’яті програми після компіляції порожня. Надалі карта заповниться компонувальником програми.

Файл з розширенням SYM після компіляції всіх модулів буде містити таблицю всіх символів програми. Кожен символ модуля розміщується в таблиці один раз і має в таблиці свою адресу. Такий підхід прискорює роботу і обсяг програми.

Файл з розширенням BRO містить інформацію для браузера коду.

Команда PROJECT/BUILD (Alt+F9). Команда компілює модулі програми і будує програму для виконання. Для відбору модулів порівнюються дата і час створення об’єктних файлів модулів з датою і часом змін у файлах на вихідній мові. Компілюються тільки ті модулі, що були змінені.

Якщо модулі застосовують файли ресурсів, які були змінені, то перед побудовою програми експерти коду поновлять код у модулях проекту. Команда будує файли ресурсів і інтерактивної довідки.

Команда PROJECT/REBUILD ALL (Ctrl+Alt+F9). Команда працює аналогічно команді PROJECT/BUILD, але компілюються всі модулі незалежно від дати і час їх створення.

Команда PROJECT/LINK ONLY (Shift+F9). Команда не компілює модулі. Вона будує програму для виконання.

Команда PROJECT/STOP BUILDING (Alt+F10). Команда припиняє компіляцію або побудову програми.

Команда PROJECT/RUN (F9). Команда працює аналогічно команді PROJECT/BUILD і запускає програму на виконання.

Команда PROJECT/DEBUG (Ctrl+Shift+F9). Автоматично створюється вікно проекту і запускається програма DEBUG. У вікні проекту можна обрати модуль, налагодження якого відбувається. При налагоджувані можна відкрити кілька вікон для перегляду результатів прогону. Повернутися у середовище VDE можна за опцією File/Exit (Alt-x).

Команда PROJECT/ PROJECT TREE (Ctrl+T). Після компіляції проекту командою можна подивитися дерево проекту. Щоб одержати текст модуля у вікні достатньо натиснути двічі лівою кнопкою миші на файлі. На рисунку 4.1 подано дерево проекту для модульної програми з розділу 4.1.

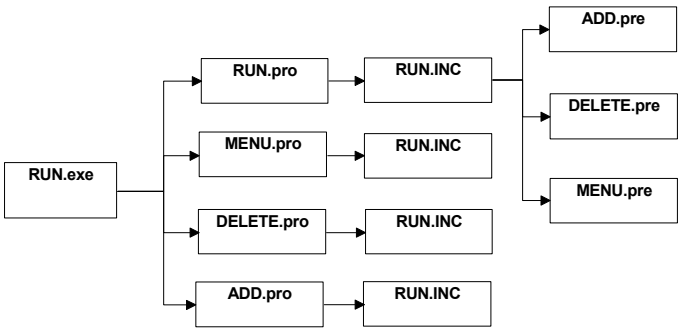


Рисунок 4.1 – Приклад дерева модульної програми

Настроювання можливостей компілятору, виконуються з опції меню Options/Project/Compiler Options.

Команда Options/Project/Compiler Options/Errorlevel. Команда дозволяє впливати на код програми, що компілюється.

1. Режим MINIMAL застосовується для створення оптимального коду програми, що виконується (малого за об'ємом і ефективним в роботі). Режим застосовується для вже налагоджених програм. В код вставляється слідкування за мінімальною кількістю помилок при виконанні програми. За помилками виводяться номери помилок.

2. Режим MEDIUM застосовують за замовчанням. В код програми вставляється код слідкування за помилками. Виводяться номери помилок та вказується їх місце в програмі (назва модуля, назва включеного файлу за Include, номер байту від початку вихідної програми).

3. Режим MAXIMAL аналогічний режиму medium, але код слідкує за максимальною кількістю помилок (переповненням стеку, кучі, тощо).

Команда Options/Project/Compiler Options/Runtime Checking. Команда контролює цілі числа на переповнення. Якщо відмітити опцію, то при переповненні розрядів фіксується помилка «Арифметичне переповнення», інакше в знаковий розряд попадає одиниця і число змінює знак. Результат роботи програми буде невірний.