МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

СТРУКТУРНЕ ПРОГРАМУВАННЯ

Методичні вказівки до виконання індивідуальних завдань

з курсу «Технологія програмування»
для студентів всіх рівнів та форм навчання спеціальностей
113 «Прикладна математика» та
122 «Комп'ютерні науки»

Затверджено
редакційно-видавничою
радою університету,
протокол № 2 від 25.06.2020 р.

Харків
НТУ «ХПІ»
2020

Структурне програмування : методичні вказівки до виконання індивідуальних завдань з курсу «Технологія програмування» для студентів всіх рівнів та форм навчання спеціальностей 113 «Прикладна математика» та 122 «Комп'ютерні науки» / уклад. О. О. Ларін, М. І. Шаповалова. — Харків : НТУ «ХПІ». — 48 с.

Укладачі О. О. Ларін,
М. І. Шаповалова

Рецензент Г. Ю. Мартиненко

Кафедра динаміки та міцності машин

ВСТУП

Для закріплення знань, вироблення умінь і навичок застосування технологій розробки програмних продуктів з курсу «Технологія програмування», потрібне практичне опрацювання основних методів структурного програмування. Це видання повинне допомогти студентам освоїти основні методи та технології побудови діаграм структурного програмування, розібрати поняттям життєвого циклу програмного забезпечення як методології проєктування сучасного програмного забезпечення (ПЗ). На прикладах розібрати побудову блок-схем, діаграм потоків даних та функціональних діаграм.

Перед виконанням індивідуальних лабораторних робіт необхідно вивчити розділи теорії. Оцінка роботи виводиться за такими основними критеріями:

– повнота і правильність виконання;

– актуальність оформлення;

– своєчасність здачі.

1. КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Основні поняття технології програмування. Історія розвитку

Технологією програмування називають сукупність методів і засобів, що використовуються в процесі розробки програмного забезпечення. Як будь-яка інша технологія, технологія програмування являє собою набір технологічних інструкцій, що включають:

- послідовність виконання технологічних операцій;
- умови, при яких виконується та чи інша операція;
- опис самих операцій, де для кожної операції визначено вихідні дані, результати, а також інструкції, нормативи, стандарти, критерії та

методи оцінки та ін.

Щоб розібратися в існуючих технологіях програмування і визначити основні тенденції їх розвитку, доцільно розглядати ці технології в історичному контексті, виділяючи основні етапи розвитку програмування, як науки.

1 етап — «стихійне» програмування. Цей етап охоплює період від моменту появи перших обчислювальних машин до середини 60-х років XX ст. У цей період практично були відсутні сформульовані технології, і програмування фактично було мистецтвом.

2 етап — структурний підхід до програмування (60–70-ті роки XX ст.), являє собою сукупність рекомендованих технологічних прийомів, що охоплюють виконання всіх етапів розробки програмного забезпечення. В основі структурного підходу лежить декомпозиція (розбиття на частини) складних систем з метою подальшої реалізації у вигляді окремих невеликих (до 40–50 операторів) підпрограм. З появою інших принципів декомпозиції (об'єктного, логічного та інше). Даний спосіб отримав назву процедурної декомпозиції.

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

4 етап — компонентний підхід і *CASE*-технології (з середини 90-х років XX ст. до нашого часу). Компонентний підхід передбачає побудову програмного забезпечення з окремих компонентів, окремо існуючих частин програмного забезпечення, які взаємодіють між собою через стандартизовані виконавчі інтерфейси. На відміну від звичайних об'єктів

об'єкти-компоненти можна зібрати в бібліотеки, що динамічно викликаються або у виконувані файли, поширювати в двійковому вигляді (без вихідних текстів) і використовувати в будь-якій мові програмування, що підтримує відповідну технологію. На сьогодні ринок об'єктів став реальністю, так в Інтернеті існують вузли, що надають велику кількість компонентів, рекламою компонентів забиті журнали. Це дозволяє програмістам створювати продукти, що хоча б частково складаються з повторно використаних частин.

1.2. Життєвий цикл програмного забезпечення

Життєвим циклом програмного забезпечення (ПЗ) називають період від моменту появи ідеї створення деякого програмного забезпечення до моменту завершення його підтримки фірмою-розробником або фірмою, яка виконувала супровід.

Склад процесів життєвого циклу регламентується міжнародним стандартом ISO / IEC 12207: 1995 «*Informational Technology — Software Life Cycle Processes*» («Інформаційні технології — Процеси життєвого циклу програмного забезпечення»). Цей стандарт описує структуру життєвого циклу програмного забезпечення і його процеси. Процес життєвого циклу визначається як сукупність взаємопов'язаних дій, що перетворюють деякі вхідні дані у вихідні.

За стандартом процес розробки включає такі дії:

1) постановка задачі (стадія «Технічне завдання»);
2) аналіз вимог і розробка специфікацій (стадія «Ескізнийпроєкт»);
3) проєктування (стадія «Технічний проєкт»);
4) реалізація (стадія «Робочий проєкт»).
5) супровід. Це процес створення і впровадження нових версій програмного продукту.

Моделі життєвих циклів. Найбільш простою і природною є, так звана *каскадна* або водоспадна (*waterfall*) модель життєвого циклу. Ця модель передбачає послідовне виконання різних видів діяльності, починаючи з вироблення вимог і закінчуючи супроводом, з чітким визначенням меж між етапами, на яких набір документів, створений на попередній стадії, передається як вхідні дані для наступної (Рис. 1). «Класична» каскадна модель передбачає тільки рух вперед за цією схемою: все необхідне для проведення чергової діяльності повинно бути підготовлено в ході попередніх робіт.

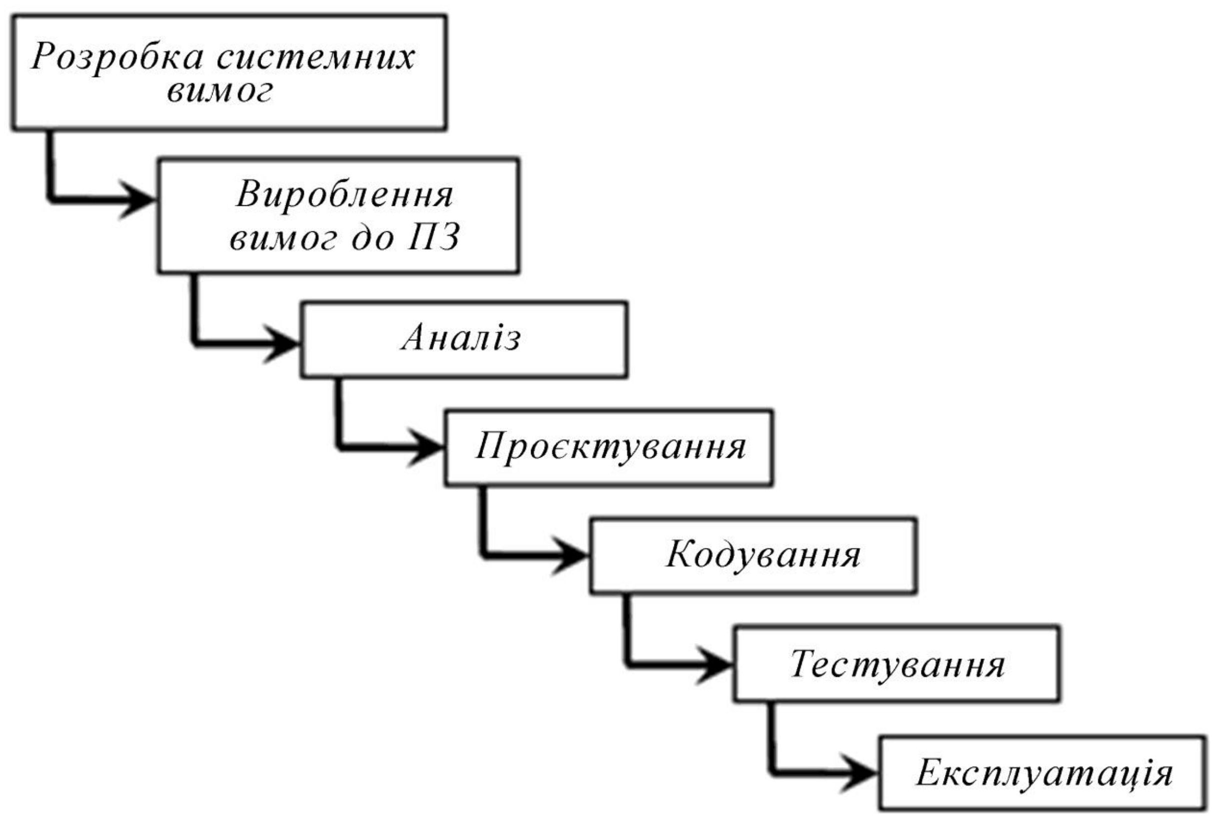


Рисунок 1 — Каскадна модель життєвого циклу

Ітеративні, або інкрементальні моделі припускають розбиття створюваної системи на набір блоків, які розробляються за допомогою декількох послідовних проходів всіх робіт або їх частини (Рис. 2). Ітеративний процес передбачає, що різні види діяльності остаточно не

прив'язані до певних етапів розробки, а виконуються у міру необхідності, іноді повторюються, доти, поки не буде отримано потрібний результат.

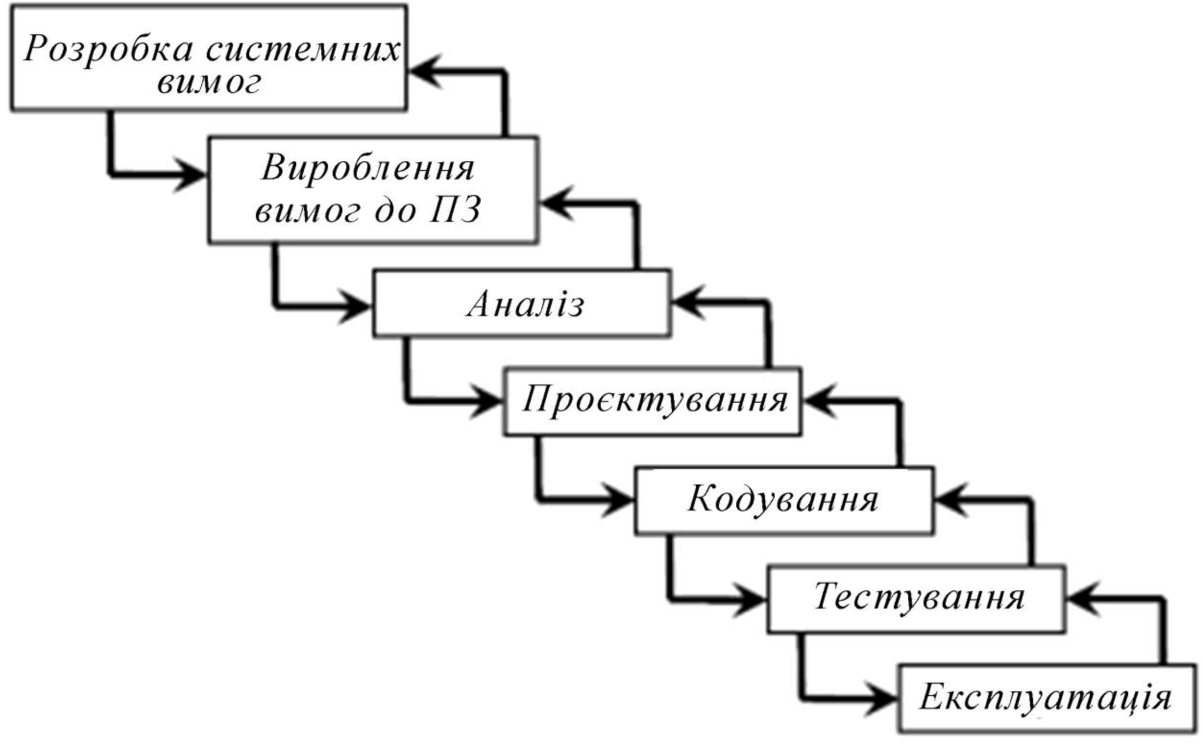


Рисунок 2 — Ітераційна модель (або модель з проміжним контролем) життєвого циклу ПЗ

Розвитком ідеї ітерацій є спіральна модель життєвого циклу ПЗ, запропонована Боема (*Boehm*). Вона пропонує кожну ітерацію починати з виділення цілей і планування чергової ітерації, визначення основних альтернатив і обмежень при її виконанні, їх оцінки, а також оцінки ризиків, що виникають і визначення способів позбавлення від них, а закінчувати ітерацію оцінкою результатів проведених в її рамках робіт.

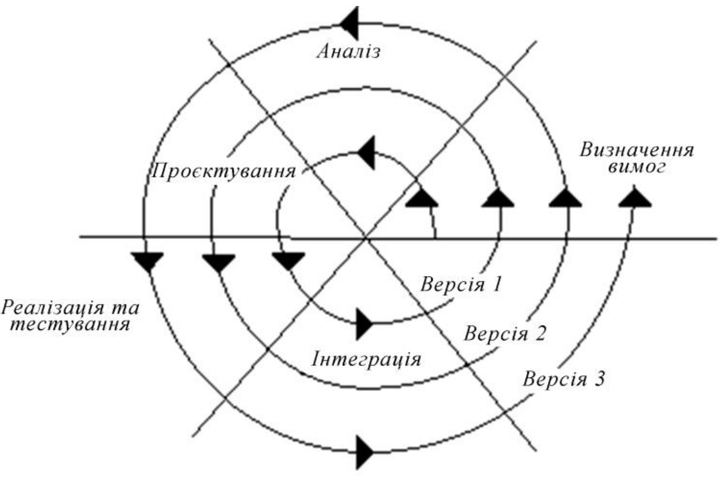


Рисунок 3 — Спіральна модель життєвого циклу ПЗ (спрощений варіант)

Рисунок 4 — Спіральна модель життєвого циклу ПЗ

Модель отримала назву спіральної, через зображення ходу робіт в «полярних координатах», в яких кут відповідає виконуваному етапу в рамках загальної структури ітерацій, а віддалення від початку координат — витраченим ресурсам (Рис. 3–4).

Виділяють дві основні методології ітеративної розробки ПЗ:

▪ «Жорсткі» («важкі») методології або уніфікований процес розробки (*Rational Unified Process, RUP*)

▪ «Гнучкі» («легкі») методології: ***XP*** *— extreme programming* (екстремальне програмування), *Crystal, ASD* (адаптивна розробка), *SCRUM* (бійка), *DSDM — dynamic system design method*,

▪ «Гнучкий» *RUP: dX* (Роберт Мартін), процес Крейга Ларман.

Ключові ідеї *RUP*:

1. Розробка починається з виділення варіантів використання (use cases) і на кожному кроці контролюється ступенем наближення до їх реалізації. *Варіантом використання* називається сценарій взаємодії результуючої програмної системи з користувачами або іншими системами, при виконанні яких користувачі отримують значні для них результати і послуги.

2. Основним рішенням, прийнятим в ході проєкту, є архітектура результуючої програмної системи. Архітектура встановлює набір компонентів, з яких буде побудовано ПЗ, відповідальність кожного з компонентів (підзадачі в рамках загальних завдань системи), чітко визначає інтерфейси, через які вони можуть взаємодіяти, а також способи взаємодії компонентів один з одним. Архітектура є одночасно основою для отримання якісного ПЗ і базою для планування робіт. Вона оформляється у вигляді набору графічних моделей на мові *UML*.

3. Основою процесу розробки є плановані і керовані ітерації, обсяг яких (реалізована в рамках ітерації функціональність і набір компонентів), визначається на основі архітектури.

«Гнучкі» (*Agile*) методи (*Kent Beck, Ward Cunningham і Ron Jeffries*)

розробки роблять упор на використанні хороших розробників, а не добре налагоджених процесів розробки. Живі методи уникають фіксації чітких схем дій, щоб забезпечити більшу гнучкість в кожному конкретному проєкті, а також виступають проти розробки додаткових документів, які не роблять безпосереднього внеску у здобуття готової працюючої програми.

Згідно з цими методиками в процесі розробки ПЗ необхідно застосовувати такі техніки:

1. Живе планування (*planning game*) — все планування здійснюється в реальному часі і виходить виключно з бажання замовника.

2. Часта зміна версій (*small releases*) — час виходу і початку використання версії не повинен перевищувати двох місяців.

3. Метафора системи (*metaphor*) — поняття заміняє архітектуру ПЗ. У вигляді 1–2х пропозицій (схем) описує суть прийнятих технічних рішень.

4. Прості проєктні рішення (*simple design*) — непотрібно додавати надлишкових функцій. Будь-яка функція повинна бути вимогою замовника.

5. Розробка на основі тестування (*test-driven development*) — тести пишуться одночасно з розробкою робочих кодів.

6. Постійна переробка.

7. Програмування парами (*pair programming*) — працюють по 2 програміста за комп'ютером (один за клавіатурою, а другий стежить і дає поради). Пари змінюються.

8. Колективна відповідальність — немає спеціалізації.

9. Постійна інтеграція — збір системи і повне тестування після кожної закінченою функції парою програмістів (кілька разів на день).

10. 40-ка годинний робочий тиждень (без понаднормової роботи).

11. Включення замовника в команду (*on-site customer*).

12. Ясність коду основний пріоритет.

13. Відкритий загальний робочий простір.

14. *JUST RULES*.

Як видно із застосовуваних технік, *XP* розраховане на використання в рамках невеликих команд (не більше 10 програмістів), що підкреслюється і авторами цієї методики. Більший розмір команди руйнує необхідну для успіху простоту комунікації та унеможливлює застосування багатьох перерахованих прийомів (Рис. 5).

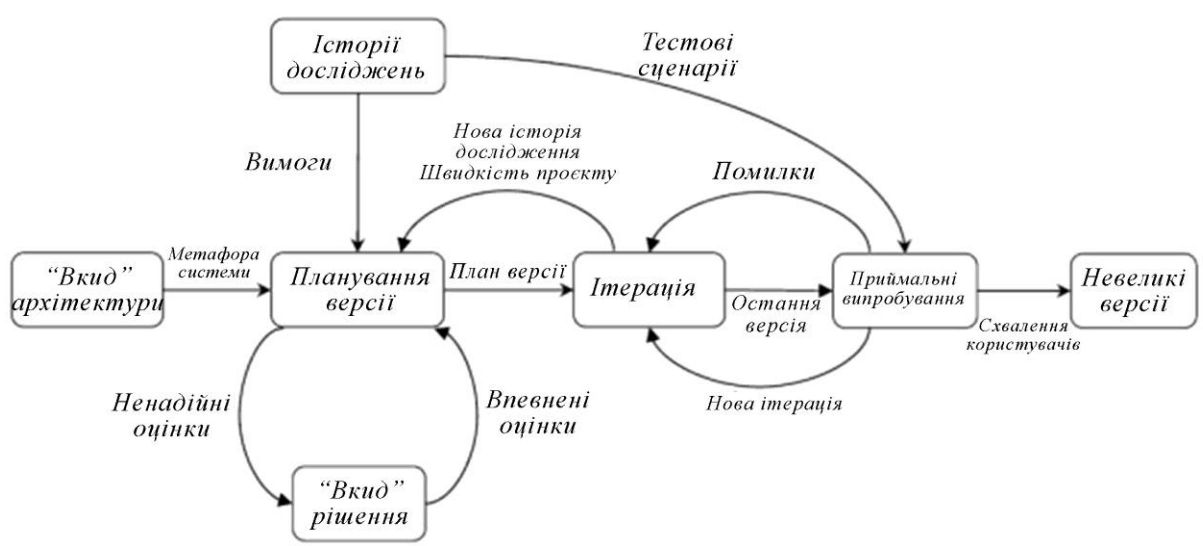


Рисунок 5 — Загальна схема *Agile* процесу розробки ПЗ

До <u>переваг</u> такого підходу відносять: велику гнучкість, можливість швидко і акуратно вносити зміни в ПО у відповідь на зміни вимог і окремі побажання замовників, висока якість виходить в результаті коду та немає необхідності переконувати замовників в тому, що результат відповідає їхнім очікуванням.

До <u>недоліків</u> можна віднести: нездійсненність в такому стилі досить великих і складних проєктів, неможливість планувати терміни і трудомісткість проєкту на досить довгу перспективу і чітко передбачити результати тривалого проєкту в термінах співвідношення результату до витрат часу і ресурсів. А також непристосованість *XP* для тих випадків, в яких можливі рішення не знаходяться відразу на основі раніше отриманого досвіду, а вимагають проведення попередніх досліджень.

1.3. Структурний підхід. Аналіз і визначення специфікацій

В рамках структурного підходу на етапі аналізу і визначення специфікацій використовують три типи моделей:

● орієнтовані на функції;

● орієнтовані на дані;

● орієнтовані на потоки даних.

Кожну модель доцільно використовувати для свого специфічного класу програмних розробок. Слід мати на увазі, що всі функціональні специфікації описує одні і ті ж характеристики розроблюваного програмного забезпечення: перелік функцій і склад оброблюваних даних. Вони розрізняються тільки системою пріоритетів (акцентів), яка використовується розробником в процесі аналізу вимог і визначення специфікацій. Діаграми *переходів станів* визначають основні аспекти поведінки програмного забезпечення в часі, діаграми *потоків даних* — напрямок і структуру потоків даних, а концептуальні діаграми *класів* — відношення між основними поняттями предметної області.

1). Специфікації орієнтовані на функції. Функціональні діаграми.

Функціональними називають діаграми, що в першу чергу відображають взаємозв’язки функцій розроблюваного програмного забезпечення. Як приклад функціональної моделі, розглянемо модель активностей, запропоновану Д. Россом в складі методології функціонального моделювання *SADT* (*Structured Analysis and Design Technique*) — технологія структурного аналізу і проєктування у 1973 р.

Правила побудови *SADT*-діаграм:

♦ графічне представлення моделей у вигляді набору діаграм (ієрархічне дерево);

♦ обмеження кількості вузлів на кожному рівні від 3 до 6;

♦ діаграми зв’язуються за допомогою дуг;

♦ унікальність міток і найменувань;

♦ розділення входів та управлінь;

♦ розділення процесу організації від функцій.

Результатом застосування методології *SADT* є модель, яка складається з діаграм, нотацій функцій і глосарію, що мають посилання один на одного. Діаграми — головні компоненти моделі, всі функції інформаційної системи (ІС) та інтерфейси на них представлені як блоки і дуги. Місце з’єднання дуги з блоком визначає тип інтерфейсу. *Керуюча* інформація входить в блок зверху, в той час як інформація, яка піддається обробці, показана з лівого боку блоку, а результати виходу показані з правого боку. *Механізм* (людина або автоматизована система), який здійснює операцію, представлений дугою, що входить в блок знизу (Рис. 6).

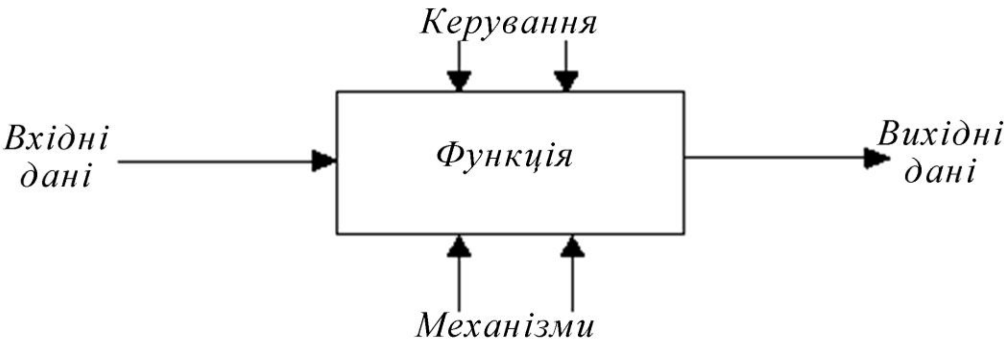


Рисунок 6 — Функціональна діаграма

Фізично дуги вихідних даних, результатів і управління є наборами даних, що передаються між функціями. Дуги, що визначають механізм виконання функції, в основному використовуються для опису специфікацій складних інформаційних систем, які включають як автоматизовані, так і ручні операції. Блоки і дуги підписуються природною мовою.

Блоки на діаграмі розміщують за «ступінчастою» схемою відповідно до послідовності їх роботи або домінування, яке розуміється як вплив, який чиниться одним блоком на інші. У функціональних діаграмах *SADT* розрізняють п’ять типів впливів блоків один на одного (Рис. 7):

1) управління — вихід блоку використовується як управління для блоку з меншим домінуванням (Рис. 7, *а*);

2) вхід — вихід блоку подається на вхід блоку з меншим домінуванням, тобто наступного (Рис. 7, *б*);

3) зворотний зв'язок з управління — вихід блоку використовується як керуюча інформація для блоку з великим домінуванням (попередній) (Рис. 7, *в*);

4) зворотній зв'язок по входу — вихід блоку подається на вхід блоку з великим домінуванням (попереднього) (Рис. 7, *г*);

6) вихід-виконавець — вихід блоку використовується як механізм для іншого блоку (Рис. 7, *д*).

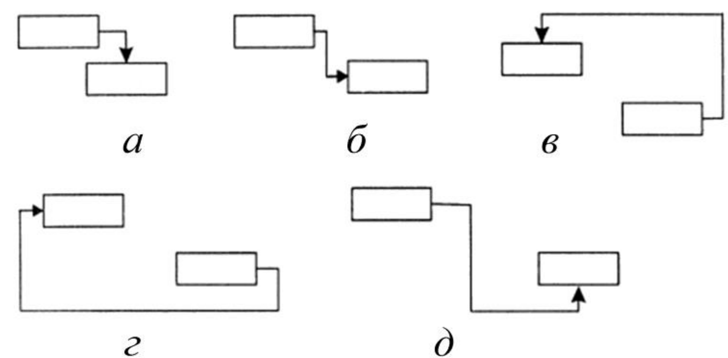


Рисунок 7 — Типи впливу блоків один на одного:
а — управління; *б* — вхід; *в* — зворотний зв'язок з управління;
г — зворотний зв'язок по входу; *д* — вихід-виконавець

Однією з найбільш важливих особливостей методології *SADT* є поступове введення все більших рівнів деталізації у міру створення діаграм, що відображають модель.

Побудова ієрархії функціональних діаграм ведеться поетапно зі збільшенням рівня деталізації: діаграми кожного такого рівня уточнюють структуру батьківського блоку. Побудову моделі починають з єдиного блоку, для якого визначають вихідні дані, результати, управління і механізми реалізації. Потім він послідовно деталізується з використанням

методу покрокової деталізації. При цьому рекомендується кожну функцію представляти не більше ніж 3–7 блоками. У всіх випадках кожна підфункція може використовувати або продукувати тільки ті елементи даних, які використані або продукуються батьківською функцією, причому ніякі елементи не можуть бути опущені, що забезпечує несуперечність побудованої моделі.

Стрілки, що приходять з батьківської діаграми або йдуть на неї, нумерують, використовуючи символи і числа. Символ позначає тип зв'язку: ***I*** (*input*) — вхідні, ***C*** (*control*) — керуючі, ***M*** (*mechanism*) — механізм, ***R*** (*result*) — результат. ***Число*** — номер зв'язку по відповідній стороні батьківського блоку, починаючи зверху вниз і зліва направо.

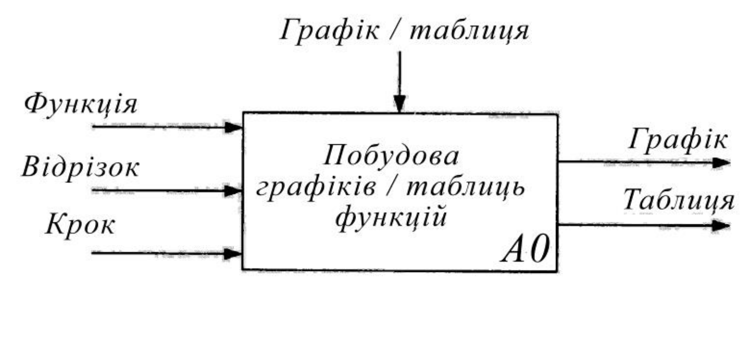
Всі діаграми пов'язані одна з одною ієрархічною нумерацією блоків: перший рівень — ***A0*,** другий — ***A1*, *A2*** і так далі, третій — ***A11*, *A12*, *A13*** і так далі, де перша цифра — номер батьківського блоку, а остання — номер конкретного субблока батьківського блоку.

Деталізацію завершують після отримання функцій, призначення яких добре зрозуміло як замовнику, так і розробнику. Ці функції описують, використовуючи природну мову або псевдокод.

В процесі побудови ієрархії діаграм фіксують всю уточнюючу інформацію і будують словник даних, в якому визначають структури та елементи даних, показані на діаграмах.

Таким чином, в результаті отримують специфікацію, яка складається з ієрархії функціональних діаграм, специфікацій функцій нижнього рівня і словника, що мають посилання один на одного. Словник в цьому випадку повинен містити опис всіх даних, які використовуються в системі.

Приклад функціональної діаграми дослідження функцій (Рис. 8).



а

б

Рисунок 8 — Функціональні діаграми дослідження функцій:

а — діаграма верхнього рівня; *б* — уточнююча діаграма

2). Специфікації орієнтовані на потоки і структуру даних.

Методології структурного аналізу і проєктування, засновані на моделюванні потоків даних, зазвичай використовують комплексне уявлення проєктованого програмного забезпечення у вигляді сукупності моделей:

- діаграм переходів станів (*STD* — *State Transition Diagrams*), що характеризують поведінку системи в часі;
- діаграм потоків даних (*DFD* — *Data Flow Diagrams*), що описують взаємодію джерел і споживачів інформації через процеси, які повинні бути реалізовані в системі;
- діаграм «сутність-зв'язок» (*ERD* — *Entity-Relationship Diagrams*), що описують бази даних системи, що розробляється;
- специфікацій процесів;
- словників термінів.

Взаємозв'язок елементів такої узагальненої моделі показана на Рис. 9.

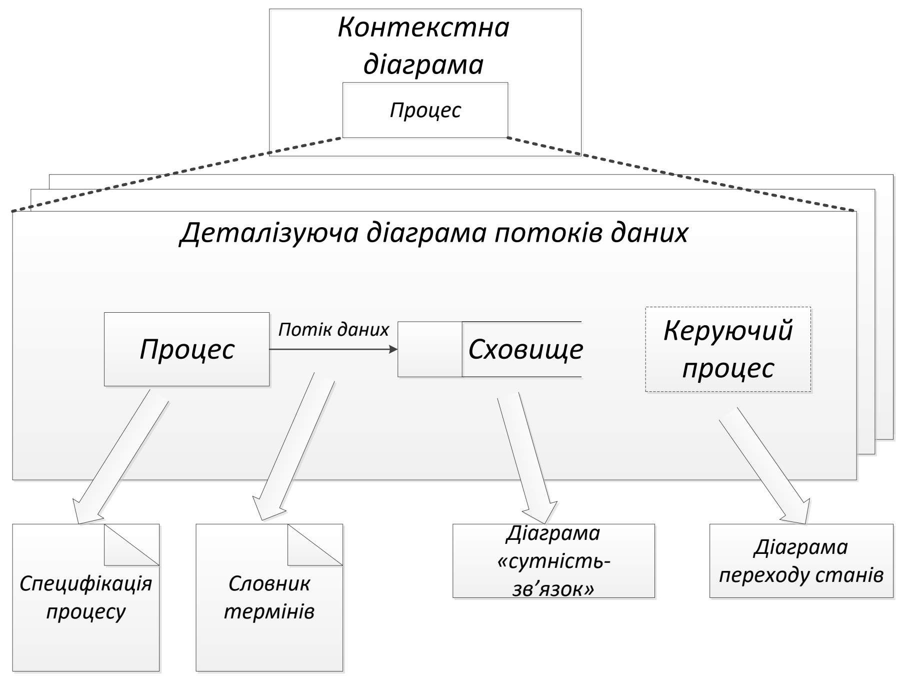


Рисунок 9 — Загальний вид специфікацій орієнтованих на потоки і структури даних

Діаграма переходу станів є графічною формою надання кінцевого автомата — математичної абстракції, що використовується для моделювання детермінованої поведінки технічних об'єктів або об'єктів реального світу.

На етапі аналізу вимог і визначення специфікацій, діаграма переходу

станів демонструє поведінку розроблюваної програмної системи при отриманні керуючих впливів. Під керуючими впливами або сигналами в даному випадку розуміють керуючу інформацію, що отримується системою ззовні. Наприклад, керуючими впливами вважають команди користувача і сигнали датчиків, підключених до комп'ютерної системи. Отримавши керуючий вплив, система повинна виконати певні дії та залишитись в тому ж стані, або перейти в інший стан взаємодії із зовнішнім середовищем.

Діаграми потоків даних дозволяють специфікувати як функції розроблюваного програмного забезпечення, так і оброблювані їм дані. При використанні цієї моделі систему представляють у вигляді ієрархії діаграм потоків даних, що описують асинхронний процес перетворення інформації з моменту введення в систему до видачі користувачеві. На кожному такому рівні ієрархії відбувається уточнення процесів, поки черговий процес не буде визнаний елементарним (моделі потоків даних були незалежно запропоновані спочатку Е. Йорданом (1975), потім Ч. Гейне і Т. Сарсон (1979)).

В основі моделі лежать поняття зовнішньої сутності, процесу, сховища (накопичувача) даних і потоку даних.

▪ *зовнішня сутність* — матеріальний об'єкт або фізична особа, що виступають як джерело або приймач інформації, наприклад, замовники, персонал, постачальники, клієнти, банк та ін.

▪ *процес* — перетворення вхідних потоків даних у вихідні відповідно до певного алгоритму. Кожен процес в системі має свій номер який пов'язаний з виконавцем, що здійснює дане перетворення. Як у випадку функціональних діаграм, фізичне перетворення може здійснюватися комп'ютерами, вручну або спеціальними пристроями. На верхніх рівнях ієрархії, коли процеси ще не визначені, замість поняття «процес» використовують поняття «система» і «підсистема», які позначають

відповідно систему в цілому або її функціонально закінчену частину.

▪ *сховище даних* — абстрактний пристрій для зберігання інформації. Тип пристрою і способи переміщення, вилучення та зберігання для такого пристрою не деталізуються. Фізично це може бути: база даних, файл, таблиця в оперативній пам'яті, картотека на папері та ін.

▪ *потік даних* — процес передачі деякої інформації від джерела до приймача. Фізично процес передачі інформації може відбуватися по кабелях під керуванням програми або програмної системи чи вручну.

Для зображення діаграм потоків даних традиційно використовують два види нотацій: нотації Йордана і Гейне-Сарсона (табл. 1).

Таблиця 1 — Нотації діаграм потоків даних

Поняття	Нотація Йордана	Нотація Гейна-Сарсона
Зовнішня сутність	Наймену вання	Номер Найменування
Система, підсистема чи процес	Найме нуван ня	Номер Найменування Механізм
Накопичувач даних	Найменування	№ Найменування
Потік	Найменування →	Найменування →

Побудова ієрархії діаграм потоків даних починають з діаграми особливого виду — *контекстної діаграми*, яка визначає найбільш загальний вигляд системи. На такій діаграмі показують, як система буде взаємодіяти з приймачами і джерелами інформації не враховуючи виконавців. Діаграми описують інтерфейс між системою і зовнішнім світом. Зазвичай початкова контекстна діаграма має форму зірки.

Якщо проєктована система містить велику кількість зовнішніх сутностей (більше 10-ти), має розподілену природу або включає вже існуючі підсистеми — прийнято будувати ієрархії контекстних діаграм. При розробці контекстних діаграм відбувається деталізація функціональної структури майбутньої системи, що особливо важливо, якщо розробка ведеться кількома колективами розробників. Отриману таким чином модель системи перевіряють на повноту вихідних даних, на об'єкти системи та ізольованість об'єктів (відсутність інформаційних зв'язків з іншими об'єктами).

На подальшому етапі кожну підсистему контекстної діаграми деталізують за допомогою діаграм потоків даних. В процесі деталізації дотримуються правила балансування — при деталізації підсистеми можна використовувати компоненти тільки тих підсистем, з якими у розроблюваної підсистеми існує інформаційний зв'язок.

Рішення про завершення деталізації процесу приймають в таких випадках:

1) коли процес взаємодіє з 2–3-ма потоками даних;

2) коли опис процесу можливий за допомогою послідовного алгоритму;

3) коли процес виконує єдину логічну функцію перетворення вхідної інформації у вихідну.

На недеталізовані процеси встановлюють специфікації, які повинні містити опис логіки (функцій) даного процесу. Такий опис може виконуватися: на природній мові, із застосуванням структурованої природної мови.

Для полегшення сприйняття процеси деталізації підсистеми нумерують, дотримуючись ієрархії номерів: так процеси, отримані при деталізації процесу або підсистеми «*1*», повинні нумеруватися «*1.1*», «*1.2*» та ін. Крім цього бажано розміщувати на кожній діаграмі від 3-х до 6–7-ми

процесів і не захаращувати діаграми деталями, неістотними на даному рівні.

Приклад розробки ієрархії діаграм потоків даних програми побудови графіків/таблиць функцій.

Для побудови контекстної діаграми визначимо зовнішні сутності і потоки даних між програмою і зовнішніми сутностями. У даної системи єдина зовнішня сутність — *«Учень»*. Він вводить або вибирає зі списку функцію, задає інтервал і кількість точок, а потім отримує таблицю значень функції і її графік. На Рис. 10 представлена контекстна діаграма системи.

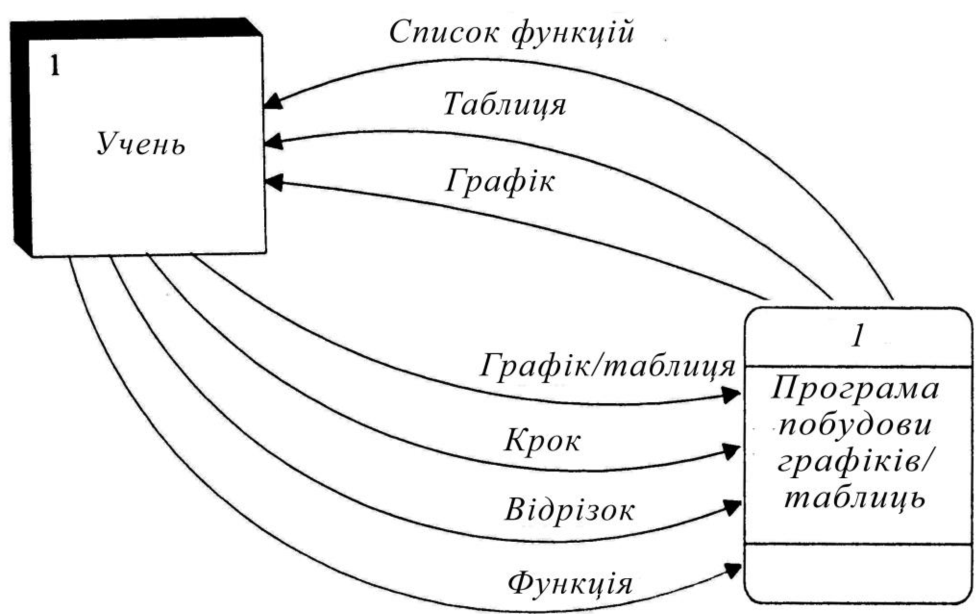


Рисунок 10 — Контекстна діаграма

Деталізуючи цю діаграму, отримуємо три процеси: *Введення/вибір* функції і її розбір, *Побудова таблиці* значень функції та *Побудова графіка* функції. Для зберігання даних додаємо сховище функцій. Потім визначаємо потоки даних (Рис. 11).

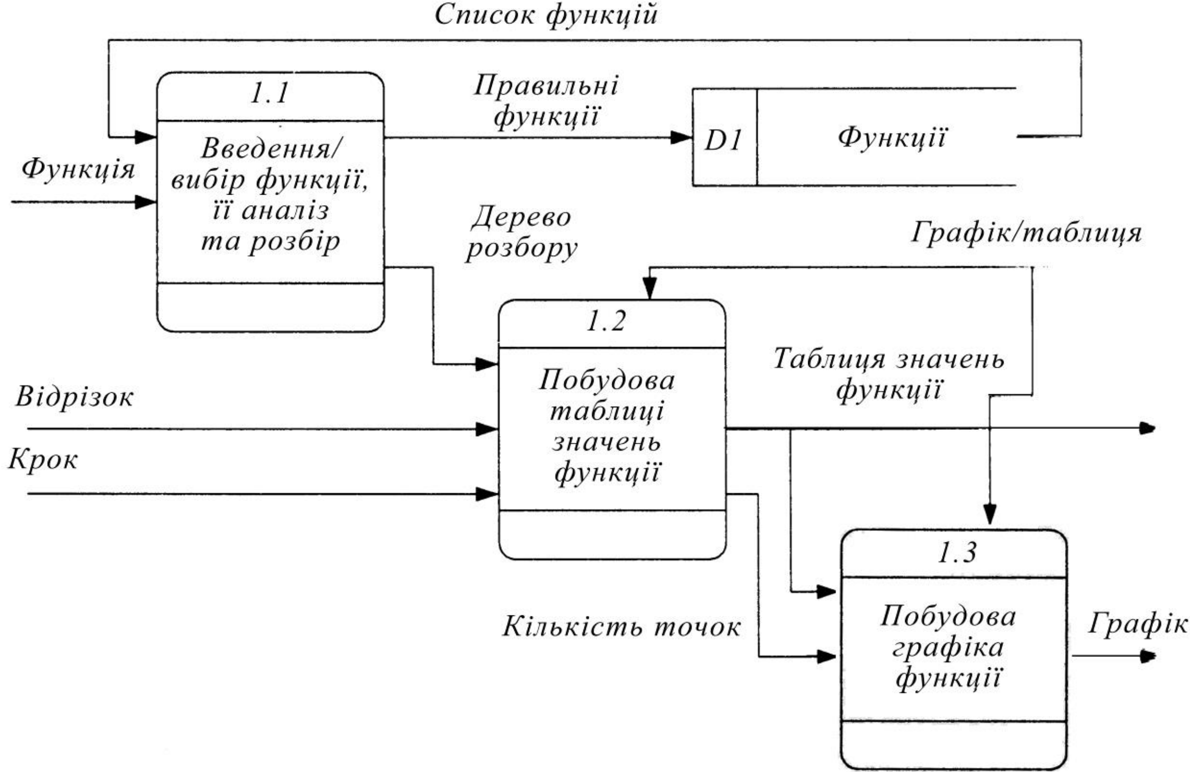


Рисунок 11 — Деталізована діаграма

Повна специфікація процесів (при орієнтації на потоки даних) включає також опис структур даних, що використовуються як при передачі інформації в потоці, так і при зберіганні в накопичувачі. Описувані структури даних можуть містити альтернативи, умовні входження і ітерації. Умовне входження означає, що відповідні елементи даних в структурі можуть бути відсутні.

Структурою даних називають сукупність правил і обмежень, які відображають, зв'язки, що існують між окремими частинами даних. Структури даних можуть бути досить складними і для їх опису застосовуються спеціальні моделі, які прийнято ділити на *ієрархічні* і *мережеві*.

Ієрархічні моделі описують відносини входження елементів даних в компонент більш високого рівня (таблиці і їх комбінації). Ці моделі позначаються діаграмами Джексона або Орра.

У мережевій моделі вводиться поняття *сутності* — об'єкт даних має істотне значення для розглянутої предметної області. Кожна сутність має унікальне ім'я з атрибутами. А також поняття *зв'язку* — поіменована асоціація між сутностями. В результаті будується діаграма «сутність-зв'язок». На рисунках 12–13 наведені приклади діаграм.

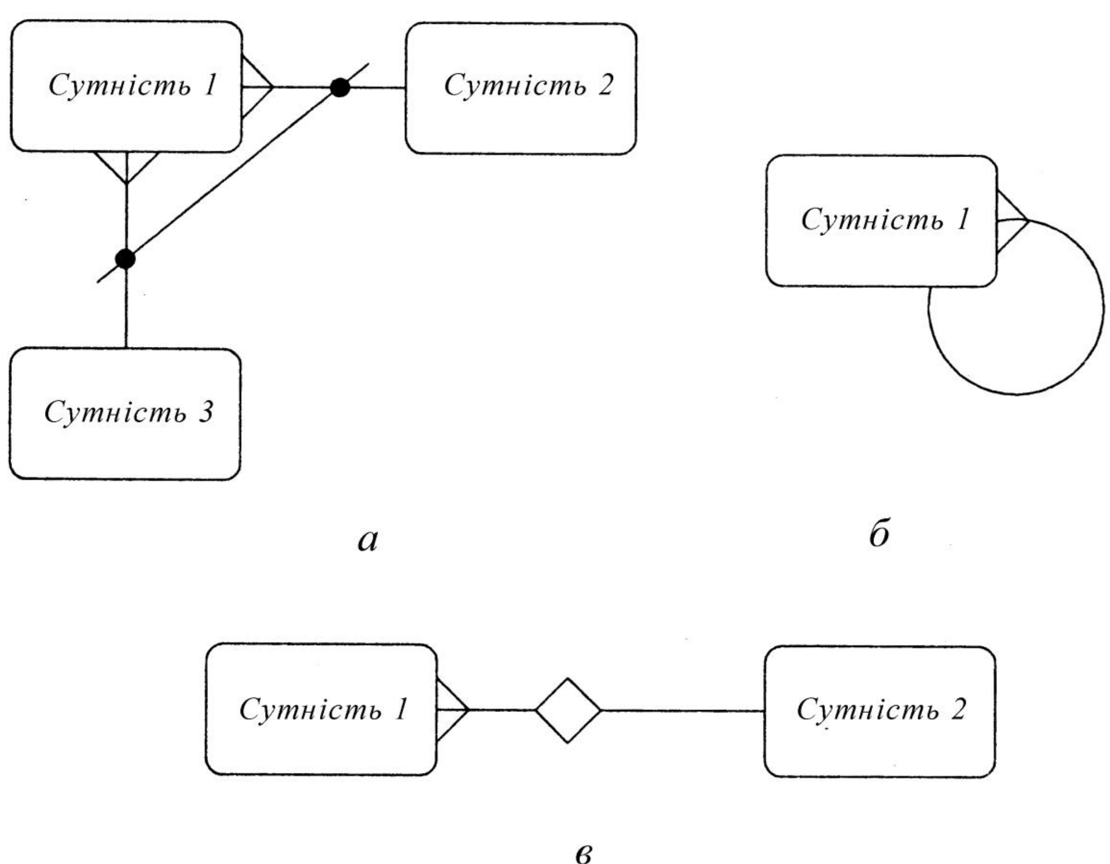


Рисунок 12 — Позначення видів зв'язку:
а — взаємно виключає; *б* — рекурсивна; *в* — непереміщуваними

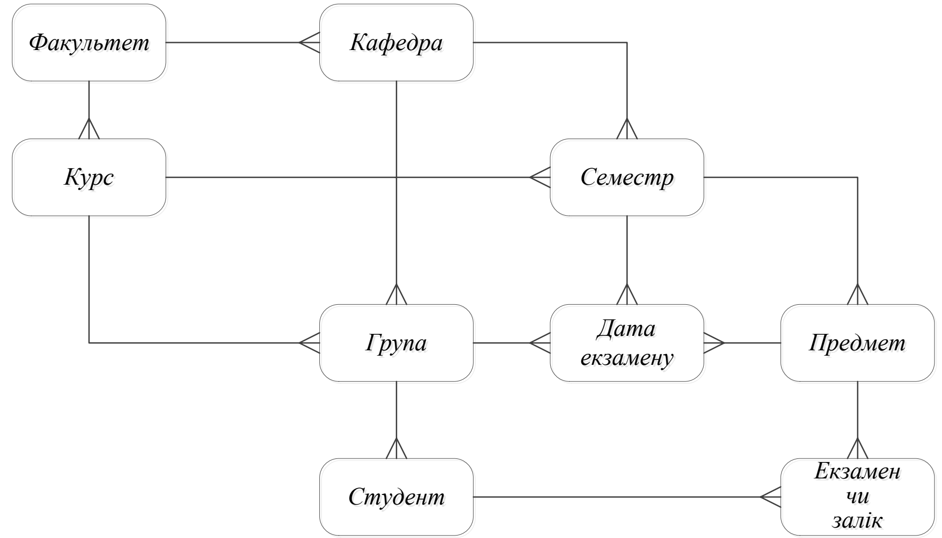


Рисунок 13 — Приклад діаграми «сутність-зв’язок» для опису БД системи успішності студентів

При необхідності створення більш складної структури БД використовують ***реляційні моделі***.

Для специфікації процесів (або функцій) використовують 4 види нотацій:

► блок-схеми (Рис. 14);

► псевдокод — формалізований текстовий опис алгоритму (текстова нотація);

► *Flow*-форми (Рис. 15);

► нотації Насіі-Шнейдермана (Рис. 16).

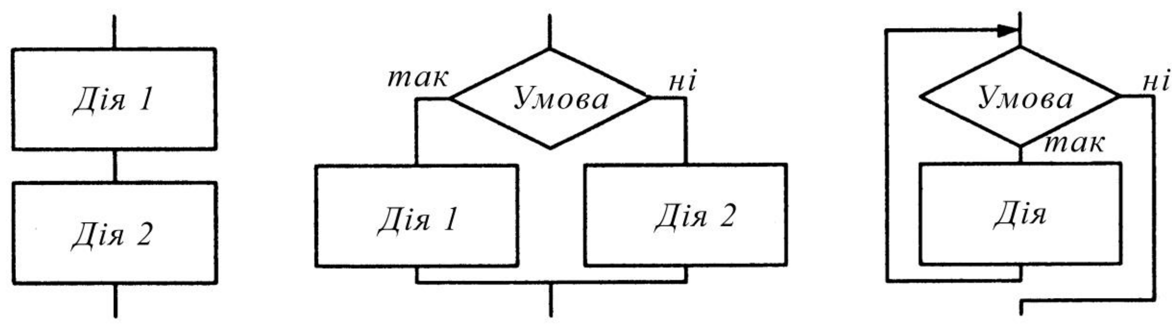


Рисунок 14 — Блок -схема

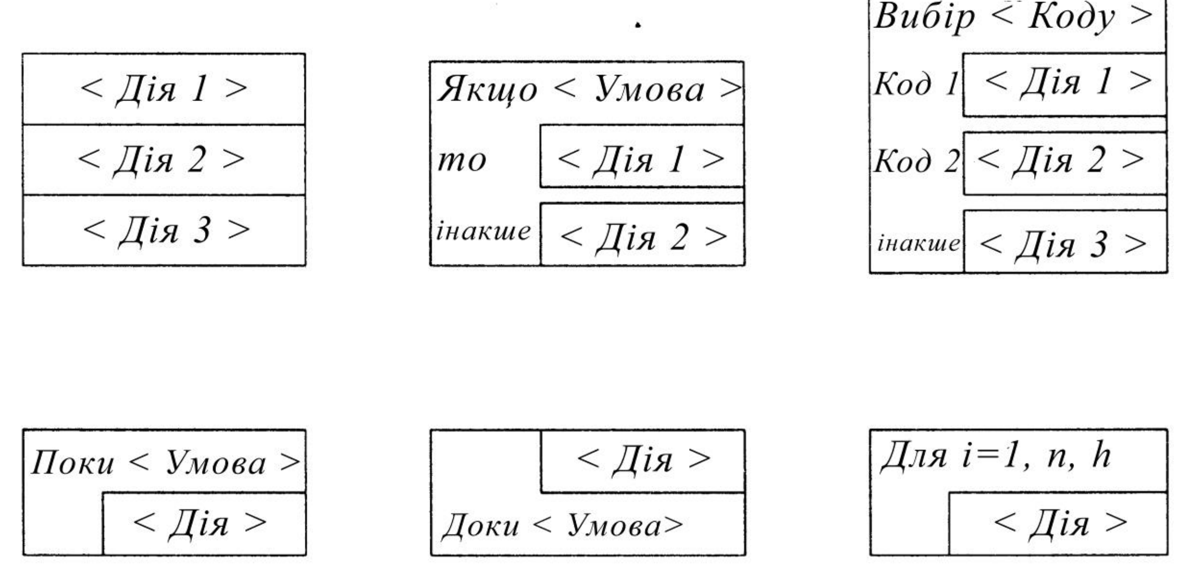


Рисунок 15 — *Flow*-форми

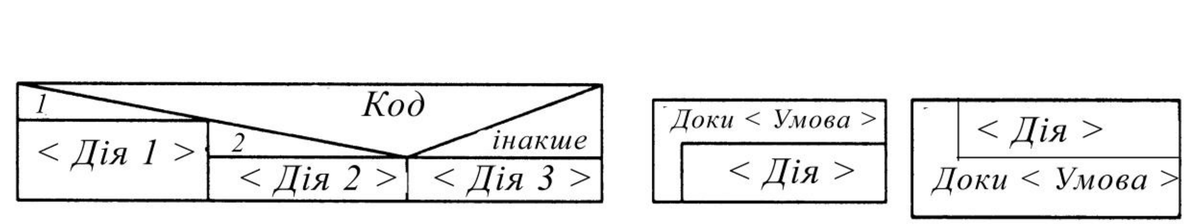


Рисунок 16 — нотації Насс-Шнейдермана

Для зручності проєктування і його технологічності проводиться розбиття на модулі, виходячи з рекомендованих розмірів (20–60 рядків) і складності структури (дві–три вкладені керуючі конструкції). У вигляді модуля (підпрограми) можна реалізувати рішення підзадач, сформульованих на будь-якому етапі процесу деталізації, однак визначальну роль при розбитті програми на модулі відіграють принципи забезпечення технологічності модулів.

Для аналізу технологічності отриманої ієрархії модулів доцільно використовувати *структурні карти* Константайна або Джексона. На структурній карті відносини між модулями представляють у вигляді графа, вершинам якого відповідають модулі та загальні області даних, а дугам — міжмодульні виклики і звернення до загальних областей даних. При цьому окремі частини програмної системи (програми, підпрограми), можуть викликатися послідовно, паралельно або як співпрограми.

Блок схема — графічне зображення ходу виконання алгоритму. У табл. 2 наведені основні графічні елементи схеми.

Типи блок-схем:

а) *послідовна блок схема*, всі дії (кроки) виконуються один за одним;

б) *розгалужувальна блок схема* (алгоритм розгалуження). Залежно від умови виділяють два види: коротка форма — дія передбачена тільки в тому випадку, якщо умова буде виконана; повна форма — дія складається з двох комплектів, в разі виконання умови «Так» — один набір дій, в разі не виконання умови (варіант «Ні») — інший набір дій;

в) *циклічний алгоритм*. Виконання послідовно якоїсь дії декілька разів. Цикл безумовний — якщо кількість повторень заздалегідь відомо; умовний цикл — кількість повторень невідомо, а залежить від умови.

– Цикл з передумовою (умовний). Спочатку перевіряємо умову, потім виконуємо дію (тіло циклу).

– Цикл з подальшою умовою (умовний). Умова перевіряється після виконання тіла циклу. Тіло циклу виконується хоча б 1 раз.

Тіло циклу — набір дій (операторів), які повинні повторюватися певну кількість разів.

– Безумовний цикл. Кількість повторень відомо і не залежить від умови. Задано: початкове, кінцеве значення і крок. Після кожного виконання тіла циклу автоматично відбувається збільшення значення на величину кроку.

Таблиця 2 — Основні елементи блок-схеми

Графічне зображення	Смислове значення	Графічне зображення	Смислове значення
	Початок / кінец		Без умовний цикл
	Ввід / вивід		Перехід
	Дія		Сторінковий перехід
	Умова		Підпрограма

Приклад послідовної блок схеми. Завдання: Вирішити рівняння виду $S = 2 \cdot a + b$, якщо a, b відомо (Рис. 17, *а*).

Приклад розгалужувальної блок схеми. **Завдання**: Вирішити рівняння виду $S = a / b$, якщо a, b відомо. Примітка: ділити на «0» не можна. Необхідно захистити блок схему введенням умови (Рис. 17, *б*, *в*).

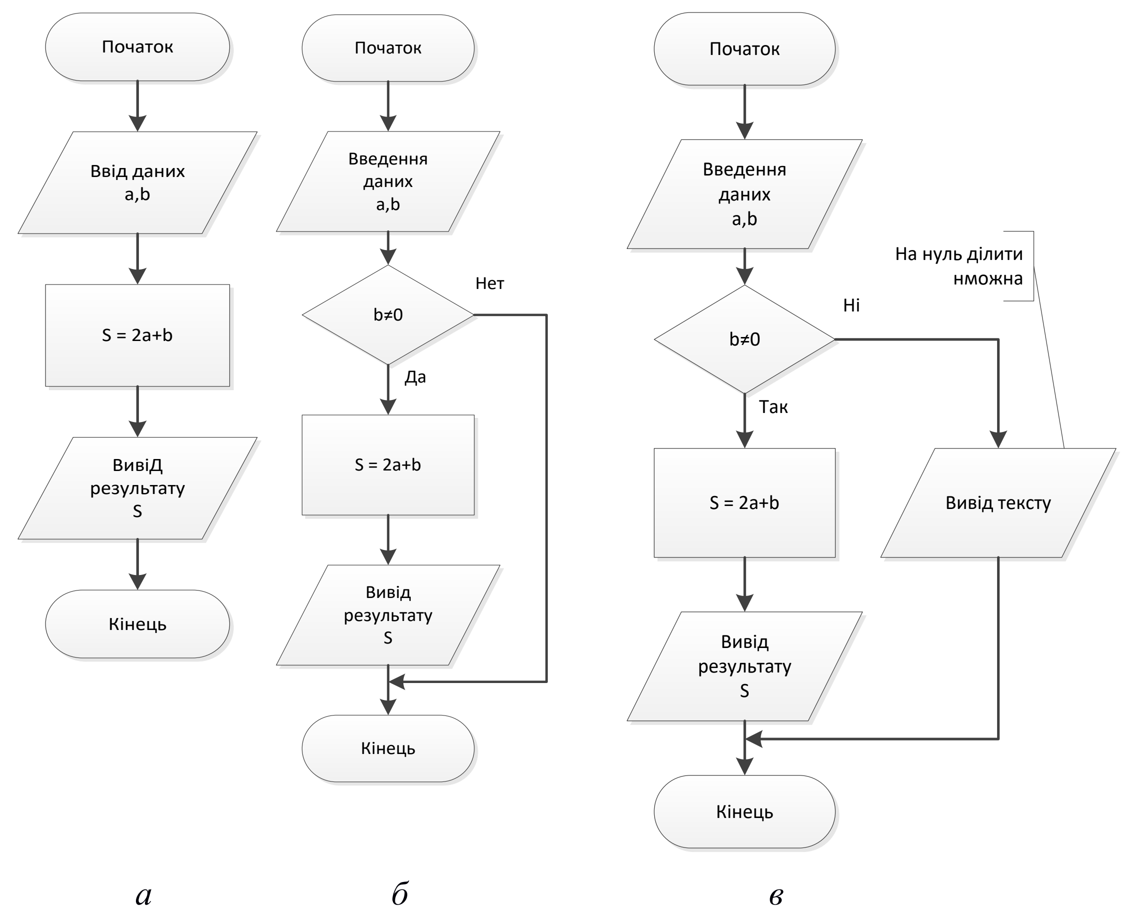


Рисунок 17 — Блок схеми:

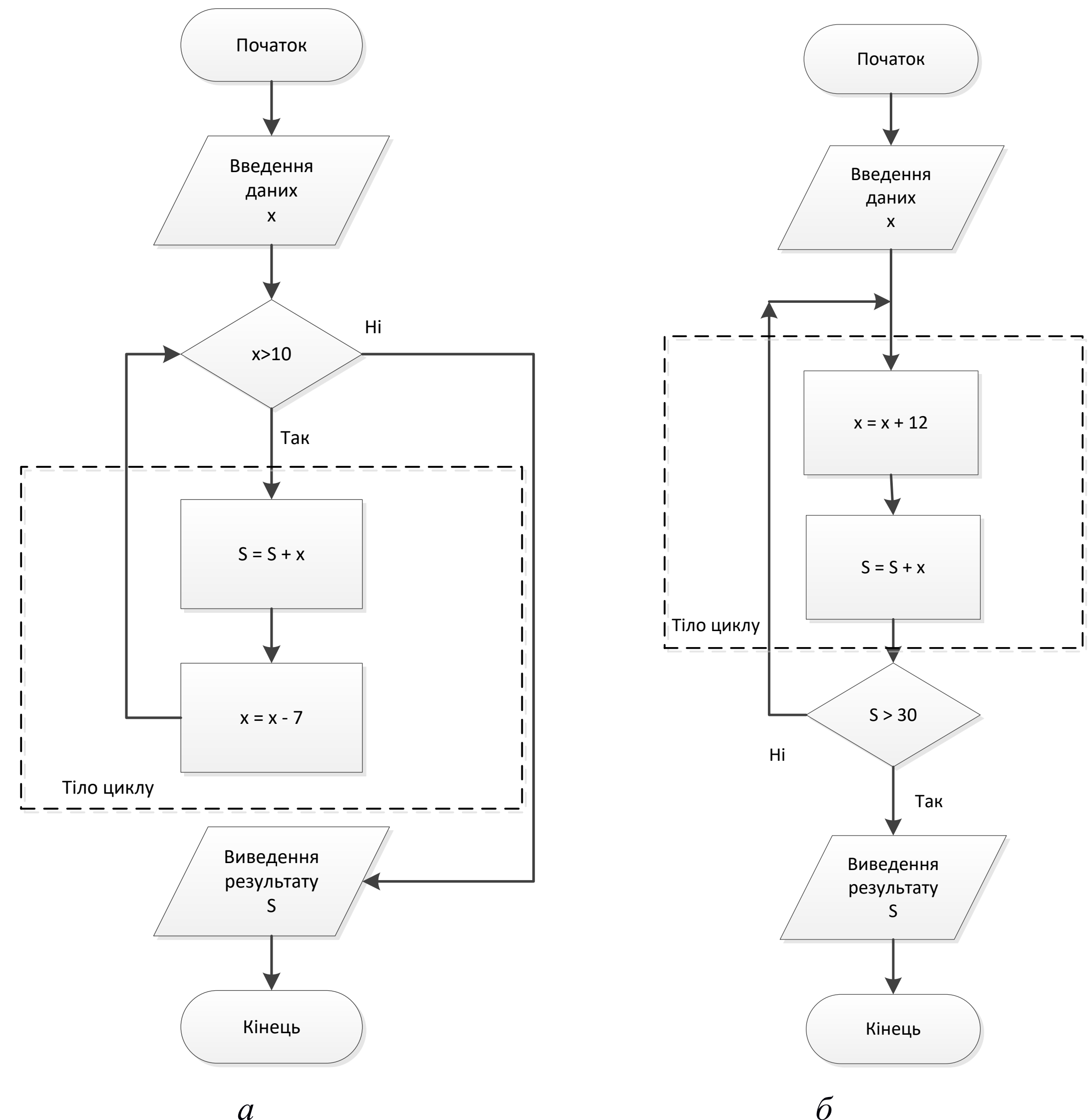
а — послідовна; *б* — розгалужувальна, коротка форма;

в — розгалужувальна, повна форма

Приклад циклічного алгоритму. Правило! *Змінна*, що стоїть в *Умові* повинна змінювати своє значення у тілі циклу, інакше така блок-схема ніколи не виконатися (не дійде до *Кінця*). Отримаємо нескінченний цикл.

1. Цикл з передумовою (Рис. 18, *а*). **Завдання**: Знайти суму S множини x, якщо відомо початкове значення $x = 20$, приріст $x = x - 7$ та умова перевірки — x повинен бути більше 10. Примітка: Якщо змінним не задаються початкові значення, прийнято вважати їх рівними «0» — для числових, і символом нового рядка — для строкових даних.

2. Цикл з подальшою умовою (Рис. 18, *б*). Виконується хоча б 1 раз. **Завдання**: Знайти суму S множини x, якщо відомо початкове значення $x = 10$, приріст $x = x + 12$ і умова перевірки — сума S більше 30.



а *б*

Рисунок 18 — Блок схеми:

а — цикл з передумовою; *б* — цикл з подальшою умовою

3. Безумовний цикл (Рис. 19). Кількість повторень — відомо. **Завдання**: Знайти суму S множин значень a і b, якщо відомо початкове значення $a = 10$, $b = 8$, кількість повторень — 3, приріст $a = a - 2$ за умови що $a > b$, інакше $b = b - 3$.

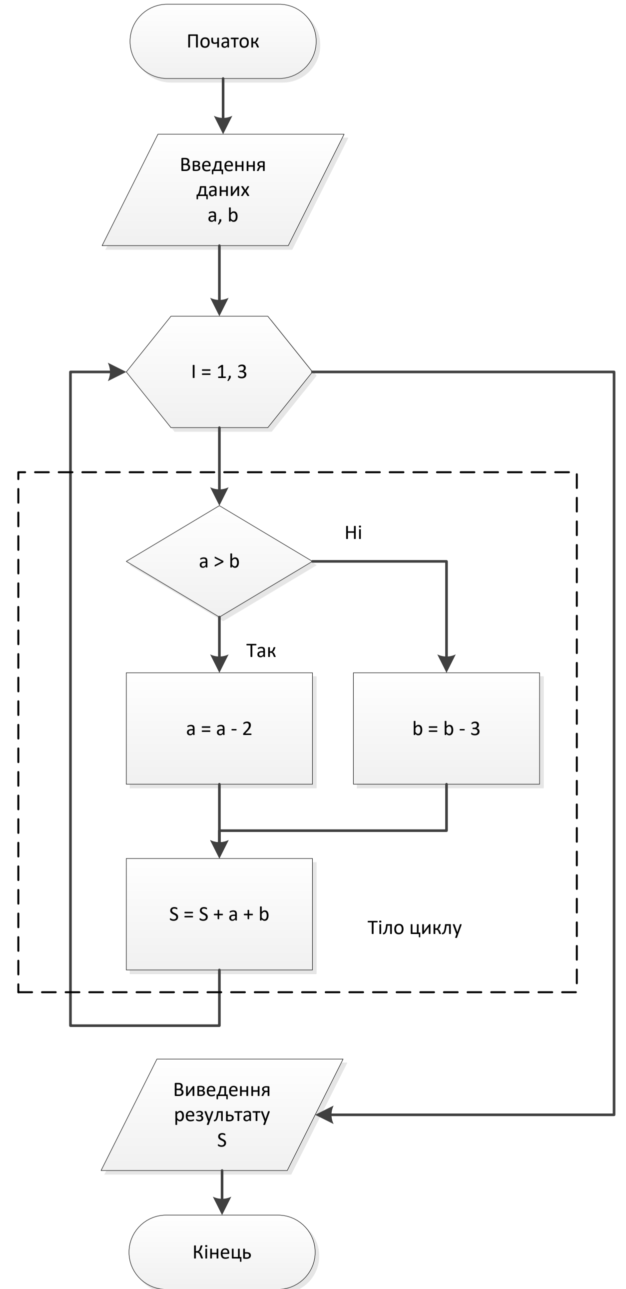
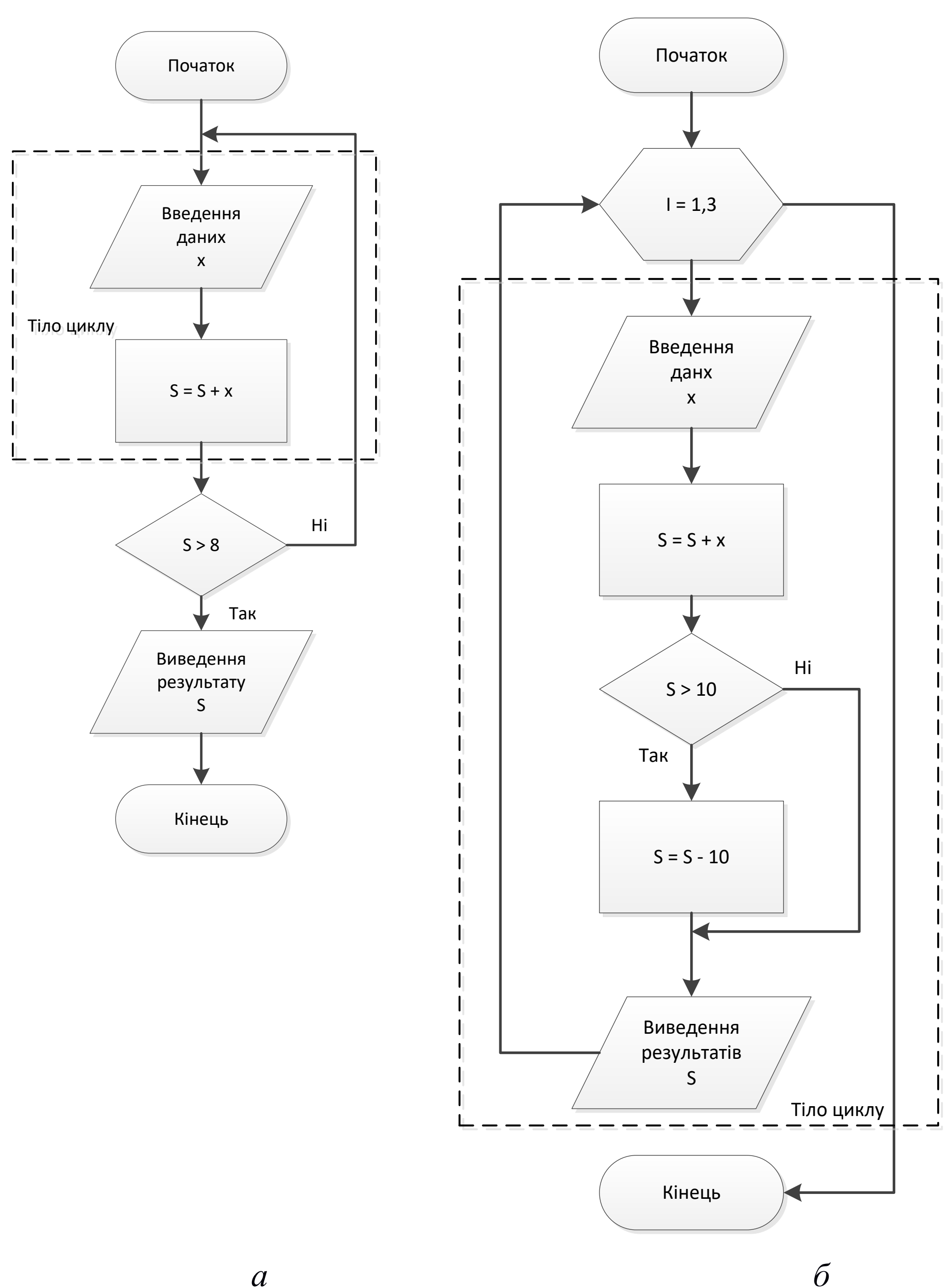


Рисунок 19 — Безумовний цикл

4. Множинне введення даних, умовний циклічний алгоритм. **Завдання**: Знайти суму S задану послідовністю чисел x = (4, 0, 5, 0, 7, 1, 12, 3, 0, 9). Введення чисел з послідовності триває до моменту перевищення значення $S < 8$ (Рис. 20, *а*).

5. Множинне введення / виведення даних, безумовний цикл. **Завдання**: Знайти суму S, якщо відомо що S повинно бути більше 10,

$S = S - 10$. Ведення даних задано послідовністю чисел x = (6, 5, 8) (Рис. 20, *б*).



а *б*

Рисунок 20 — Множинне введення даних:

а — умовний циклічний алгоритм; *б* — безумовний цикл

2. ІНДИВІДУАЛЬНІ ЛАБОРАТОРНІ РОБОТИ

2.1. ЛАБОРАТОРНА РОБОТА 1

Тема: Створення консольного додатку на мові програмування C# без використання інтегрованого середовища розробки (*IDE*).

1. Встановіть компілятор за таким посиланням:

https://dotnet.microsoft.com/learn/dotnet/hello-world-tutorial/install

2. Щоб переконатись у правильності встановлення *SDK*, запустіть командну строку (*cmd*), та введіть команду: *dotnet*.

Якщо команда запускається і виводить до консолі правила користування — все зроблено вірно.

3. Створення першого консольного додатку.

Покрокове виконання лабораторної роботи:

♦ Створіть через консоль папку, де будуть зберігатись лабораторні роботи з дисципліни, використовуючи потрібні команди — створити папку *TP/Labs/Laba1* : *md* «ім'я» — створити директорію; *dir* — переглянути вміст директорії; *D*: — змінити диск на «*D*»; *cd* «ім'я» — перейти до відповідної директорії.

♦ Далі, знаходячись у *Laba1* введіть такий код, для створення програми консольного типу. Параметр «*–o*» створює каталог з назвою *Laba1_Part1*, де зберігається ваш додаток, і заповнює його необхідними файлами: *dotnet new console -o Laba1_Part1*.

♦ Зайдіть у щойно створений каталог: *cd Laba1_Part1*. Основним файлом у папці *Laba1_Part1* є *Program.cs*. За замовчуванням він вже містить необхідний код для виводу «*Hello World!*» до консолі. Для перевірки — наберіть команду: *dotnet run.*

♦ Замініть файл *Program.cs*. Для цього треба відкрити його у блокноті чи *Notepad++*, та замінити кодом, наведеним нижче. Запропонована програма для знаходження дискримінанта. Збережіть файл, та знову запустіть його у командному рядку (Рис. 21).

```
using System.Linq;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Discriminant(); //під час запуску викликаємо процедуру, що описана нижче
        }
        static void Discriminant()              //описання процедури
        {
            double a, b, c, x1, x2, D;
            //зчитуємо значення, введені користувачем
            Console.WriteLine("Enter value of a variable A");
            a = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter value of a variable B");
            b = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("Enter value of a variable C");
            c = Convert.ToDouble(Console.ReadLine());
            D = Math.Pow(b, 2) - (4 * a * c);   //знаходимо дискримінант
            Console.WriteLine("Discriminant = " + D);
            if (D > 0)                          //якщо два кореня
            {
                Console.WriteLine("Two roots of the equation: ");
                x1 = (-b + Math.Sqrt(D)) / (2 * a);
                x2 = (-b - Math.Sqrt(D)) / (2 * a);
                Console.WriteLine("X1 = " + x1);
                Console.WriteLine("X2 = " + x2);
            }
            else if (D == 0)                     //якщо один корінь
            {
                Console.WriteLine("One root of the equation: ");
                x1 = -b / (2 * a);
                Console.WriteLine("X = " + x1);
            }
            else                                 //якщо нема коренів
            {
                Console.WriteLine("There are no equation roots!");
            }
            //запит для повторного запуску
            Console.WriteLine("Press - Enter - to solve again.");
            ConsoleKeyInfo k = Console.ReadKey();
            //якщо користувач натиснув кнопку Enter - повторити виклик процедури
            if (k.Key == ConsoleKey.Enter) {
                Console.Clear();
                Discriminant();
            }}}}
```

Рисунок 21 — Програма для знаходження дискримінанта

♦ Знаходячись у каталозі *TP/Labs/Laba1*, створіть другу частину лабораторної роботи, для цього введіть: *dotnet new console -o Laba1_Part2.*

♦ Замініть файл *Program.cs* у новоствореному каталозі *Laba1_Part2*. Запропонована програма для знаходження похідної функції у точці.

f(x)=sin3(x/2); x=(Pi/2); f'(x)=?

Похідна функції — поняття диференційного обчислення, що характеризує швидкість зміни функції в даній точці. Визначається як границя відношення приросту функції до приросту її аргументу при прагненні збільшення аргументу до нуля, якщо така межа існує.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp
{
 class Program
    {
        static void Main(string[] args)
        {
            double x = Math.PI / 2;
            double dx = 1E-12;
            double rate;
            rate = (f(x+dx)-f(x))/dx;
            Console.WriteLine("x={0}, dx={1}, f'(x)={2}", x, dx, rate);
            Console.ReadKey();
        }
         static double f(double x)
        {
            return Math.Pow(Math.Sin(x/2),3);
        }}}
```

Рисунок 22 — Програма для знаходження похідної функції у точці

► **ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ 1:**

1. Зробити перевірку для першої частини лабораторної роботи *Laba1_Part1*, заборонити введення будь-яких символів крім числових.

2. Знайти похідну функції в точці:

2.1 *f(x)* = $(x-5)(2x-5)$, в точці $x=5$

2.2 *f(x)* = $x+\sin x$, в точці x = Pi/2

2.3 $f(x)=x^3+3x^2-72x+90$ в точці $x=5$

2.4 $f(x)=\frac{x-5}{2x-5}$ в точці x = -2

2.5 $f(x)=(2x-1)\sqrt{x}$ в точці x = 4

2.6 $y=3x^4-\frac{1}{\sqrt[3]{x}}$, в точці x = 1

2.7 $y=\sqrt{x}\sin x\cos x$, в точці x = Pi/6

2.8 $y=x^5-4x^3+2x^2-7x$ в точці x = -1

2.9 $f(x)=\frac{x^2}{x+2}$ в точці $x=-4$

2.10 $f(x)=\frac{x^2 arctg5x}{2}-\frac{x}{10}+\frac{1}{50}arctg5x$ в точці $x=\frac{1}{5}$.

3. Побудувати функціональну діаграму, діаграму потоків даних та блок-схему роботи програми.

2.2. ЛАБОРАТОРНА РОБОТА 2

Тема: Створення віконного додатку.

1. Створіть новий проєкт за допомогою *Microsoft Visual Studio*. Виберіть тип проєкту *Visual C#,* додаток *Windows* форми. Задайте назву і вкажіть шлях до робочої директорії проєкту.

2. Вкажіть властивості форми. Виділіть форму. Правою кнопкою миші виберіть в меню «властивості» (*Properties*). У вікні властивостей форм змініть назву форми на «Лабораторна робота № ПІБ», вкажіть розміри (ширина 800, висота 450), та стартове положення «в центрі екрану» (*CenterScreen*).

3. Редагуємо головне меню. Натисніть на вкладку «інструменти» (*ToolBox*). У вікні інструментів виберіть «Меню» -> «Головне меню» і перетягніть іконку на робочу форму. Заповніть поля головного меню як показано на Рис. 23.

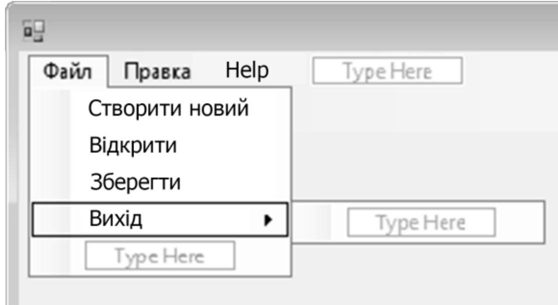


Рисунок 23 — Головне меню

Натискаючи правою кнопкою миші на відповідному полі головного меню (наприклад, «Вихід») з’явиться вікно властивостей елементів головного меню. Додайте можливість закриття форми через $Alt + X$.

4. Програмування вимкнення (виходу) програми. Подвійний клік на порожнє поле поруч з подією *Click*. У вікні з кодом додаємо обробку події закриття програми: *Application.Exit ()*.

5. Розміщення картинки на формі. В інструментах *ToolBox-> CommonControl* виберіть елемент *PictureBox*. В його властивості вкажіть файл із зображенням *Image*. Виберіть файл із зображенням стрижня (файл попередньо намалювати в графічному редакторі), та помістіть його на форму.

6. Створення закладок в правій частині форми. В інструментах *ToolBox->Containers* виберіть елемент *TabControl*. У властивостях *TabControl* задайте розміщення в правій частині форми *Dock-> Right*.

Активуйте першу закладку. Змініть напис на ній (властивість *TabPages*) на «Тип перетину». Властивість *AutoScroll* відзначити як *True*. Помістіть на цю вкладку *Panel* (елемент з *ToolBox-> Containers*) і задайте на ній властивості *BackColor = LightGray; BorderStyle = Fixed3D*; *Cursor = Hand*.

Скопіюйте таким чином *Panel* три рази розмістивши колонкою. Додайте зображення поперечного перерізу балки. Повторіть операції для всіх панелей (Рис. 24).

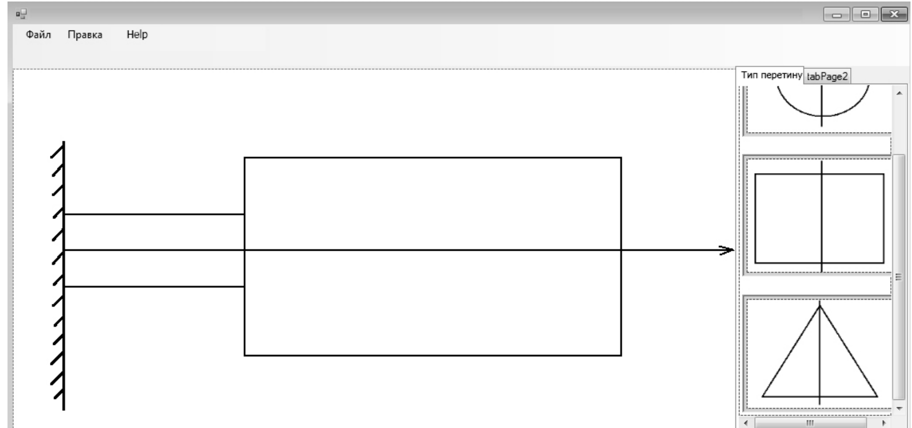


Рисунок 24 — Розташування об’єктів на формі

7. Встановіть мітку і поле введення (*Label*, *textBox*) на другій закладці.

8. Задайте операції виділення панелей кольором. Фону панелі — червоний, якщо на неї натиснути кнопкою миші *panel1.BackColor = Color.Red.*

9. Додайте компонент «*RichText*».

10. Створіть подію для виконання розрахунків жорсткості (Рис. 25), як подію на натискання кнопки «Обчислити» в головному меню.

```
double k;
double R = Convert.ToDouble (textBox1.Text);
double b = Convert.ToDouble (textBox2.Text);
double E1 = Convert.ToDouble (textBox3.Text);
double L1 = Convert.ToDouble (textBox4.Text);
double L2 = Convert.ToDouble (textBox5.Text);

double S = 0;
if (Panel1.BackColor == Color.Red) S = 3.14159265 * R * R;
if (Panel2.BackColor == Color.Red) S = R * b;
if (Panel3.BackColor == Color.Red) S = 1/2 * (R * b);

double k1 = E1 * S / L1;
double k2 = E1 * S / L2;
k = (k1 * k2 / (k1 + k2));

richTextBox1.AppendText ("Жорсткість ділянок дорівнює: \ n");
richTextBox1.AppendText ("\ N");
richTextBox1.AppendText ("K1 =" + Convert.ToString (k1) + "\ N");
richTextBox1.AppendText ("K2 =" + Convert.ToString (k2) + "\ N");
richTextBox1.AppendText ("K =" + Convert.ToString (k) + "\ N");
richTextBox1.AppendText ("\ N");
richTextBox1.AppendText ("-----------------------------------");
richTextBox1.AppendText ("\ N");
richTextBox1.AppendText ("\ N");
```

Рисунок 25 — Програма для виконання розрахунків жорсткості

Результат роботи програми наведено на Рис. 26.

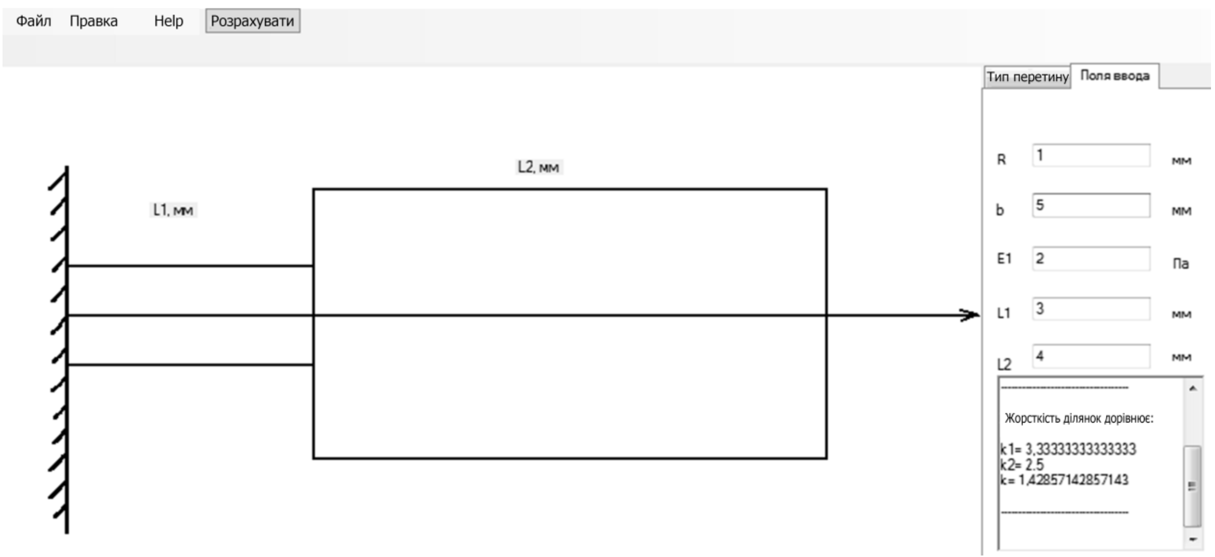


Рисунок 26 — Результат роботи програми

► **Завдання до лабораторної роботи 2:**

1. Перевірити роботоспроможність приведеного коду.

2. Розрахувати жорсткість для випадку 3-х різних стрижнів.

3. Побудувати функціональну діаграму, діаграму потоків даних та блок-схему роботи програми.

2.3. ЛАБОРАТОРНА РОБОТА 3

Тема: Розміщення графіки на формі. Інтерактивне управління.

Завдання. Створити віконний додаток, що відображає балку. Передбачити можливість зміни крайових умов у візуальному режимі. Зробити розрахунок невідомих коефіцієнтів і вивести графік прогину балки при натисканні на кнопку.

1. Створіть віконний додаток *Windows.*

2. Визначте властивості форми (розміри, текст підпису).

3. Відобразьте балку на формі. Опис події *Form1_Paint* (Рис. 27). Всередину цієї функції передається покажчик на об’єкт *e* класу

System.Windows.Forms.PaintEventArgs, як метод, в якому представлені найпростіші інструменти малювання.

```
Graphics gr = CreateGraphics ();
 Pen BlPen = new Pen (Color.Black); // Створити новий об'єкт класу олівець
 BlPen.Width = 8.0F; // Визначаємо його товщину
 gr.DrawLine (BlPen, 100, 100, 500, 100); // Відображаємо лінію (вісь балки)

 BlPen.Width = 2.0F; // Змінюємо товщину

 // Малюємо умовне позначення рівномірно чинного тиску
 for (int i = 0; i <10; i ++)
 {
 gr.DrawLine (BlPen, 120 + i * 400/10, 100, 120 + i * 400/10, 60);
 gr.DrawLine (BlPen, 120 + i * 400/10, 100, 120 + i * 400/10 - 5, 85);
 gr.DrawLine (BlPen, 120 + i * 400/10, 100, 120 + i * 400/10 + 5, 85);
 };
 gr.DrawLine (BlPen, 120, 60, 480, 60);

// Малюємо трикутник позначає «Шарнір» на лівому краю балки
 gr.DrawLine (BlPen, 102, 104, 80, 120);
 gr.DrawLine (BlPen, 80, 120, 120, 120);
 gr.DrawLine (BlPen, 120, 120, 102, 104);
```

Рисунок 27 — Опис події для Form1_Paint

4. Додайте панель на форму і визначте її властивості (колір фону — білий, положення — під лівим краєм балки, край виділити).

5. Розмістіть компонент *checkBox1* і *checkBox2* на панелі, задайте властивості (текст напису — «Шарнів» і «Закладення»; «Шарнір» – виділити).

6. Опис процедури перемикання (вибору) між крайовими умовами, опишіть як подію *CheckedChanged* (Рис. 28).

```
{
if (checkBox2.Checked == true) // Якщо checkBox2 виділений
{
checkBox1.Checked = false; // Відзначити checkBox1 невиділений
}
this.Refresh (); // Оновити вікно форми
Application.DoEvents ();
}
```

Рисунок 28 — Процедура перемикання крайових умов

7. Відкоригуйте подію в подія *Form1_Paint*, що дає можливість враховувати вибір типу крайової умови (Рис. 29).

```
if (checkBox1.Checked == true)
// Малюємо трикутник позначає «Шарнир» на лівому краю балки
{
gr.DrawLine (BlPen, 102, 104, 80, 120);
gr.DrawLine (BlPen, 80, 120, 120, 120);
gr.DrawLine (BlPen, 120, 120, 102, 104);
}
// якщо виділено «Закладення», то змінити відображення на вертикальну лінію
if (checkBox2.Checked == true)
{
gr.DrawLine (BlPen, 100, 70, 100, 130);
}
```

Рисунок 29 — Корективи події Form1_Paint

```
// перевірка на натискання правої кнопки миші
// і потрапляння курсора в це момент в прямокутник обмежує лівий край балки
 {
 if ((e.Button == System.Windows.Forms.MouseButtons.Right) && (eX> 80)
     && (eX <120) && (eY> 100) && (eX <120))
 {
 panel1.Visible = true; // Показати панель
 }
 }
// аналогічно запрограмувати зникнення панелі при натисканні лівою
//кнопкою миші на формі!
```

Рисунок 30 — Процедура зміни крайової умови

8. Відмітьте панель, як невидиму. Напишіть процедуру, що дозволяє візуально (шляхом натискання правою кнопкою миші на відповідному краї), міняти тип крайової умови (подія *Form1_MouseClick*). Всередину цієї функції передайте покажчик на об’єкт *e* класу *System.Windows.Forms.MouseEventArgs* (Рис. 30).

► **ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ 3:**

1. Аналогічно пунктам (4-8) опишіть вибір крайової умови на правому краї.

2. Помістіть знизу форми кнопку і поле введення (*Edit*).

3. Опишіть процедуру визначення констант рішення в залежності від типу крайових умов і відобразьте рішення (прогини балки) тонкою лінією.

4. Побудуйте функціональну діаграму, діаграму потоків даних та блок-схему роботи програми.

2.4. ЛАБОРАТОРНА РОБОТА 4

Тема: Основи *OpenGl*. Побудова 2D геометричних фігур.

OpenGL — це програмний інтерфейс, що складається з окремих команд, які використовуються для вказівки об’єктів і операцій для роботи з тривимірною графікою.

У наступному списку коротко описані основні графічні операції, які виконує OpenGL для виведення зображення на екран:

– конструювання фігур з геометричних примітивів, створюючи математичний опис об’єктів (примітивами в *OpenGL* вважаються точки, лінії, полігони, бітові карти і зображення);

– позиціонування об’єктів у тривимірному просторі та вибір точки спостереження для огляду отриманої композиції;

– обчислення кольору для всіх об’єктів. Кольори можуть бути визначені додатком, отримані з розрахунку умов освітленості, обчислені за допомогою текстур, накладених на об’єкти або з будь-якої комбінації цих факторів;

– перетворення математичного опису об’єктів і асоційованої з ними колірної інформації в пікселі на екрані. Цей процес називається растеризуванням (або растровою розгорткою).

Вершини і примітиви. Побудова всіх базових елементів зводиться до точок. У разі комп’ютерної графіки точками створюються лінії, з яких можуть бути побудовані найпростіші геометричні фігури — полігони. Полігони можуть утворювати собою сітку, яка представляє в тривимірному просторі будь-яку модель. Це область в просторі, яка обмежується однією

замкненою ламаною лінією, причому кожен з відрізків даної ламаної лінії задається за допомогою двох вершин на його кінцях. Однак, описаний таким чином полігон може виявитися дуже складним геометричним об'єктом. Тому не можна, щоб ребра полігонів перетиналися, а також він обов'язково повинен бути опуклим. Для побудови примітивів в *OpenGL* використовуються різні режими візуалізації.

Вершини в *OpenGl* використовують спеціальну команду:
void glVertex [2 3 4] [s i f d] (type coords) та *void glVertex [2 3 4] [s i f d] v (type * coords),* де 2, 3, 4 — це кількість координат, якими задані вершини. У *OpenGL* точки насправді задаються 4-а координатами: *x, y, z, r*, положення точки описується як *x / r, y / r, z / r.* Коли *r* не вказується, воно приймається рівним одиниці. Якщо вказуються точки у двовимірній системі координат, *z* вважається рівним нулю. *i, f, d* — це можливий тип прийнятих значень. Наприклад, якщо написати команду *glVertex3f (,,)* то в дужках повинні бути три змінні типу *float*. Для *d* — double, для *I* — int.

Виклик функції *glVertex* може мати результат тільки між викликами функцій: *glBegin ()* та *glEnd ().*

Координатні осі розташовані так, що точка (0,0) знаходиться в лівому нижньому кутку екрану, вісь *x* спрямована вліво, вісь *y* —вгору, а вісь *z* — з екрану. Вершини в *OpenGL* об'єднуються у примітиви, до яких відносяться точки, лінії, пов'язані або замкнуті лінії, трикутники і так далі. Створення примітиву відбувається всередині командних дужок:

void glBegin (GLenum mode) та *void glEnd (void)*

Параметр *mode* визначає тип примітиву, який задається всередині і може набувати таких значень:

GL_POINTS кожна вершина задає координати деякої точки.

GL_LINES кожна окрема пара вершин визначає відрізок; якщо задано непарне число вершин, то остання вершина ігнорується.

GL_LINE_STRIP кожна наступна вершина задає відрізок разом з

попередньою.

GL_LINE_LOOP останній відрізок визначається останньою і першою вершиною, утворюючи замкнуту ламану.

GL_TRIANGLES кожна окрема трійка вершин визначає трикутник; якщо задано не кратно трьом число вершин, то останні вершини ігноруються.

GL_TRIANGLE_STRIP кожна наступна вершина задає трикутник разом з двома попередніми.

GL_TRIANGLE_FAN трикутники задаються першою і кожною наступною парою вершин (пари не перетинаються).

GL_QUADS кожна окрема четвірка вершин визначає чотирикутник; якщо задано не кратна чотирьом число вершин, то останні вершини ігноруються.

GL_QUAD_STRIP чотирикутник з номером n визначається вершинами з номерами $2n-1$, $2n$, $2n+2$, $2n+1$.

GL_POLYGON послідовно задаються вершини опуклого багатокутника.

Для роботи з *OpenGL* на мові С# у *Microsoft Visual Studio* використовують бібліотеку ***Tao Framework*.** Для роботи з бібліотеками необхідно підключити відповідні простори імен:

using Tao.OpenGl; // для роботи з бібліотекою OpenGL

using Tao.FreeGlut; // для роботи з бібліотекою FreeGLUT

using Tao.Platform.Windows; // для роботи з SimpleOpenGLControl

Для роботи з графічними елементами на форму додається ***SimpleOpenGLControl*.** Основні налаштування об’єкту наведені на Рис. 31.

```
private void Form1_Load (object sender, EventArgs e)
 {
 Glut.glutInit (); // ініціалізація Glut
 Glut.glutInitDisplayMode (Glut.GLUT_RGB | Glut.GLUT_DOUBLE | Glut.GLUT_DEPTH);
Gl.glClearColor (0, 0, 0, 1); // очищення вікна (0,0,0 - чорний; 1,1,1 - білий)
Gl.glViewport (0, 0, simpleOpenGlControl1.Width, simpleOpenGlControl1.Height);
// установка порту виведення відповідно до розмірів елемента
Gl.glMatrixMode (Gl.GL_PROJECTION); // настройка проекції
 Gl.glLoadIdentity (); // настройка параметрів OpenGL для візуалізації
 // коректно налаштовуємо 2D ортогональную проекцію в залежності від розмірів
 // сторін вікна візуалізації
 if ((float) simpleOpenGlControl1.Width <= (float) simpleOpenGlControl1.Height)
 {
 // настройку 2D ортогональної проекції
 Glu.gluOrtho2D (0.0,30.0f * (float) simpleOpenGlControl1.Height /
 (float) simpleOpenGlControl1.Width, 0.0, 30);
 }
 else
 {
 // настройку 2D ортогональної проекції
 Glu.gluOrtho2D (0.0,30.0f * (float) simpleOpenGlControl1.Width /
 (float) simpleOpenGlControl1.Height, 0.0, 30);
 // Glu.gluOrtho2D поміщає початок координат в лівий нижній квадрат,
 // а спостерігач в цьому випадку перебувати на осі Z
 }
 Gl.glMatrixMode (Gl.GL_MODELVIEW);
 Gl.glLoadIdentity ();
 }
```

Рисунок 31 — Налаштування SimpleOpenGLControl

Як параметри *Glu.gluOrtho2D* передаються координати області видимості вікном проєкції: *left, right, bottom* і *top.* У даному випадку, для виключення спотворення, пов’язані з тим, що область виводу не квадратна, параметр *top* розраховується як добуток 30 і відношення висоти елемента *simpleOpenGlControl1* (в якому буде йти візуалізація) до його ширини. І навпаки.

Для *Form1()* задається функція ініціалізації:

{ InitializeComponent (); simpleOpenGlControl1.InitializeContexts (); }

► Завдання до лабораторної роботи 4:

1. Придумати і реалізувати побудову двомірної фігури як подію натискання на кнопку.

2. Побудувати функціональну діаграму, діаграму потоків даних та блок-схему роботи програми.

2.5. ЛАБОРАТОРНА РОБОТА 5

Тема: Створення примітивів, вибір кольору, трансформація, перенесення, масштабування 3 D об'єктів в OpenGL.

1. Створіть віконний додаток *Windows*.

2. Визначте властивості форми.

3. Розташуйте на формі такі компоненти: *Label, trackBar, comboBox, simpleOpenGlControl, checkbox, timer*.

Приклад розміщення компонент наведено на Рис. 32.

Рисунок 32 — Розміщення компонентів на формі

4. Для *Form1()* задається функція ініціалізації:

{ InitializeComponent (); simpleOpenGlControl1.InitializeContexts (); }

5. Основні налаштування об'єкту SimpleOpenGLControl для роботи з графічними елементами наведені на Рис. 33.

```
private void Form1_Load (object sender, EventArgs e)
{
Glut.glutInit ();
// ініціалізація бібліотеки Glut
Glut.glutInitDisplayMode (Glut.GLUT_RGB | Glut.GLUT_DOUBLE);
// ініціалізація режиму екрану
Gl.glClearColor (1, 1, 1, 1);
// установка кольору очищення екрана (RGBA)
Gl.glViewport (0, 0, simpleOpenGlControl1.Width, simpleOpenGlControl1.Height);
// установка порту виведення
Gl.glMatrixMode (Gl.GL_PROJECTION);
// активація проекційної матриці
Gl.glLoadIdentity (); // очищення матриці
// установка перспективи
Glu.gluPerspective (45, (float) simpleOpenGlControl1.Width /
(float) simpleOpenGlControl1.Height, 0.1, 200);
Gl.glMatrixMode (Gl.GL_MODELVIEW);
Gl.glLoadIdentity ();
// початкова настройка параметрів openGL (тест глибини, перше джерело світла)
Gl.glEnable (Gl.GL_DEPTH_TEST);
//Gl.glEnable (Gl.GL_LIGHTING);
Gl.glEnable (Gl.GL_LIGHT0);
```

Рисунок 33 — Налаштування об’єкту SimpleOpenGLControl

```
private void Draw () //
 {
 Gl.glClear (Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
 // очищення буфера кольору і буфера глибини
 Gl.glClearColor (1, 1, 1, 1); // очищення поточної матриці
 Gl.glLoadIdentity (); // поміщаємо стан матриці в стек матриць,
// подальші трансформації торкнуться тільки візуалізацію об'єкту
 Gl.glPushMatrix ();
 Gl.glTranslated (a, b, c); // переміщення
 Gl.glRotated (d, os_x, os_y, os_z); // поворот за встановленою осі
 Gl.glScaled (zoom, zoom, zoom); // масштабування
```

Рисунок 34 — Налаштування об’єкту SimpleOpenGLControl

На Рис. 34 наведена функція відрисовки, де *Gl.glTranslated* відповідає за переміщення об’єкту по осях, *Gl.glRotated* — повороти за встановленою віссю, *Gl.glScaled* — зміна розміру моделі. Далі наведені стандартні об’єкти у сітковій та полігональній формах:

Gl.glColor3f ((float) 0.0, (float) 0.0, (float) 0.0) — встановленя кольору.

Glut.glutWireSphere (1, 32, 32) — сіткова сфера.

Glut.glutSolidSphere (1, 32, 32) — полигональная сфера.

Glut.glutWireCylinder (0.7, 1.2, 32, 32) — сітковий циліндр.

Glut.glutSolidCylinder (0.7, 1.2, 32, 32) — полігональний циліндр.

Glut.glutWireCube (0.7) — сітковий куб.

Glut.glutSolidCube (0.7) — полігональний куб.

Glut.glutWireCone (0.7, 1.2, 32, 32) — сітковий конус.

Glut.glutSolidCone (0.7, 1.2, 32, 32) — полігональний конус.

Glut.glutWireTorus (0.2, 0.8, 32, 32) — сітковий тор.

Glut.glutSolidTorus (0.2, 0.8, 32, 32) — полігональний тор.

Пістя завершення побудови фігур необхідно повернути стан матриці *Gl.glPopMatrix (),* завершити малювання *Gl.glFlush (),* та оновити елемент *simpleOpenGlControl1.Invalidate ().*

► **ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ 5:**

1. Провести побудову стандартних тривимірних фігур.

2. Реалізувати побудову власної тривимірної фігури.

3. Додати можливість масштабування, повороту та переміщення моделі.

4. Побудувати функціональну діаграму, діаграму потоків даних та блок-схему роботи програми.

3. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Поняття «технології програмування» забезпечення технологічності складних програмних систем.

2. Основні етапи розвитку технології програмування.

3. Життєвий цикл і етапи розробки програмного забезпечення. еволюція моделей.

4. Поняття технологічності ПЗ.

5. Модулі та їх властивості.

6. Структурне програмування. Засоби опису структурних алгоритмів.

7. Ефективність і технологічність ПЗ.

8. Основні експлуатаційні вимоги до програмних продуктів.

9. Вимоги та визначення специфікацій ПЗ при структурному підході.

10. Розробка технічного завдання.

11. Поняття блок-схеми. Інші види опису алгоритмів.

12. Поняття специфікації ПЗ при структурному підході.

13. Діаграми переходу станів.

14. Функціональні діаграми.

15. Діаграми потоку даних.

16. Діаграми відносин компонентів даних.

17. Розробка користувальницьких інтерфейсів.

18. Типи користувальницьких інтерфейсів.

19. Класифікація діалогів.

20. Основні компоненти графічних користувальницьких інтерфейсів.

21. Інтелектуальні елементи призначених для користувача інтерфейсів.

22. Інтерфейси прямого маніпулювання і їх проєктування.

23. Особливості суб'єктивного сприйняття користувачем інформації відтворюється програмним продуктом.

24. Тестування та налагодження ПЗ.

25. Ручний контроль ПЗ.

26. Класифікація помилок.

27. Методи налагодження ПЗ.

28. Структурне тестування.

29. Функціональне тестування.

30. Комплексне тестування.

ЗМІСТ

Вступ …………………………………………………………….….. 1

1. Короткі теоретичні відомості …………………………………… 1

1.1. Основні поняття технології програмування. Історія розвитку ……………………………...…………...…… 1

1.2. Життєвий цикл програмного забезпечення……………..... 3

1.3 Структурний підхід. Аналіз і визначення специфікацій … 10

2. Індивідуальні лабораторні роботи ………………………………. 30

2.1. Лабораторна робота 1 30

2.2. Лабораторна робота 2 33

2.3. Лабораторна робота 3 36

2.4. Лабораторна робота 4 39

2.5. Лабораторна робота 5 43

3. Контрольні запитання ……………………………………………... 45

Навчальне видання

СТРУКТУРНЕ ПРОГРАМУВАННЯ

Методичні вказівки до виконання індивідуальних завдань
з курсу «Технологія програмування»
для студентів всіх рівнів та форм навчання спеціальностей
113 «Прикладна математика» та
122 «Комп'ютерні науки»

Укладачі: ЛАРІН Олексій Олександрович
ШАПОВАЛОВА Марія Ігорівна

Відповідальний за випуск доц. Водка О. О.
Роботу до видання рекомендував проф. Бреславський Д. В.

В авторській редакції

План 2020 р., поз. 98

Самостійне електронне видання