МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний авіаційний університет
М. О. Сидоров ВСТУП ДО ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
Курс лекцій
Київ
Видавництво Національного авіаційного університету «НАУ-друк» 2010
УДК 004.4(042.4)
ББК з 973.20-018.2я7 С 347
Затверджено методично-редакційною радою Національного авіаційного університету (протокол № 14 від 03.07.2008р.).

Сидоров М. О. Вступ до інженерії програмного забезпечення : курс лекцій / М.О.Сидоров. - К.: Вид-во Нац. авіац. ун-ту «НАУ-друк», 2010. -112 с.
ISBN 978-966-598-626-3
У курсі лекцій викладено основні положення інженерії програмного забезпечення. Для студентів напряму 6.050103 "Програмна інженерія".

ВСТУП

Видання містить матеріал лекцій з дисципліни «Вступ до програмної інженерії», яка викладається в бакалавраті «Програмна інженерія».

Матеріал книги є базовим під час вивчення інших дисциплін бакалаврату «Програмна інженерія».

Курс лекцій складається з двох, модулів та семи розділів, у яких викладено основні принципи, методи та засоби інженерії програмного забезпечення.

У першому розділі розглянуто умови виникнення інженерії програмного забезпечення та її місце в контексті інших інженерій.

У другому розділі висвітлено культуру інженерії програмного забезпечення і моделі зрілості як засоби реалізації культури.

У третьому розділі викладено результати інженерної діяльності - програми, програмні продукти і системи.

У четвертому розділі наведено складові життєвого циклу програмного забезпечення.

У п'ятому розділі розглянуто складові інженерії програмного забезпечення; їх застосування і зв'язок.

У шостому розділі подано моделі життєвого циклу програмного забезпечення.

У сьомому розділі викладено окремі матеріали з менеджменту проектів, а саме - оцінка вартості програмного забезпечення.

Курс лекцій може застосовуватися під час вивчення дисципліни «Методологія розробки великих програмних комплексів і систем», яка викладаєтеся в бакалавраті «Комп'ютерні науки».

Модуль І
ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ОСНОВНІ ПОНЯТТЯ

Розділ 1. ІСТОРИЧНИЙ АСПЕКТ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Незважаючи на відносно недовгий період, що пройшов з моменту появи інженерії програмного забезпечення (1968), вона має свою історію.

У розділі 1 розглянуто процеси, діячі і події, які призвели до по- яви інженерії програмного забезпечення. Показано, що інженерія програмного забезпечення — це одна з Інженерних дисциплін. Наведено визначення інженерії програмного забезпечення. Розглянуто професійні властивості інженера з програмного забезпечення,

1.1. Умови виникнення інженерії програмного забезпечення

Як відомо, перші комп'ютери, що працювали під управлінням Програм**,** які зберігаються в пам'яті, з'явилися у 40 - 50-х рр. XX ст. Разом з ними постало нове завдання, суть якого полягає у створенні програм, і процес, спрямований на її вирішення - програмування**.** Тому подальший розвиток обчислювальної техніки пов'язаний не тільки з удосконаленням комп'ютерів і їх розповсюдженням, але й з розвитком програмування.

Майже одночасно з появою комп'ютерів відбулося розділення розробників програм на два типи - прикладних і системних програмістів.

До першого типу (прикладні програмісти) увійшли фахівці з прикладних галузей (доменів) - математики, фізики, економіки, освіти**;** технологій. Вони писали програми мовами високого рівня *(Cobol, Fortran)* для вирішення тих завдань, що виникають у галузях. Їх діяльність називалася прикладним програмуванням.

До другого типу (системні програмісти) увійшли фахівці, від яких не вимагалося знань доменів, оскільки вони займалися автоматизацією процесів розробки програм. Системні програмісти за- звичай писали програми в машинному коді або мовою АСЕМБЛЕР. Їх діяльність називалася системним програмуванням. Сукупність прикладних і системних програм називається програмним забезпеченням.

У 60 - 70-х рр. XX ст. були створені високопродуктивні обчислювальні машини (швидкістю близько 1 млн., опер./с БЕСМ-6 в СРСР і UNIVAC в США). З їх появою виникла можливість вирішення великих і складних завдань. Це, своєю чергою, потребувало розробки великих програм (від 100 тис. до 1 млн. рядків). Великі програми спричинили проблеми, пов'язані з їх створенням і вико- рисканням.

Із збільшенням продуктивності, кількості обчислювальних машин і розширенням сфери їх застосування, з'явилися програми двох типів. Програми першого типу створювалися і продавалися разом з машинами (транслятори, операційні системи, бібліотеки підпрограм). Програми другого типу створювалися на замовлення і призначалися для вирішення завдань з різних предметних галузей. Таким чином, з'явився замовник - організація, яка ставила завдання, призначала терміни, виділяла бюджет і оплачувала роботу. У зв'язку з цим дуже швидко з'явилось завдання - супровід Про- грами і проблема - непорозуміння між розробником і замовником.

У 60-х рр. XX ст. унаслідок розповсюдження застосування комп'ютерів, зросла роль і утвердилась важливість програмного забезпечення. Цьому сприяла поява значної кількості проектів програмного забезпечення, що характеризувалися такими аспектами:

- наявністю замовника або ринкової ніші;
- великими розмірами і витратами;
- жорсткими вимогами до процесів реалізації і результатів;
- мілітаризацією.

Контекст, у якому розроблялося і використовувалося програмне забезпечення, сприяв особливому стану програмного забезпечення в обчислювальній техніці і характеризувався такими чинниками:

- розроблялося дуже велике за обсягом програмне забезпечення, характерним представником якою була тоді операційна система OS360 для ЕОМ серії IBM. Це програмне забезпечення містило більше 500 тис. операторів і розроблялося значним, для того часу,

колективом розробників (близько 1000). Досвід цього проекту був узагальнений і став відомий;

- програмне забезпечення вирішувало настільки серйозні зав- дання, що виникла проблема з його супроводом, суть якого зводилася не тільки до виправлення помилок, допущених при розробці, але й до модифікації програмного забезпечення у зв'язку із зміною вимог замовника або середовища, у якому експлуатувалося програмне забезпечення, або бажанням розробника продовжити експлуатацію, шляхом випуску вдосконалених версій;

- часті зриви термінів розробки і перевищення бюджету потре- бували не тільки нового підходу до організації процесу розробки, але і нових методів і засобів, що забезпечують обґрунтований розрахунок параметрів проекту, які характеризували фінансування, терміни, об'єми програмного забезпечення, кількісний і якісний склад колективу розробників. Існуюча і широко використовувана одиниця вимірювання - «людино-місяць» не працювала на таких масштабних проектах;

- досвід розробки програмного забезпечення, який нагромаджений за цей період, показав, що все рідше розроблялися принципово нові проекти. Лише 15% усіх проектів потребували розробки «з нуля». Нині 85% належать до проектів, що повторюються. Тому актуальним стає використання досвіду, нагромадженого в програмному забезпеченні До того ж, поступово стало зрозуміло, що має вирішуватися завдання використання досвіду не лише безпосередньо програмування, у вигляді частин програм, але і досвіду результатів виконання інших процесів, наприклад, проектування. Для вирішення цього завдання в 1984 р. були широко розгорнені роботи з дослідження програмного забезпечення в аспекті повторного використання *(reuse);*

- техніка програмування і процеси, що були ефективні в 50-х і ранніх 60-х рр. XX ст. («програмування в малому») для розробки невеликих програм малими колективами, стали неефективними при розробці великого за обсягом, складного програмного забезпечення, що складається з мільйонів рядків коду, та вимагає декількох років роботи сотень фахівців різних спеціальностей. Були потрібні нові технології, що почали вважати «програмуванням у великому».

Почали з'являтися нові процеси, що потребували певної організації (табл.1.1). Таким чином, склалася ситуація, яка призвела до кризи в програмному забезпеченні і необхідності пошуку шляхів виходу з неї, так званої «срібної кулі». Виходом з цієї ситуації стало обговорення на конференції NATO в 1968 р. нової дисципліни, яку назвали «інженерія програмного забезпечення» *(software engineering).* До того ж, уперше акценти в методах, засобах і процесах розробки програмного забезпечення були зміщені: по-перше, з кодування програм на інші процеси їх розробки, а по-друге, з якісних аспектів у бік кількісних, інженерних. Окрім цього, додатковий стимул отримали ті, хто виконував наукові і практичні роботи з економічного напряму і менеджменту проектів програмного забезпечення.

Таблиця 1.1

1.2. Інженерія програмного забезпечення - інженерна галузь

Інженерна галузь характеризується діяльністю, що ґрунтується на таких принципах**:**

- ефективність - результати отримують за допомогою заданих ресурсів, які відповідають висунутим вимогам і стандартам;
- практичність - результати мають конкретних замовників;
- фундаментальність - результати отримують на основі знань фундаментальних наук;
- успадкованість - результати отримують на основі нагромад-женого досвіду, виключаючи діяльність «з нуля»;
- відчутність - результати є відчутними продуктами, які мож-на застосовувати, руйнувати, а також досліджувати за допомогою емпіричних методів пізнання;
- супроводження - результати, знаходячись в експлуатації, обов'язково

супроводжуються (обслуговуються),

У процесі розвитку людства з'явилося багато інженерних галу-зей, але їх становлення проходило один і той же шлях, у якому роз-різняють три фази (рис. 1.1),

Рис. 1.1. Фази розвитку інженерної галузі

У кожній фазі мають місце виконавці, ресурси, методи реаліза-ції і використання продуктів галузі. Для фаз характерне:

- фаза I: виконавці - віртуози і талановиті одинаки; ресурси - інтуїція і груба сила; методи - випадкова передача досвіду, екстра-вагантне застосування матеріалі»; використання - виробництво для себе;

- фаза II: виконавці - майстерні-виробники; ресурси - окремі інструменти; методи - механічний тренінг, облік економічних чин-ників у виборі матеріалів; використання - виробництво для прода-жу, утворення ринку;

- фаза III: виконавці - досвідчені професіонали; ресурси - машини і комплекси, що використовуються в технологіях**;** методи - теоретичні і емпіричні, передавання знані, шляхом диференційова-ного навчання, супровід; використання - сегментація ринку.

Для інженерії програмного забезпечення характеристика зазначених фаз наводиться в табл. 1.2.

Інженерна діяльність реалізується інженерами в Контексті технологій. Технологія - це організована сукупність процесів, спрямованих на отримання з початкових матеріалів кінцевих продуктів за допомогою методів і засобів технологій. Інженери - це професіонали, чия освіта дозволяє їм, використовуючи знання фундаментальних наук і конкретних технологій, реалізовувати процеси, застосовуючи методи і засоби технологій для створення надійних, широко використовуваних продуктів.

Таблиця 1.2

Нині інженерія програмного забезпечення - це систематизований, регламентований і кількісний (інженерний) підхід до вирішення завдань розробки, експлуатації, супроводу й утилізації програмного забезпечення. До того ж, процеси і програмне забезпечення мають відповідати заданим технічним, економічним, соціальним і правовим вимогам.

Технічні вимоги обов'язково відображають відповідність процесів і продуктів життєвого циклу вимогам, специфікованим замовником.

Економічні вимоги обов'язково містять вимоги щодо виконання проекту в рамках заданого фінансового бюджету.

Соціальні вимоги обов'язково відображають те, що створювані Програмні продукти повинні мати властивості корисності.

Правові вимоги обов'язково відображають те, що виконання програмного проекту повинне здійснюватися законними методами. Особливо це важливо, коли під час розробки застосовується успадковане програмне забезпечення або компоненти багаторазового використання.

Як і інші інженерні дисципліни, інженерія програмного забезпечення характеризується такими аспектами:

- творчість - інженерія концентрується на проблемах аналізу і проектування;

- інструментальність ключові проблеми в інженерії - це ви- бір і використання інструментів;

- стандартизація - кращі практичні досягнення інженерії у вигляді інженерних принципів є основою створення стандартів;

- успадкованість (повторне використання) - в інженерії повтор- не використання знань і продуктів фаз життєвого циклу с найважливішим чинником підвищення продуктивності і якості;

- професіоналізм - інженерія програмного забезпечення - це професія.

Остання властивість характеризує інженерію програмного забезпечення не як академічну, а швидше як практичну дисципліну. Професійний інженер з програмного забезпечення мас такі риси:

- ухвалює рішення, оцінюючи стан і кожного разу вибираючи підходи для вирішення конкретних завдань і в конкретному контексти знаходячи баланс між витратами і прибутком;
- вимірює, калібрує і оцінює вимірювальні інструменти;
- виконує в практичній діяльності одну з багатьох ролей, на- приклад, дослідник, розробник, архітектор, виробник, тестер, експлуатаційник, керівник, продавець, консультант, викладач;
- результати праці інженера можуть бути різні, від пристроїв і систем, до процесів і структур;
- застосовує знання з інших дисциплін (на додаток до своїх власних), наприклад, з математики, базових наук і економіки. При цьому; основними дисциплінами **є** «Комп'ютерні науки», «Дискретна математика» і «Групова динаміка». Дисципліни «Фізика» і «Без- перервна математика» використовуються в деяких застосуваннях, але вони менш важливі, ніж вказані;
- створює: інструменти;
- працює дисциплінованим і систематичним чином;
- працює в колективах разом з іншими фахівцями, розвиваючи навички взаємин колективної роботи;
- дотримується стичних і професійних принципів, захищаючи суспільство, замовників і себе;
- постійно поповнює свої знання, освоюючи нові методи, техніку і технології;
- спирається на специфічні знання і досвід, працюючи всередині специфічних доменів, враховуючи особливість доменних рішень, виробництва, матеріалів, вимог;
- уміє визначати, які частини можна повторно використовувати, а які слід розробляти заново.

Розділ 2. КУЛЬТУРНИЙ АСПЕКТ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Культура - безліч цінностей, цілей і принципів, які керують діями, пріоритетами і рішеннями окремих осіб або групи, що працюють у напрямі спільної мети. Культура групи розробників програмного продукту дуже сильно впливає на якість продукту, продуктивність розробників і мікроклімату в групі.

2.1. Культура інженерії програмного забезпечення

Культура інженерії програмного забезпечення - це сукупність позицій розробників, людських відносин і технічних процесів, орієнтованих на якість у широкому сенсі. Кожна організація і особа має власну культуру, набуту шляхом копіткої роботи. Культура інженерії програмного забезпечення характеризується такими чинниками:

- чіткими організаційними цілями;
- зобов'язанням менеджменту вести організацію до досягнення встановлених цілей;
- середовищем, яке дає змогу кожному розробникові вдосконалювати і ефективно застосовувати свої знання і навички;
- вимірюваннями, що дають змогу добирати ефективні процеси. Будь-яка «здорова» культура повинна містити три істотні компоненти:
- персональне зобов'язання кожного розробника створювати якісні продукти шляхом систематичного застосування передового досвіду інженерії програмного забезпечення;
- зобов'язання менеджерів усіх, рівнів забезпечувати середовище, в якому якість програмного забезпечення (у всіх його аспектах) є фундаментальною концепцією і кожен

розробник може реалізовувати цю концепцію;

- зобов'язаній всіх членів організації постійно вдосконалювати процеси, в яких воли беруть участь і продукти, які вони створюють.

На рис. 2.1 показано, як культура інженерії програмного забезпечення пов'язана з цілями, пріоритетами і технічною практикою розробки програмного забезпечення [25]. Культура інженерії програмного забезпечення визначає якісний рівень дій, процесів і технічних можливостей організації, задає проектні цілі і можливості організації. Рівень проектних цілей залежить від того технічного досвіду, який нагромоджено в організації. Чим більший цей досвід, тим вищі проектні цілі і тим краще відповідне програмне забезпечення здатна створювати організація. Технічний досвід, очевидно, впливає на загальну культуру організації і чим він більший, тим вища культура (А). Культура організації є, з одного боку, основою для діяльності кожного розробника, а з іншого - культура організацій буде тим вища, чим вища культура кожного розробника (В). Нарешті, культура інженерії програмного забезпечення допомагає встановлювати пріоритети менеджменту. Очевидно, вищі пріоритети менеджменту визначають (підсилюють) загальну культуру організації (С).

Рис. 2.1. Зв'язок культури з організацією

Сьогодні відомо декілька моделей культур організацій. Найвідоміші з них: Константіноса та Де Грака. Модель Константіноса розглядає чотири організаційні парадигми:

- закрита - характеризується стабільністю, секретністю, незначною гнучкістю, низхідним процесом прийняття рішень, неінноваційністю, авторитарністю;
- відкрита - характеризується інноваційністю, співпрацею, переговорністю, адаптивністю, колективним ухваленням рішень, ви- рішенням проблем;
- синхронна - характеризується гармонійністю, рівністю, ефективністю, консерватизмом у змінах, координативністю, непрактичністю керівництва;
- випадкова - характеризується незалежністю ініціатив, творчістю і винахідливістю, індивідуальністю, нестабільністю, автономністю, неформальністю.

Модель Де Грака розглядає дві організаційні парадигми, які називаються римським і грецьким шляхами розвитку організацій (табл. 2.1), Де Грак вказує, що римський шлях характерний для старих, крупних і консервативних організацій, а грецький - для нових, середніх і малих мобільних організацій (ательє).

Таблиця 2.1

Римський шиях	Грецький шлях
— використовують структурні методи;	— використовують об'ектно-оріентовані методи;
— застосовують більше формалізмів;	— Застосовують менше формалізмів;
—Створюють крупкі програми; - застосовують *CASE*- інструменти;	— створюють малі програми;
— проекти повністю керовані;	— застосовують окремі інструменти;
— створюють максимум документації;	— проекти частково керовані;
— налаштовані управляти проектами	— створюють мінімум документації;
	— налаштовані писати програми

Р. Гласс *(R.Gluss)* додає «варварську» культуру для найменш цивілізованих, малих програмних ательє, порівнюючи її з грецькою і римською культурами таким чином:

- греки організовують речі, римляни - людей, варвари організовують «що-небудь»;
- у греків методології неформальні, у римлян - формальні, у варварів - відсутні;
- греки пишуть програми, римляни управляють проектами, варвари «стрибають» кодуючи;
- греки мотивуються «підручною» проблемою, римляни - груповими цілями, варвари -

героями;

- греки мінімізують документацію, римляни максимізують її, варвари гордяться її відсутністю;
- греки працюють у маленьких групах, римляни - у великих, варвари - поодинці;
- греки використовують речі як інструменти, римляни - людей, варвари взагалі не використовують інструментів;
- греки - демократи, римляни - імперіалісти, варвари - анархісти;
- греки - емпірично-індуктивні, римляни - аналітично-дедуктивні, варвари - емоційні;
- греки - інтуїтивні, римляни - логіки, варвари - імпульсні;
- греки надають увагу субстанції, римляни - формі, варвари - лініям коду;
- греки роблять речі, римляни планують речі, варвари руйнують речі.

Нині для оцінювання культури персоналу розроблено кодекс стики інженера програмного забезпечення, а для оцінювання рівня зрілості культури організацій використовується широко відома модель, розроблена в *SEI. Capability Maturity Model* (CMM) і її подаль- ший розвиток CMM1 *(Integrity).*

2.2. Моделі зрілості процесів, що відбуваються на підприємстві

Процес програмного забезпечення - це фундаментальний компонент культури організації. Зрілість процесу програмного забезпечених визначається як ступінь, за якого можна стверджувати, що процес явно визначуваний, керований, вимірюваний, контрольований і ефективний.

На сьогодні відомо декілька моделей зрілості процесів, що відбуваються на підприємстві, що розробляє програмне забезпечення.

2.2.1. Моделі зрілості можливостей (CMM)

До моделей CMM належать три: CMM - модель зрілості підприємств, P-CMM - модель зрілості персоналу, CMM1 - інтегрована модель зрілості підприємств.

На рис. 2.2 показано три шкали, що описують зрілість організацій у виробництві якісного програмного забезпечення.

Рис. 2.2. Шкала зрілості

На шкалі *Humphrey* була побудована модель CMM. Вона була розроблена як механізм для вибору субпідрядників, що виконують проекти Міністерства оборони США. Зараз ця модель використовується для аналогічних цілей ряду компаній.

Модель CMM забезпечує засоби для оцінювання можливостей організації створювати якісне програмне забезпечення. Окрім цього модель надає рекомендації, які повинні бути реалізовані в організації, щоб підвищити її можливості для виробництва якісного програмного забезпечення.

На рис. 2.3 показано схему моделі, яка визначає п'ять рівнів зрілості організації: початковий, повторюваний, визначуваний, керований, оптимізований. Кожен рівень (окрім першого) має безліч процесів (ключові області), що асоціюються з ним. Ці процеси визначають цілі, які повинні бути досягнуті організацією, щоб її зрілість відповідала певному рівню. Кожна мета досягається конкретними діями процесу. Організація може не виконувати всіх дій про-цесу, але вона повинна продемонструвати, що всі цілі ключових областей досягнуті на даному і всіх попередніх рівнях. Пропуск рівнів, що знаходяться нижче в шкалі CMM, не допускається, оскіль-ки області нижніх рівнів забезпечують цілі в галузях вищих рівнів,

Рис. 2.3. Модель СМ SEI

Початковий рівень - зрілість організації ґрунтується на фольк-лорі та індивідуальних здібностях персоналу. Досвід передається тільки шляхом тренінгу, стандарти не враховуються; розмір, витра-ти й інші характеристики проекту не оцінюються. Успіх проекту залежить від менеджера і окремих програмістів («героїв»). Менеджер зазвичай не бачить, що відбувається всередині проекту, Харак-теристики цього рівня такі: хаос, недисциплінованість, непередбачуваність, анархія. 11а цьому рівні немає областей ключових процесів. Робота буде зроблена, коли вона буде зроблена.

Повторюваний рівень - організація використовує проектний менеджмент і управління, документуючи процеси в різних аспектах програмного проекту. Метрики застосовуються, щоб прослідкувати прогрес у виконанні проекту та ідентифікувати проблеми. Передбачається, що організація може повторювати проекти. На цьому рівні розглядаються такі області ключових процесів:

- управління вимогами - спрямовано так: на управління змінами у вимогах шляхом реалізації відповідних планів і дій;
- планування проекту - зорієнтовано на оцінювання і планування проекту;
- стеження і контроль проекту - результати виконання проекту повніші бути описані і внесені до планів проекту. Якщо мають міс-це відхилення, то вони мають бути виправлені;
- управління субпідрядниками - забезпечує керування і регулювання відносин з субпідрядниками;
- гарантування якості - забезпечує аудит і контроль якості, особлива увага приділяється застосуванню стандартів;
- управління конфігурацією програмного забезпечення - продукти фаз життєвого циклу контролюються за допомогою плану управління конфігурацією. Ідентифікуються процедури необхідних змін,

Визначуваний рівень - організація використовує стандартний документований процес для розробки і супроводу програмного забезпечення. Він може налаштовуватися на конкретні проекти, створюються групи для координації процесів. На цьому рівні розглядаються такі області ключових процесів:

- організація процесу - створюються групи для координації про-цесів розробки і дій, спрямованих на підвищення їх ефективності;
- визначення процесу - організація розробляє стандартний про-цес і організовує збір та аналіз інформації, пов'язаної з використанням стандартного процесу для окремих проектів;
- навчальні програми - організація повинна мати цілі, плани і програми з навчання персоналу для реалізації проектів на третьому рівні;
- управління інтеграцією - організація забезпечує вибір і налаштування процесів для конкретного проекту зі стандартного процесу;
- інженерія продуктів - організація визначає інструменти і методи для виконання процесів фаз життєвого циклу, пов'язаних з аналізом і проектуванням, кодуванням, тестуванням, документу-ванням**;**
- міжгрупова координація - дії груп, які координують процеси розробки**,** повинні ідентифікуватися, відстежуватися, а проблеми, що витікають, - вирішуватися;
- експертні оцінки - використовуються експертні оцінки для ідентифікації дефектів, що мають місце в продуктах.

Керований рівень - організація використовує збирані метрики для прогнозу і управління якістю продукту.

- На цьому рівні розглядаються такі області ключових процесів:
- управління процесом на основі кількісних показників - ство-рення плану з використанням метрик, який організація застосовує для розуміння і управління процесом;
- управління якістю продукту - визначаються показники якості продукту, з їх допомогою здійснюються кваліфікація і управління якістю.

Оптимізований рівень - організація зосереджує увагу на двох метапроцесах: попередження дефектів і безперервне вдосконалення якості і продуктивності процесів створення програмного забезпе-чення. Витрати на ці процеси закладені до бюджету і планів дій організації. На цьому рівні розглядаються такі області ключових процесів:

- запобігання дефектам - організація створює і реалізує плани, що забезпечують ідентифікацію, аналіз і усунення причин виник-нення дефектів у продуктах;
- управління змінами технології - вивчаються технології, вико-ристовувані в інших організаціях і нові технології, що забезпечать підвищення можливостей організації у напрямі створення якісних продуктів і зменшення тривалості циклу;
- управління змінами процесу створення програмного забезпечення - безперервне вдосконалення процесу.

Певною мірою організації можуть претендувати на п'ятий рівень СММ. Наприклад, цього рівня досягла *IBM FSC,* коли вона виконувала проект програмного забезпечення для *Space Shuttle.* Це програмне забезпечення мало бути високої якості, без дефектів. Для кожного дефекту, знайденого в продукті, викопувалися такі кроки: визначення причин появи дефекту і їх усунення; розуміння корекція аспекту процесу, що призвів до дефекту; коригування дій, що забезпечують управління якістю, які допустили дефект; перевірка продуктів з подібним дефектом, що дефекти могли бути про- пущені через контроль.

Таким чином, суть СММ полягає в тому, що всі дії з розробки і супроводу програмного забезпечення плануються, контролюються і систематично поліпшуються.

2.2.2. Модель P-CMM

Для оцінювання рівня зрілості персоналу використовуються різні моделі. Найпоширенішою моделлю розвитку персоналу є *People-Capability Maturity Model* (P-CMM), що розроблена *SEI.* Також, як і СММ, ця модель складається з п'яти рівнів зрілості. Кожен рівень містить області ключових процесів, які вказують, що слід виконати на певному рівні, щоб досягти зрілості персоналу відповідного рівня моделі. Зміст рівнів моделі такий:

- початковий - в організації немає процедур управління персоналом і перевірки правильності використання персоналу, якість робіт і графік виконання непередбачуваний;
- повторюваний - в організації складаються процедури управ-ління, що можуть повторюватися під час виконання нового проекту, організація проводить навчання персоналу, керує продуктивністю; на робочих місцях є комунікації і є відповідне середовище розробки. Проте успіх проекту залежить від діяльності окремих менеджерів і «героїв»-виконавців;
- визначуваний - в організації є стандартний процес і тому вводяться відповідні організаційні форми, що гарантують виконання стандартного процесу; значна увага приділяється підвищенню кваліфікації, аналізу навичок, знань і прийомів роботи персоналу; вводяться норми, котрі регулюють культуру сумісних дій персоналу;
- керований - основна увага приділяється підвищенню керованості організації; формується ролева структура персоналу, що забезпечує вирівнювання інтенсивності праці організації;
- оптимізований - головні завдання організації цього рівня зрі-лості - безперервне вдосконалення персоналу, систематичне навчання і підвищення кваліфікації; витрати на вирішення цих завдань закладаються до бюджету організації, а конкретні заходи прописуються в плані діяльності.

Модель P-CMM узгоджена з моделлю СММ і тому забезпечує:

- розширення шляхом підвищення кваліфікації персоналу можливостей організацій, що займаються створенням програмного за-безпечення;
- гарантування того, що здатність високоякісної розробки програмного забезпечення є відмінною рисою всієї організації, а не декількох співробітників;

- сумісність між мотивацією окремого індивідуума і мотивацією всієї організації;
- збереження в організації цінних людських ресурсів (наприклад, співробітників, що мають рідкісні знання і навички).

2.2.3. Модель CMMI

Успіх використання CMM привів до розроблення значної кількості схожих моделей. Це викликало як конфлікти між цілями, що пов'язані з поліпшенням процесів і існуючими технологіями, так і проблеми, пов'язані з навчанням і вибором моделей. Для вирішення цих проблем SEI розгорнув роботи зі створення інтегрованої моделі CMM - CMMI, що забезпечує кращі результати існуючих моделей. Джерелами для CMMI послужили такі моделі:

- SW-CMM - модель розробки програмного забезпечення;
- *EIF/IS* 731 - модель розробки систем;
- *IPD-CMM* - модель сумісної розробки продуктів.

Модель *SW-CMM* - це поетапна модель. Модель розробки систем - безперервна модель. CMM для сумісної розробки продукту - це «гібрид» моделей, що комбінує риси поетапно і безперервно.

Використовуючи вказівки моделі, організація ,може підійти до вдосконалення процесів або з погляду підвищення продуктивності окремих областей удосконалення, або з погляду підвищення зріло-сті організації в цілому. Реалізуючи перинні підхід, фокусуються на встановленні базових, початкових станів і вимірюванні резуль-татів поліпшень для кожної області вдосконалення індивідуально. Такий підхід підтримується в безперервних моделях, ключовим терміном яких є «продуктивність» *(capability).* Другий навпаки -фокусується на групах областей удосконалення, призначених для визначення перевірених на практиці етапів у зрілості процесів усієї організації. Цей підхід використовується в поетапних моделях, ключовим терміном яких є **«** зрілість**»** *(maturity),*

З мстою збереження властивостей обох типів моделей була роз- роблена інтегрована модель CMMI - CMM *Integrated.*

Модель містить *25* областей удосконалення, що розбиваються на чотири рівні зрілості уявленні і на чотири категорії процесів у безперервному уявленні.

Розділ 3. ПРОГРАМНІ ПРОДУКТИ І СИСТЕМИ

3.1. Програми і програмування

Суть метолів, засобів і процесів, що розглядаються в інженерії програмного забезпечення, пов'язана з двома головними поняттями - «програма» (комп'ютерна програма) і «програмування». Перше поняття позначає засіб, який керує діями комп'ютера, а другий - процес, який спрямовано на створення програм.

3.1.1. Програма

Програма - це опис обчислень. Обчислення - це дії, здійснення яких доручається певному виконавцеві. Виконавці можуть бути різ- ними і обов'язкова умова - виконавець повинен «розуміти» програму, ІЗ основі опису дій лежить поняття алгоритму. Мета обчислень - отримати результат. Основними об'єктами обчислень і результатом є значення. Значення - це конкретні елементи програми, об'єкти, що в обчисленнях замінюють змінні. Змінні - це програмні об'єкти що зберігають значення**.** Програма описує багато обчислень. Залежно від конкретних значень виконавець здійснює одне з них.

3.1.2. Комп'ютерні програми

Якщо виконавець програми - комп'ютер, то вона називається комп'ютерною. Комп'ютер - це «механістичний, рутинний» пристрій, тому дуже важливо, аби комп'ютерна програма містила всі вказівки про те, як необхідно виконувати обчислення. А це озна-чає, що поняття алгоритму, що лежить в основі програми, повинно мати точний, конструктивний характер. Оскільки комп'ютер що і математична машина, то для представлення алгоритму було розро-блено багато математичних способів. Проте, незалежно від способу представлення, алгоритму комп'ютерній програмі властиві такі аспекти:

- повна деталізація і визначеність опису обчислень і, тим самим, закінченість;
- масовість обчислень
- орієнтація на загальні зміни змінної;
- виконавець обчислень - комп'ютер;
- форми програми - текст;
- засіб запису програми - спеціальна мова.

Державний стандарт (ДСТУ 2844-94) визначає комп'ютерну програму (*computer program)* як послідовність інструкцій, які може виконувати ЕОМ.

3.1.3. Програмування

Процеси, пов'язані З написанням комп'ютерних програм, нази-ваються програмуванням, У найзагальнішому вигляді програму-вання - це виконання трьох процесів:

- складання схеми програми - опис обчислень за допомогою спеціальних засобів (мови специфікацій) - псевдокоди, блок-схеми або формальні (математичні) мови;
- складання тексту програми - опис обчислень за допомогою спеціального засобу - мова програмування з використанням схеми програми;
- налагодження програми - виконання програми з використан-ням спеціально підібраних значень з метою пошуку і усунення дефек-тів, допущених у програмах при реалізації перших двох процесів,

Два ключові об'єкта беруть участь у Програмуванні - програміст і мова програмування.

Програміст - це спеціально підготовлений суб'єкт (професіонал), який знає одну або декілька мов специфікацій і програмування та вміє виконувати процеси програмування.

Мова програмування — це штучна знакова система, призначена для запису комп'ютерних програм. Як будь- яка знакова система мова програмування задається синтаксисом - множина правил, що визначають вид пропозицій (мови), і семантикою - множина правил, що визначають операційне значення (сенс) пропозицій мови. Кожна мова програмування за допомогою синтаксису і семантики описує певного носія мови, яким ця мова визначається однозначно. Носій мови входить до складу перекладачів (трансляторів), які здійснюють переклад програми з мови програмування на мову ви-конавця. Теоретичну основу мов програмування складають алгоритмічні мови (засоби для запису алгоритмів), при цьому набори опи-сових засопів мов програмування перевищують мінімальні набори, необхідні для їх алгоритмічної універсальності, що викликане практичною орієнтацією МОВ програмування. За весь період розвит-ку програмування було розроблено значну кількість мов програму-вання (рис.3.1).

Рис 3.1. Мови програмування

3.2. Продукти інженерії програмного забезпечення, продукція і програмне забезпечення

Нині для комп'ютерів розроблено і продовжує розроблялися значна кількість програм. Серед них особливе місце займають програми, які називаються продуктами інженерії програмного забезпе-чення і продукцією.

3.2.1. Продукти інженерії програмного забезпечення

Стандарт *ISO/IEC* 14598-1 визначає продукт інженерії програмного забезпечення (продукт програмного забезпечення, програм**ний** продукт, *software product)* як безліч комп'ютерних програм, процедур і пов'язаних з ними документації та даних. До того ж, підкреслюється, що продукти можуть бути таких типів:

- для постачання користувачеві;
- інтегральні частини інших продуктів;
- для розробників і супроводжуючих.

Таким чином, ті, хто використовує продукт інженерії програмного забезпечення, можуть бути таких типів;

- користувач *(user)* - особа або організація, які використовують продукт для виконання своїх специфічних функцій;
- розробник *(developer)* - особа або організація, які виконують специфічні дії в контексті життєвого циклу програмного забезпечення та, які зорієнтовані на розробку продукту;
- супроводжуючий *(mainteiner)* - особа або організація, які виконують специфічні дії, пов'язані з супроводом продукту.

Український стандарт ДСТУ 2844-94 визначає продукт інженерії програмного забезпечення як програмний засіб (програмне забезпечення, *software),* призначений для постачання користувачеві**.** Якщо розглядати користувачів вказаних трьох типів, то це визначення і визначення *ISO/IEC* збігаються.

Таким чином, термін «продукт програмного забезпечення» використовується для позначення двох типів об'єктів:

- по-перше, так називаються комп'ютерні програми, що відповідають додатковим вимогам, що пов'язані з їх тривалим застосуванням користувачами. Ці вимоги здійснюються шляхом створення додаткових описів, інструкцій **і** даних;
- по-друге, так називаються результати виконання фаз життєвого циклу програмного забезпечення (робочий продукт), коли результати **є** інтегральними частинами розробленого продукту. Тому, продуктом може бути не тільки комп'ютерна програма**,** й специфікація вимог, документація або проект програми**,** що розробляється.

3.2.2. Продукція інженерії програмного забезпечення

У зв'язку з розвитком і застосуванням інженерних методів у розробці і використанні комп'ютерних програм змінилося ставлення до результатів праці в цій галузі. Розширення асортименту розробників і користувачів продуктів призвело до необхідності класифікувати їх і визначити належність до того або іншого виду продукції.

Продукція інженерії програмного забезпечення - це сукупність Програм, програмних засобів і продуктів, що мають загальну класифікаційну ознаку (за приналежністю, місцем розробки, призначенням).

Нині розрізняють продукцію інженерії програмного забезпечення країни, галузі і підприємства.

3.2.3. Програмне забезпечення

Таким чином, у комп'ютері може знаходитися безліч комп'ютер**них** програм двох типів. До першого типу належать будь-які комп'ютерні програми**,** які користувач

комп'ютера встановив з тих чи інших причин. До другого - належать комп'ютерні програми - продукти (продукція).

Безліч комп'ютерних програм у комп'ютері називається програмним забезпеченням *(software)* і є однією з двох найважливіших частин комп'ютерної системи. Інша частина - апаратне забезпечення (*hardware) є* власне комп'ютером. Зараз починає вирізнятися ще і третя частина - інтелектуальне забезпечення *(know ware).*

Стандарт *ISO/IEC* визначає програмне забезпечення *(software)* як все або частина програм, процедур, правил і пов'язаної з ними документації інформаційної обчислювальної системи.

Український стандарт визначає програмне забезпечення (software) як програмний засіб, тобто взаємозв'язану сукупність програм, процедур, правил, документації і даних, що стосуються функціонування обчислювальної системи.

3.3. Системи програмного забезпечення

Поняття системи програмного забезпечення, або програмної сис-теми *(software system) є* дуже важливим в інженерії програмного забезпечення. Але чіткого визначення цього пон**яття** не існує. У цьому розділі для того, щоб виявити, що таке програмна система, застосовується системний підхід. Тому, перш ніж визначити понят-тя програмної системи розглянемо властивості комп'ютерних програм з позицій системного аналізу.

3.3.1. Комп'ютерні програми як системи

Із системотехніки відомо, що об'єкт повинен володіти щонайменше чотирма властивостями, аби його можна було вважати системою, тому комп'ютерну програму можна розглядати як систему, якщо вона має такі властивості: цілісність і подрібненість, наявність зв'язків, наявність організації, наявність інтегральної якості,

Властивості цілісності і подрібненості виражаються в тому, що, з одного боку, програма - це цілісна конструкція, а з другого, - в її складі можна розрізняти елементи.

Елементи - це такі частини програми, які в даній програмі на певному рівні її розгляду не підлягають подальшому поділу на частини. Поза програмою елементи мають властивості, що є загальними і називаються системнозначимими. Увійшовши в програму, елемент набуває властивостей, що мас значення лише в цій програмі і яке називається системовизначеним. Наприклад, підпрограма обчислювальна *t* g*(x)* (системовнзначена властивість), увійшовши до складу програми управління польотом певного об'єкта, може обчислювати значення деформації (зрушення) шарів плоскої о тіла (системовизначена властивість).

Завдяки властивостям цілісності і подрібненості програма може розглядатися як єдине ціле, але таке, що складається з частин, що взаємодіють.

Властивість - наявність зв'язків *(couple);* виражається в наявності стійких зв'язків між елементами програми, що перевершують по силі зв'язки цих елементів з елементами, що не входять в дану програму. Зв'язки можуть бути такими, що сполучають і передають інформацію. Наявність сильних зв'язків дає змогу говорити про межу програми (там, де вони стають слабкими), а наявність межі веде до поняття зовнішнього середовища програми, яка є сукупністю елементів, що не належать програмі, але пов'язаних з нею і, що виливають на неї. Зазвичай цей вплив інтерпретується як надходження в програму вхідних значень і отримання з програми значень (результату).

Частіша межі, через яку здійснюється передавання вхідних значень і отримання результату, називається Інтерфейсом програми. Звичайна взаємодія програми і середовища має двонапрямлений характер (рис. 3**.** 2)**.**

Рис. 3.2. Програма і середовище

Властивість - наявність організації; виражається у взаємозалежній поведінці частин програми завдяки їх упорядкованості в часі і просторі. У результаті складається внутрішня структура програми, а завдяки процесам, що відбуваються в програмі, функції частин програми трансформуються у функції (функцію) цілої програми. Функції можуть бути корисні, даремні і шкідливі.

Корисність програми є однією з її властивостей і визначається відповідністю функції І конструкції програми вимогам, що висуваються до неї.

Властивість - наявність інтегральної якості; виражається в тому, що має місце якість, що властива програмі в цілому, але не властива жодній з її частин окремо. Тому програма **не** зводиться **до** простої сукупності її частин. Розчленувавши програму на частини і вивчивши їх окремо, не можна зрозуміти інтегральну якість програми.

3.3.2. *Програмні системи*

Програмною системою називається продукт інженерії програмного забезпечення, що має системні властивості та створений для застосування в певній наочній галузі.

Як правило, програмні системи є частиною складного апаратно-програмного комплексу, що включає крім комп'ютера різноманітні пристрої, робота яких заснована на різних принципах дії.

3.3.3. Класифікація програмних систем

Застосування обчислювальної техніки, що постійно розширюється, призвело до появи великої різноманітності в програмних системах. Відомо багато класифікацій програмних систем. На рис. *3.3* показано одну з класифікацій, що не «претендує» на повноту. Як видно, всі програми системи поділяються на дві групи - автоматичні і автоматизовані.

Рис. 3.3. Типи програмних систем

У першій групі розрізняють автоматичні вбудовані системи **(** АВСТС**)** і автоматизовані системи управління технологічними процесами (АСУТП). До другої групи належать:

- автоматизовані інформаційні системи (АІС), що поділяються на документальні (повнотекстові) - АІДС і фактографічні - АІФС;
- автоматизовані моделюючі системи (АМС);
- автоматизовані системи управління (АС), що поділяються на системи організаційного управління - АСОУ і управління експериментами - АСУЕ;
- системи автоматизованого проектування (САПР), що поділяються на системи проектування об'єктів різною призначення -*CAD* і системи проектування програмного забезпечення - CASE, системи зворотної інженерії (CARE).

Розділ 4. ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. СКЛАДОВІ

Життєвий цикл програмного забезпечення - це фундаментальне поняття інженерії програмного забезпечення.

Уведення цього поняття дало змогу деталізувати розроблення програмного забезпечення і зробити його керованим.

У цьому розділі розглядаються складові життєвого циклу програмного забезпечення - процеси, продукти і ресурси.

4.1. Життєвий цикл програмного забезпечення

Життєвий цикл програмного забезпечення, як і будь-якого іншого виробу, відображає його відповідність у часі і просторі, процеси, які пов'язані з виробом і засоби здійснення процесів,

У період загострення кризи програмного забезпечення велися спори щодо ефективності (необхідності) використання життєвого циклу. Характер суперечок відображає гасло, типове для того часу -«Зупиніть життєвий цикл, я кочу вийти!». Як тепер зрозуміло - спори вирішилися на користь використання поняття життєвого циклу і, більше того, це поняття є фундаментальним в інженерії програмного забезпечення.

Життєвий цикл програмного забезпечення складається з фаз, Суть фаз - виконання процесів за допомогою ресурсів (методи, засоби, інструменти, персонал) і отримання продуктів. Тому основ- ними складовими життєвого циклу програмного забезпечення є процеси , ресурси, продукти (рис. 4**.** 1).

Рис. 4.1. Основні складові життєвого циклу програмного забезпечення

4.2. Процеси

Розробка програмного забезпечення - складне і трудомістке завдання та існує безліч альтернативних способів його вирішення. Визначення поняття процесу взагалі і процесу розробки програмного забезпечення зокрема може допомогти вирішити проблему вибору способу виконання процесу. Завдяки визначенню процесу можна краще розуміти, що слід робити і чекати, що потрібно забезпечити в першу чергу. Процес - це ряд взаємозв'язаних видів діяльності (дій).

4.2.1. Основні визначення

Дія, зазвичай, має очікувану тривалість, прогнозовану вартість **і** очікувані вимоги до ресурсів. Розрізняють стандартний і визначуваний процеси. Стандартний процес - це безліч фундаментальних елементів процесу, які будуть включені в кожен визначуваний процес. Визначуваний процес - це покрокове виконання ряду певних видів тривалості, спрямоване на досягнення певної мети, Розробка програмного забезпечення часто не схожа на звичайний вид діяльності, який може бути побудований **і** впорядкований як виробництво, що повторюється, або «конторська» процедура. Тому процес повинен розглядатися, як інтелектуальна діяльність, що адаптується до творчих потреб професіоналів і їх завдань. При цьому потрібне досягнення компромісу між індивідуальною потребою в гнучкості і організаційною потребою в стандартах. Деякі з чинників, які слід при цьому враховувати, такі:

- процеси, що реалізуються в ході здійснення того чи іншого програмного проекту (розрізнятимуться, оскільки програмні продукти мають різний характер);
- здійснення стандартного процесу розробки програмного забезпечення (потребує від організації і проекту визначення процесів, які задовольняють їх власні унікальні потреби);
- процес, використовуваний для реалізації даного проекту (повинен враховувати досвід членів команди, поточний статус продукту, а також доступні інструменти і можливості).

Згідно з стандартом IEEE610 процес життєвого циклу програмного забезпечення визначається як послідовність етапів, спрямова-них на досягнення конкретної мети, якою може бути створення програмного продукту або програми. Детальніше, процес - це обмежений ряд взаємозв'язаних дій, у процесі здійснення яких використовуються один або більше типів початкових продуктів, які за допомогою однієї або декількох змін перетворюються на вихідний продукт, що становить цінність для замовника.

Процес розглядають в трьох різних аспектах, визначаючи три типи процесів:

- метапроцес - дії, які виконує організація під час проведення підприємницької діяльності, пов'язаної з розробкою програмного забезпечення. Основна увага при виконанні цього типу процесу приділяється економіці організації, довготривалій стратегії і поверненню інвестицій у програмне забезпечення;

- мікропроцес - дії, які виконуються в організації в разі реалізації певного проекту програмного забезпечення. Основна увага приділяється вартості, термінам і якості;

- мікропроцес - дії, які виконуються командою розробників у певній фазі проекту, спрямовані на отримання конкретних результатів. Основна увага приділяється створенню проміжного продукту такої якості, яка адекватна функціональним можливостям і настільки економічно і швидко, наскільки це здійснено на практиці.

Ці три типи процесу перекриваються, коли виконуються паралельно, але у них різні цілі, учасники, метрики, виробничі відносини і часовий масштаб (табл. 4.1),

Таблиця 4.1

Стандарт IEEE 1074 описує процеси і дії. Цим стандартом передбачено 17 підпроцесів і 65 дій, що входять до складу підпроцесів.

Розрізняють процеси «важкі» і «полегшені». Для процесів першого типу характерне таке:

- реалізація фіксованих вимог великою групою розробників;

- повний прогноз робіт, які слід виконати;

- строго усталений порядок виконання. Для процесів другого типу характерне таке:

- реалізація невеликою кількістю розробників за умови частих змін вимог, непередбачуваність;

- адаптивність під час виконання;

- участь замовника;

- відсутність повного порядку і документування.

Як приклад реалізації стандартного процесу можна навести уніфікований процес (*Rational United process)* , а як приклад визначуваних процесів - робочі процеси.

В уніфікованому процесі термін «процес» належить до концепції, що працює подібно до шаблону, який може бути багато разів використаний для створення екземплярів - процесів конкретного проекту.

Процес розробки програмного забезпечення - це назва для повного набору дій, необхідних для перетворення вимог користувачів в узгоджений набір артефактів, що є програшнім продуктом, а пізніше - для перетворення змін в цих вимогах у новий узгоджений набір артефактів.

Значущий результат процесу - це узгоджений набір артефактів, що представляє одну програмну систему або сім'ю тих систем, що С програмними продуктами. Таким чином, процес покриває не тільки перший цикл розробки (перший випуск), але **і** більшість подальших випусків. У подальших випусках до екземпляра процесу вносяться послідовні зміни у вимоги. На виході, відповідно, виходять змінені набори артефактів.

В уніфікованому процесі розрізняють чотири стадії, які групуються відповідно по дві;

- стадія розробки, дії спрямовані на виконання робіт з проектування і синтезу (включає початкову стадію і стадію уточнення — детального проектування);

- стадія виготовлення, дії зорієнтовані на створення, тестування і введення в дію програм або програмних продуктів (включає стадію реалізації і стадію введення в дію).

Процес розробки описується в поняттях робочих процесів. Робочим процесом є набір видів діяльності. Термін «робочий процес» використовується для позначення потоку зв'язаних і послідовних дій. Передбачається сім робочих процесів:

- управління проектом - контроль ходу робіт і гарантія умов Досягнення успіху для всіх зацікавлених сторін;

- створення робочого середовища - автоматизація процесу і розвиток середовища

супроводу та експлуатації;

- управління вимогами - аналіз проблемної області і вдосконалення робочих продуктів вимог;
- проектування - моделювання рішення і отримання робочих продуктів проектування (архітектура);
- реалізація - програмування компонентів **і** вдосконалення робочих продуктів;
- оцінка - оцінювання тенденцій і якості продукту;
- впровадження - передача кінцевих продуктів користувачеві.

У термінах UML процес - це стереотип кооперації, в якій беруть участь співробітники і артефакти. Не існує такого процесу розробки програмного забезпечення, який міг би застосовуватися у всіх випадках. Основні чинники, що призводять до відмінностей у процесах, такі:

- організаційні - структура і культура організації, організація управління проектом, наявні здібності і знання, попередній досвід і створені програмні системи;
- наочні області - наочна область програмного забезпечення, бізнес-процес підтримки, співтовариство користувачів і пропозиції конкурентів;
- життєвого циклу - час виходу на ринок (проведення під час розроблення програми експертизи технології та персоналу і планування майбутніх випусків);
- технічні - мови програмування, засоби розробки, бази даних, покладена в основу розробки архітектура.

Процеси змінюються, оскільки вони викопуються в різних контекстах, призначені для розробки різних типів систем і мають різні типи бізнес-обмежень (плани, ціна, якість і надійність). Тому реаль-ний процес розробки програмного забезпечення повинен мати можливість адаптуватися і конфігуруватися під поточні потреби конкретного проекту і/або організації. Розробляючи уніфікований процес, у нього була включена можливість спеціалізації. Будь-яка організація, що використовує уніфікований процес, з часом спеціалізує його, пристосувавши до своїх умов.

4.2.2. Контроль виконання процесу

Для контролю виконання процесу використовуються контрольні точки. Контрольна точка - це місце, визначене часом у процесі, коли він зупиняється і здійснюється оцінювання параметрів складових фази життєвого циклу.

Розрізняють три типи контрольних точок:

- основні - встановлюються в кінці кожної стадії розробки на рівні всієї системи. Дають змогу виявити великі проблеми, узгодити точки зору управління та розробки і підтвердити, що цілі даної стадії досягнуті;
- другорядні - встановлюються в кінці ітерації (у рамках стадії) для детальної перевірки результатів ітерації і для санкціонування подальшої роботи;
- оцінки стану - встановлюються довільно і дають змогу контролювати хід процесів.

Для основних контрольних точок в кінці кожної стадії використовуються формальні, схвалені зацікавленими сторонами критерії оцінки і опису версій. Для другорядних контрольних точок застосовуються неформальні критерії, встановлювані командою розробників.

Критерії точок оцінки стану виробляються залежно від ситуації, що склалася.

Рівень і кількість контрольних точок змінюються залежно від таких параметрів, як: масштаб проекту, кількість зацікавлених сторін, стан бізнесу, технічний ризик і чутливість проекту до змін витрат і термінів. Для більшості процесів потрібно встановити всі основні контрольні точки (відповідно до кількості стадій). Лише у виняткових випадках слід додавати основні контрольні точки або оперувати меншою їх кількістю. У простіших проектах для контролю проміжних результатів може знадобитися менша кількість другорядних контрольних точок або їх не потрібно буде зовсім, а періодичність оцінок стану

може бути невеликою (наприклад, щоквартальною).

4.3. Ресурси

Ресурси - це другий тип складових, що забезпечують підтримку виконання процесів життєвого циклу програмного забезпечення. Розрізняють такі ресурси: інструменти, методи, виконавці.

4.3.1. Інструменти

Інструменти - це засоби, за допомогою яких здійснюються процеси життєвого циклу. Інструменти можуть бути реалізовані як програмно, так і апаратно. Далі розглядатимуться лише програмно реалізовані інструменти.

Перехід від «програмування в малому» до «програмування у великому» розширив погляд на розробку програмного забезпечення в двох напрямах:

- визнання важливості фаз аналізу і проектування зажадало розробки нових методів;
- нові методи потребували відповідної інструментальної підтримки.

Тому в 1970р. зусилля дослідників були спрямовані на уточнення поняття процесу і розробку відповідних методів, а, починаючи з 1980-х, почали розроблятися інструменти для реалізації процесів на основі нових методів.

Типи інструментів. Інструменти поділяють на два типи: вертикальні або окремі, горизонтальні або інтегральні.

Вертикальні інструменти призначені для виконання окремих дій або процесів у життєвому циклі програмного забезпечення. Наприклад, таких: створення людино-машинних інтерфейсів; створення баз даних і знань; специфікація вимог; виконання доменного аналізу; кодування (програмування); тестування; супровід; документування; реструктурування.

Вертикальні інструменти можуть бути таких типів: синтаксичні, семантичні, структурні.

Синтаксичні інструменти - принцип їх дії ґрунтується на використанні синтаксичного аспекту представлення інформації. Це сканери, синтаксичні аналізатори, реструктуризатори і мовно-орієнтовані редактори.

Семантичні інструменти - принцип їх дії ґрунтується на використанні семантичного аспекту представлення інформації, Це інтерпретатори, компілятори, верифікатори і валідатори.

Структурні інструменти - принцип їх дії ґрунтується на структурному представленні інформації. Це контролери версій, діаграмери.

Горизонтальні інструменти призначені для виконання всіх процесів життєвого циклу або лише декількох процесів (стадій) у певному аспекті. Наприклад, таких:

- аналіз і проектування - інструменти *Computer Aided Software Engineering-Analysis and Design (CASE-AD);*
- управління проектом - інструменти CASE - Project Management (CASE-PM);
- управління якістю - інструменти *CASE - Quolity Management (CASE-QM);*
- реверсивна інженерія - інструменти *Computer Aided Reverse Engineering (CARE).*

Горизонтальні інструменти, як правило, складаються з декількох інструментів, об'єднаних методологією побудови. Розрізняють такі методології побудови інтегральних інструментів: мовно-орієнтована, структурно-орієнтована, методо-орієнтована, набір інструментів.

Мовно-орієнтована методологія. Відповідно до неї, весь набір інструментів побудований, орієнтуючись на конкретну мову. Зазвичай, це мова програмування зі своїм середовищем програмування. Особливістю таких середовищ є те, що вони, як правило, орієнтовані на швидку розробку. Середовище містить всі інструменти, що необхідні для створення, тестування, налагодження і швидкої зміни коду. Інструмент будується як

монолітна система і працює в інтерактивному режимі. До того ж у середовищі зберігається вся синтаксична і семантична інформація (результат синтаксичного розбирання, структура і семантичний опис), що забезпечує розуміння програми.

Структурно-орієнтована методологія. Спочатку суть методології полягала в тому, щоб дати користувачеві інтерактивний інструмент, наприклад! синтаксично орієнтований редактор для введення програм у термінах мовних конструкцій. Потім ця ідея була поширена на інтерактивну підтримку семантичного аналізу, виконання і налаштування програм. У кінцевому результаті ця методологія заснована на маніпулюванні програмними структурами І забезпечує різні погляди на програми, які генеруються з однієї і тієї ж програмної структури, стеження за інформацією, важливою для користувача. Тому структурно-орієнтована методологія забезпечує такі аспекти:

- багаторівневий погляд на програмні структури;
- семантичні обчислення;
- генерацію структурно-орієнтованих середовищ. Методо-орієнтована методологія. Ця методологія орієнтує на

використання одного методу при побудові середовища. Розглядають методи двох типів:

- розробки - орієнтовані на реалізацію робочого процесу;
- управління - орієнтовані на реалізацію макро- і мікро-процесів,

Прикладами методів першого типу є *SREM, SADT, OOD, PSL/PSA,* а також діаграми даних і управління, ER-діаграми, мережі Петрі, мови специфікацій, *PDL*. Методи другого типу використовуються для побудови *CASE,* які зазвичай бувають двох типів, орієнтовані на виконання декількох або всіх робочих процесів і на ви- конання процесів управління проектом.

Методологія - набір інструментів. Суть її полягає у відсутності якої-небудь методології. Середовища будуються з сукупності невеликих інструментів, орієнтованих на виконання фази кодування (редактор, компілятор, налагоджувальник, лінкер). Сюди можуть Входити інструменти контролю версій і управління конфігураціями.

Середовища розробки програмного забезпечення. Важливу роль в інструментальній підтримці процесів життєвого циклу відіграють середовища розробки. Середовище розробки *(Software Development Environment - SDE)* - це сукупність апаратних **і** програмних інструментів, використовуваних для розробки програм і програм-них продуктів. Розрізняють середовища програмування, орієнтовані на реалізацію окремих процесів (вертикальні) **і** середовища розробки програмного забезпечення, орієнтовані на весь життєвий цикл (горизонтальні). Раніше розглядалися методології побудови середовищ розробки програмного забезпечення. Далі розглянемо їх моделі.

Загальна схема SDE моделі така:

SDE Model = ({Структури},
{Механізми},
{Управління}).

Структури - це об'єкти і агрегати об'єктів, на які впливають механізми в процесі розробки програмного забезпечення.

Прості структури - файлові. Складніші будуються на основі абстрактних синтаксичних дерев, графів, баз даних. Основні функції структур такі:

- підтримувати інтеграцію інструментів у *SDE;*
- мати достатньо інформації, необхідної для роботи механізмів І виконання управління;
- забезпечувати управління змінами програмного забезпечення, що розробляється.

Зазвичай, для вирішення завдань у *SDE* пропонується використовувати різні типи структур.

Механізми - це мови, інструменти і фрагменти інструментів, які маніпулюють структурами в *SDE.* Стосовно програміста, механізми можуть бути видимі і приховані.

Якщо детальніше вивчені механізми, то більше вони мають тенденцію вбудовуватися всередину середовища і складати основу генераторної частини інструментів. Наприклад, *Jass*

- генератор синтаксичних аналізаторів. До того ж, чим більш узгоджуються механізми і структури, тим глибшим є аналіз взаємозв'язків між частинами програмного забезпечення і вірогідніша поява адекватної методології, наприклад, аналіз діаграм даних, використовува-ний для управління змінами або програмно-базоване тестування.

Управління - це вимоги, що впливають на користувача *SDE* у процесі розробки програмного забезпечення. Виражаються вимоги у формі правил, стратегій, методик. Зазвичай інструменти і структури, створюючи *SDE,* визначають управління. Розрізняють типи управління: підтримуване і захищене.

Підтримуване управління припускає наявність у *SDE* механізмів і структур, що забезпечують підтримку управління. Наприклад, низхідна розробка програмного забезпечення - управління, яке підтримується в багатьох *SDE.*

Захищене управління припускає не тільки підтримку, але і не- можливість порушити його в *SDE.* Захист може бути реалізований за допомогою механізмів і структур *SDE* або організаційно - зовні в *SDE* у формі певного консенсусу розробників.

Розрізняють типи управління: спрямовані на механізми і структури і спрямовані на управління.

Другий тип управління називається високорівневим або метауправ-лінням. Наприклад, усі проект повинні бути реалізовані мовою *ADA.*

4.3.2. Методи

Метод - це систематичний процес дослідження, який у контексті інженерії реалізується для дослідження і створення програмного забезпечення.

В інженерії програмного забезпечення доводиться розробляти і описувати в різних аспектах велику і різноманітну кількість компонентів, використовуючи такі засоби:

- описи - техніка і нотації, представлені за допомогою синтаксису, діаграм, таблиць, що використовуються в документуванні програмного забезпечення;
- моделі - математичні структури, що представляють аспекти компонентів програмного забезпечення;
- моделювання - створення моделі та експериментування для отримання інформації про програмне забезпечення.

Вказані засоби використовуються в рамках трьох таких процесів *(рис . 4.2):*

- аналіз домена (специфікування вимог) - у результаті виконання процесу визначається «що повніша робити» програмна система;
- аналіз вимог - у результаті виконання процесу визначається «що робить» програмна система;
- проектування - у результаті виконання процесу визначається «як робить» програмна система.

Очевидно, що опис і розробку програмного забезпечення не можна виконати в рамках одного методу. Усі використовувані в інженерії програмного забезпечення метоли поділяють на дві групи: загальнонаукові методи і методи інженерії програмного забезпечення.

Рис. 4.2. Процеси програмного забезпечення

Своєю чергою загальнонаукові методи поділяють на три такі групи:

- теоретичні методи - абстрагування, формалізація, аксіоматика, узагальнення;
- емпіричні методи - спостереження**,** порівняння, контроль, розрахунок, вимірювання, ідентифікація, науковий експеримент;
- емпірико-теоретичні методи - аналіз і синтез, індукція і дедукція, перевірка гіпотез, моделювання.

Теоретичні методи пізнання спрямовані на дослідження абстракт-них об'єктів, їх властивостей і відносин. Ці методи: дають можливість отримувати нові знання про об'єкти і

явища шляхом дослідження властивостей абстрактних об'єктів і відносин між ними. Теоретичні методи є найбільш потужним інструментом для прогнозування, створення нових областей знань і служать основою фундаментальних наук. Вони ж лежать в основі багатьох методів інженерії програмного забезпечення,

Емпіричні методи пізнання поділяють па дві групи. Перша - методи якісної оцінки: спостереження, порівняння, контроль. Друга група - кількісної оцінки: розрахунок, ідентифікація, вимірювання і експеримент. Велике значення серед усіх експериментальних ме-тодів пізнання для інженерії програмного забезпечення мас. вимірювання, за допомогою якого отримують кількісну інформацію щодо програмного забезпечення. Наявність вимірювальної інформації про досліджуваний об'єкт дає можливість забезпечити ефек-тивну реалізацію всіх емпіричних методів пізнання - від спостере-ження до експерименту.

Емпірико-теоретичні методи дають змогу досліджувати різні сторони об'єктів і явищ, розчленовуючи їх на складові для деталь- нішого вивчення. Найповніше отримання додаткової інформації, що міститься в неявному вигляді в результатах, отриманих за допомогою емпіричних методів, забезпечують емпірико-теоретичні методи встановлення ступеня істинності гіпотез, а також складають основу методів проектування нових технічних засобів і технологіч-них процесів, основу технічних наук і методів підвищення продук-тивності праці на виробництві.

Методи інженерії програмного забезпечення можна розглядати в контексті двох доменів: прикладного і реалізаційного (рис. 4.3).

Суть розробки програмного забезпечення полягає в реалізації процесу, який починається з ідентифікації вимог у прикладному домені і закінчується створенням програмного продукту, який від-повідає цим вимогам у реалізаційному домені. На практиці процес розробки має ітераційний характер і кожна ітерація використовує два Види аналізу (рис. 4,4).

Рис. 4.4. Зв'язок доменів і моделей

Перший вид аналізу спрямований на розуміння вимог, а другий - на розуміння того, як програмний продукт повинен задовольняти цю вимогу. Па рис. 4.5 показано як спочатку відрізняються погляди замовника і розробника на одне й те саме питання і як, реалізуючи доменні знання, за допомогою ітерацій «сходиться» розуміння то-го, «що повинно робити» і «що робить» програмне забезпечення.

Рис. 4.5. Розбіжність поглядів замовника та розробника

На рис. 4.5 ∆ показує початкову розбіжність поглядів замовника і розробника на продукцію програмного забезпечення.

Моделі можуть бути дескриптивні і прескриптивні. Дескриптивна модель показує, як програмний продукт повинен «поводитися». Ця модель має бути перетворена в прескриптивну модель, що показує, який програмний продукт «поводитиметься» так, як визначає дескриптивна модель. Мета першої моделі - показати, як програмний продукт відповідатиме вимогам, а мета другої - забезпечити однозначне виконання вимог тими, хто конструюватиме програмний продукт.

Дескриптивні моделі - концептуальні, прескриптивні - формальні. Обидві категорії моделей повинні бути точними і недвозначними. Проте формальні моделі повинні містити додаткові критерії коректності для створюваного програмного продукту. Оскільки мета концептуальної моделі - описувати вимоги, то і якість моделі визначає те, наскільки добре програмні продукти відповідають вимогам прикладного домена. Визначення такої відповідності - процес суб'єктивний і називається валідацією. Якщо формальна модель існує, то основні властивості програмного продукту встановлені і можна встановлювати його коректність відносно до формальної моделі. Процес встановлення коректності називається верифікацією. Таким чином, валідація має справу з проблемою (вимоги), а верифікація з

продуктом (реалізація вимог). Очевидно, що для однієї проблеми може бути декілька концептуальних моделей, а для кожної концептуальної - декілька формальних моделей, У цьому сенсі немає кращої реалізації проблеми, а формальні моделі відіграють важливу роль у процесі розробки програмного продукту, оскільки без них не можна визначити коректність проектування і реалізації.

Таким чином, усі методи інженерії програмного забезпечення можна розділити на два типи. Перший тип - проблемно-орієнтовані методи, що забезпечують краще розуміння проблеми і пропонованого рішення. Другий тип - продукто-орієнтовані методи, що забезпечують коректну трансформацію формальної специфікації в супроводжувану реалізацію. Очевидно, що можуть бути методи, які забезпечують обидва аспекти процесу розробки. У табл. 4.2 наведено методи інженерії програмного забезпечення.

Таблиця 4.2

Розглянемо ці методи докладніше.

Рівні абстракції (Е. Дейкстра, 1968). Ґрунтуючись на досвіді системи мультипрограмування Т.Н.Е., Дейкстра запропонував розробляти програмне забезпечення за рівнями. Перший низький рівень забезпечує балові сервіси для реалізації наступних, рівнів і ґрунтується на можливостях тієї машини, що реально існує. Кожен наступний рівень використовує сервіси попереднього. Процес Створення рівнів продовжується до тих пір, доки по буде реалізований найвищий рівень абстракції. Наводиться приклад з управлінням пам'яті. На базовому рівні (машинна мова) відомі канальні команди для обміну даними між різними типами пам'яті. На першому рівні проектується механізм управління перериваннями - вплив на сигнали переривання, але без деталей маскування, відкриття, закривання, блокування, черговості. На наступному рівні проектується канальний супервізор - зберігається можливість управління каналами без зайвих деталей і, нарешті, на ще вищому рівні адміністратор пам'яті організовує обмін. Метод можна віднести як до методів розробки, так і управління.

Покрокове уточнення (Н. Вірт, 1971). Цей метод ґрунтується на таких принципах:

- розкладання простору рішень на підпростори;
- ітеративне вирішення завдання (повторення дій);
- аналіз ефективності розкладання (наступне поліпшення виконується лише у вигідному напрямі).

Мета покрокового уточнення полягає в зменшенні ризику, пов'язаного із застосуванням рішень під час програмування. Розділений процесу розробки програмного забезпечення на кроки дає змогу планувати його створення, відкладаючи реалізацію певних модулів, шляхом заміни їх заглушками, що імітують міжмодульні інтерфейси.

Наприклад, метод МЕТА (Х. Ледгард, 1973) використовує низхідне уточнення.

Необхідна умова роботи методу - наявність точного і стабільного опису завдання, а результат S - програма.

Функціональна декомпозиція — це метод низхідного аналізу програмного забезпечення шляхом опису функцій, котрі йога утворюють і уявлення про їх реалізацію у вигляді функціональних зв'язаних модулів. Модуль називається функціонально зв'язаним, якщо містить компоненти, спрямовані на виконання однієї функції. Додаток, побудований з функціонально зв'язаних модулів, легко будувати і супроводжувати.

Модуляризація (Д. Парнас, 1972). Цей метод заснований на трьох фундаментальних принципах: з'єднання, скріплення, приховування інформації. Метод призначений для вирішення завдання розбиття програми на модулі, яка називається модуляризацією. Використовуються такі критерії модуляризації:

- з'єднання - поєднання частин модуля із зовнішнім оточенням повинне бути слабким;
- скріплення - скріплення частин, що входять до складу модуля має бути сильним;
- приховування - інформація, що міститься в модулі, повинна бути прихованою.

Модуль мас тіло, яке містить опис функції, що вирішується модулем, і інтерфейс, що забезпечує зв'язок модуля з його користувачами.

Структурне програмування (Е. Дейктра, 1972). Цей метод обмежує написання програм шляхом використання лише трьох типів операторів: послідовне з'єднання, вибір і повтор. Такий операторний базис достатній для написання будь-якої програми. Програми називаються структурними, а процес програмування - структурним. Метод передбачає також способи переходу від неструктурних програм до структурних.

Абстрактні типи даних (Б, Лісков, С. Зайліс, 1975). Суть методу полягає в розширенні концепції структур даних визначуваними операціями над ними. Перші реалізації абстрактних типів даних могли здійснюватися вже в підпрограмних мовах програмування *(Paskal, С) шляхом* використання процедурного типу або вказівного тину на підпрограму. У подальшому в мовах програмування були введені спеціальні конструкції (модуль, клас), спрямовані на ефективну реалізацію абстрактних типів даних.

Структурний аналіз (Т. де Марко, 1978). Метод аналізу специфікацій вимог за допомогою діаграм (управління, дані, перехід станів) - *PSL/PSA* (Д. Тіхроєв, 1977) шляхом ієрархічної декомпозиції вирішення проблеми. Людино-машинна техніка реалізована у вигляді мов і призначена для структуризації документації і аналізу інформації. Мови *PSL/PSA* використовують графічні діаграми *{System Analisys and Design Techik - SADT),* які застосовують для опису даних, перетворення інформації і процесів, що відбуваються в системах (рис. 4.6).

Рис. 4.6. Діаграма SADT

ER-моделювання (С. Чен, 1976). Метод опису прикладного домена за допомогою спеціальних діаграм «суть-зв'язок» *(Enitity -Relation - ER),* Метод використовує три типи нотацій: сутність, зв'язок (відношення), з'єднання.

Наприклад, діаграма на рис. 4.7 описує домен «комп'ютер — програма».

ER-модель використовується для опису логічних схем баз даних.

Рис. 4.7. ER-Діаграма:

□-сутність; ◊-(зв'язок,відношення) − − - з’єднання

Позначення 1 і *m* біля з'єднань показують їх тип (у цьому прикладі і «один - до багатьох»).

Системне програмування Джексона (К. Джексон, 1977). Метод використовує структурні оператори (проходження, вибір, повторения) для представлення логічних структур програмного забезпечення.

Віденський метод (IBM, 1970). Метод ґрунтується па операційному підході до опису семантики мов програмування. Суть методу полягає в тому, що завдання семантики здійснюється шляхом опису певних системних пристроїв - автоматів, що володіють па-м'ятно і структурними станами.

Метод був розроблений для представлення семантики мови програмування *PL/1,* а потім використовувався для опису програмного забезпечення.

Simula 67 (У. Дал, Л. Кей, 1976), Мова програмування, до якої вперше введено поняття класу. Основним поняттям у мові був об'єкт - екземпляр блоку. Безліч об'єктів утворюють клас. Класи можуть складатися з підкласів (ієрархія, контейнер) у сучасному розумінні Згодом мова *Simula 67* стала основою побудови мов об'єктно-орієнтованого програмування.

Об'єктно-орієнтоване проектування (Г. Буч, 1980). Суть методу полягає у використанні поняття об'єкта при аналізі наочної об- ласті і проектуванні програмного забезпечення,

Доменний аналіз (Р. Прісто-Діаз, 1991). Метод спрямований на аналіз наочних

областей з метою визначення повторно використовуваних рішень.

Об'єктнно-орієнтований аналіз (Е. Йодон, П. Коад. і 978). **Є** розширенням методу структурного аналізу шляхом використання об'єктів.

Усі перелічені методи можна розташувати в матриці (рис. 4.8),

Рис. 4.8. Відповідність типів моделей і методів

4.3.3. Персонал

Програмні процеси реалізують фахівці. Більшість програмних Процесів є настільки складними, що не можуть бути виконані одним або двома особами, тому шин зазвичай, утворюють певну структуру (організацію), характер якої, як і властивості окремих осіб, відіграють дуже важливу роль у розробці програмних продуктів.

Отже, розглянемо дві складових персоналу:

- кваліфіковані фахівці, здатні викопувати роботи зі створення програмного забезпечення;
- організації - об'єднання людей, що ставлять за мету виконання робіт зі створення програмного продукту.

Фахівці. Нині відомий дуже широкий спектр фахівців, які можуть залучатися до розробки програмного забезпечення.

Фахівцями комісії SEEPP розроблений етичний кодекс інженера з програмного забезпечення. Він включає такі аспекти:

- суспільні інтереси - дії програмістів повинні відповідати суспільним інтересам;
- клієнт і працедавець - програмісти повинні вчиняти так, щоб якнайкраще задовольнити вимоги клієнта і працедавця, але при цьому дотримуватись суспільних інтересів;
- продукт — програмісти повинні бути впевнені в тому, що створюванні ними програмні продукти і пов'язані з продуктами модифікації відповідають професійним найвищим стандартам;
- критицизм - інженери-програмісти повинні дотримуватися цілісності і незалежності своїх думок, формуючи здоровий професійний критицизм мислення;
- Менеджмент - менеджери і лідери, керівники груп з розробки ПО, зобов'язані дотримуватися стичних норм у процесі розробки і супроводу програм;
- професіоналізм - програмісти зобов'язані бути чесними і підтримувати репутацію професіоналів, не забуваючи про дотримання суспільних інтересів;
- колегіальність - програмісти зобов'язані підтримувати своїх колег;
- самовдосконалення - програмістам слід постійно підвищувати свою кваліфікацію, що сприятиме їх професійному зростанню, а також формувати етичний підхід до професійної діяльності.

Культуру складають безліч цінностей, цілей і принципів, що керують діями, пріоритетами і рішеннями окремих осіб або групи, які працюють задля реалізації загальної мети. Культура груп - розробників програмного продукту дуже сильно впливає на якість продукту, продуктивність розробників і робочу обстановку в групі.

На ринку праці використовуються різні методи і моделі, які можна розділити на дві групи:

- підбору персоналу;
- розвитку персоналу.

До моделей підбору персоналу належать такі:

- індикатор типів (модель) Майерса-Брігтса *(MBTI)* - застосовується для ідентифікації чотирьох біполярних видів поведінки на основі 16 різних описів персональних властивостей особи;
- модель функціональних міжособових відносин орієнтації поведінки *(FIRO-B)* -

застосовується для ідентифікації типів міжособових відносин шляхом вимірювання трьох аспектів, які позначаються як «залучення», «контроль» **і** «прихильність»;

- модель сортування темпераментів Кейрси - застосовується для тестування осіб (шляхом опитування) на основі чотирьох типів темпераментів;

- модель міжпроцесорної взаємодії Келера (РСМ) - застосовується для ідентифікації типу особи на основі шести тилів осіб з використанням транзакційного аналізу. Транзакції - це «мінісценарії» поведінки. Застосування здійснюється шляхом спостереження. Модель враховує результати змін характеристик особи впродовж життя.

Наведені моделі дають змогу розпізнавати особові шаблони і передбачати характер взаємодій між співробітниками в організації.

Моделі розвитку персоналу застосовуються для підвищення кваліфікації фахівців.

Організація. В інженерії програмного забезпечення розглядається два поняття, що характеризують організації, розробляючи [програмне забезпечення: «культура» і «структура».

Існують такі визначення організацій:

- система, яка обмінюється матеріалами, кадрами, робочого си- лою і енергією з навколишнім середовищем;

- група людей, що мають певну формальну мету;

- група людей, що координують свої дії для досягнення організаційних цілей.

Структура сучасних організацій визначена в XVIII ст., у роботах французького гірського інженера Генрі Файола. Він запропонував ряд принципів менеджменту, які лежать і сьогодні в основі функціонування більшості організацій. Наприклад, розподіл праці; централізація; управління; дисципліна; ієрархічна структура; функціональне орієнтування; «стартова ніша» спеціальності; шлях рішень і просувань, який спрямований вертикально вгору зі «стартової ніші», нарівні з діями в рамках проекту, розподіленим на категорії відповідно до дисциплін і спеціальностей,

Основною характеристикою організації є її зрілість. Незрілу організацію характеризують такі аспекти:

- спеціальні процеси, імпровізовані їх виконавцями і керівництвом;
- процеси і правила, строго не дотримувані або не обов'язкові;
- високий ступінь залежності від розробників;
- можливість виникнення проблем, пов'язаних з цінами і графіками;
- невідповідність графіків функціональним властивостям, якості продукту або послуги;
- непередбачувана якість продукту. Характеристики зрілої організації такі:
- визначені, документовані **і** постійно удосконалювані процеси;
- документовані процеси, які є сумісними з фактичним способом виробництва;
- видима підтримка з боку керівництва і розробників;
- добре контрольована поведінка, що перевіряється;
- виконувані і використовувані вимірювання продуктів і процесів;
- виконувані технології.

У структурному аспекті відомо багато різних типів структур організацій, які варіюються навколо трьох основних тинів структур: функціональна, проектна, матрична.

Функціональна структура - припускає розділення співробітників за функціональними обов'язками.

Проектна структура - припускає розподіл структури за функціональними підрозділами, орієнтованими на виконання проекту.

Матрична структура - припускає розділення співробітників з функціональних підрозділів за виконуваними проектами з підпорядкуванням їх менеджерові матричного проекту.

4.4. Продукти

У загальному випадку продуктом може бути будь-який компонент (артефакт) або документи, що з'явилися в результаті виконання процесу.

Продуктами як складовими фаз життєвого циклу разом ***з*** кінцевим продуктом с результати кожної окремої фази життєвого циклу (вимоги, проект, реалізація, тести). Так само, як процеси, продукти поділяють на вертикальні і горизонтальні. До останніх належать результати виконання горизонтальних процесів.

4.4.1. Комплекти розробки

В уніфікованому процесі продукти як результати виконання процесів називаються робочими продуктами і утворюють комплек-ти. Розрізняються такі комплекти: вимог, проектування, реалізації, впровадження, управління (планування та експлуатації).

Представлення робочих продуктів у вигляді комплектів робить процес розробки керованим.

Комплект вимог. Для опису загальної концепції системи використовується структурований текст, що дає змогу докумен-тувати домен проекту. В основі тексту лежить контракт між особою, відповідальною за фінансування з боку замовника і групою розробників. Для додаткових специфікацій можна також використовувати спеціалізовані формати (наприклад, законодавчі вимоги), а також призначені для користувача макети прототипів, у яких містяться вимоги. Для робочого представлення моделей вимог використовується нотація UML**.** (моделі варіантів використання, моделі наочної області). Комплект вимог є головним робочим контекстом для визначення трьох інших комплектів, що стосуються розробки, і саме на його основі формуються текстові варіанти.

Робочі продукти з комплекту вимог оцінюються і вимірюються такими комбінаціями:

- несуперечність до специфікацій версій з комплекту управління;
- несуперечність між загальною концепцією і моделями вимог;
- несуперечність, повнота і смислова відповідність між інформацією з різних комплектів;
- аміни в поточній версії робочих продуктів вимог порівняно з попередніми версіями (тенденції до зменшення кількості дефектів і доопрацювань).

Комплект проектування. У проектних моделях, що описують тримувані рішення, використовується мова. У комплекті проектування представлені різні рівні абстракції, які відповідають різним компонентам з області рішень (їх індивідуальні властивості, атрибути, статичні зв'язки і динамічні взаємодії). У цих моделях міститься достатня кількість інформації про структуру і поведінку для визначення специфікації робочих продуктів (кількість і специфікації складових частин і матеріалів, витрати праці та інші прямі витрати). Інформація, що міститься в проектних моделях, може бути безпосередньо (часто автоматично) переведена в підмножину продуктів, що належить до комплектів реалізації і впровадження. Окремі продукти з комплекту проектування включають проектну модель, тестову модель і опис архітектури ПО (ту частину інформації з проектної моделі, яка мас відношення до опису архітектури).

Робочі продукти проектування оцінюються і вимірюються комбінацією таких чинників:

- внутрішня несуперечність і якість проектної моделі;
- несуперечність до моделей вимог;
- перехід у комплект реалізації і впровадження у відповідні нотації (наприклад, трасування, генерація початкового коду, компіляція, редагування зв'язків);
- несуперечність, повнота і смислова відповідність між інформацією з різних комплектів;
- зміни в поточній версії проектної моделі порівняно з попередніми версіями (тенденції до зменшення кількості дефектів і доопрацювань).

Ступінь автоматизації аналізу проектних моделей натепер є обмеженим, тому слід покладатися на аналіз, що виконується фахівцем.

Комплект реалізації включає початковий код, який є реалізацією компонентів (їх форму, інтерфейс і залежність), та всі потрібні для автономного тестування компонентів виконувані файли. Виконувані файли є простими складовими частинами, необхідними для створення кінцевого продукту, включаючи компоненти, створені на замовлення *(COTS)*, програмні інтерфейси *(API)* комерційних компонентів, а також *API* компонентів повторного використання або компонентів, наявних у початковій мові програмування. Робочі продукти комплекту також можна транслювати в підмножину комплекту впровадження (виконувані файли в остаточному вигляді). Конкретні робочі продукти включають самодокументований початковий код продукту і пов'язані з ним файли (сценарії компіляції, інфраструктура для управління конфігурацією, файл з даними), самодокументований тестовий початковий код і пов'язані з ним файли (файли з вхідними даними для тестування, файли з результатами тестування), виконувані файли для незалежного запуску компонентів і файли для проведення тестування компонентів.

Комплекти реалізації мають формати, придатні для читання людиною і оцінюються та вимірюються комбінацією таких чинників:

- несуперечність стосовно до проектних моделей;
- несуперечність і повнота різних комплектів робочих продуктів;
- оцінювання початкових або виконуваних файлів компонентів на відповідність певним критеріям за допомогою перевірок, аналізу, демонстрації або тестування;
- виконання тестових варіантів для незалежних компонентів з автоматичним порівнянням очікуваних результатів з отриманими;
- аналіз зміни в поточній версії компонента реалізації порівняно з попередніми версіями (тенденції до зменшення кількості дефектів і доопрацювань).

Комплект упровадження містить файли, що поставляються користувачеві, записи машинною мовою, сценарії збірки, сценарії інсталяції і дані, необхідні для використання продукту в тому середовищі, для якого він призначений. Записи машинними мовами Представляють компоненти продукту в кінцевому вигляді, службовцеві для розповсюдження серед користувачів. Уміст комплекту впровадження може бути інстальованим, виконаним відповідно до сценаріїв використання (протестовано) і динамічно переналаштовує для підтримки тих властивостей, які повинні бути в кінцевому продукті. Конкретні робочі продукти, які можуть бути потрібними в період виконання, оцінюються і вимірюються такою комбінацією:

- тестування сценаріїв використання і характеристик якості, визначених у комплекті вимог для оцінювання несуперечності, повноти і смислової відповідності між інформацією, що міститься в комплектах вимог і впровадженням;
- тестування стратегій розподілу, реплікації і розміщення сценаріїв використання, таких як: інсталяція, динамічна зміна конфігурації, основне застосування і управління в аномальних ситуаціях;
- тестування на відповідність описаних у керівництві користу-вача сценаріїв використання, таких як: інсталяція, динамічна зміна конфігурації, основне застосування і управління в аномальних ситуаціях;
- аналіз змін у поточній версії комплекту впровадження порівняно з попередніми версіями (тенденціями зменшення кількості дефектів, зміни в продуктивності).

4.4.2. Робочі продукти, пов'язані з тестуванням

Тестуючи, керуються підходом, у якому визначальним є документація. Команди розробників складають документацію з вимог, проектну документацію верхнього рівня і детальну проектну документацію до того, як починають створюватися початкові або виконувані файли. Команди розробників, провідні тестування складають документацію щодо планів тестування процедур тестування, планів інтеграційного тестування, планів тестування модулів і процедур, тестування модулів до початку Створення яких-небудь тестуючих

драйверів, заглушок або інструментів. Для такого підходу, з попереджуючим створенням документації, характерні ті ж проблеми тестування, що і при Проведенні розробки.

Одним з визначальних принципів сучасного процесу є застосування тих же комплектів, нотацій і робочих продуктів для продуктів тестування, які використовуються під час розробки самого продукту. Фактично лише визначається інфраструктура, потрібна для виконання тестування, як одна з обов'язкових підмножин кінцевого продукту. У процесі тестування виконуються певні правила, властиві періоду розробки:

- робочі продукти тестування повинні створюватися паралельно з продуктом від початку до впровадження, оскільки тестування - це діяльність, властива всьому життєвому циклу, а не тільки його пізнім етапам;
- робочі продукти тестування узгоджуються і створюються в рамках тих же комплектів, що і сам продукт;
- робочі продукти тестування реалізуються в програмованому і відтворюваному форматах;
- робочі продукти тестування документуються тим же чином, що і сам продукт;
- розробники тестів використовують ті ж інструменти, методи і процес навчання, що і розробники, котрі створюють основний продукт.

Модуль II
МОДЕЛІ ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ПРОЦЕСИ, ПРОДУКТИ, РЕСУРСИ

Розділ 5. ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розглядаються основні положення трьох інженерій програмного забезпечення - прямої, оберненої і емпіричної; показується взаємо- зв'язок двох інженерій програмного забезпечення, які нині часто використовуються разом. Розглядаються основні методи і засоби, які використовує пряма, обернена і емпірична інженерії.

5.1. Пряма інженерія

Пряма інженерія забезпечує процеси розробки програмного забезпечення з високорівневих абстракцій у вигляді специфі-кацій вимог і закінчуючи реалізацією програмного продукту у вигляді виконуваного коду. Як найповніше процеси прямої інженерії представлені в життєвому циклі програмного забез-печення (рис. 5.1).

Рис. 5.1. життєвий цикл програмного забезпечення

Доменний аналіз. Компоненти цієї фази життєвого циклу такі:

- процеси - орієнтовані на аналіз існуючого досвіду, нагромадженого в домені рішень з метою виділення елементів архітектури, коду, типів для подальшого використання їх у розробці;
- продукти - архітектура, код, тести, методи;
- ресурси - інструменти доменного аналізу, доменні експерти, доменні інженери.

Специфікування вимог. Компоненти цієї фази життєвого циклу такі:

- процеси - дії зі складання вимог до програмного забезпе-чення;
- продукти - специфікації вимог;
- ресурси - мови специфікацій, інженери вимог, комунікатори із з замовником.

Архітектурне і детальне проектування. Компоненти цієї фази такі:

- процеси - орієнтовані на створений архітектури І детального проекту;

- продукти - архітектура і детальний проект програмного забезпечення;

- ресурси - CASE, архітектура і системні програмісти. *Кодування і тестування.* Компоненти цієї фази такі:

- процеси - кодування і тестування програм;

- продукти - програми;

- ресурси - засоби програмування, програмісти і тестери. *Супровід.* Компоненти цієї фази такі:

- процеси - що коригують, адаптують, удосконалюють і онов-люють супровід. Коригуючий супровід - зміна програмного забезпечення з мстою виправлення помилок, допущених на попередніх фазах життєвого циклу. Адаптуючий супровід - зміна програмного забезпечення у відповідь на зміни навколишнього середовища. Вдосконалюючий супровід - зміна програмного забез-печення задля вдосконалення його властивостей. Відновлюючий супровід - зміна програмного забезпечення задля відновлення його працездатності;

- продукти - супроводжуване програмне забезпечення;

- ресурси - засоби програмування, програмісти, інженери з супроводу.

Ліквідація. Компоненти цієї фази такі:

- процеси - відновлення, переробка, повторне використання І знищення програмного забезпечення. Відновлення - це віднов-лення працездатності програмною забезпечення. Переробка - це реінженерія або міграція програмного забезпечення. Повторне використання - це виділення з програмного забезпечення частин компонентів, які можна використовувати знову в розробці нового програмного забезпечення.

Знищення - це знищення неутилізованих залишків програмного забезпечення;

- продукти - повторно використані компоненти;

- ресурси — екстрактори, програмісти, експерти.

5.1.1. Інструменти прямої інженерії

Computer Aided Software Environment (CASE) - це інструментарій, призначений для автоматизації виконання процесів життєвого циклу програмного забезпечення. *CASE* відіграє, таку ж роль у програмно-му забезпеченні, як *CAD/CAM в* інженерії фізичних систем. Функці-онування *CASE* ґрунтується на моделі процесів життєвого циклу (рис. 5.2).

Рис. 5.2. Зв'язок CASE моделі процесів ;життєвого циклу

До того ж, використання *CASE* в організації може розглядатися як шлях до отримання несуперечливих, повторюваних і визначува-них процесів. Це веде до того, що визначення *CASE* може виводити організацію на другий (повторюваний) або третій (визначуваний) рівень зрілості по моделі *CMML*

Таким чином, *CASE* повніша орієнтуватися на виконання всіх процесів життєвого циклу, що задаються відповідною моделлю (рис. 5.3.)

Рис. 5.3. Процеси, що автоматизують CASE

Крім моделі процесів розробки програмного забезпечення CASE повинна включати інструменти розробки і модель програмного продукту (рис. 5.4).

Рис. 5.4. Модель основи CASE

Очевидно, що *CASE* належать до інструментів горизонтального типу. Прикладами *CASE* є *IBM Rational, Doors (Telelogic).*

5.2. Обернена інженерія

Виконання процесів супроводу програмного забезпечення і ви-ділення з нього програмних компонентів призвело до необхідності реконструювання програм і розробки відповідного розділу про-грамної інженерії, який називається реверсивною *(reverse)* або оберненою (*backward)* інженерією. Традиційний, низхідний підхід до розробки програмного забезпечення (від вимог до коду) назива-ється прямим або інженерією вперед (*forward).*

Завдання оберненої інженерії протилежне прямій і полягає в за-безпеченні процесів низькорівневого представлення програмного забезпечення (частіше початкового і рідше об'єктного коду), високо рівневого його уявлення, наприклад, проектної інформації або специфікацій вимог

Загалом, обернена інженерія забезпечує два такі процеси:

- ідентифікацію системних компонентів і відношень між ними;
- створення високорівневих представлень компонентів і програмного продукту в цілому.
- Тому в оберненій інженерії доводиться вирішувати два завдання;
- вибір відповідного рівня представлення абстракцій, стандартів і уявлень дня інформації про програмну інженерію;
- створення інструментів, що полегшують розпізнавання відповідної інформації в існуючому початковому коді.

Досвід показує, що тієї інформації, яка є в низькорівневому пред-ставленні програмного забезпечення, як правило, недостатньо для по-будови високорівневого опису, тому процеси оберненої інженерії складні і потребують значного інформаційного і інструментального забезпечення,

На рис. 5.5 показано принципову відмінність процесів прямої і оберненої інженерії. Якщо для процесів прямої інженерії в разі створення програмного забезпечення характерне цілеспрямоване звуження області ухвалюваних рішень, то для процесів оберненої інженерії характерне розширення області рішень, що виводяться, яку постійно доводиться звужувати для того, щоб вийти на такс високорівневе уявлення програмного забезпечення.

Рис. 5.5. Відмінність прямої та оберненої інженерії

Потреба в оберненій інженерії натепер виникає в трьох випадках:

- у разі створення компонентів з існуючого програмного забезпечення;
- під час відновлення програмного забезпечення, наприклад, у процесі супроводу;
- у процесі переробки програмного забезпечення, наприклад, під час міграції.

Обернена інженерія не веде до зміни наявного програмного забезпе-чення і використовується лише, для того, щоб тримати інформацію про його низькорівневі уявлення. Тому за винятком декількох завдань (на-приклад, завдання розуміння програмного забезпечення) обернена ін-женерія зазвичай використовується у поєднанні з методами прямої ін-женерії,

5.2.1. Методи оберненої програмної інженерії

Місце методів, використовуваних у рамках оберненої інженерії в життєвому циклі, показано на рис. 5.6.

Рис. 5.6. Методи оберненої інженерії

До цих методів належать такі;

- відновлення проектної інформації;

- реструктуризація;
- редокументування;
- реінженерія.

Поняття реверсивної інженерії стосовно технічних систем викори-стовується для визначення процесів розробки специфікацій системи шляхом дослідження її задля створення безлічі подібних систем.

Стосовно програмного забезпечення основні цілі оберненої Ін-женерії полягають не в створенні дубліката системи, а в отриманні інформації для кращого розуміння системи, щоб підвищити ефек-тивність супроводу, переробити систему або виділити з неї певні компоненти, що відповідають заданим вимогам.

Відновлення проектної інформації (design recovery) - це метод оберненої інженерії, у якому разом з початковим кодом під час від-новлення проектної інформації використовуються всі доступні ві-домості про систему: проектна документація, досвід розробників і експлуатаційників, знання про домен. Головна мета відновлення проектної інформації - розроблення структур, які допоможуть ін-женерові програмного забезпечення зрозуміти програму або прог-рамну систему. Отже, кінцевою метою е не специфікація вимог, яка відповідає аналізованому початковому коду, а проектна інфор-мація.

Реструктуризація (restructuring) - це метод трансформації прог-рамного забезпечення на одному рівні його уявлення шляхом вико-ристання інформації, котру отримали в процесі виконання реверси-вної інженерії. Трансформація не приводить до зміни первинних вимог до програмного забезпечення (наприклад, реструктуризація - це перетворення неструктурної форми коду в структурну).

Редокументування (redocumentaiion) - це метод створення або перегляду семантично еквівалентних уявлень програмного забез-печення в рамках одного і того ж рівня. Прикладом може слугувати створення діаграм управління, описи структури програмного за-безпечення у формі зручної для сприйняття людиною. Ключова роль цього методу полягає в тому, щоб забезпечити візуалізацію відношень, що мають місце між програмними компонентами для того, щоб розпізнати їх і управляти ними,

Реінженерія (reengineering) - це метол зміни програмного за-безпечення шляхом використання методів прямої інженерії на ос-нові відновленої (за допомогою оберненої інженерії) проектної ін-формації. До того ж, реінженерія веде до зміни системних і функ-ціональних вимог програмного забезпечення і є методом ного пе-реробки.

5.2.2. Інструменти оберненої інженерії

Усі інструменти оберненої інженерії утворюють інтегроване се-редовище - *Computer Aided Reverse Software Environment (CARSE).* Загальну архітектуру середовища зображено на рис. 5.7.

Рис. 5.7. Архітектура інструментів оберненої інженерії

5.3. Емпірична інженерія програмного забезпечення

Емпіричні методи досліджень відіграють «впливову» роль в ін-женерії програмного забезпечення і їх застосування складають од-ну з інженерій - емпіричну інженерію програмного забезпечення.

На відміну від прямої та оберненої інженерії мста емпіричної інженерії - не розробка або переробка програмного забезпечення, а здобуття знань про програмне забезпечення. Тому її основу складають два кола методів та засобів. Перше пов'язане із збиранням інформації щодо властивостей програмного забезпечення. Переваж-но це робиться шляхом застосування вимірювань. Друге складають методи та засоби обробки нагромадженої

інформації і здобуття знань стосовно програмного забезпечення, що досліджується.

5.3.1. Методи емпіричної інженерії програмного забезпечення

Головним методом досліджень програмного забезпечення є вимірювання. Для контролю процесів, продуктів та ресурсі в життєвого циклу програмного забезпечення слід використовувати величини характеризуючи їх властивості, що називаються метриками.

Величина - це певна властивість предмета, з якою можна зіста-вити значення. Для синтезу величини варто визначити властивість (семантику величини), систему значень (шкалу) та спосіб зістав-лення значень з величиною.

У теорії вимірювання виділяють три основні шкали вимірювань - номінальну, порядкову і кількісну. Номінальна (класифікаційна) шкала включає значення, що проявляє себе лише у відношенні еквівалентності або може бути зіставлена з властивістю предмета (не упорядкованих один стосовно іншого). Наприклад, можна зістави-ти з вихідним текстом програми величину «мова програмування», значенням якої може бути назва однієї з мов (наприклад, «С», «С++», «Pascal», «Java» тощо). Такий же тип має шкала класифі-кування призначення модулів програмного забезпечення (наприк-лад, «Бази даних», «Математичні пакети», «Операційні системи» тощо). До номінальних величин застосовується тільки операція пе-ревірки на еквівалентність. Порядкова (ординальна) шкала спосте-рігає за упорядкуванням одного значення стосовно іншого, до яких належать операції порівняння. Порядкову пікапу можна задати для більшості експертних оцінок, наприклад, оцінювання читабельнос-ті тексту програм - «незадовільно», «задовільно», «добре», «від-мінно» або для оцінювання рівня інкапсуляції програмних компо-нентів - «лексичний», «операторний», «процедурний», «класний», «модульний». Кількісна шкала включає в себе значення, що про-явили себе стосовно еквівалентності, порядку і адекватності. Такі величини дають змогу виковувати адекватні і мультиплікативні операції над значеннями (віднімання, множення, ділення). До них належать, наприклад, такі кількість рядків коду, складання коментарю, оцінювання трудозатрат на створення коду.

Очевидно, що велику цінність являють собою кількісні (адитив-ні) величини, оскільки вони не тільки відображають властивість програмного забезпечення, які можна використовувати для оброб-ки і аналізу, а й мають більш повиний набір операцій над значення-ми. Однак для використання величини з більш високою інкалою вимірювань слід мати достатні знання про характер відношень між цими значеннями.

Як правило, значущу шкалу вдається досить просто задати для величин, маючи вузький, добре інтерпретований зміст. Очевидно, що перехід від номінальної шкали до порядкової і від порядкової до кількісної потребує підвищення знань про характер відношень між значеннями величин. Ці знання мають імперичну природу і з'являються шляхом досвідченого виявлення залежно від значення.

Отримане значення величини, виявлене стосовно еквівалентно-сті, зводиться до завдання класифікації стану об'єкта вимірювання, що визначається сукупністю ознак, значення яких дають змогу іден-тифікувати кожен стан. Наприклад, під час визначення мови програмування, яку використано для написання програмного модуля («с», *«cpp», «h», «pas», «ada», «htm»). Як* датчик ознаки має висту-пати прилад, що виділяє розширення з імені файла. Вирішальний пристрій має реалізувати виявлення значень на основі правил виду: «if», «розширення» = «.pas» or «.dpr», zen значення = «Pascal». Для тримання значення номінальної величини достатньо, щоб вибрані ознаки також мали властивості еквівалентності.

Для величин, які виявили себе стосовно адитивності, значення дорівнює числовій оцінці сумарної виміряної величини Nx_e, що ви-никла в результаті складання порівнюваних однорідних величин, і має дорівнювати сумі числових оцінок цих порівнюваних, а сума іменованих чисел x_{Nj} відображає порівняння, що повинне дорівню-вати іменному числу x_{Ne} відображаючи сумарну величину:

;

,

тоді

і $q_x = q_{xi}$ при будь-якому i ,

де q_x - значення величини, що має числове значення 1.

Наявність властивості адитивності у величині дає змогу використовувати для визначення значення міру, яка забезпечує відновлюваність величини заданого розміру.

Оскільки тексти програм мають дискретну природу, то для виз-начення значень адитивності величини необхідно мати одиничну міру, що дорівнює кванту величини і пристрій для додавання міри та рахунку квантів. Наприклад, під час вимірювання довжини прог-рами, як кількісна міра може вважатися рядок, а як вимірювальний пристрій - сканер тексту, підраховуючи кількість рядків,

У випадку номінальних і порядкових величин значення виявля-ється певною функцією від показників, які також є величинами:

,

де V- номінальна чи порядкова величина; $v_1 ....v_n$ - величини по-казників.

Оскільки властивості номінальних і порядкових величин не за-довольняють вимоги вимірювань, то для реалізації процесу вимі-рювання як показники, можна вважати лише адитивні величини.

Основними частинами статистичного аналізу стосовно програм-ного забезпечення можна вважати первинний статистичний аналіз, кореляційний аналіз та регресійний аналіз (рис. 5.8).

Первинний статистичний аналіз - не визначення закону розпо-ділу випадкової величини. На етапі первинного статистичного ана-лізу досліджуються вхідні статистичні дані. У ході дослідження спочатку виявляється графічний вигляд (гістограма) закону розпо-ділу. Для уточнення законів розподілу визначаються статистичні характеристики, такі як: математичне сподівання, середнє квадра-тичне відхилення, коефіцієнти асиметрії та ексцесу. На основі зна-чень математичного сподівання проводиться вилучення аномаль-них явищ (відхилень), при якому за допомогою квантилів розподі-лу Стьюдента визначаються «грубі» значення, тобто такі, що не потрапляють під заданий закон розподілу і значно віддалені від математичного сподівання. Після видалення аномальних явищ об-числюються коефіцієнти асиметрії та ексцесу.

Рис. 5.8. Схема використання статистичного аналізу

Далі всі статистичні характеристики обчислюються у зсуненому та в незсуненому виглядах. Зсунені дані являють собою обчислені результати вимірів, незсунені - теоретичні значення, що повинні приймати характеристики за «нормальності» розподілу. Потім про-водиться інтервальне оцінювання параметрів. Для всіх отриманих значень, що пройшли попередній етап, проводиться порівняння коефіцієнтів асиметрії та ексцесу із заданим теоретично. Після цьо-го на основі гістограм та висновків аналізу числових характеристик робиться висновок про закон розподілу величини.

На цьому етапі дослідник програмного забезпечення може отри-мати декілька результатів для подальшого використання.

Кореляційний аналіз пар метрик проводиться таким чином. Спочатку, за визначеними раніше законами розподілу, всі дослі-джувані значення класифікуються на ті, що мають нормальний розподіл і ті, що його не мають. Для пар метрик, що мають нормаль-ний закон розподілу, проводиться просте визначення коефіцієнта кореляції та його оцінювання. Якщо

коефіцієнт кореляції дорівнює нулю, ніякого зв'язку в парі немає. У разі знаходження коефіцієнта кореляції між мінус 1 і плюс 1, наявний лінійний регресійний зв'язок. Якщо ж коефіцієнт кореляції дорівнює 1, то має місце функ-ціональний зв'язок. Далі проводиться визначення значущості кое-фіцієнта кореляції (висувається гіпотеза, що коефіцієнт кореляції дорівнює 0), при якому використовується t - тест на основі статис-тичної характеристики, яка має t розподіл Стьюдента. Якщо це значення значущості менше, ніж задане табличне, ця пара відсію-ється з подальших досліджень. У разі значущості проводиться дос-лідження на довірчі інтервали. Під час потрапляння коефіцієнта в довірчі інтервали можна зробити висновок про те, що досліджувані величини мають між собою лінійну регресійну залежність. В іншо-му випадку вони відсіюються.

Для пар досліджуваних величин, які не мають нормального за-кону розподілу, проводиться парна рангова кореляція. Суть парної рангової кореляції полягає в порівнянні не самих значень величин чи їх статистичних характеристик, а рангів, тобто номерів дослі-джуваних величин у відповідних матрицях (наборах статистичних даних). Визначається парна рангова кореляція методом обчислення коефіцієнта Спірмена чи Кендала. Якщо значення коефіцієнта ви-явилось рівним 0. то робиться висновок про відсутність кореляції і пара досліджуваних величин «відкидається». Якщо коефіцієнт кореляції набуває значення 1, чому відповідає повний збіг коефіцієн-тів, то робиться висновок про прямо пропорційну залежність (тобто лінійну), якщо мінус 1, то робиться висновок про обернено пропорційну залежність (тобто також лінійну). Якщо ж коефіцієнт коре-ляції набуває іншого значення, то далі його перевіряють на значущість, перевіряючи гіпотезу, що коефіцієнт дорівнює 0.

Отже, результатом цього етапу с відсіювання незалежних між собою пар досліджуваних величин та визначення за можливістю виду залежності для інших пар.

Регрессійний аналіз залежних величин - останній етап у дослі-дженні залежностей. Спочатку проводиться ідентифікація регресії. Вона передбачає як графічну побудову, так і аналітичні досліджен-ня. Рис. 5.9. Кореляційні поля: *а* - вписується в коло; *б*- вписується в еліпс (спадного вигляду); в - вписується в еліпс (вихідного вигляду); *г*- складної конфігурації

Графічна побудова розпочинається з визначення кореляційного поля. Приклади кореляційних полів показано на рис. 5.9.

Рис. 5.9. Кореляційні поля: а - вписується в коло; б- вписується в еліпс (спадного вигляду); в - вписується в еліпс (вихідного вигляду); г- складної конфігурації

Якщо кореляційне поле мас форму еліпса, робиться висновок про лінійний регресійний зв'язок. Далі проводиться побудова лі-нійної регресії і її оцінювання. Якщо побудовані точки кореляційного поля потрапляють у коло, то робиться висновок про відсут-ність залежності. Якщо ж кореляційне поле не вписується в коло чи еліпс, а має інший вигляд, то робиться висновок про нелінійну за-лежність у лінії регресії. Потім будуються і аналізуються найімовір-ніші наближені лінії регресії. Серед них вибирається найточніша шляхом обчислення відхилення значень залежної Змінної Висно-вок про найточніше припущення робиться для функції, у якої від-хилення найменше. Для нелінійної залежності проводиться лінеа-ризація коефіцієнтів, тобто зведення функції до лінійного вигляду.

Завершальним етапом є довірче оцінювання ліній регресій [8]. Довірче оцінювання регресії відбувається в декілька етапів. Пер-шим етаном є визначення коефіцієнта детермінації, який показує ступінь залежності між величинами. Далі проводиться оцінювання відхилення окремих значень залежної величини від емпіричної рег-ресії шляхом порівняння практичних та теоретичних значень зале-жної змінної. Останнім кроком є побудова довірчого інтервалу лінії регресії.

Якщо пара пройшла всі етапи і не була відсіяною, робиться вис-новок, що одна метрика залежить певним чином від іншої з силою, що показує коефіцієнт детермінації, а вигляд

залежності визначає лінія регресії.

5.3.2. Засоби емпіричної інженерії програмного забезпечений

Для реалізації емпіричних методів в інженерії програмного за-безпечення створюються спеціальні середовища - *Computer Aided Empirical Software Engineering (CAESE)* (рис. 5.10). Як видно, *CAESE* забезпечує вивчення проблем, пов'язаних з програмним за-безпеченням на основі емпіричних даних, отриманих шляхом про-ведення експериментів.

Рис. 5.10 Структура CAESE

5.4. Взаємозв'язок інженерій

Взаємозв'язок прямої і оберненої інженерії у вигляді програмного забезпечення показано на рис. 5.11. Видно, що оберненій інженерії відводиться інформаційна роль, наприклад, для формування депози-тарію для інформації про програмне забезпечення.

Рис. 5.11. Обіг програмного забезпечення

На основі взаємодії обох інженерій будується методологія розро-бки і супроводу програмного забезпечення. Суть її полягає в тому, що з наявного програмного забезпечення застосуванням методів оберненої інженерії будується репозитарій проектних вирішень домена, відтак на основі репозитарію методами прямої інженерії ство-рюються нові застосування.

Розділ 6. ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. МОДЕЛЮВАННЯ

Стандарт *ISO /IEC* 12207:1995 визначає модель життєвого циклу як схему, що відображає процеси, дії і завдання, які залучаються до розробки, експлуатації і супроводу програмного продукту, почи-наючи з визначення вимог і закінчуючи зняттям з експлуатації.

Головні цілі моделювання життєвого циклу полягають у тому, щоб, абстрагуючись від деталей, визначити склад і порядок вико-нання фаз, а також критерії переходу від однієї фази до іншої.

Першими моделями життєвого циклу були такі: «кодуй і вип-равляй (code-and-fix*);* «крокова» (stage wise*);* «водоспад» *(waterfall).*

6.1. Базові моделі

Модель «кодуй і виправляй» містила дві фази (рис. 6.1):

— написання коду;

- встановлення помилок у коді.

Недоліки моделі: після безлічі змін код ставав погано структурований; навіть добре спроектоване програмне забезпечення не відповідало вимогам; супровід коду був дорогим, оскільки код по-гано пристосований для тестування і модифікації.

Рис. 6.1. Модель «кодуй і виправляй»

Крокова модель була розроблена на основі моделі «кодуй і вип-равляй» шляхом усунення недоліків, значної деталізації кроків роз-робки програмного продукту.

Модель, що передбачає послідовне виконання наступних кроків, така (рис. 6.2):

планування дій, специфікування дій, кодування, тес-тування, асемблювання, *shakedown*, оцінювання розробленої сис-теми. Основні недоліки моделі полягали в тому, що вказані кроки ви-конувалися у строїти послідовності (був відсутній зворотний зв'язок) і не передбачалося швидке прототипування програм (рис. 6.3). Удоско-наленням крокової моделі була каскадна модель, оскільки вона забез-печувала:

Рис. 6.2. Крокова модель

- зворотні зв'язки між стадіями (тим самим зменшувалася переробка програмних продуктів окремих стадій);
- прототипування як спосіб розробки програмного забезпечення двічі;
- введені фази проектування;
- кожна фаза продукту проходить верифікацію, валідацію або тестування.

Проте головний недолік каскадної моделі - це обов'язкове завер-шення фаз специфікації вимог і проектування перш, ніж може бути продовжено виконання інших фаз життєвого циклу. Якщо для окремих класів програмних систем, наприклад, операційних сис-тем цей підхід був ефективний, то для широкого класу інтерактив-них застосувань він не працював. Це зумовило необхідність розроб-ляти інші моделі життєвого циклу (рис. 6.3).

Рис. 6 .3 . Каскадна модель

Каскадна модель не знайшла практичного застосування, проте виявилася важливою теоретичною базою для розробки моделей інших типів, а також стандартів, Наприклад, такі особливості тех-нологій, як інкрементна і паралельна розробка, сімейство програм, еволюційні зміни, формальна розробка і верифікація, ризик-аналіз були вперше введені як розвиток каскадної моделі.

6.2. Типи моделей життєвого циклу

Натепер моделі життєвого циклу можна поділити на три групи. Основою моделей усіх груп є каскадна модель.

Першу групу утворюють «послідовні» моделі - модифікації каскад-ної моделі орієнтування на розробку «з нуля». До цієї групи входять:

класична, водоспад, спіральна, інкрементна, покрокова, модель швидкої розробки, модель прототипування, еволюційна модель.

Розробка нових методів побудови програмного забезпечення, заснованих на багатократному і повторювальному використанні, призвела до появи другої групи моделей, До її складу входять моделі компонентної розробки і моделі, засновані на повторному використанні.

Одним із шляхів підвищення ефективності розробки ПЗ є усу-нення фаз життєвого циклу шляхом автоматизації відповідних процесів, У зв'язку з цим з'явилася третя група моделей автоматично-го синтезу програмного забезпечення.

6 .2.1. Моделі, орієнтовані на розробку «з нуля»

Класична модель водоспаду - це узагальнення моделі водоспаду Райса, Модель містить п'ять фаз і, зазвичай, використовується в теоретичних побудовах (рис. 6.4).

Рис. 6.4. Класична модель водоспаду

Спіральна модель запропонована Боемом, як уточнення моделі водоспаду в результаті виконання ряду проектів, Процес розробки пред-ставлений у вигляді спіралі. Кожен виток

спіралі - фаза (рис. 6.5).

Рис. 6.5. Спіральна модель

Спіраль розташована в чотирьох квадратах. У кожному квадраті виконуються певні дії:

- квадрат 1 - визначаються цілі альтернативи і обмеження - визна-чення вимог і специфікація для критичних частин системи з погляду продуктивності, функціональних властивостей, здібності до акомодації змін, програмного/апаратного інтерфейсу, критичних чинників успіху;

- квадрат *2* - розробляється прототип, ідентифікуються і вирі-шуються ризики - визначення вимог і специфікацій для найбільш потенційно небезпечних частин уявної системи задля виконання оцінювання і визначення ступеня ризику; розділення на окремі час-тини відповідно до ступенів ризику;

- квадрат 3 — розробляється продукт;

- квадрат **4** - планується така фаза: застосування Інформації, що належить до фази розробки продукту на наступному рівні, до планування на наступному кроці фази проекту.

Таким чином, у відповідному квадраті відбуваються такі дії: планування, прототипування, конструювання, оцінювання замов-ником і планування наступних дій.

Вертикальна вісь показує накопичувану вартість, а горизон-тальна - прогрес у розробці продукту.

Інкрементна модель забезпечує процеси побудови версії програм-ного забезпечення, що розробляється (рис *6.6).* Модель об'єднує кас-кадну модель з ітераційним підходом до розробки програмного за-безпечення, Інкрементна модель може комбінуватися з іншими.

Рис. 6.6. Інкрементна модель

Покрокова модель припускає визначення пріоритетних функцій розробки програмного забезпечення у відповідній фазі життєвого циклу. Відповідно до певних пріоритетів визначається кількість кроків і кожна фаза реалізується покроково (рис. 6-7). Після вико-нання кроку виходить програмний продукт, який передається замов-никові для експерименту і перевірки.

Модель швидкої розробки (рис. 6.8) введена у використання фір-мою IBM. Суть моделі полягає в такому:

- користувач програмного забезпечення бере участь у всіх фазах життєвого циклу;

- скорочується час переходу від вимог до створення повного програмного забезпечення за рахунок використання відповідних інструментальних засобів і повторного використання.

Рис. 6.7. Покрокова модель

Рис. 6.8. Модель швидкої розробки

Модель прототипування забезпечує створення програмного за-безпечення у двох примірниках. Перший примірник називається прототипом і використовується для вимог. Після того, як вимоги узгоджені, прототип викидається і програмне забезпечення ство-рюється наново (рис. 6.9). Гасло моделі "давайте будувати двічі»,

Еволюційна модель - розробляється перша версія програмного продукту, яка передасться замовникові. Потім вона доопрацьову-ється і знову передасться замовникові, І так до тих пір, доки не бу-де побудована остаточна версія продукту. Еволюційна модель за-безпечує еволюційний процес розробки програмного забезпечення (рис 6. 10),

Рис. 6.9. Модель прототипування.

Рис. 6.10. Еволюційна моделі.

V-модель життєвого циклу була введена для ідентифікації дій, пов'язаних з тестуванням на всіх стадіях розробки програмного продукту. Лівий бік моделі (рис. 6.11) містить традиційні фази кас-кадної моделі, проте окрім робочого продукту виробляється відпо-відний тест. Правий бік моделі пов'язаний з інтеграцією і тесту-ванням.

Рис. 6 ***. 11. V*** *- модель*

W-модель життєвого циклу є модифікацією V-моделі і реалізує метод, відповідно до якого результат кожної фази перевіряється на коректність, змістовність і завершеність (рис. *6.* 12). Суть моделі полягає у проведенні аудиту, перегляданні і тестуванні робочих продуктів, які здійснюються паралельно з виконанням фаз.

Рис. 6. 12. W - модель

Стадійна модель життєвого циклу. Ця модель є модифікацією еволюційної моделі (рис, 6.13). Суть її полягає в розгляді супрово-ду розробленого програмного продукту як процесу розробки.

Рис. 6 . 13. Стадійна модель

Моделі, орієнтовані на використаних готових компонентів. Розрізняють дві групи моделей:

- засновані на застосуванні компонентів багатократного використання;
- засновані на повторному використанні успадкованого програм-ного забезпечення.

Моделі компонентної розробки орієнтуються на багатократне використання готових компонентів, наприклад, методом *об* 'ектно-орієнтованого програмування. Для розроблення компонентів пе-редбачається три можливості (рис. 6.14):

- розробка «з нуля»;
- використання існуючих класів;
- повторне використання успадкованого програмного забезпе-чення.

Рис. 6.14. Компонентна модель розробки

Існує три типи моделей, заснованих на повторному використан-ні; швидка, ітеративна і повна.

Швидка модель передбачає розробку шляхом зміни *коду* успад-кованого програмного продукту з подальшою зміною інших робо-чих продуктів фаз життєвого циклу (рис. 6.15).

Рис. 6.15. Швидка модель

Ітеративна модель припускає аналіз успадкованого програм-ного забезпечення і побудову нового продукту шляхом послідо-вних змін робочих продуктів успадкованого програмного проду-кту (рис. 6.16).

Рис. 6.16. Ітеративна модель

Повна модель передбачає побудову на основі успадкованого програмного продукту

репозитарія повторно використовуваних компонентів і потім створення з його допомогою нового програм-ного продукту (рис. 6.17).

Ряс. 6.17. Повна модель

Моделі, орієнтовані на автоматичне виконання фаз життє-вого циклу. Моделі автоматичного синтезу забезпечують автома-тичну побудову програмного продукту шляхом переходу від не-формальної специфікації до формалізованої специфікації завдя-ки автоматичному виконанню однієї або декількох фаз життєво-го циклу (рис. 6.18).

Рис. 6.18. Модель автоматичного синтезу

6.2.2. Вибір моделей життєвого циклу

Зазвичай для кожного проекту програмного забезпечення виби-рають моделі життєвого циклу. Під час вибору використовуються характеристики таких елементів розробки:

- вимога;
- команда розробників;
- колектив користувачів;
- тип проекту і ризики.

Після вибору моделі здійснюється її підлаштування під процеси конкретного проекту.

Розділ 7. МОДЕЛІ, МЕТОДИ І ЗАСОБИ ОЦІНЮВАННЯ ВАРТОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розвиток моделей, методів і засобів оцінювання вартості прог-рамного забезпечення (ПЗ) сягнув рівня практичного застосування. Проте через відсутність інформації, засобів і фахівців зазначене не використовують під час розробки ПЗ в Україні.

У розділі наводяться результати аналітичного огляду літератури з урахуванням досвіду авторів ПЗ за вказаною темою. Розділ складаєть-ся з трьох частин. У першій розглядаються одиниці розміру ПЗ, які використовуються в моделях, методах і засобах оцінювання вартості ПЗ, у другій - моделі *і* методи оцінки, у третій - засоби оцінки.

7.1. Одиниці розміру програмного забезпечення

Під час оцінювання вартості ПЗ використовують дві одиниці розміру: рядок коду (Line of Code - LOC) і функціональну крапку (Function Point - FP).

Line of Code - це рядок початкового коду ПЗ (виключаються порожні рядки, коментарі і специфічні оператори). До переваг ви-користати! LOC, як одиниці розміру ПЗ, відносять простоту, а не-доліками є: розмір проекту в LOC може бути визначений лише піс-ля його завершення; LOC залежить від мови програмування; LOC не враховує якість коду. Продуктивність (S) програміста з викори-станням LOC розраховується ПЗ за такою формулою:

,

де n - кількість рядків коду, написаних програмістом (LOC); m -час роботи програміста (у людино-годинах). Видно, що чим більше рядків коду, тим вище продуктивність розробника. Проте можна реалізувати одну і ту ж функцію, написавши меншу кількість ряд-ків коду. Одиниця розміру LOC не відображає функціональні влас-тивості коду. Тому, якщо розробник прагне оптимізувати процес розробки задля зменшення трудовитрат На реалізацію проекту, то при використанні LOC як основної одиниці розміру проекту під

зменшенням трудовитрат мається на увазі Зменшення кількості рядків коду в програмі, при цьому не оцінюється його функціональ-ність.

Існують також проблеми із застосуванням LOC і в проектах, що використовують декілька мов програмування. Наприклад, 10.000 LOC мови C++, очевидно, не можна порівнювати з 10.000 LOC мо-ви COBOL, а в разі застосування автоматизованих або заснованих на шаблонах візуальних засобів розробки розрахунок LOC тим менш ефективний, ніж більше коду створюється автоматично.

Function Point була введена як альтернатива LOC. Методика аналізу функціональних точок була розроблена А. Дж. Альбректом (A. J. Albrecht) для компанії IBM у середині 70-х років XX ст., коли виникла потреба и підході до оцінювання витрат праці на розробку ПО, який би не залежав від мови і середовища розробки. З 1986 р. просування методики і розробку відповідного стандарту продовжує International Function Point User Group (IFPUG). Ця організація розробила керування програмним забезпеченням на практиці засто-сування розрахунку функціональних точок (Function Point Counting Practices Manual - FPCPM) і остання версія цього документа (4.1) офіційно визнана ISO як стандарт оцінювання розміру ПЗ.

Методика аналізу FP грунтується на концепції розмежування взаємодії. Суть її полягає в тому, що програма розділяється на кла-си компонентів ПЗ формату і типу логічних операцій. В основі цьо-го поділу лежить припущення, що область взаємодії програми роз-діляється на внутрішню - взаємодія компонентів додатка, і зовніш-ню - взаємодія з іншими застосуваннями.

Відповідно до прийнятого стандарту використовуються п'ять класів компонентів, на яких грунтується аналіз:

- внутрішній логічний файл (Internal Logical File - ILF) — група логічно пов'язаних даних, меж додатка, що знаходяться всередині, і підтримуваних введенням ззовні;
- зовнішній інтерфейсний файл (External Interface File - EIF) -група логічно пов'язаних даних, меж додатка, що знаходяться ззов-ні, і що є внутрішнім логічним файлом для іншого застосування;
- зовнішнє введення (External Input - EI) - транзакція, в разі ви-конання якої дані перетинають межу додатка ззовні. Це можуть бути як дані, що отримуються від іншого застосування, так і дані, що вводяться в програму користувачем. Отримувані дані можуть бути командами управління або статичними даними. В останньому випадку може виникнути необхідність відновити внутрішній логіч-ний файл;
- зовнішній вивід (External Output - EO) - транзакція, при вико-нанні якої дані перетинають межу додатка зсередини. З ILF і EIF створюються файли виведення або повідомлення і відправляються іншому застосуванню. Виведення також містить похідні дані, що отримуються з ILF;
- зовнішній запит (External Inquiry -EQ) - транзакція, в разі ви-конання якої відбувається одночасне введення і виведення. У ре-зультаті інформація вертається з одного або більше ILF і EIF, Ви-ведення не містить похідних даних, а ILF не оновлюється.

Класи компонентів оцінюються ПО за складністю і належать до категорії високого, середнього або низького рівня складності. Для транзакцій (EI, EO, EQ) рівень визначається ПО за кількістю фай-лів, на які посилається транзакція (File Types Referenced — FTR) і кількості типів елементів даних (Data Element Types - DET). Для ILF і EIF мають значення типи елементів записів (Record Element Types - RET) і DET. Типи елементів записів - цс підгрупа елементів даних у ILF або EIF. Типи елементів даних - це унікальне, не реку-рсивне поле підмножини ILF або EIF. Рівні складності і відповідні ним значення FTR і DET описані в FPCPM.

Наприклад, для EI з кількістю FTR від трьох і більше, і DET(від 5 до 15) рівень складності визначається як високий. Далі компонен-ти розподіляються ПЗ за «ваговими категоріями» залежно від рівня їх складності. Наприклад, ILF середньої складності має значення 10, а EQ високої складності - значення 6. Після цього, робиться розрахунок

нескоригованих функціональних точок (Unadjusted Function Point - UFP) ПЗ у відповідній формулі [1]:

,

де N_{ij}- і W_{ij} - кількість екземплярів класу і складності j і ного вагоме значення відповідно. Результат розрахунку може бути скоригова-ний за допомогою чинника регулювання вартості (Value Adjustment Factor - VAF). Під час розрахунку VAF враховуються чотирнадцять загальних характеристик системи (General System Characteristic -GSC), які оцінюють загальну функціональність застосування, що розробляється. Ці характеристики відображають можливість пов-торного використання коду, продуктивність, можливість розподі-леної обробки та інші властивості додатка. Кожній GSC привлас-нюється значення від 0 до 5. Після того, як враховані всі чотирнад-цяті, загальних характеристик системи, розраховується чинник ре-гулювання вартості ПО за формулою [4]

,

де C_j-ступінь виливу і-ї GSC. Останньою розраховується кількість повних функціональних точок: FP-UAF* VAF,

Існують додатки, під час оцінювання яких використання стан-дартних функціональних точок не ефективне. Ці застосування такі: управління процесом у реальному часі, математичні обчислення, симуляція, системні застосування, інженерні застосування, вбудо-вані системи. Перераховані застосування відрізняються високою інтенсивністю обчислень, часто заснованих на алгоритмах підви-щеної складності. Для вирішення завдань розрахунку розміру вка-заних застосувань у 1986 році організацією Software Productivity Research (SPR) була розроблена методика аналізу характерних то-чок ПЗ (feature points). Суть її полягає в тому, що оцінюється кіль-кість алгоритмів у програмі і частково модифікується ступінь зна-чущості (weighting values) для розрахунку FP. Ця методика вважа-ється експериментальною.

7.2. Методи і моделі оцінювання вартості програмного забезпечення

Методи і моделі оцінювання вартості ПЗ можна розділити на дві групи: неалгоритмічні методи і алгоритмічні моделі. До неалгорит-мінних методів належать Price-to-win, оцінка ПЗ Паркінсона, екс-пертна оцінка, оцінка за аналогією. До алгоритмічних моделей, на-лежать SLIM і COCOMO.

Суть неалгоритмічних методів полягає в тому, що при оціню-ванні вартості ПО використовуються певні схеми і принципи, а не математичні формули. Нижче проаналізуємо ці методи.

Price-to-win. Метод грунтується на принципі «клієнт, завжди має рацію». Суть методу полягає в тому, то незалежно від передбачу-ваних реальних витрат на розробку проекту, оцінка вартості ПО коригується відповідно до побажань замовника. Price-to-win фак-тично є політикою проведення переговорів з клієнтом, тому оціню-вання часто застосовується компаніями, що не мають засобів для якісного оцінювання проектів. Застосування методу може мати для розробника певні негативні наслідки: брак ресурсів для виконання проекту, невиконання термінів здачі проекту і як результат — втрата контракту або банкрутство.

Оцінка за Паркінсоном. Метод грунтується на принципі «Обсяг роботи зростає так, як це потрібно, щоб зайняти час, виділений на її виконання». Принцип, пізніше названий «законом», був уперше висловлений С.Н. Паркінсоном і описував природу взаємодії бю-рократичної системи в адміністративних інститутах, відображаючи процес неефективного використання ресурсів. У застосуванні до розробки програмних проектів закон Паркінсона використовується у вигляді такої схеми: щоб підвищити продуктивність праці розроб-ника, слід зменшити час, відведений на розробку.

Експертна оцінка. Метод грунтується на принципі експертної оцінки і застосовується в

таких проектах, що використовують нові технології, нові процеси або вирішальні інноваційні завдання. До процесу оцінювання залучаються інженери-розробники, які самі оцінюють частину проекту, що займається ними. Після цього скли-каються збори, на яких результати окремих оцінок інтегруються в єдину, цілісну систему. Припущення, на яких грунтувалася оцінка окремих експертів, заносяться в протокол і відкрито обговорюють-ся. При опитуванні експертів використовуються Дельфійськая ме-тодика або розширена методика, орієнтована на приведення експер-ті» до консенсусу. У результаті досягається баланс оцінки при інте-грації окремих компонентів у загальну систему. Далі йде чергова стадія компонентного оцінювання, і у межах збільшення кількості ітерацій точність оцінки збільшується.

Оцінка за аналогїєю - будучи різновидом експертної оцінки, час-то виділяється в окремий метод. Метод ґрунтується на принципі аналогії, Оцінка аналогічно алгоритмічним моделям використовує емпіричні дані про характеристики завершених проектів. Головна відмінність полягає в тому, що алгоритмічні моделі використо-вують ці дані непрямим чином, наприклад, для калібрування пара-метрів моделей, а метод оцінювання ПО за аналогією за допомогою емпіричних даних дає змогу відібрати схожі проекти. Схема оцін-ки, заснована на вказаному принципі, складається з декількох ета-пів. На першому етапі здійснюється збір даних за проектом, що розробляється, У рамках життєвою циклу ПО оптимальними фор-мами для цього с аналіз вимог і проектування. На основі експертної оцінки проводиться відбір характеристик ПО за якими порівнюва-тимуться проекти, Вибір характеристик залежить від типу додатка, середовища розробки і набору відомих параметрів додатка. Наступ-ний етап включає пошук і аналіз проектів «аналогічних» ПО, що розробляються за вибраними характеристиками, Результатом цього етапу є, як правило, декілька проектів, що мають найменші відмін-ності в числових значеннях характеристик оцінки. Для відбору найбільш близьких проектів, що розробляються, може використо-вуватися метод вимірювання евклідової відстані в n-мірному прос-торі. Кожній характеристиці привласнюється значення ваги (множ-ник), що визначає значущість характеристики для проекту, У спрощеному варіанті вага дорівнює одиниці, тобто всі характе-ристики проекту вважаються рівнозначними ПО за важливістю. Далі проекти і їх відповідні характеристики відображаються в n-мірному просторі, як точки (n дорівнює кількості змінних, для кожної змінної використовується своє вимірювання), після чого обчислюється евклідова відстань між відповідними точками:

де a і b- точки в просторі; $a_1 ... a_n$ і $b_1 ... b_n$ - координати точок у відповідних площинах.

Проекти, що мають найбільшу схожість, будуть розташовані щонайближче, тобто евклідова відстань у них буде найменшою. Останнім станом с експерт на оцінка проекту, що розробляється, в якій значення, взяті а аналогічного проекту, використовуються як база оцінки.

Моделі оцінювання вартості 173. Модель оцінювання вартості програмного забезпечення - цс одна або декілька функцій, які опи-сують залежність між характеристиками проекту і витратами на його реалізацію. Моделі поділяють за типом використовуваних фу-нкцій на лінійні, мультиплікативні, статичні; за використанням іс-торичних даних на емпіричні та аналітичні. Моделями, що часто реалізуються і є добре документованими, є моделі Путнема (стати-чна, аналітична) і COCOMO (статична, емпірична).

Модель Путнема (SLIM). Найбільш поширена модель аналітич-ної групи. Створена для проектів обсягом понад 70 000 рядків коду, модель ґрунтується на твердженні, що витрати на розробку ПО розподіляються згідно з кривими Нордена-Рейлі, які є графіками функцій, що розподіляє робочу силу за часом. Загальний вигляд подібної функції:

де v - набуте значення; t- час, а v_0 і t_p - параметри, що визначають функцію. Для

великого значення t крива прагне до параметра v_0 , який називається cost scale factor parameter, функція зростає найшвидше при $t = t_p$ Основною причиною такої поведінки моделі було те, що спочатку дослідження Нордена ґрунтувалися не на теоретичній основі, а на спо-стереженнях за проектами, не пов'язаними з ПО (машинобудуван-ня, будівництво). Тому немає наукового підтвердження, що прог-рамні проекти потребують такого ж розподілу робочої сили. Нав-паки, часто кількість людино-годин, потрібних проекту, може різко змінитися, зробивши оцінку непридатною до використання. Після ряду емпіричних спостережень Путнем виразив робоче рівняння моделі у формі:

де Size - розмір коду в LOC; C - технологічний фактор; E- загаль-на вартість проекту в людино-годинах; t - очікуваний час реалізації проекту.

Технологічний фактор включає в себе характеристику проекту в таких аспектах: методи управління розуміння процесу, якість вико-ристаних методів інженерії ПО, рівень використаних мов програ-мування, рівень розвитку середовища, навички та досвід команди розробників, складність додатків.

Рівняння для загальної вартості E мас вигляд:

де D_0 - коефіцієнт, що виражає кількість необхідної робота (зна-чення від 8 до 12 означає, що ПО повністю нове, з великою кількіс-тю зв'язків; значення до 27 - потрібне перероблення наявного ко-ду)- Зв'язуючи два рівняння, отримаємо таке

i

які показують, що витрати пропорційні розміру коду в степені 9/7 ≈ 1/286. Це досить близько до моделі Б. Боема, де даний чин-ник знаходиться у межах від 1,05 до 1,20 [10].

У 1991 році Путнемом була представлена альтернативна реалі-зація моделі, виконана за замовленням Quantitative Software Management (QSM) Inc. і застосована в комплексі SLIM Estimate для оцінювання вартості ПЗ [14]. Повне рівняння в цій реалізації виг-лядає як: $E = 12^5$ ∙ $B(SLOC/P)^3$ ∙ $(1/Schedule^4)$.

Якщо на загальний час реалізації проекту обмеження не накла-даються, то можливе використання спрощеного рівняння

тут B - чинник спеціальних навичок; P - чинник продуктивності; Schedule - час розробки ПЗ графіку (у місяцях), Рівняння може бу-ти використане, якщо передбачувані витрата понад 20 люднно-місяців.

Використання наведених рівнянь потребує знання параметра P. Для його визначення використовується спеціальна таблиця, що міс-тить значення параметра P, залежні від середовища застосування, що розробляється.

Модель COCOMO. Сім'ю моделей COCOMO було створено в 1981 році на основі бази даних про проекти консалтингової фірми TRW.

COCOMO є третьою моделлю, орієнтованою на використання в трьох фазах життєвого циклу ПО: базова (Basic) - застосовується на етапі вироблення специфікацій, вимог, розширена (Intermediate) - після визначення вимог до ПО; поглиблена (Advanced) - використ-товується після закінчення проектування ПЗ. У загальному вигляді рівняння моделей має вигляд:

де E - витрати праці на проект (у людино-місяцях); S - розмір коду (у KLOC); EAF -

чинник уточнення витрат (effort adjustment factor). Параметри a і b залежать від виду застосування, що розробляється, який може бути таким:

- відносно простий проект, робота над яким ведеться однорід-ною командою розробників, вимоги носять рекомендаційний харак-тер, відсутня заздалегідь вироблена вичерпна специфікація (напри-клад, нескладне прикладне програмне забезпечення);
- проект середньої складності, робота над яким ведеться зміша-ною командою розробників, Вимоги до проекту визначаються спе-цифікацією, проте можуть змінюватися в процесі його розробки (наприклад, програмне забезпечення системи управління банківсь-ким терміналом);
- проект, який повинен бути реалізованій! у жорстких рамках заданих вимог (наприклад, програмне забезпечення системи управ-ління польотами),

У базовій моделі чинник EAF береться рівним одиниці. Для виз-начення значення цього чинника в розширеній моделі використо-вується таблиця, що містить ряд параметрів, які визначають вар-тість проекту. Використовуючи поглиблену модель, спочатку ви-конують оцінювання з використанням розширеної моделі на рівні компонента, після чого кожен параметр вартості оцінюється для всіх фаз життєвого циклу ПЗ.

COCOMO II також є сімейством моделей і є розвитком базової (Basic) моделі COCOMO. COCOMO II включає три моделі - ство-рення додатків (Application Composition Model, ACM), ранній етап розробки (Early Design Model, EDM) і пост-архитектурна (Post Architecture Model, PAM).

ACM використовується на ранньому етапі реалізації проекту, для того, щоб оцінити таке: інтерфейс користувача, взаємодія з сис-темою, продуктивність. За початковий розмір береться кількість екранів, звітів і 3GL-компонентів. Якщо припустити, що в проекті буде використано r % об'єктів з раніше створених проектів, кіль-кість нових об'єктних точок у проекті (Object Points, OP) можна розрахувати як:

OP=(object points)*(100-r)/100.

Тоді витрати можна розрахувати за формулою:

E=OP/PROD,

де PROD - табличне значення.

EDM - це високорівнева модель, якій потрібна порівняно неве-лика кількість початкових параметрів. Вона призначена для оціню-вання доцільності використання тих або інших апаратних і програмних засобів у процесі розробки проекту. Для визначення розміру використовується функціональна точка (Unadjusted Function Point). Для її перетворення в LOC використовуються таблиці перетворень, Рівняння моделі раннього етапу розробки мас вигляд:

E=a∙LOC∙EAF,

де a - константа 2,45; EAF визначається так само, як і в оригіналь-ній моделі COCOMO.

Параметри для EDM отримують комбінуванням параметрі» для постархітектурної моделі.

PAM є найбільш деталізованою моделлю, яка використовується, коли проект повністю готовий до розробки, Для оцінювання вартості ПЗ за допомогою PAM необхідний пакет опису життєвого циклу проекту, який містить докладну інформацію про чинники вартості і дозволяє провести точніше оцінювання. PAM використовується на етапі фактичної розробки і підтримки проекту. Для оцінювання роз-мірів можуть використовуватися як рядки коду, так і функціональні точки з модифікаторами, що враховують повторне використання ко-ду. Модель використовує 17 чинників вартості і 5 чинників, що виз-начають масштаб проекту (у моделі COCOMO масштаб визначався параметрами виду додатка). Рівняння PAM має вигляд

a взято за 2,55, a,

де W_i - параметри, що відображають властивості проекту, наприклад, схо-жість з раніше виконаними проектами, ризик вибору архітектури для реалізації, розуміння процесу розробки, спрацьованості команди роз-робників. Значення параметрів є табличними.

7.3. Засоби оцінювання вартості програмного забезпечення

Широко відомі засоби оцінювання ПЗ, засновані на моделях SLIM і COCOMO.

SLIM Estimate компанії QSM є найбільш часто використовува-ним програмним засобом для оцінювання вартості програмного забезпечення, у якому реалізована модель Путнема. Засіб входить до складу пакету прикладного програмного забезпечення і призна-чений для роботи над проектом ПЗ на початкових стадіях життєво-го циклу. У пакет, окрім засобу оцінки, також входять засоби зби-рання і зберігання даних про реалізовані проекти (SUM DataManager), аналізу цих даних (SLIM Metrics), загального конт-ролю над процесом розробки (SLIM Control). Цей пакет використо-вується для оцінювання вартості, що розробляється програмним забезпеченням у таких організаціях: Alcatel Telecom, AT&T, Athens Group, Australian Department of Defence, BAE, Bell South Communications, Hewlett-Packard. IBM Rational Software, Lockheed Martin, Motorola Communications, Nokia, US Air Force Cost Analysis Agency. SLIM Estimate дає змогу виконувати оцінювання вартості розробки програмного забезпечення різними способами: майстер швидкого оцінювання, оцінювання розміру, оцінювання PI, оціню-вання непередбачених обставин, оцінювання, засноване на історичних чинниках. Першим і найчастіше використовуваним є майстер швидкої оцінки (Quick Estimate Wizard). Для цього використову-ються такі параметри: тип застосування, що розробляється; макси-мально можливий час роботи над проектом; бюджет проекту; орієн-товна загальна кількість рядків; індекс продуктивності команди розробників; відсоток повторно використовуваного коду. Форму-ються таблиці і будуються діаграми, що відображають загальну кількість задіяної робочої сили і її розподіл програмного забезпе-чення за графіком робіт.

Шаблон робочої книги (workbook) проекту SLIM Estimate під-тримує близько 50 різних форматів проведеного оцінювання прог-рамного забезпечення. Створені робочі книги можуть служити ша-блонами для оцінювання вартості подальших проектів. За умов-чанням, SLIM Estimate оцінює трудовитрати з 50% вірогідністю успішної реалізації проекту. Для зміни цього значення слід, відко-ригувати значення вірогідності за допомогою майстра налаштуван-ня вірогідності [12], Результатом оцінки розміру є загальна кіль-кість рядків коду, які може створити команда розробників у певних умовах. Результат оцінки індексу продуктивності є PI, необхідний для реалізації проекту в заданих умовах. Оцінка непередбачених обставин використовується для генерації плану реалізації із зада-ною вірогідністю успішного завершення проекту. Ці способи мо-жуть використовуватися як незалежно, так і для уточнення результа-тів, отриманих у результаті використання майстра швидкої оцінки.

За допомогою функції Edit Historical Projects такі оцінки мо-жуть бути експортовані в SLIM DataManager. Для порівняльного аналізу результатів оцінки можливий імпорт даних з програми SLIM Metrics або іншої робочої книги SLIM Estimate.

Для оцінювання розміру проекту разом зі SLIM Estimate «постав-ляється» реалізована в Microsoft Excel таблиця, значення з якої мо-жуть бути імпортовані в робочу книгу проекту. У ранніх версіях SLIM Estimate основною одиницею вимірювання був логічний ви-раз у початковому коді (Logical Source Statement, LSS). Починаючи з версії 5.0, в SLIM Estimate використовуються рядки коду, функці-ональні і об'єктні точки (безпосередньо, без перетворення в LSS). Найбільш широко використовуваним способом калібрування моделі в SLIM Estimate є використання історичних параметрів налашту-вання (Historical Tuning Factors). Програмний комплекс SLIM Estimate може експортувати дані звітів у найбільш популярні фор-мати файлів, такі як: Microsoft Word, Microsoft Excel, Enhanced Metafile, Microsoft Project, HTML.

У комплект постачання SLIM Estimate входить база реалізованих проектів, котрі можна використовувати для калібрування вико-ристовуваної моделі - установки значень параметрів вартості для опису характеристик проекту. Найпоширенішим способом калібру-вання моделі в SLIM Estimate с використання історичних параметрів налаштування (Historical Tuning Factors), У разі його використан-тання значення параметрів вартості для проекту обчислюються

програмним комплексом на основі обраних проектів з бази реалізо-ваних проектів.

Модель Путнема надзвичайно чутлива до значення технологіч-них чинників, тому точне визначення їх значення є дуже важливим для правильного оцінювання на основі SLIM. Перевагою моделі Путнема перед COCOMO 1.1 або COCOMO 2.0 є невелика кіль-кість параметрів, необхідних для оцінки.

Засоби оцінювання вартості розробки ПЗ, засновані на моделі SLIM, не потребують обов'язкового використання історичної бази проектів. Тому вони можуть застосовуватися безпосередньо органі-зацією, що виконує проектування програмного забезпечення. Ви-користовуючи історичну базу даних, потрібна участь фахівця для порівняння реалізованих і описаних проектів з бази з проектом, що знаходиться в розробці. Залучення сторонньої організації при ви-конанні оцінювання вартості також може бути необхідне внаслідок наявності у неї достатньо великої історичної і деталізованої бази реалізованих проектів.

Costar (SoftStar Systems), Cost Xpert (Marotz), SoftwareCost Calculator (SofiwareCost.com) — засоби, засновані на моделі COCOMO. Допускається використання всіх реалізацій моделі COCOMO, моделей життєвого циклу програмного забезпечення Waterfall і MBASE/RUP, підтримується робота з проектом, складе-ним з компонентів, для кожного з яких можна виконати роздільне оцінювання.

Costar, дає змогу проводити оцінювання в двох режимах: покроковому (за допомогою майстра оцінювання вартості); інтерактив-ному (що забезпечує безпосередню вказівку значень параметрів, які впливають на вартість проекту). Для визначення розміру оці-нюваного проекту використовуються функціональні точки або ряд-ки коду. Для перекладу значень, указаних у рядках коду, в програмі є конвертатор, що розраховує значення розміру коду у функціона-льних точок, виходячи з мови програмування, яка використовуєть-ся для реалізації проекту. Costar підтримує обмеження проекту, заснованого на граничних фінансових витратах і крайньому терміні реалізації проекту.

Для оцінювання витрат, пов'язаних з оплатою праці працівни-ків, існує, два альтернативні підходи: розрахунок витрат для кожно-го з етапів життєвого циклу програмного забезпечення; розрахунок місячної плати за працю для кожної категорії співробітників.

Для аналізу результатів оцінки Costar створює різні форми зві-тів, графіків і діаграм. Звіти, представлені у формі таблиць, можуть бути збережені у форматі Microsoft Excel, графіки і діаграми - у форматі растрового зображення BMP.

Для проведення точного оцінювання вартості розробки ПЗ мо-дель COCOMO потребує детального і різнобічного опису проекту. Цс може утруднити застосування заснованих на її основі засобів на ранньому етапі розробки програмного забезпечення і сприяє збіль-шенню точності оцінювання на пізніх етапах розробки програмного забезпечення, аналізуючи завершений проект.

Під час використання засобів на основі моделі COCOMO або COCOMO IT чинниками, що впливають на точність оцінювання вартості, є такі: правильний вибір конкретної реалізації моделі COCOMO; точність калібрування - відповідність установок почат-ковим даним. У зв'язку з цим, для застосування засобів використо-вують персонал, який не мас прямого відношення до процесів про-ектування і розробки програмного забезпечення. Він формує спе-цифікації проекту і параметри, необхідні для оцінки, які надаються співробітникам, які виконують оцінювання.

Ефективне застосування алгоритмічних моделей оцінювання вар-тості програмного забезпечення і заснованих на їх основі засобів оцінки віддають перевагу їх сумісному використанню з неалгоритмічними методами оцінювання. Так, алгоритмічні засоби оцінки мо-жуть бути застосовані членами експертних комісій для аналізу про-екту і формування власного оцінювання. Завдяки широким можливостям експорту даних і візуалізації, використання автоматизованих засобів оцінки вартості програмного забезпечення дає можливість формувати власні бази характеристик реалізованих проектів, а також створювати

звіти, що ілюструють процес розробки проекту, що знач-но знижує трудовитрати, пов'язані з підготовкою звітності.

Параметри вартості, Параметр вартості (cost driver) - це суб'єктивна величина, яка оцінює різні тимчасові, якісні і ресурс-ні аспекти розробки програмного забезпечення. Кожен з парамет-рів може бути відкалібрований. Калібрування параметрів вартості - це коректування значень параметрів, що впливає на значення трудовитрат, а отже, на якийсь час і на вартість, оцінюючи прог-рамний проект. При калібруванні за вказаними нижче сімнадцять-ма параметрами вибирається оцінний рівень (дуже високий, висо-кий, вище номінального, номінальний, нижче номінального, низь-кий, дуже низький) параметра. У формулах цей рівень відбиваєть-ся у вигляді коефіцієнта трудовитрат і, таким чином, на кожній стадії розробки проекту впливає на вартість і тривалість тієї або іншої стадії. Виділяють такі групи параметрів (табл.7.1): продукту (product factors), платформи (platform factors), персоналу (personnel factors) і проекту (project Jactors). У табл. 7.2 подано короткий опис кожного параметра.

Таблиця 7.1

Продукт	Враховують характеристики того, що розробляється ПЗ *(RELY. DATA, CPLX, RUSE, DOCU)*
Платформа	Враховують характеристики програмно-апаратного комплексу, потрібного для функціонування ПЗ *(TIME, STOR, PVOL)*
Персонал	Враховують рівень знань і злагодженості роботи колекти-ву програмістів *(ACAP, PCAP, PCON, APEX, PLEX. LTEX)*
Проект	Враховують вплив сучасних підходів і технологій, тери-торіальну віддаленість членів колективу розробників і терміни виконання проекту *(TOOL, SITE, SCED)*

Таблиця 7.2

Параметр	**Опис**
RELY (Required Software Reliability)	**Враховує ступінь виконання програмою певної дії протягом певного часу**
DATA (Database Size)	**Враховує вплив обсягу тестових даних на розробку продукту. Рівень цього парамет-ра розраховується як співвідношення байт у тестованій базі даних до *SLOC* у прог-рамі**
CPLX (Product Complexity)	**Включає п'ять типів операцій; управління, рахункові, пристрійно-залежні, управління даними, управління, призначене для корис-тувача інтерфейсом, Рівець складності — це суб'єктивне середньозважене значення рівнів типів операцій**
RUSE (Developed for Reusability)	**Враховує трудовитрати (потрібні додатко-во для написання компонентів), призначе-ні для повторного використання *в* даному або подальших проектах. Використовує такі оцінні рівні: «у**

	проекті», «у програ-мі», «у лінійці продуктів», «у різних ліній-ках продуктів». Значення параметра нак-ладає обмеження на параметри *RELY* і *DOCU*
DOCU (Documentation Match To Life-Cycle Needs)	**Враховує ступінь відповідності докумен-тації проекту його життєвому циклу**
TIME (Execution Time Constraint)	**Враховує тимчасові ресурси, використову-вані ПЗ при виконанні поставленого зав-дання**
STOR (Main Storage Constraint)	**Враховує відсоток використання сховищ даних**
PVOL (Platform Volatility)	**Враховує термін «життя» платформи (комп-лекс апаратного і ПЗ, який потрібний для функціонування того, що розробляється ПЗ)**
ACAP (Analyst Capability)	**Враховує аналіз, здатність проектувати, ефективність і комунікативні здібності групи фахівців, які розробляють вимоги і специфікації проекту. Параметр непови-нен оцінювати рівень кваліфікації окремо взятого фахівця**
PCAP (Programmer Capability)	**Враховує рівень програмістів у колективі. При виборі значення для цього параметра слід звернути увагу на комунікативні і професійні здібності програмістів і на ко-мандну роботу в цілому**
PCON (Personnel Continuity)	**Враховує плинність кадрів у колективі**
APEX (Applications Experience)	**Враховує досвід колективу при роботі над додатками певною типу**
PLEX (Platform Experience)	**Враховує вміння використовувати особ-ливості платформ, такі як: графічний ін-терфейс, бази даних, мережевий інтер-фейс, розподілені системи**
LTEX (Language and Tool Experience)	**Враховує досвід програмістів (мови, сере-довища та інструменти)**
TOOL (Use Of Software Tools}	**Враховує рівень використання інструмен-тів розробки**
SITE (Multisite Development)	**Враховує територіальну віддаленість (від офісу до міжнародних офісів) членів ко-манди розробинків і використовувані ними засоби комунікації (від телефону до відео конференц-зв'язку)**
SCED (Required Development Schedule)	**Враховує вплив тимчасових: обмежень, що накладаються на проект і на значення тру-довитрат**

СПИСОК ЛІТЕРАТУРИ

1. *Basilli V.R.* ***Viewing*** Maintenance ***as Reuse-Oriented Software Development/V.R. Basilli //IEEE Software. 1990. - June. -P. 19-25.***

2. *Boehm B.W.* ***Improving Software Productivity / B.W. Bo ehm // Computer. - 1987. - Vol. 20, n.9. - P. 43 - 57.***

3. Boehm B.W. ***Software Engineering Economics / B.W. Boehm - Englewood Cliffs, Ш.: Prentice-Hall, 1981. - 257 p.***

4. *Boehm B.W.* ***Spiral Model of software Development and Enhancement / B.W. Boehm // Computer. - 1988, - May. - P. 71 - 73.***

5. *Bosch J.* ***Design and use of software architectures /*** J. ***Bosch. -Addison Wesley, 2000. - 325 p.***

6. *Black R.* ***Critical Testing Processes / R. Black. - Addison-Wesley, 2003.***

7. *Blum B.A.* ***Taxonomy of Software Development Methods / B.A. Blum // Coinmunication of the ACM. - 1994. - Vol. 37, n. 11,-P. 82 - 94.***

8. *Budgen D.* ***Software design: Reading / D. Budgen. - Addison-Wesley, 1994.-320 p.***

9. *Fenton K.* ***Software Metrics: A Rigorous Approach / K. Fenton,*** *E.* ***Norman. - London: Chapman & Hall, 1991. — 638 p.***

10. *Glass R.L.* ***Extrime programming the good, the bad. and the bottom line / R.L. Glass // IEEE. Software. - 2001, - Vol.18, n.7. - P.ll -P. 112,- 111 - 112.***

[\. Georgiadou E. ***Software Process and Product Improvement: A Historical Perspective / E. Georgiadou // Кибернетика и системный анализ. - 2003. № 1, - P. 147 - 177,***

12. *Jacobson I.* ***The Unified Software Development Process / I. Jacobson, G. Booch, J. Rumbaugh. - Addison-Wesley, 1999. - 310 p.***

13.Jonsson P. ***Software Reuse. Architecture. Process*** and ***Organization for Business Success. Person Education Asia / P. Jonsson. - 2002. — 497 p.***

14. *Martin J.* ***Rapid Application Development /*** J. ***Martin. -Macmillan, 1991. -250 p.***

15. *Perry D.* ***Models of Software Development Environments / D. Perry, G. Kaiser // IEEE Trans. On Soft. Engin. - 1991. - Vol. 17, n. 3.-P. 283-295.***

16. *Pricto-Diaz R.* ***Domain Analysis: An Introduction*** *II* ***Software engineering notes /R. Pricto-Diaz. - 1990. - Vol, 15, №. 2.- P. 47 - 54.***

17. *Railich V.* ***Software cultures and evolution / V. Railich, N. Wilde // Computer. 2001. - Sept. - P.25 - 28.***

IS. *Rombach H.D,* ***Software specifications: a framework / H. D. Rombach. - Carnegie mullon Univ: SET, 1990. - 30 p.***

19. *Rajlich W.* ***A stage Model for the Software Life Cycle / W. Rajlich, K. Bennett // Computer. - 2000. - July. - P. 77 - 70.***

20. *Silverman B.* ***Software Cost and Productivity Improvements: An Analogical view / B. Silverman // Computer.- 1985. - May. - P. 87 - 95.***

21. *Spillner A.* ***Software Testing Foundations / A. Spillner, T. Linz, H. Schafer. - Dpunkt: Verlag, 2007. - 277 p.***

22. *Sidorov N.A.* ***Software Stylistics / N. A. Sidorov // Proceedings of NAU. - 2005. - 2(24), - P. 98 - 103.***

23. *Toriik Matsumoto K.* ***An Environment for Computer - Aided Empirical Software Engineering / Matsumoto K. Gingerz Toriik // IEEE Trans, On Software Eng. - 1999. - Vol. 25, № . 4 - P. 474 - 485 .***

24. *Van Veendabl E.* ***Standard glossary of term used in Software testing / E. Van Veendabl. - ISTQB. - 2007. Vol. 1,2. - June. - 30 p.***

25. *Creting* ***a software engineering culture / K. Wiegers // Dorset House Publishing. - New York, 2003. - 358 p.***

26. *Вельбицкий И.В.* ***Технология программирования*** *I* ***И. В. Вель*** бицкий. ***- К.: Техніка, 1984. - 274 с.***

27. ***Ка*** *пе С.* ***Тестирование программного обеспечения / С. Капе.*** - M: DiaSoft, 2001. ***- 542с.***

28. *Саммервил И.* ***Инженерия программного обеспечения / И. Саммервил. - М.: Вильяме, 2002. - 720 с.***

29. *Сидоров Н.А.* ***Повторное использование, переработка и вос-становление программного обеспечения / Н.А. Сидоров // Управ-ляющие системы и машины. - 2000. - № 3, 4. - С, 27 - 37.***

$$S = \frac{n}{m},$$

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний авіаційний університет

М. О. Сидоров

ВСТУП ДО ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Курс лекцій

Київ
Видавництво Національного авіаційного університету
«НАУ-друк»
2010

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} N_{ij} W_{ij} \ ,$$

Періоди розвитку Об'єкти порівняння	1960±5 років програмування *"any-wich-way"*	1970±5 років «програмування в малому»	1980±5 ро-ків «програму-вання у ве-ликому»
Об'єкти	Маленькі програми	Алгоритми і програми	Системна структура
Дані	Неструктурована інформація	Структури даних і типи	Бази даних
Управління	Елементарне ро-зуміння діаграм управління	Програми виконуються і закін-чуються	Програми, що безпере-рвно вико-нуються
Простір етанів	Стан, що погано розуміється окре-мо від управління	Маленькі, прості	Великі, структуризовані
Організаційне управління	Немає	Індивідуальні зусилля	Колективні зусилля, супровід
Інструмент»	Асемблери	Компілятори, редактори, заван-тажувачі	Середовища, інтегровані інстру-менти

$$VAF = 0.65 + \left[\left(\sum_{i=1}^{14} C_i \right) / 100 \right],$$

Аспекти розглядання	Фаза (початок)		
	I (1960)	II (1970)	III (1980)
Особливості програмування	Програмування «абияк»	Програмування «в малому»	Програмування «у великому»
Підготовка кадрів	Майже відсутня	Прикладна математика	Комп'ютерні науки
Ресурси	Асемблери, машинні дампи	Транслятори, лінкери, завантажувачі, системи програмування	Середовища розробки програм
Технології	Відсутні	HIPO, формалізовані технічні завдання	R-Технологія, *PSL/PSA, SREM, SADT*
Економіко	Відсутня	Інтуїтивна	*PRICE-S, SCEP, SLIM*
Ринок	Відсутній, замовлення на програмне забезпечення	Виробництво для продажу	Сегментація ринку

$$d(a,b)=\sqrt{(a_1-b_1)^2+(a_2-b_2)^2\ldots+(a_n-b_n)^2}=$$

$$=\sqrt{\sum_{i=1}^{n}(a_i-b_i)^2}$$

Римський шиях	Грецький шлях
— використовують структурні методи; — застосовують більше формалізмів; —Створюють крупкі програми; - застосовують CASE-інструменти; — проекти повністю керовані; — створюють максимум документації; — налаштовані управляти проектами	— використовують об'ектно-оріентовані методи; — Застосовують менше формалізмів; — створюють малі програми; — застосовують окремі інструменти; — проекти частково керовані; — створюють мінімум документації; — налаштовані писати програми

$$v = v_0\left(1 - e^{-l^2/2t^2_p}\right),$$

Автор шкали	Метафора рівня	Рівні					
	Епохи	-	Анархія	Фольклор	Методологія	Метрики	Інженерія
Weinberg	Шаблони	Забутий	Мінливий	Рутинний	Керований	Передбачуваний	Узгодженний
Humphrey	Рівні	-	Початковий	Повторюваний	Визначений	Керований	Оптимізований

$$Size = CE^{1/3} t^{4/3},$$

Атрибути процесу	Метапроцесс	Макропроцес	Мікропроцес
Цілі	Стратегія бізнесу Прибутковість стратегії бізнесу Конкурентоспромо жність	Виконання проекту. Прибутковість проекту. Зменшення ризиків. Виконання бюджету проекту, термінів, якості	Виконання процесу. Управління ресурсами. Виконання проміжного бюджету, термінів, якості
Учасники	Менеджери організації. Замовники	Менеджери проекту. Розробники ПЗ	Менеджери процесів проекту. Розробники програмного забезпеченим
Метрики	Передбачуваність проекту Отримання доходу на контрольованому сегменті ринку	Виконання бюджету, термінів. Досягнення основних контрольних точок	Виконання бюджету і термінів процесу. Досягнення основних контрольних точок процесу
Часовий масштаб	Постійно	Від одного року до декількох років	Від одного до декількох місяців

$$E = D_0 t^{3d},$$

Порядковий номер	Назва методу	Автор	Рік походження
1	Рівні абстракції	Е. Дейкстра	1968
2	Покрокове уточнення	Н. Вірт	1971
3	Функціональна декомпозиція, модуляризація	-	-
4	Структурне проектування	У. Стивенс, Г. Мейерс, Л. Константіне	1991
5	З'єднання, скріплення, приховування	Д. Парнас	1972
6	Структурне програмування	Е. Дейкстра	1972
7	Абстрактні типи даних	Б, Лісков С. Зайліс	1975
8	Структурний аналіз	Т. Де Марко	1978
9	PSL/QSA	Д. Тихросв Е. Херши	1977
10	ERM	С.Чен	1976
11	JSP/JSD	К. Джексоні	1977
12	Vienna Development methud	ІБМ	1970
13	Simula 67, об'ектно-оріентоване програмування ля	У. Даал, А. Кей	1976
14	Об'сктно-орієнтованне проектування	Г. Буч	1980
15	Домєнний аналіз	Р. Прието-Діаз	1991
16	Об'єктно-орієнтований аналіз	Е. Йодон, П. Коад	1978

$$E = \left(D_0^{4/7} C^{-9/7} \right) S^{9/7} \text{ i } t_d = \left(D_0^{-1/7} E^{-3/7} \right) S^{-3/7}$$

Тип моделі	Зорієнтованість методів	
	проблемно-орієнтовані	продукто-орієнтовані
Концептуальний	I Структурний аналіз ER-модель Об'єктно-орієнтований аналіз	II Структурно проектування О'бєктно-орієнтоване проектування
Формальні моделі	III PSL/PSA JSD VDM	IV Рівні абстракцій Покрокова розробка Доведення правильності JSD ОО - програмування

$$E = 56{,}4 \cdot B(SLOC/P)^{\frac{9}{7}}$$

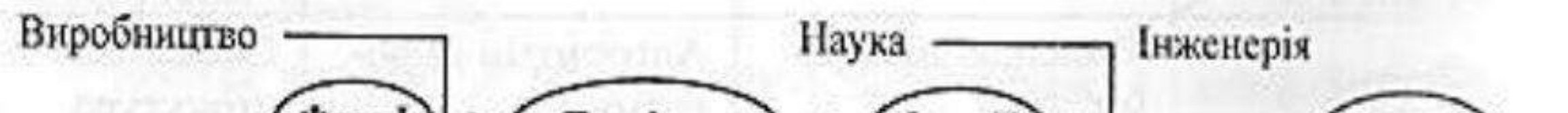
Продукт	Враховують характеристики того, що розробляється ПЗ *(RELY. DATA, CPLX, RUSE, DOCU)*
Платформа	Враховують характеристики програмно-апаратного комплексу, потрібного для функціонування ПЗ *(TIME, STOR, PVOL)*
Персонал	Враховують рівень знань і злагодженості роботи колекти-ву програмістів *(ACAP, PCAP, PCON, APEX, PLEX. LTEX)*
Проект	Враховують вплив сучасних підходів і технологій, тери-торіальну віддаленість членів колективу розробників і терміни виконання проекту *(TOOL, SITE, SCED)*

$$E = a \cdot S^b \cdot EAF,$$

Параметр	Опис
RELY (Required Software Reliability)	Враховує ступінь виконання програмою певної дії протягом певного часу
DATA (Database Size)	Враховує вплив обсягу тестових даних на розробку продукту. Рівень цього парамет-ра розраховується як співвідношення байт у тестованій базі даних до *SLOC* у прог-рамі
CPLX (Product Complexity)	Включає п'ять типів операцій; управління, рахункові, пристрійно-залежні, управління даними, управління, призначене для корис-тувача інтерфейсом, Рівець складності — це суб'єктивне середньозважене значення рівнів типів операцій
RUSE (Developed for Reusability)	Враховує трудовитрати (потрібні додатко-во для написання компонентів), призначе-ні для повторного використання *в* даному або подальших проектах. Використовує такі оцінні рівні: «у проекті», «у програ-мі», «у лінійці продуктів», «у різних ліній-ках продуктів». Значення параметра нак-ладає обмеження на параметри *RELY* і *DOCU*
DOCU (Documentation Match To Life-Cycle Needs)	Враховує ступінь відповідності докумен-тації проекту його життєвому циклу
TIME (Execution Time Constraint)	Враховує тимчасові ресурси, використову-вані ПЗ при виконанні поставленого зав-дання
STOR (Main Storage Constraint)	Враховує відсоток використання сховищ даних
PVOL (Platform Volatility)	Враховує термін «життя» платформи (комп-лекс апаратного і ПЗ, який потрібний для функціонування того, що розробляється ПЗ)

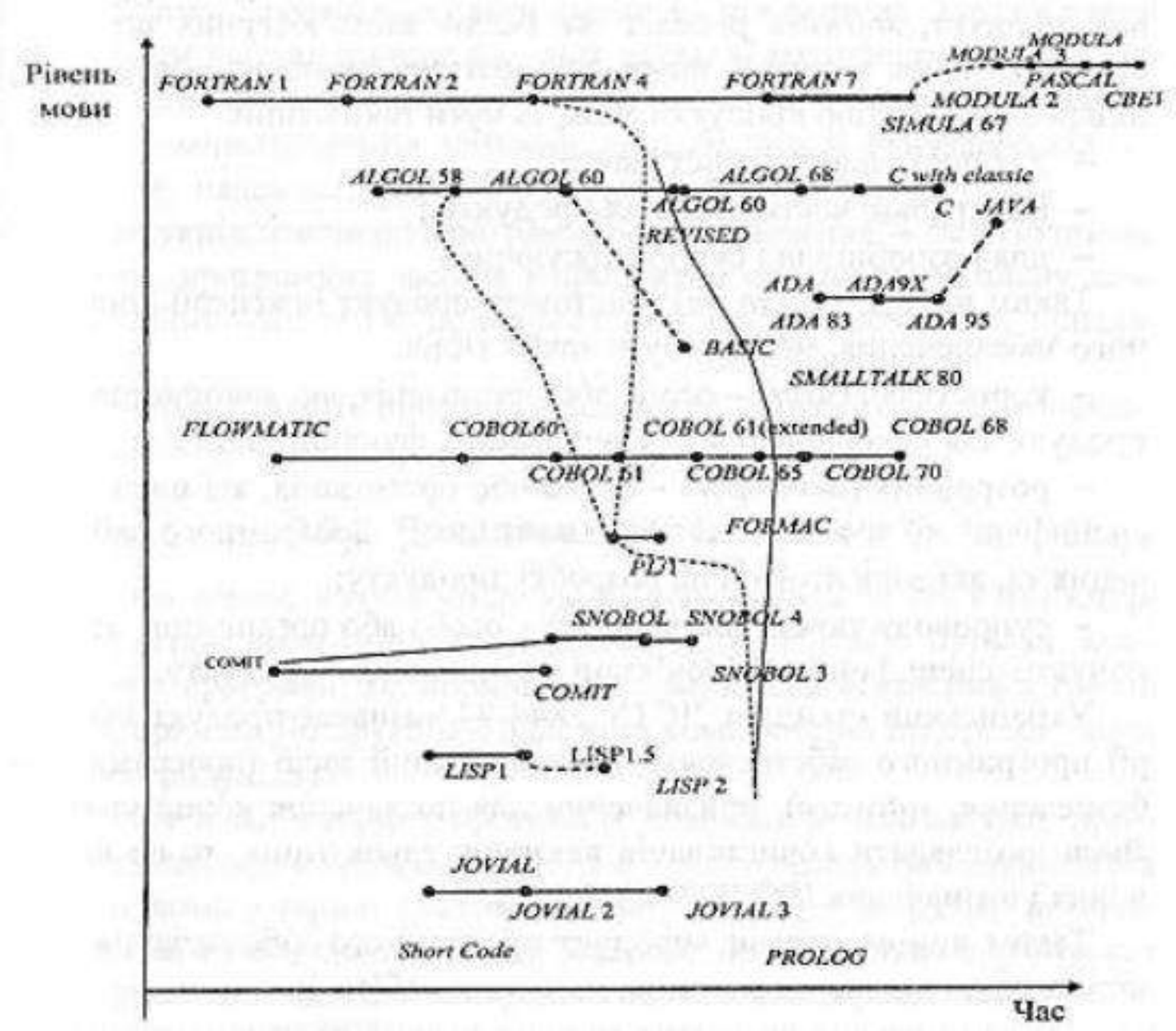
$E = a \cdot LOC^b \cdot EAF$, a взято за 2,55, а, $b = 1{,}01 + 0{,}01 \cdot \sum W_i$

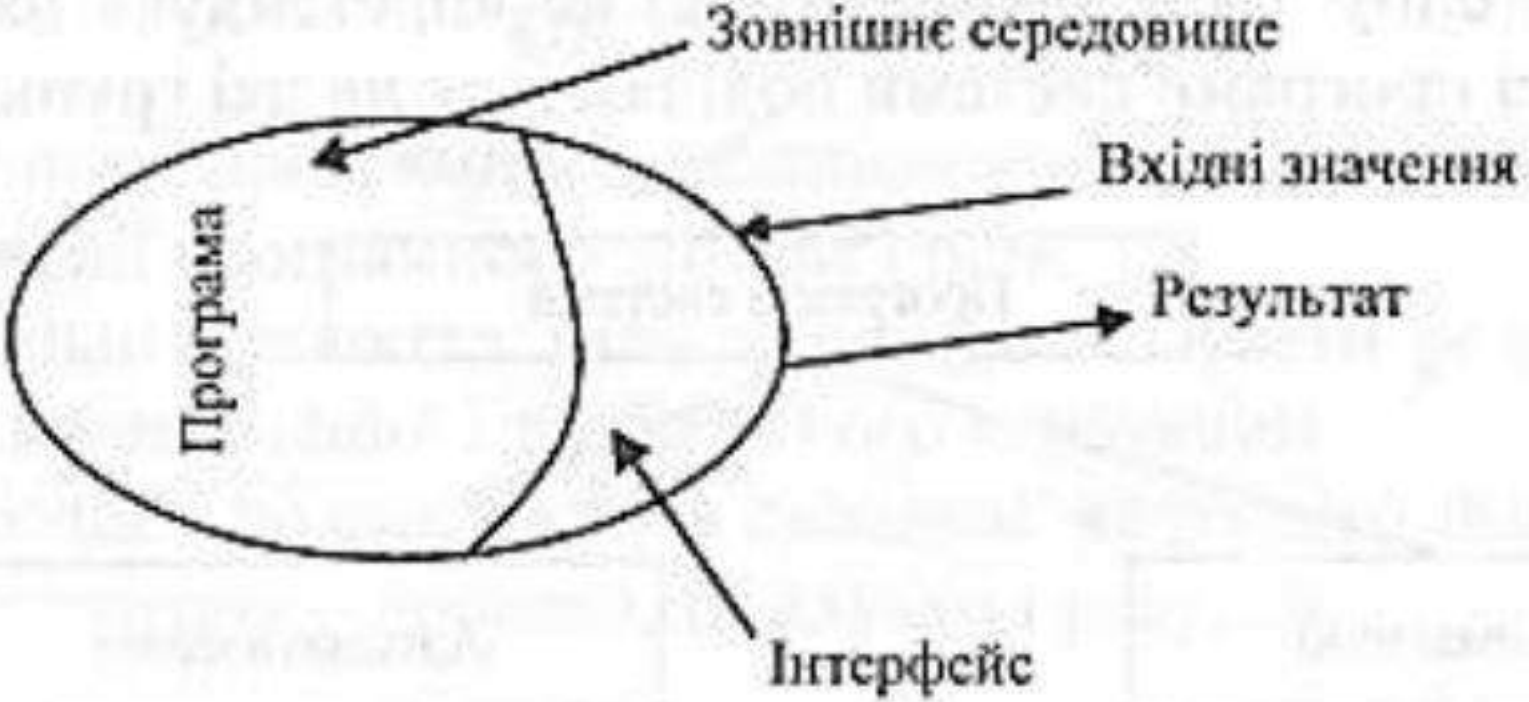
Параметр	Опис
ACAP (Analyst Capability)	Враховує аналіз, здатність проектувати, ефективність і комунікативні здібності групи фахівців, які розробляють вимоги і специфікації проекту. Параметр непови-нен оцінювати рівень кваліфікації окремо взятого фахівця
PCAP (Programmer Capability)	Враховує рівень програмістів у колективі. При виборі значення для цього параметра слід звернути увагу на комунікативні і професійні здібності програмістів і на ко-мандну роботу в цілому
PCON (Personnel Continuity)	Враховує плинність кадрів у колективі
APEX (Applications Experience)	Враховує досвід колективу при роботі над додатками певною типу
PLEX (Platform Experience)	Враховує вміння використовувати особ-ливості платформ, такі як: графічний ін-терфейс, бази даних, мережевий інтер-фейс, розподілені системи
LTEX (Language and Tool Experience)	Враховує досвід програмістів (мови, сере-довища та інструменти)
TOOL (Use Of Software Tools}	Враховує рівень використання інструмен-тів розробки
SITE (Multisite Development)	Враховує територіальну віддаленість (від офісу до міжнародних офісів) членів ко-манди розробинків і використовувані ними засоби комунікації (від телефону до відео конференц-зв'язку)
SCED (Required Development Schedule)	Враховує вплив тимчасових: обмежень, що накладаються на проект і на значення тру-довитрат


Виробництво
Ремесло
Фаза I
Торгівля
Наука
Фаза II
Інженерія
Фаза III

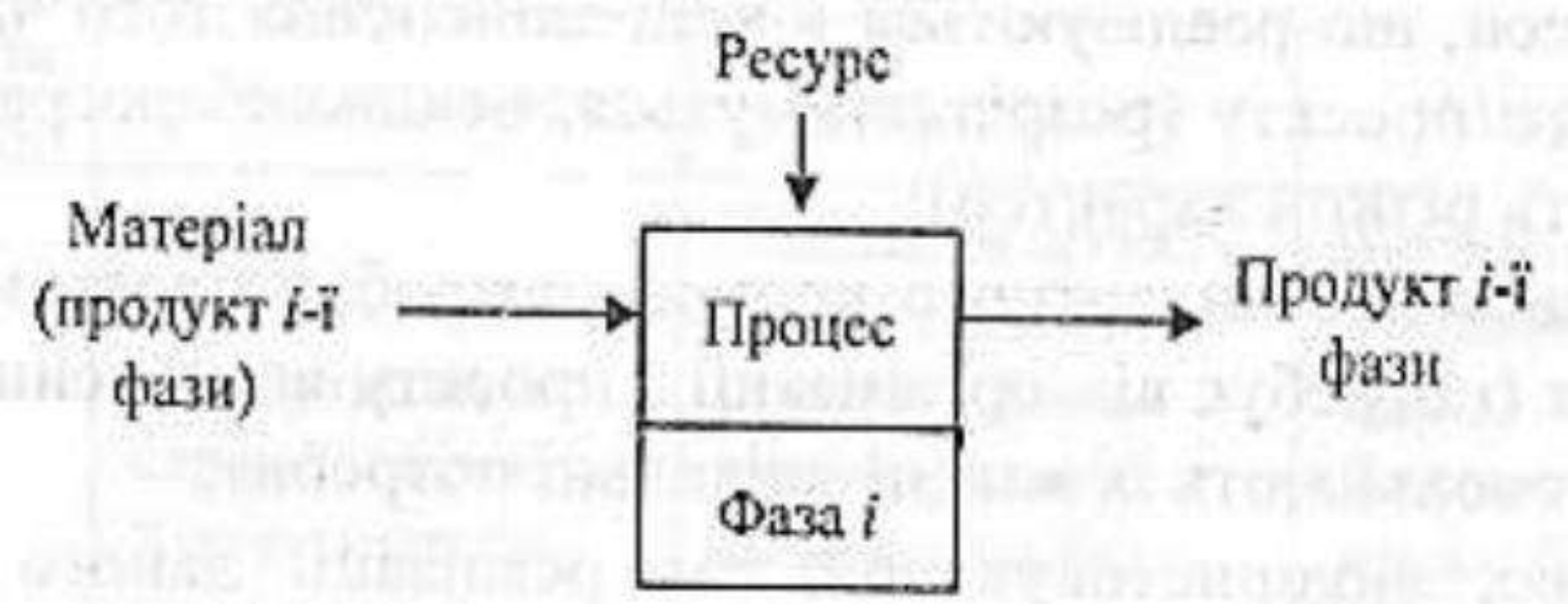

Проєктні цілі
Мають на увазі
Технічні практики
Визначення (А)
Підсилюють
Культура інженерії програмного забезпечення
Основа для
(В)
Підсилюють
Підсилюють
(С)
Допомагає встановити
Дії розробників (кожного)
Пріоритети менеджменту

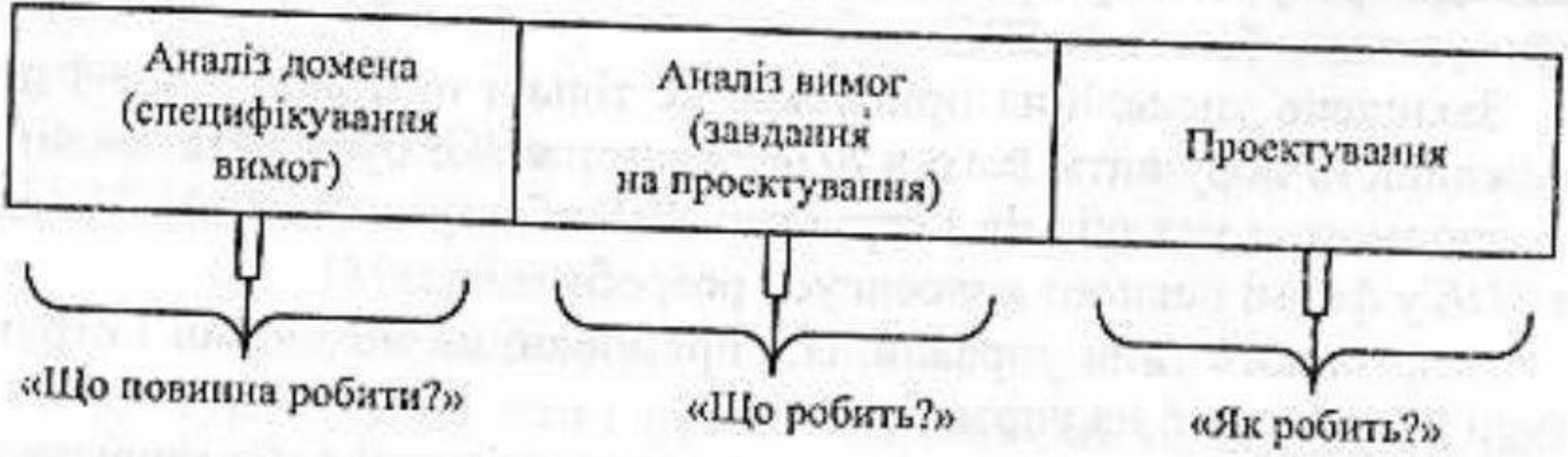

Оптимізований
Керування змінами процесів
Керування змінами технологій
Керування дефектами
Керований (4)
Керування якістю
Кількісне керування проектами
Визначений (3)
Координація
Інженерія продукту
Інтегрований проектний менеджмент
Програми тренінгу
Визначення організації проекту
Повторюваний (2)
Керування конфігурацією
Гарантія якості
Керування підрядчиками
Планування проекту
Керування вимог
CASE
Початковий

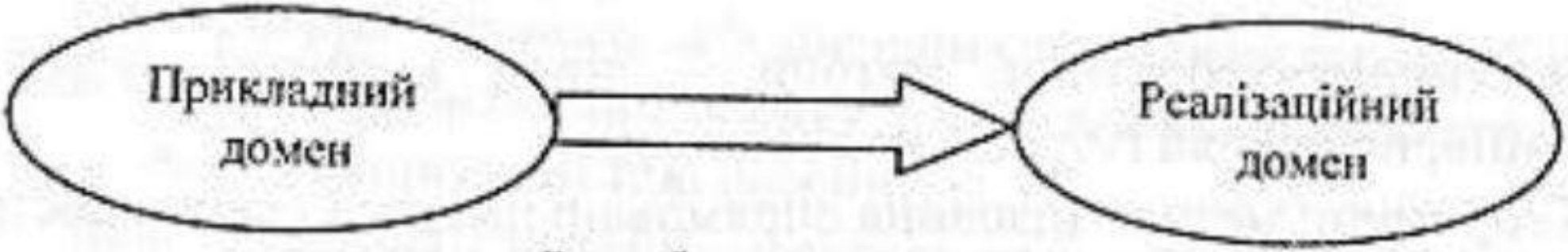

Рівень мови
FORTRAN 1
FORTRAN 2
FORTRAN 4
FORTRAN 7
MODULA
MODULA 3
PASCAL
MODULA 2
CBEI
SIMULA 67
ALGOL 58
ALGOL 60
ALGOL 68
C with classic
ALGOL 60
REVISED
C
JAVA
ADA
ADA9X
ADA 83
ADA 95
BASIC
SMALLTALK 80
FLOWMATIC
COBOL60
COBOL 61(extended)
COBOL 68
COBOL 61
COBOL 65
COBOL 70
FORMAC
PDY
SNOBOL
SNOBOL 4
COMIT
SNOBOL 3
COMIT
LISP1.5
LISP1
LISP 2
JOVIAL
JOVIAL 2
JOVIAL 3
Short Code
PROLOG
Час


Зовнішнє середовище
Вхідні значення
Результат
Програма
Інтерфейс


Програмні системи
Автоматичні
Автоматизовані
АВстС
АСУТП
АІС
АМС
АОС
АСУ
САПР
АІДС
АІФС
АСОУ
АСУЕ
CAD
CASE
CARE

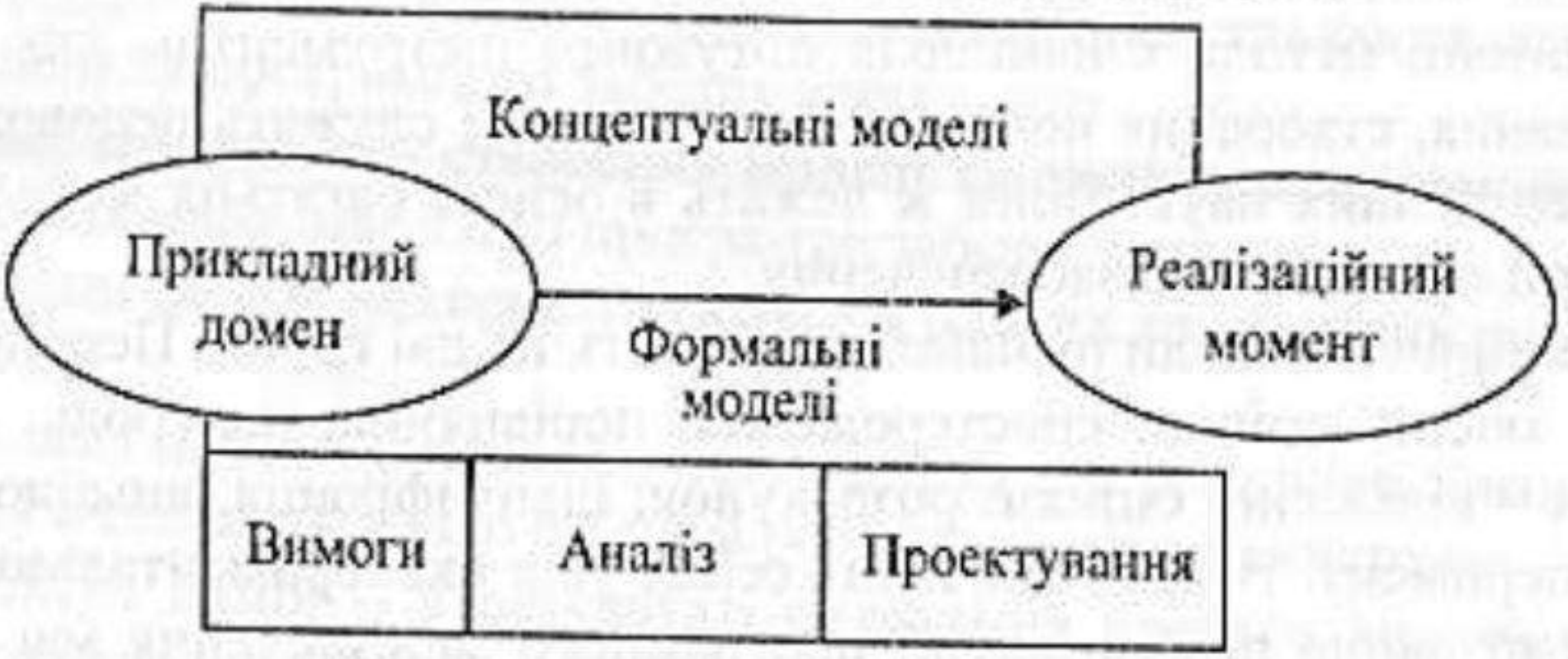

Ресурс
Матеріал (продукт *i*-ї фази)
Процес
Продукт *i*-ї фази
Фаза *i*

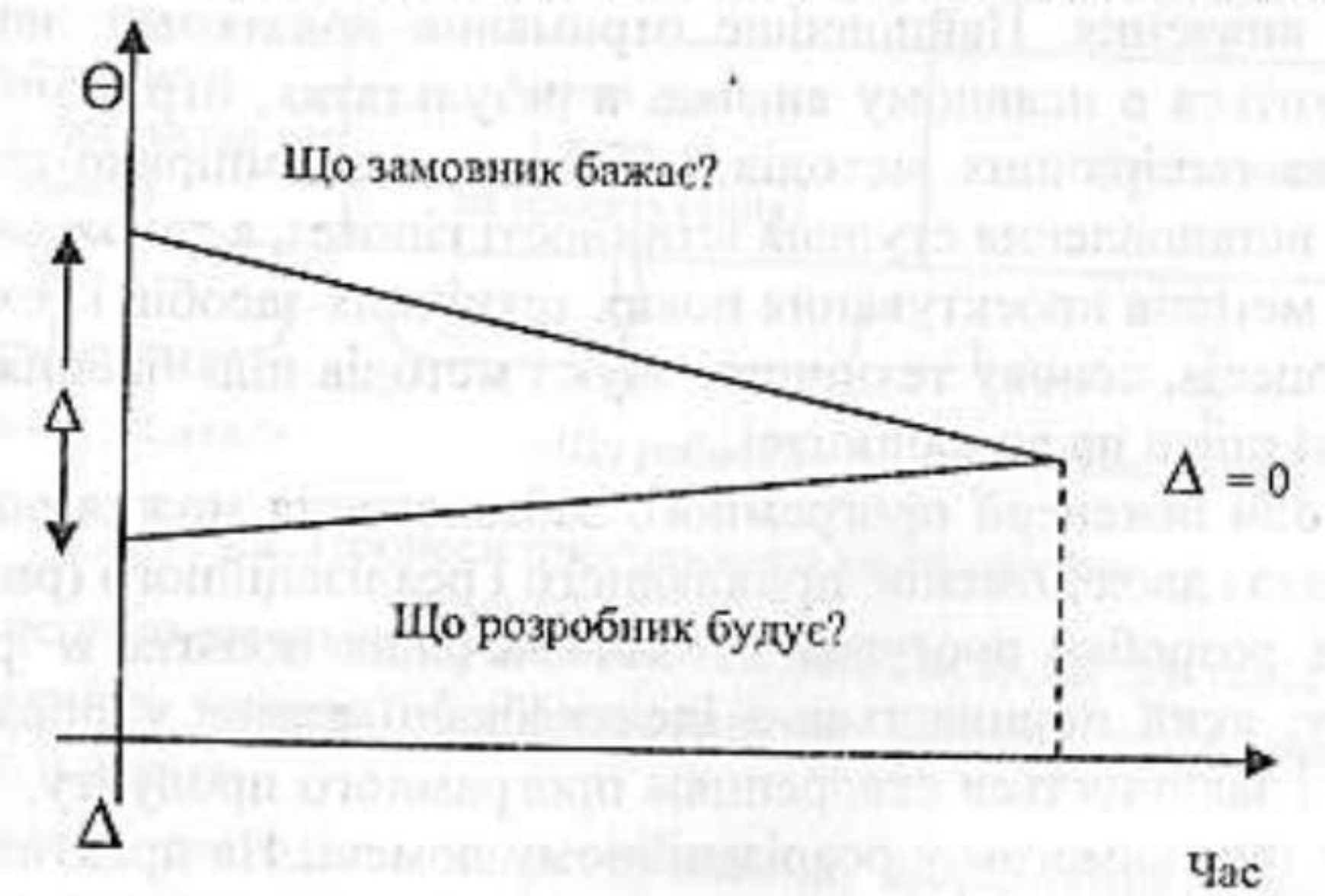

Аналіз домена (специфікування вимог)
Аналіз вимог (завдання на проектування)
Проектування
«Що повинна робити?»
«Що робить?»
«Як робить?»

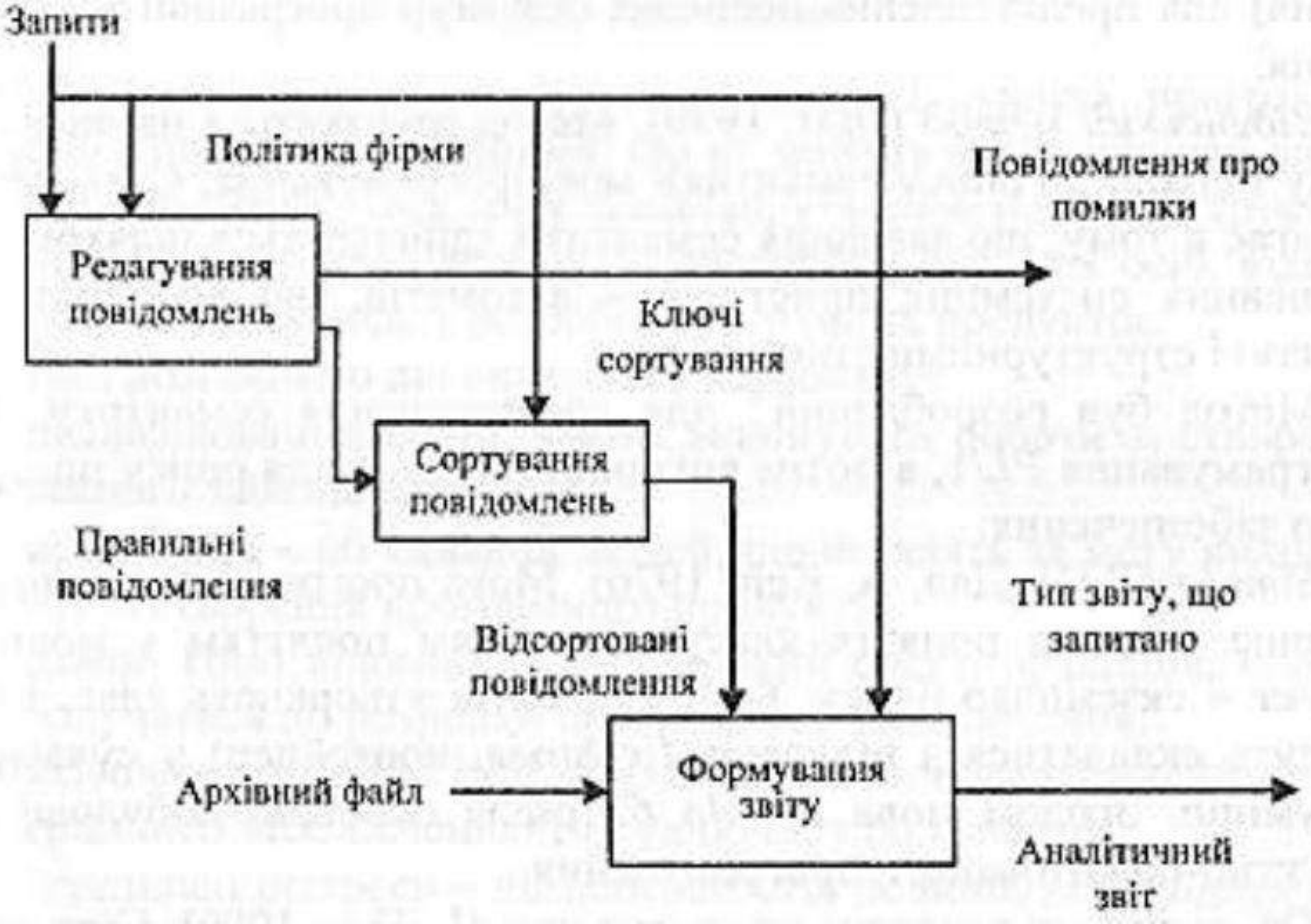


Розробка програмного забезпечення

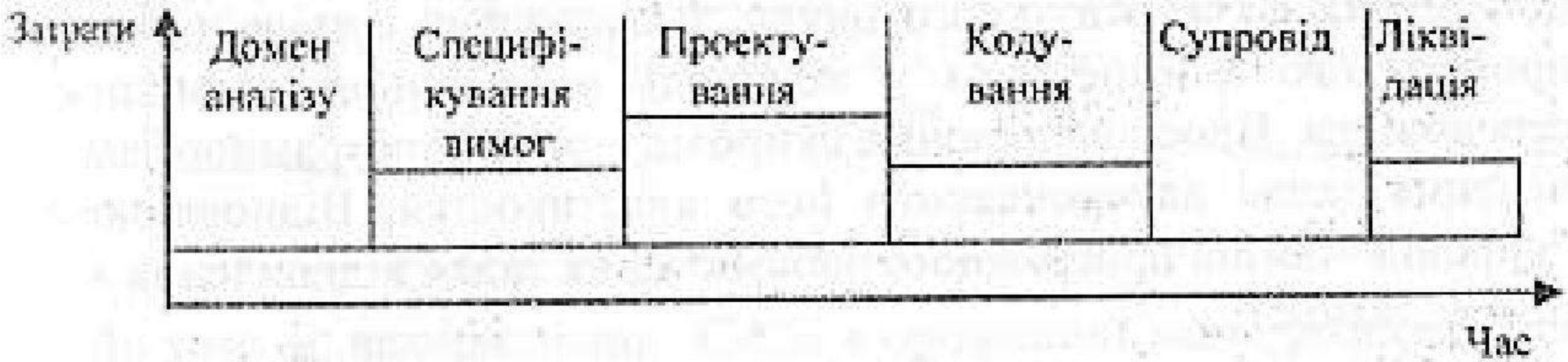
Рис. 4.3. Домени програмного забезпечення




Θ
Що замовник бажає?
Δ
Δ = 0
Що розробник буде?
Δ
Час


Запити
Політика фірми
Повідомлення про помилки
Редагування повідомлень
Ключі сортування
Сортування повідомлень
Правильні повідомлення
Тип звіту, що запитано
Відсортовані повідомлення
Архівний файл
Формування звіту
Аналітичний звіт

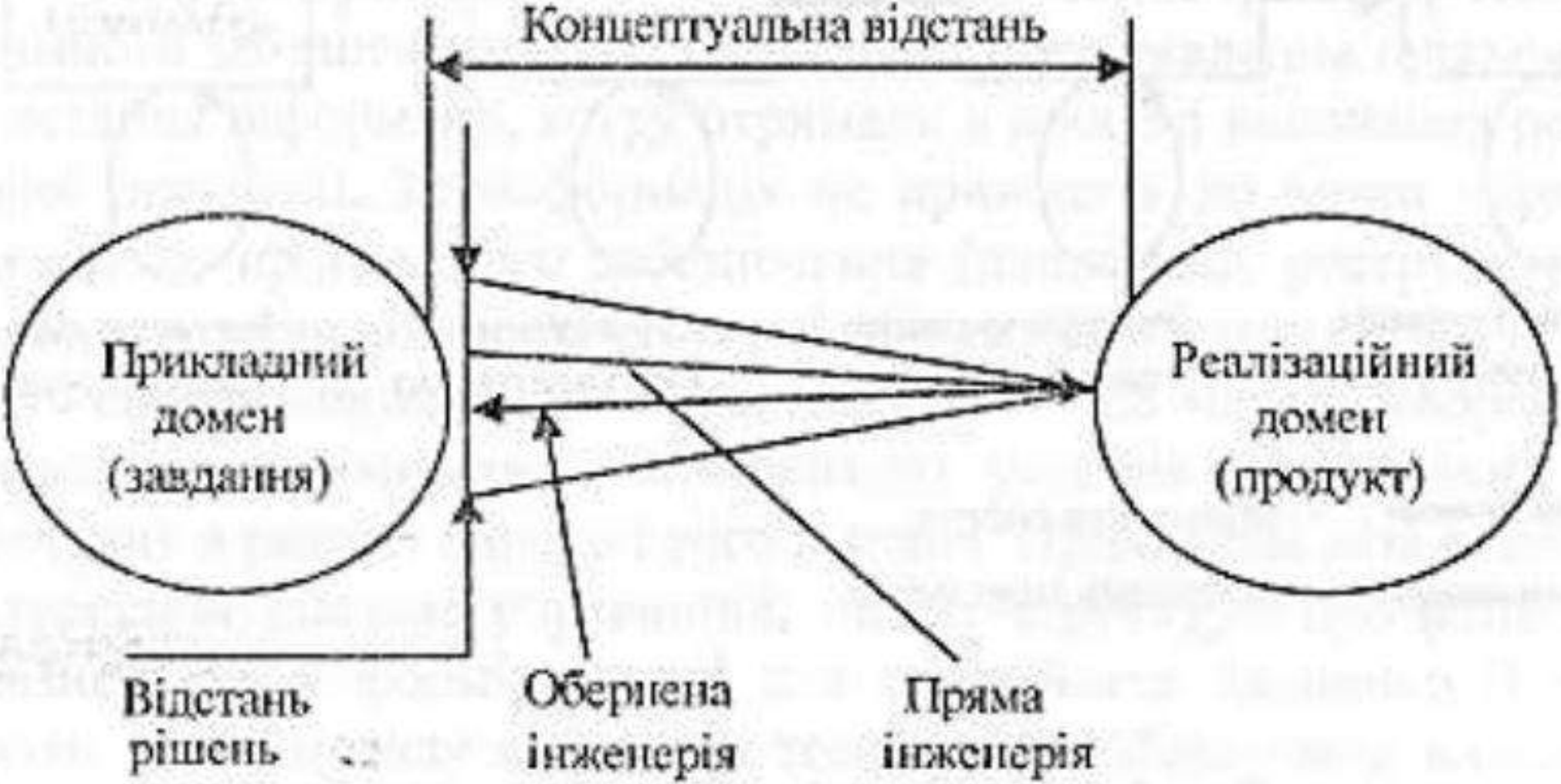

Комп'ютер
1
Виконує
m
Програму

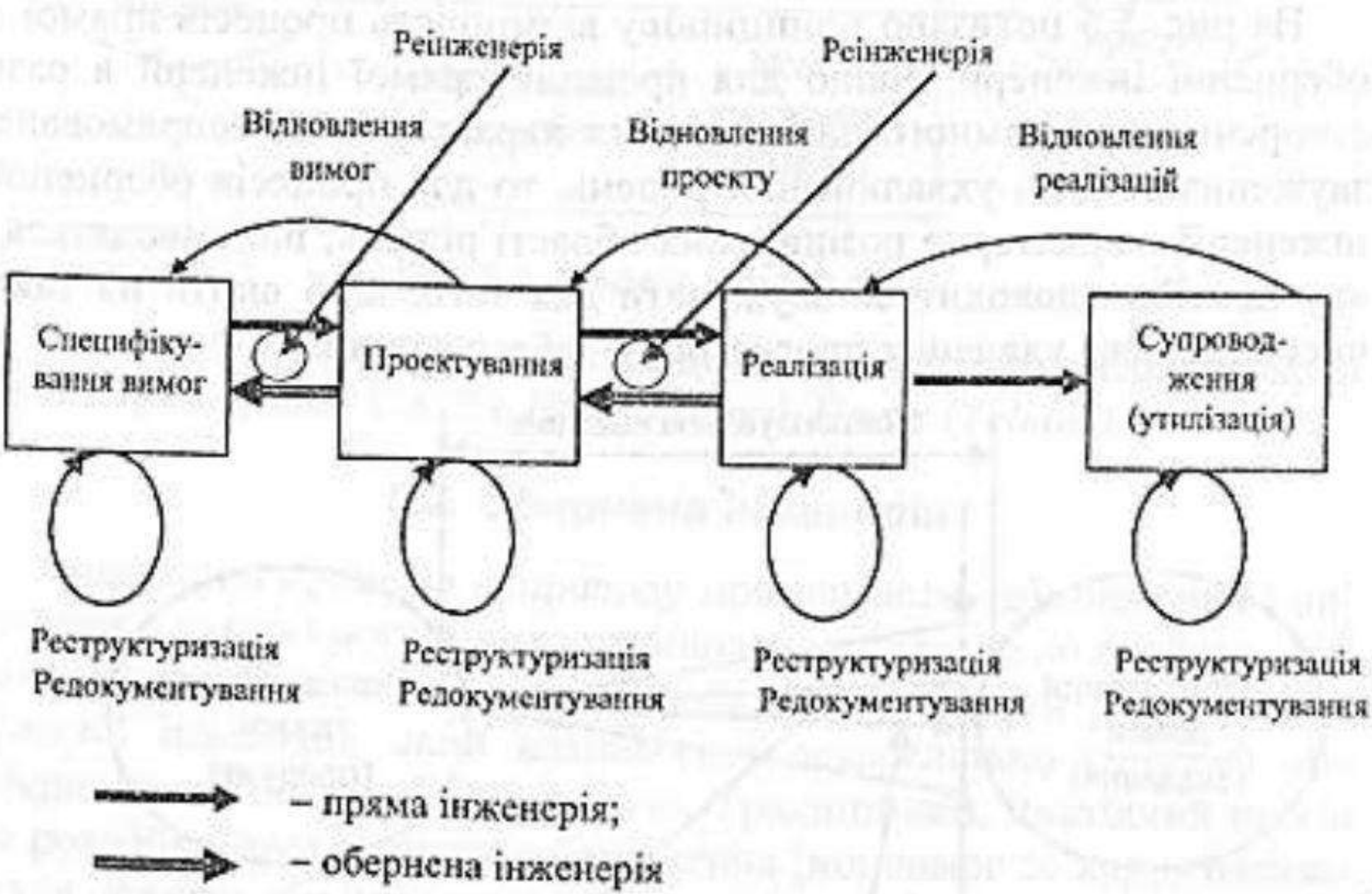

Затрати
Домен аналізу
Специфікування вимог
Проектування
Кодування
Супровід
Ліквідація
Час


Підвищення ефективності розробки програмного забезпечення
Використовують для
Визначає
CASE
Модель процесів життєвого циклу
Визначає процеси, ресурси, продукти


Вимоги замовника
Аналіз вимог
Проектування
Кодування
Тестування
Програмний продукт
CASE


Розробник програмного забезпечення
Модель процесів розробки програмного забезпечення
Вимоги замовника
Інструменти розробки
Модель програмного продукту
Програмний продукт


Концептуальна відстань
Прикладний домен (завдання)
Реалізаційний домен (продукт)
Відстань рішень
Обернена інженерія
Пряма інженерія


Реінженерія
Реінженерія
Відновлення вимог
Відновлення проекту
Відновлення реалізацій
Специфіку-вання вимог
Проектування
Реалізація
Супровод-ження (утилізація)
Реструктуризація
Редокументування
Реструктуризація
Редокументування
Реструктуризація
Редокументування
Реструктуризація
Редокументування
– пряма інженерія;
– обернена інженерія

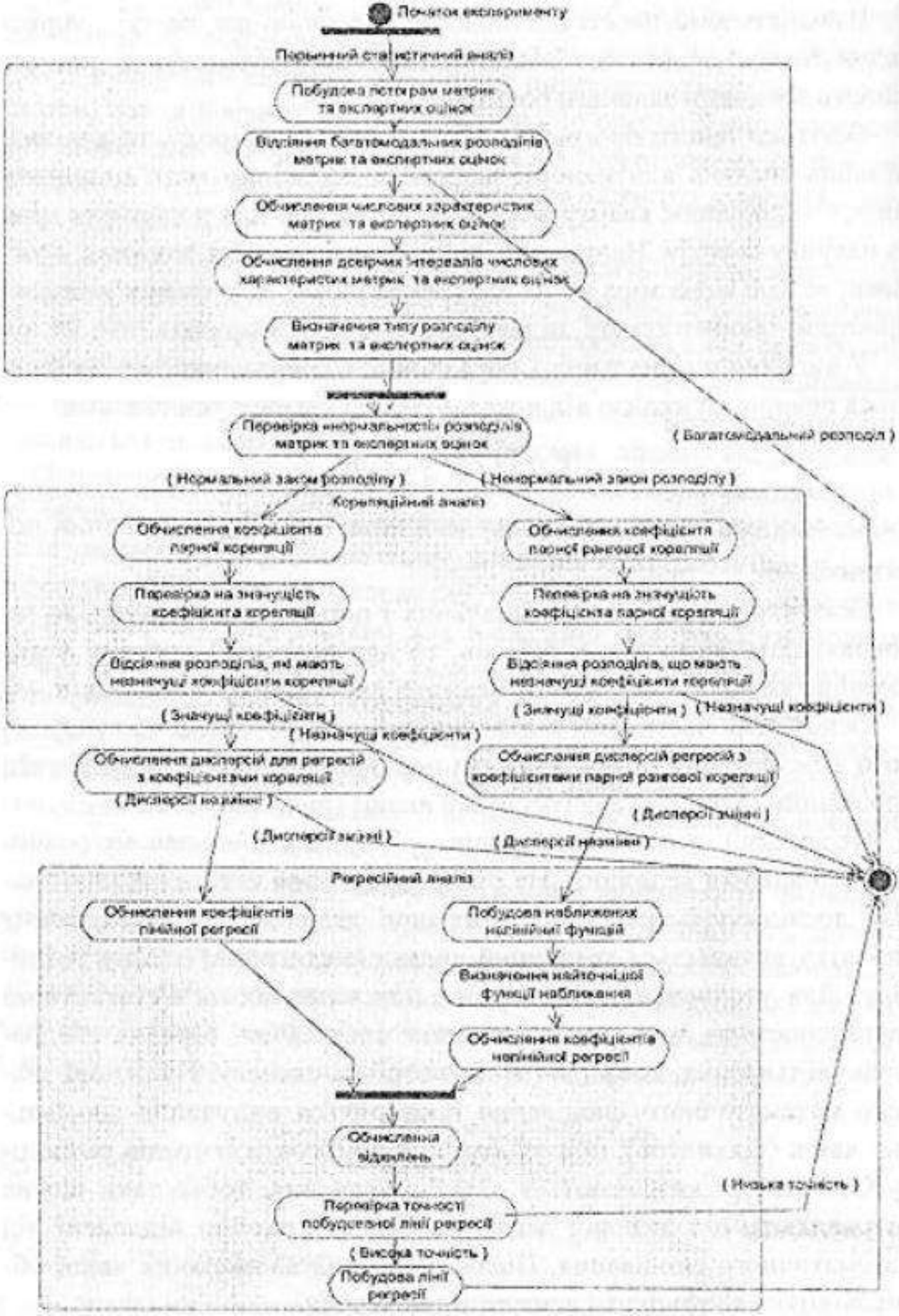

Успадковане програмне забезпечення
Аналізатори, екстрактори
Абстрактори
Представлення успадкованого програмного забезпечення
Репозитарій

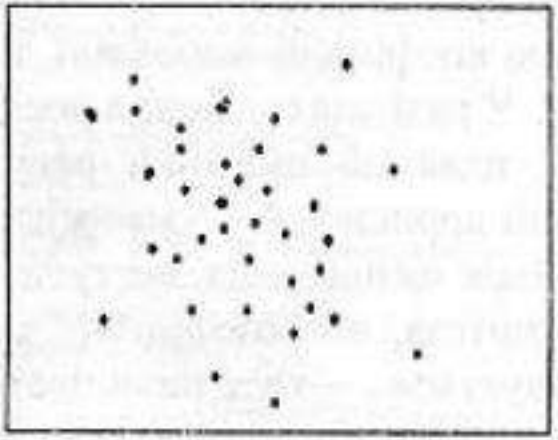
$$N_{xe} == \sum N_{x1}$$

$$N_{xe} = \sum_{x_{ne}} = \sum N_{xi} q_{xi}$$

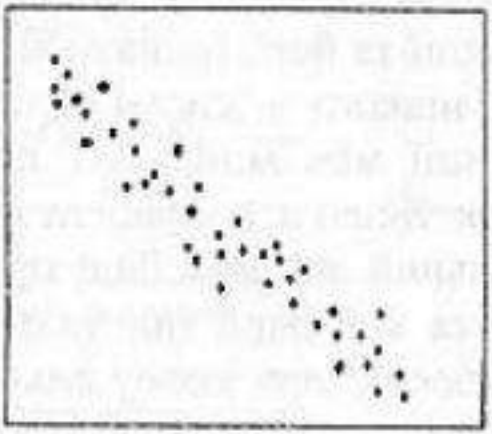
$$X_{ne} == N_{xe} q_x$$

$$V = f(V_1, V_2 \ldots V_n)$$

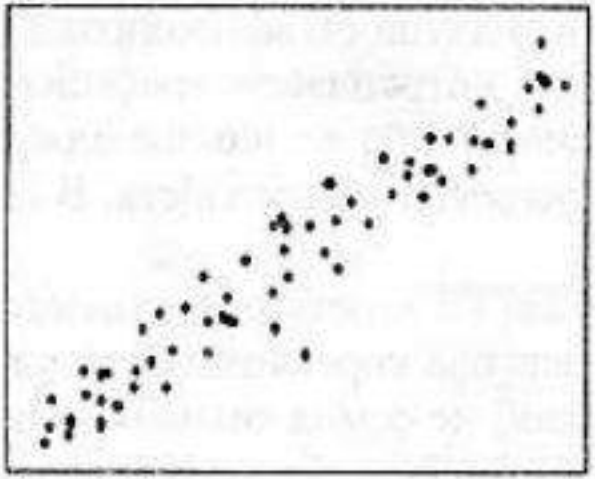

Початок експерименту
Первинний статистичний аналіз
Побудова гістограм матриць та експертних оцінок
Виділення багатомодальних розподілів матриць та експертних оцінок
Обчислення числових характеристик матриць та експертних оцінок
Обчислення довірчих інтервалів числових характеристик матриць та експертних оцінок
Визначення типу розподілу матриць та експертних оцінок
Перевірка «нормальності» розподілу матриць та експертних оцінок
(Багатомодальний розподіл)
(Нормальний закон розподілу)
(Ненормальний закон розподілу)
Кореляційний аналіз
Обчислення коефіцієнта парної кореляції
Обчислення коефіцієнта парної рангової кореляції
Перевірка на значущість коефіцієнта кореляції
Перевірка на значущість коефіцієнта парної кореляції
Відсіяння розподілів, які мають незначущі коефіцієнти кореляції
Відсіяння розподілів, що мають незначущі коефіцієнти кореляції
(Значущі коефіцієнти)
(Незначущі коефіцієнти)
(Значущі коефіцієнти)
(Незначущі коефіцієнти)
Обчислення дисперсій для регресій з коефіцієнтами кореляції
Обчислення дисперсій регресій з коефіцієнтами парної рангової кореляції
(Дисперсії незначні)
(Дисперсії значні)
(Дисперсії значні)
(Дисперсії незначні)
Регресійний аналіз
Обчислення коефіцієнтів лінійної регресії
Побудова наближених нелінійних функцій
Визначення найточнішої функції наближення
Обчислення коефіцієнтів нелінійної регресії
Обчислення відхилень
Перевірка точності побудованої лінії регресії
(Низька точність)
(Висока точність)
Побудова лінії регресії
Закінчення експерименту



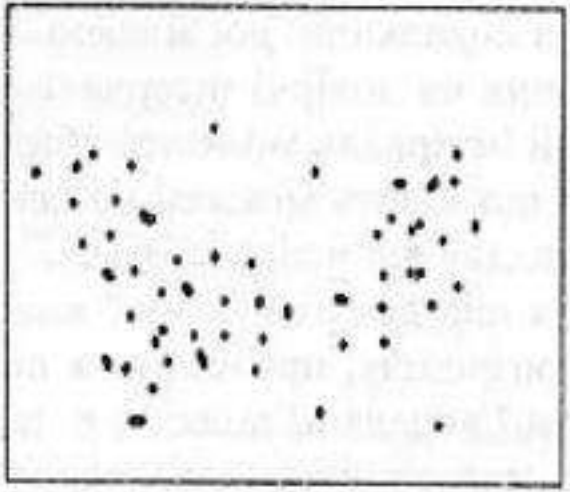
а



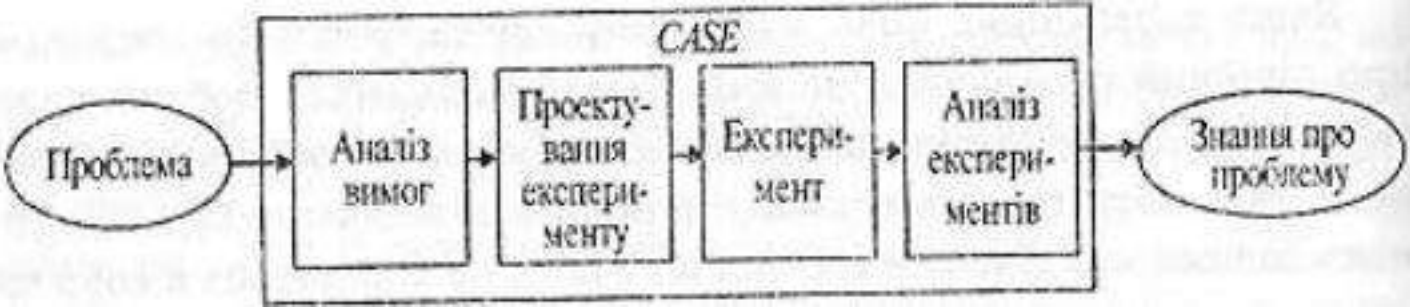
б

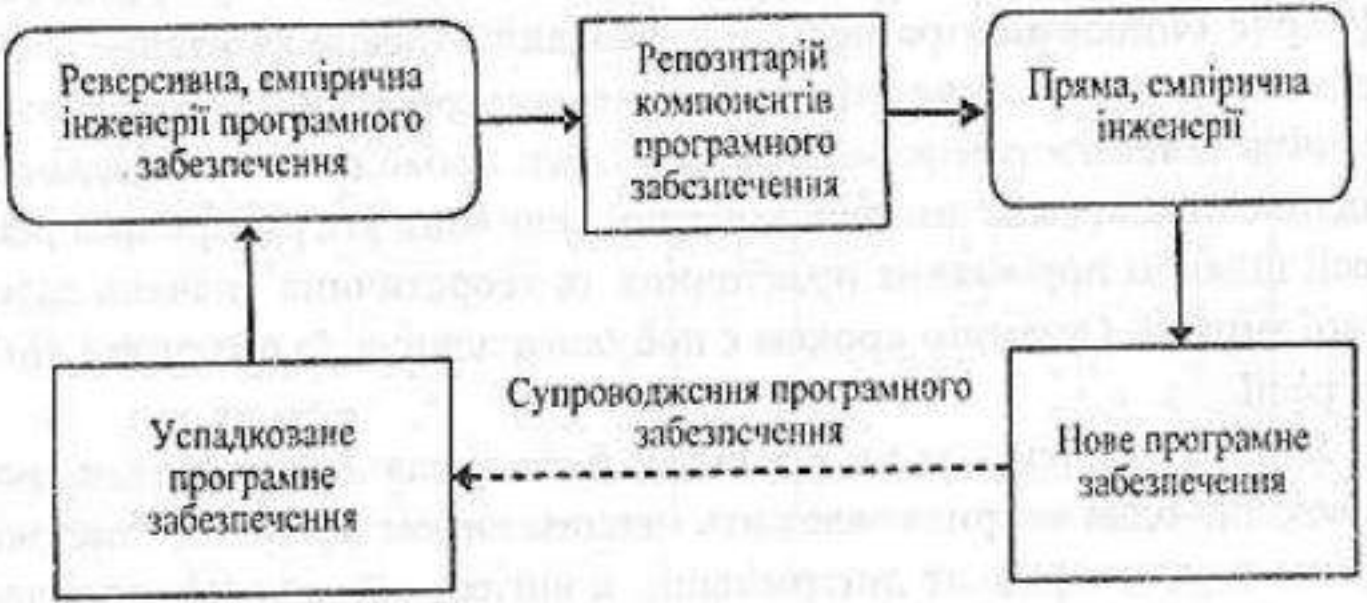


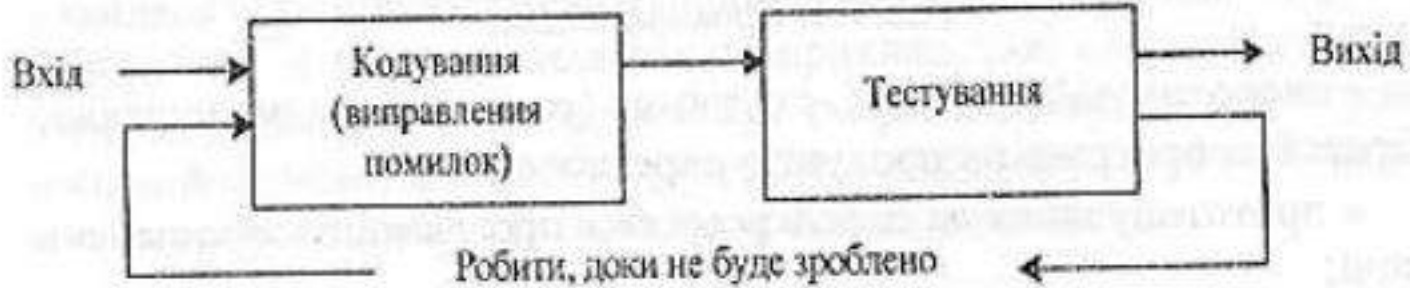
в

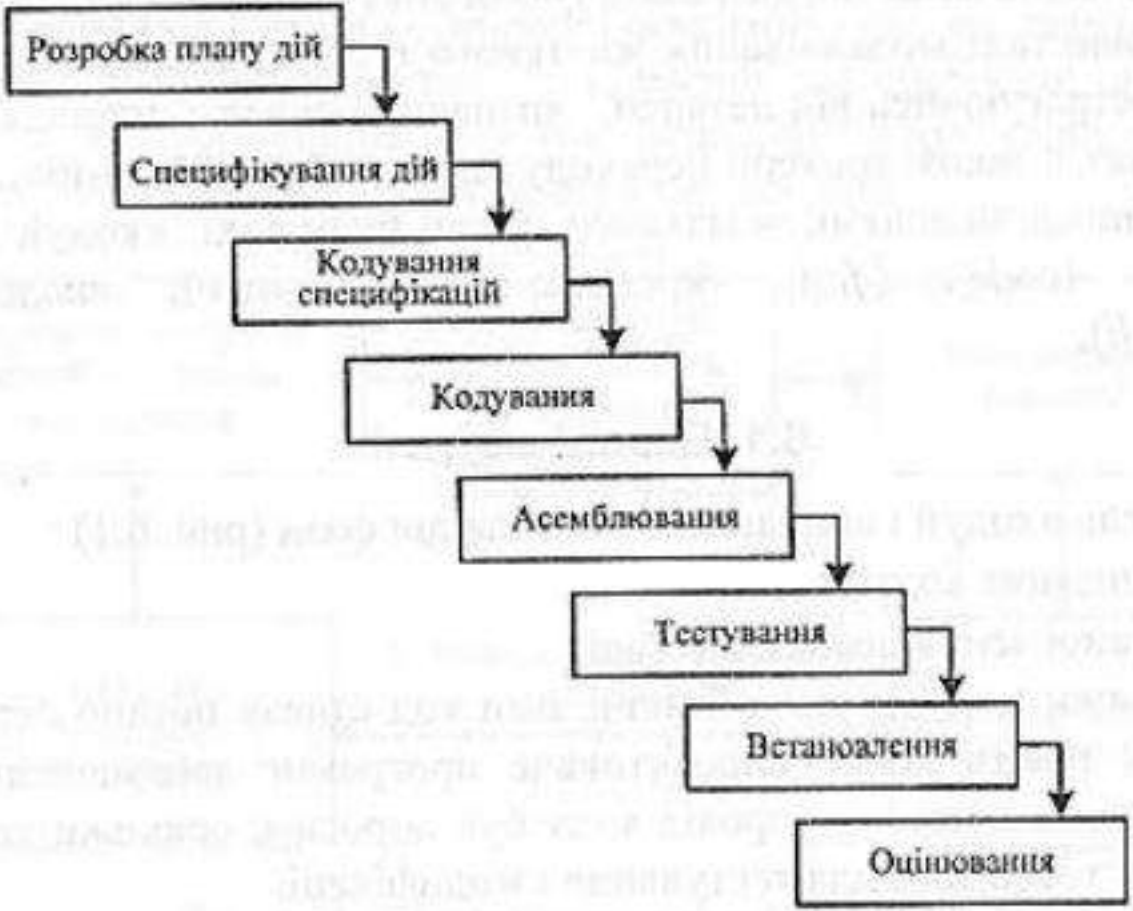


г


CASE
Проблема
Аналіз вимог
Проектування експерименту
Експеримент
Аналіз експериментів
Знання про проблему

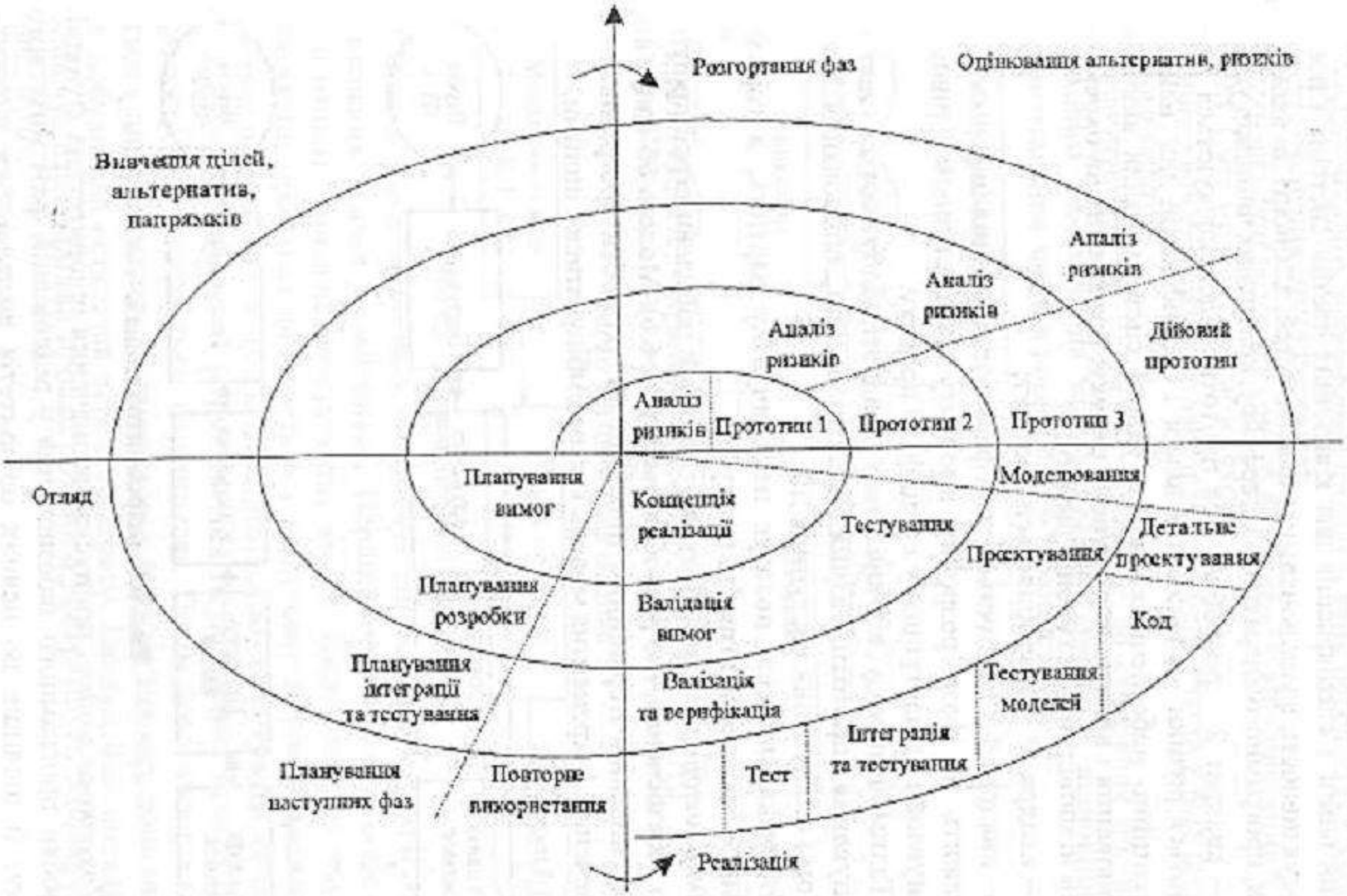

Реверсивна, емпірична інженерії програмного забезпечення
Репозитарій компонентів програмного забезпечення
Пряма, емпірична інженерії
Успадковане програмне забезпечення
Супроводження програмного забезпечення
Нове програмне забезпечення


Вхід
Кодування
(виправлення
помилок)
Тестування
Вихід
Робити, доки не буде зроблено

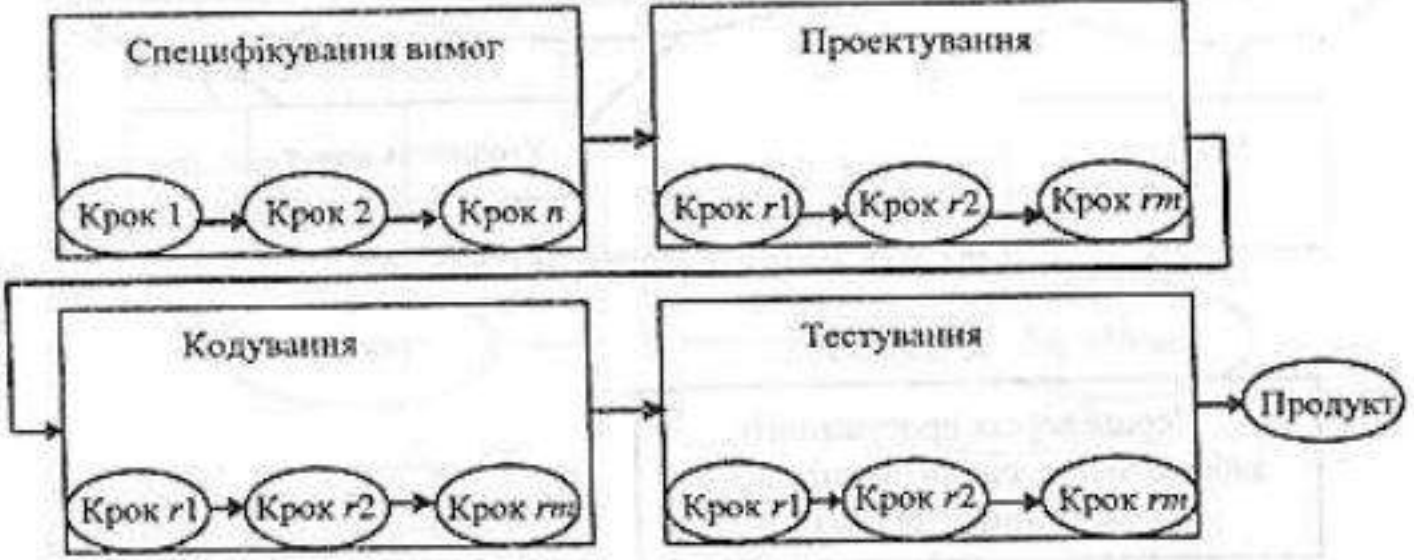

Розробка плану дій
Специфікування дій
Кодування специфікацій
Кодування
Асемблювання
Тестування
Ветанюалення
Оцінювання


Детальне проектування
Верифікування
Кодування
Тестування модулів
Інтеграція
Вірифікування продукту
Реалізація
Системне тестування
Експлуатація супровід
Валідація

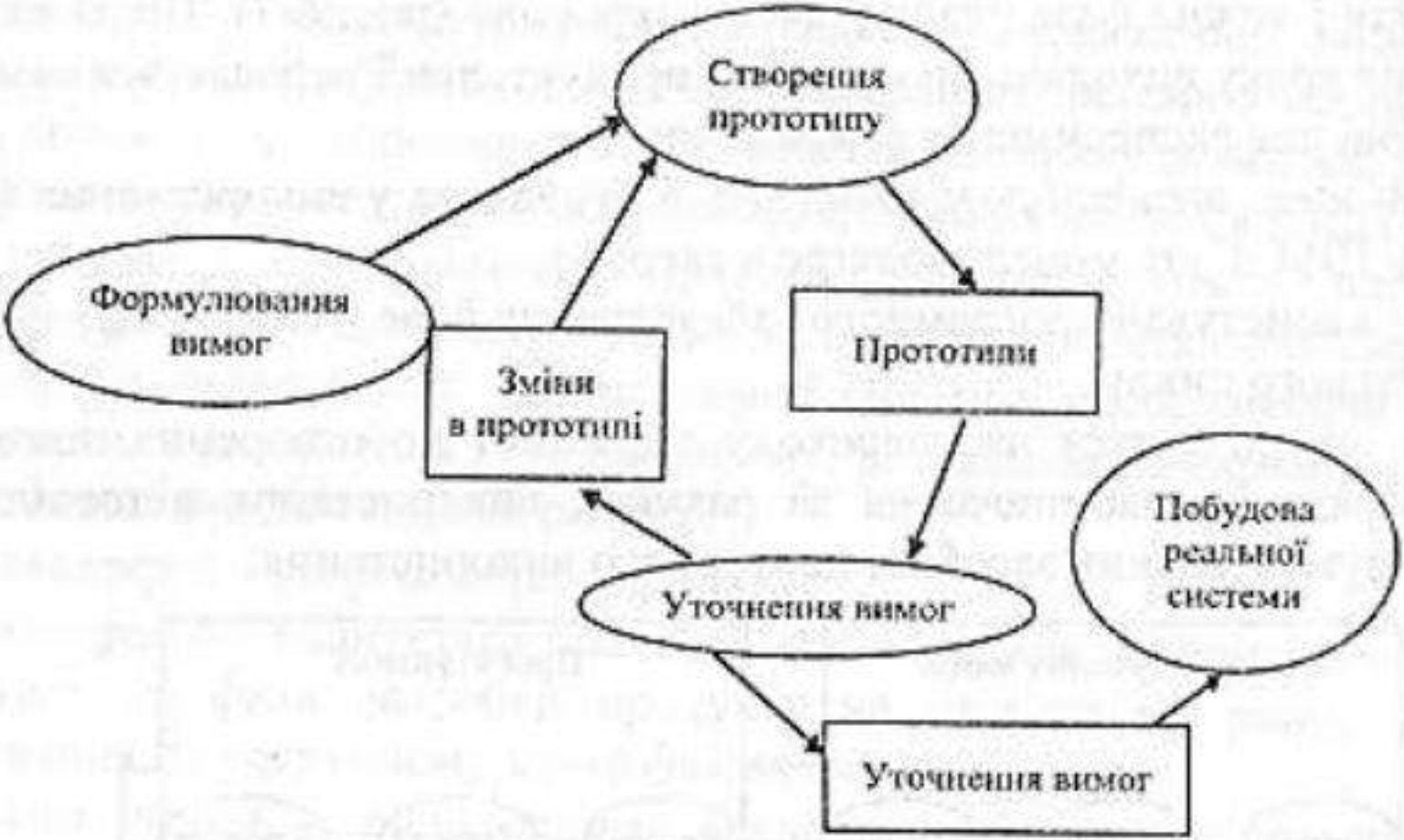

Специфікування вимог
Архітектурне проектування
Детальне проектування
Кодування, тестування
Супровід

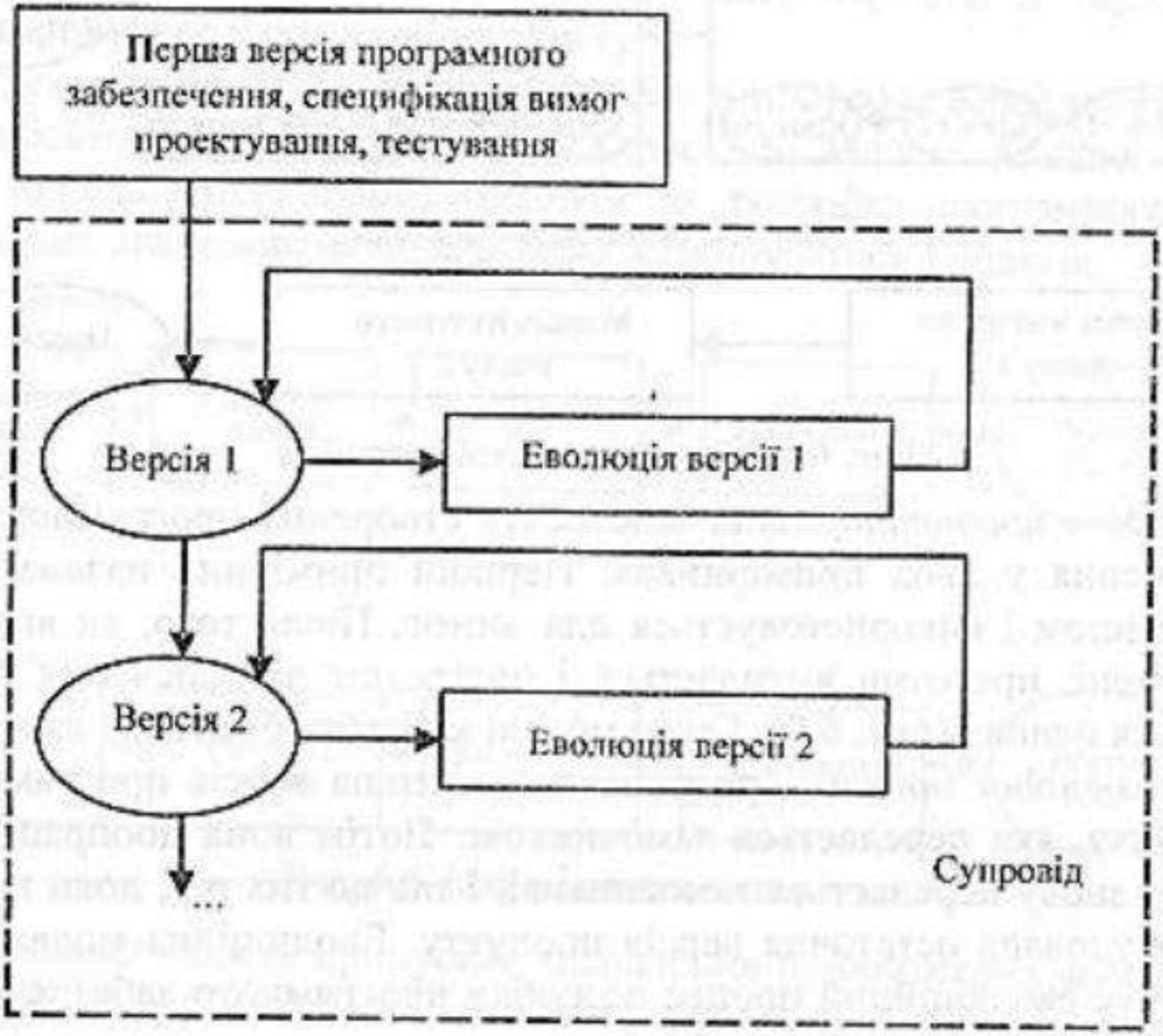

Розгортання фаз
Оцінювання альтернатив, ризиків
Визначення цілей, альтернатив, напрямків
Аналіз ризиків
Аналіз ризиків
Аналіз ризиків
Дійовий прототип
Аналіз ризиків
Прототип 1
Прототип 2
Прототип 3
Огляд
Планування вимог
Концепція реалізації
Тестування
Моделювання
Проектування
Детальне проектування
Планування розробки
Валідація вимог
Код
Планування інтеграції та тестування
Валізація та верифікація
Тестування моделей
Інтеграція та тестування
Тест
Планування наступних фаз
Повторне використання
Реалізація

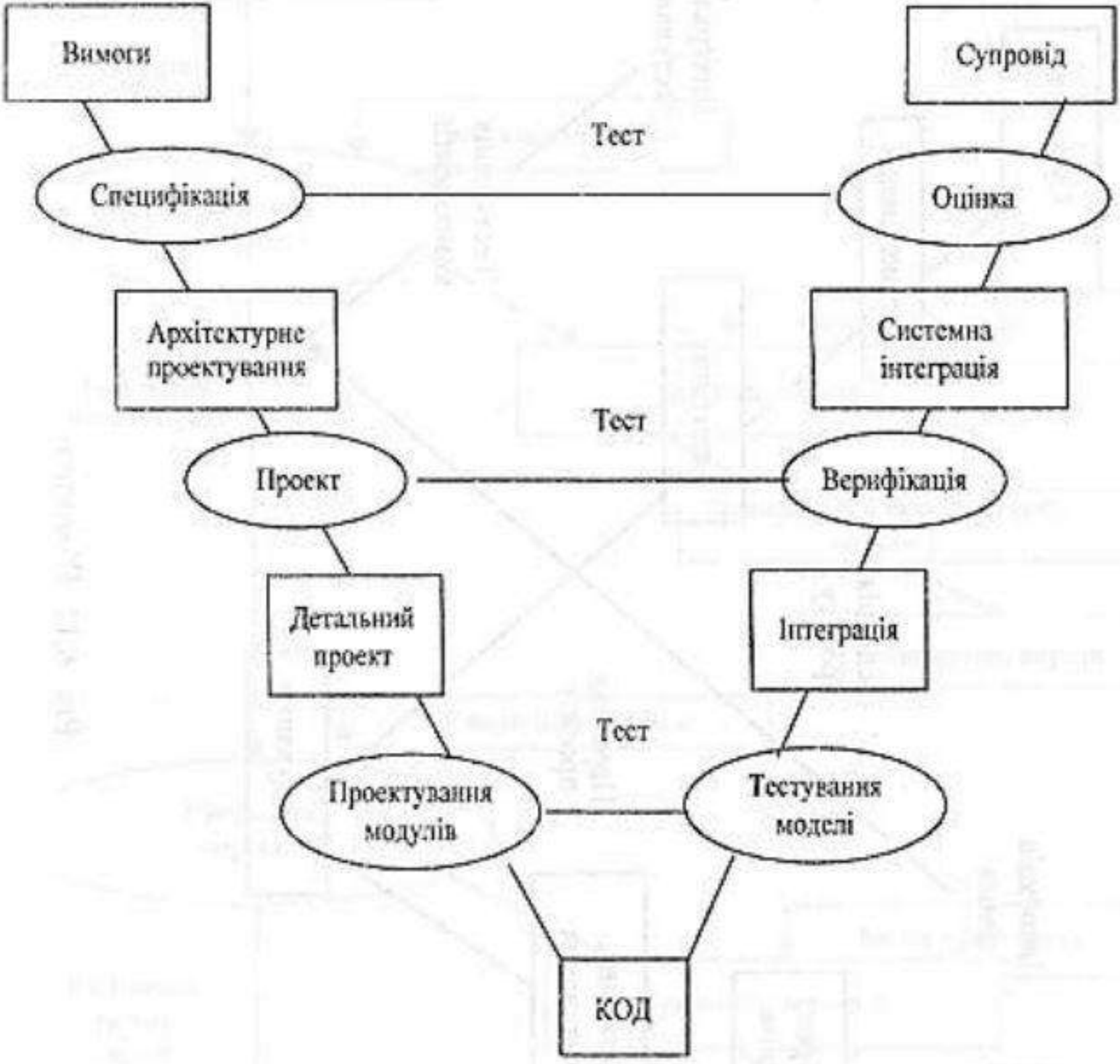

1-й інкремент
Аналіз вимог
Проектування
Кодування
Тестування
Версія ПЗ 1
n-й інкремент
Аналіз вимог
Проектування
Кодування
Тестування
Версія ПЗ 2

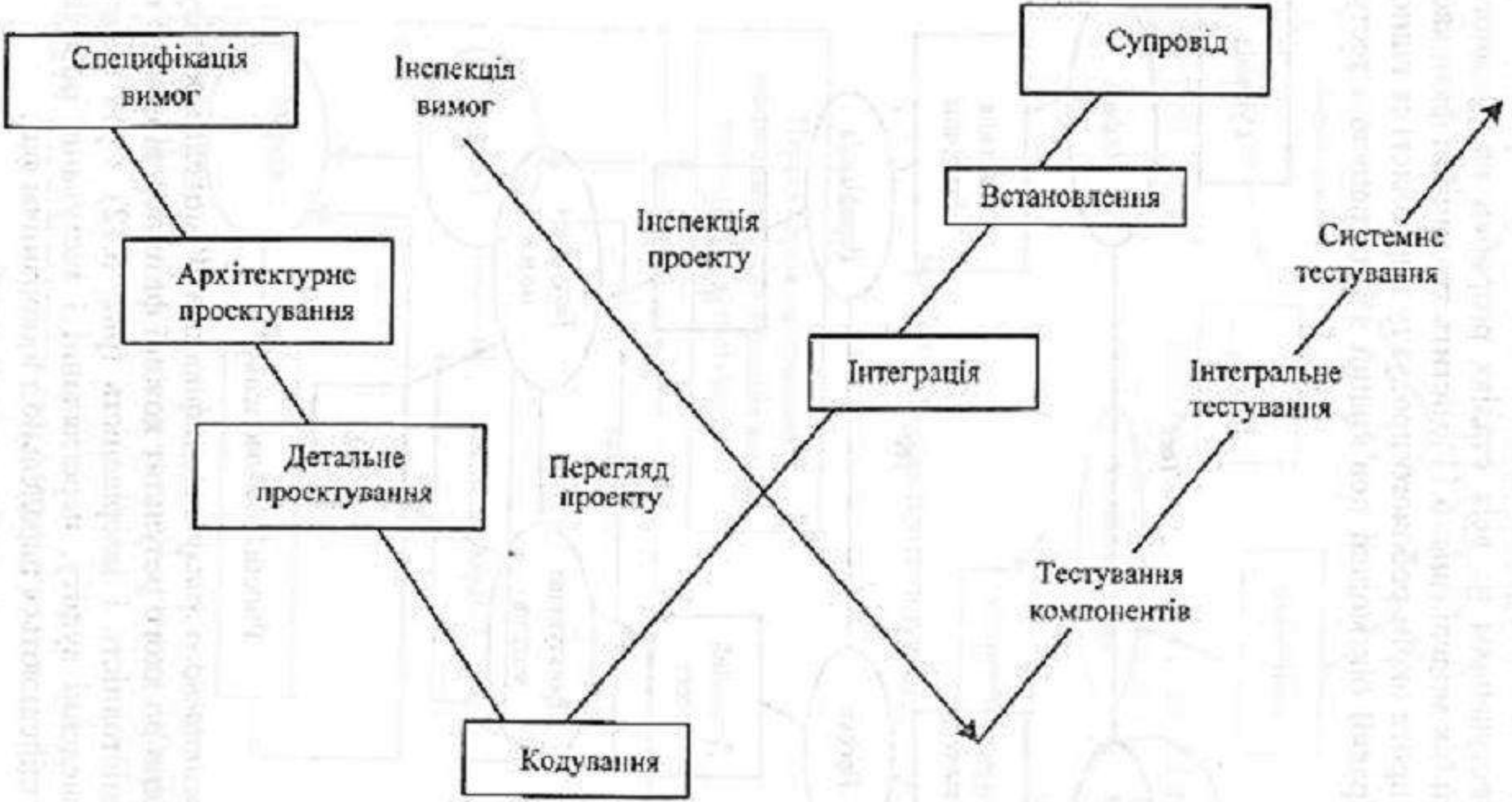

Специфікування вимог
Крок 1
Крок 2
Крок n
Проектування
Крок r1
Крок r2
Крок rm
Кодування
Крок r1
Крок r2
Крок rm
Тестування
Крок r1
Крок r2
Крок rm
Продукт

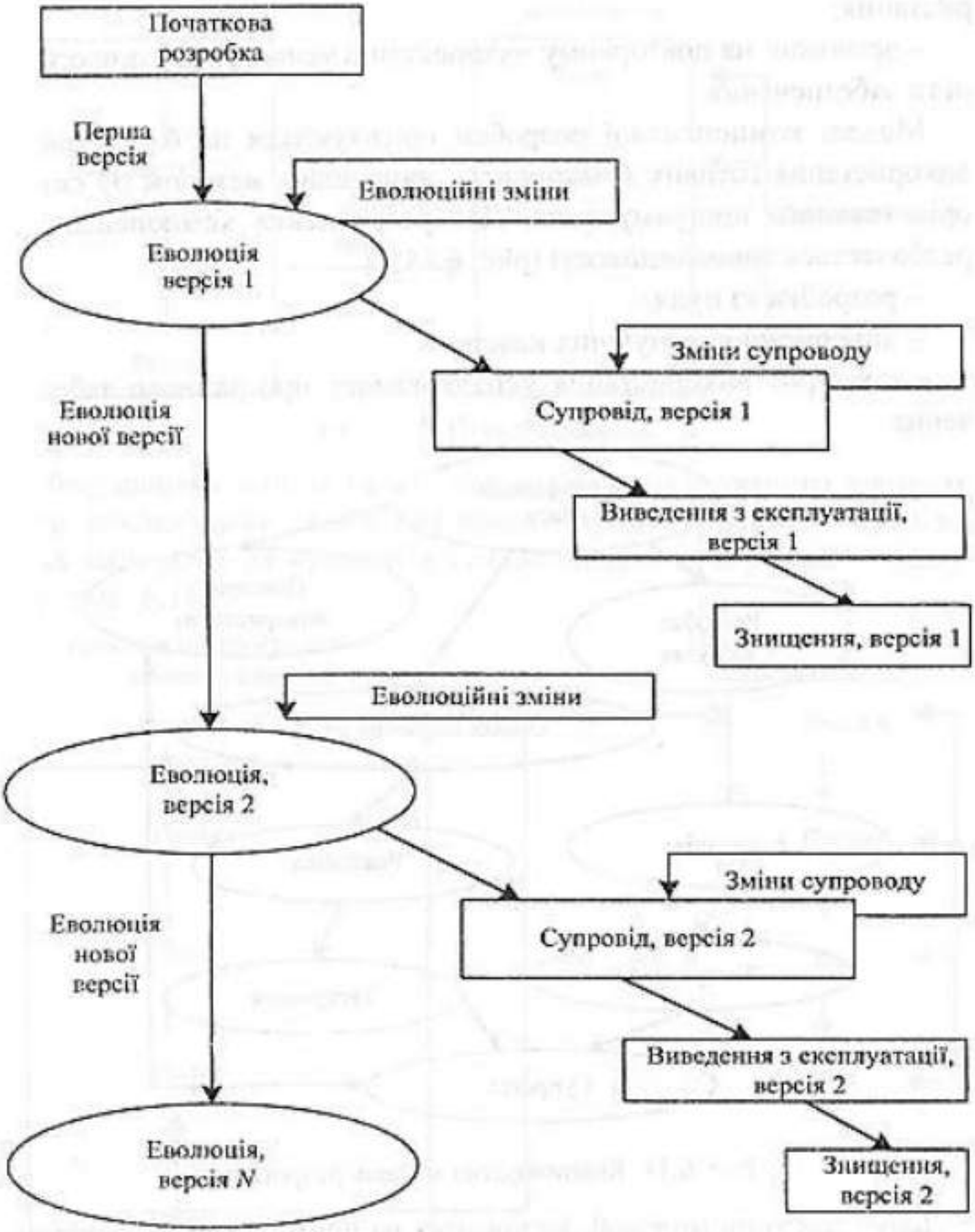

Модель життєвого циклу 1
Модель життєвого циклу 2
Продукт

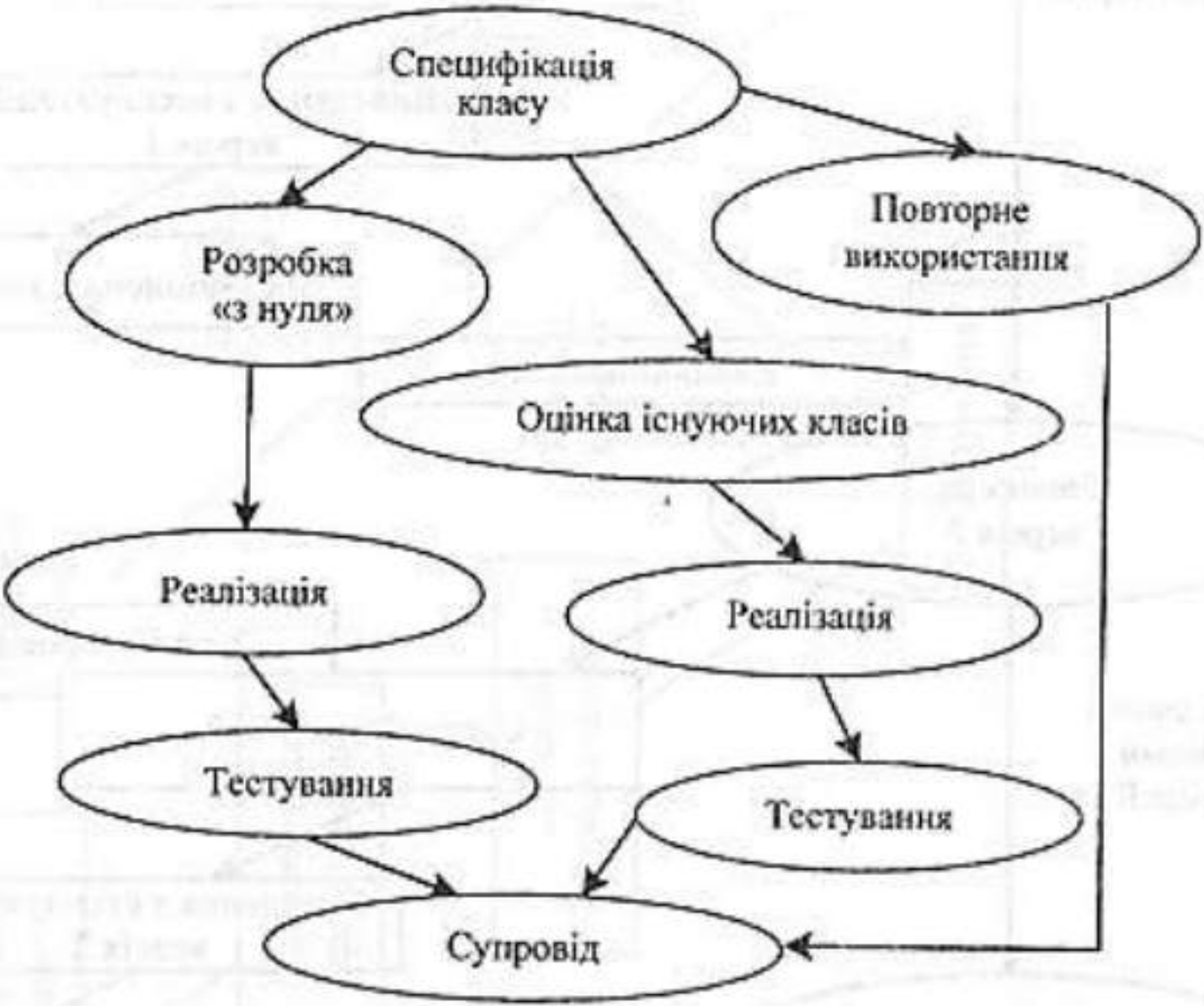

Створення прототипу
Формулювання вимог
Зміни в прототипі
Прототипи
Побудова реальної системи
Уточнення вимог
Уточнення вимог

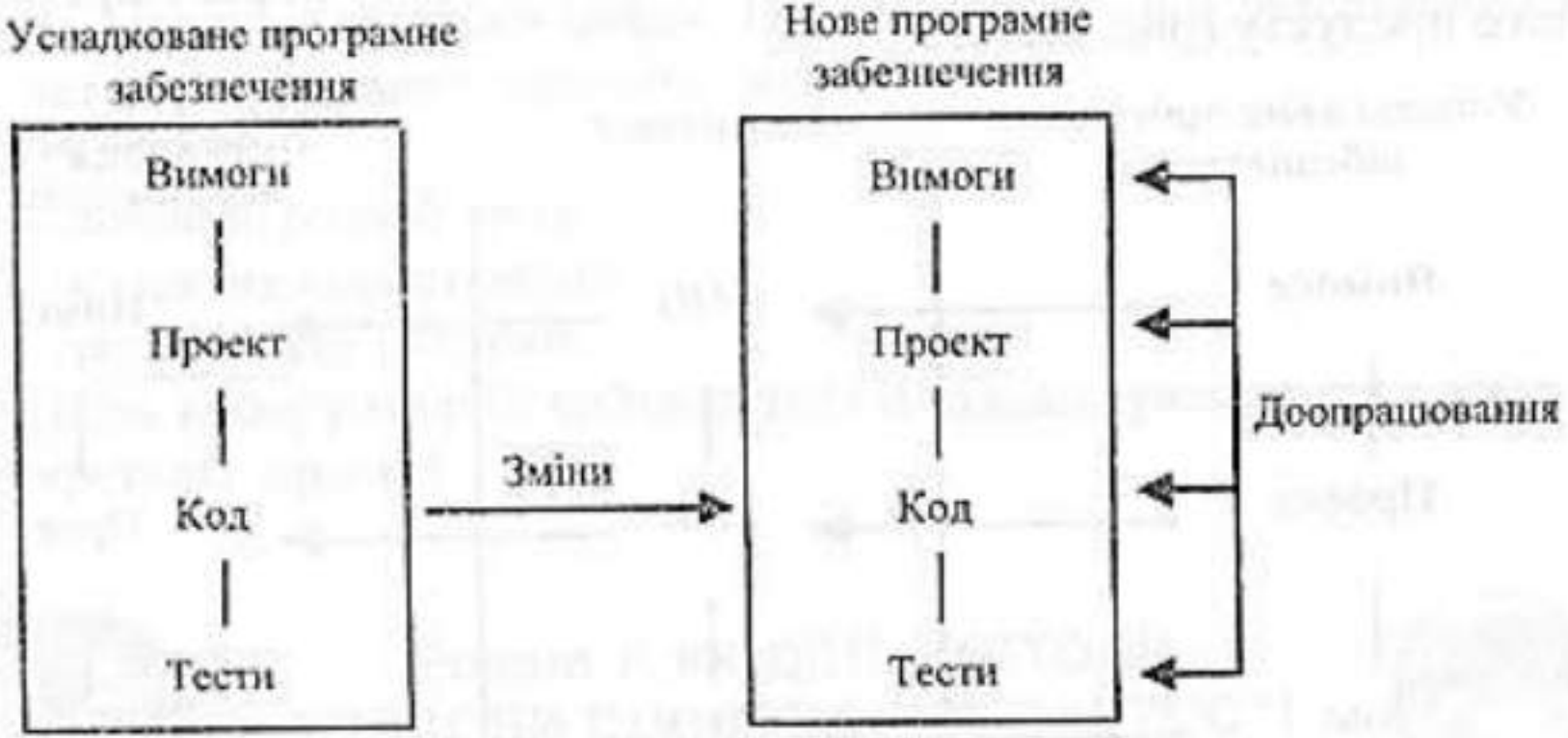

Перша версія програмного забезпечення, специфікація вимог проектування, тестування
Версія 1
Еволюція версії 1
Версія 2
Еволюція версії 2
...
Супровід

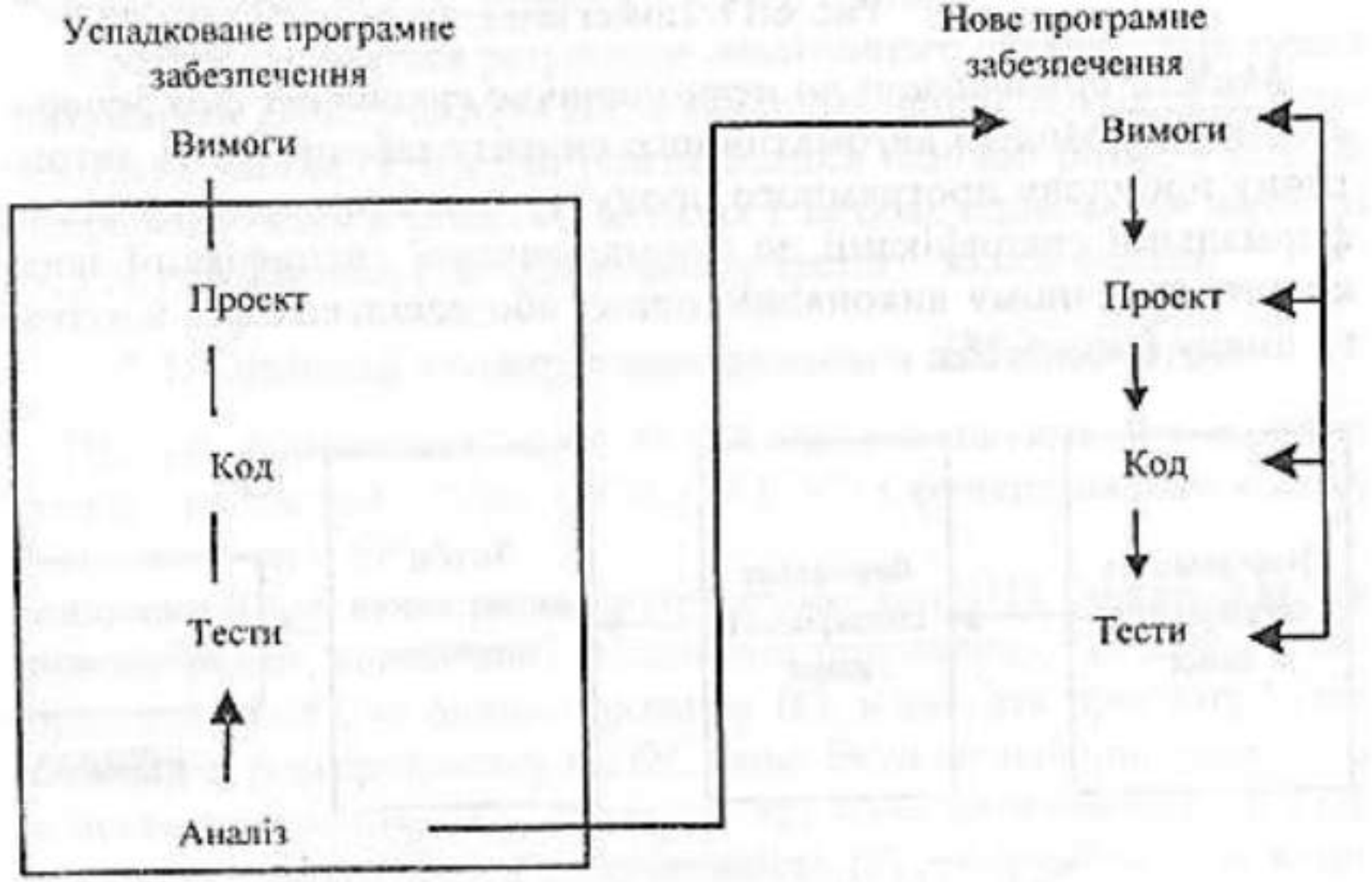

Вимоги
Супровід
Тест
Специфікація
Оцінка
Архітектурне проектування
Системна інтеграція
Тест
Проект
Верифікація
Детальний проект
Інтеграція
Тест
Проектування модулів
Тестування моделі
КОД

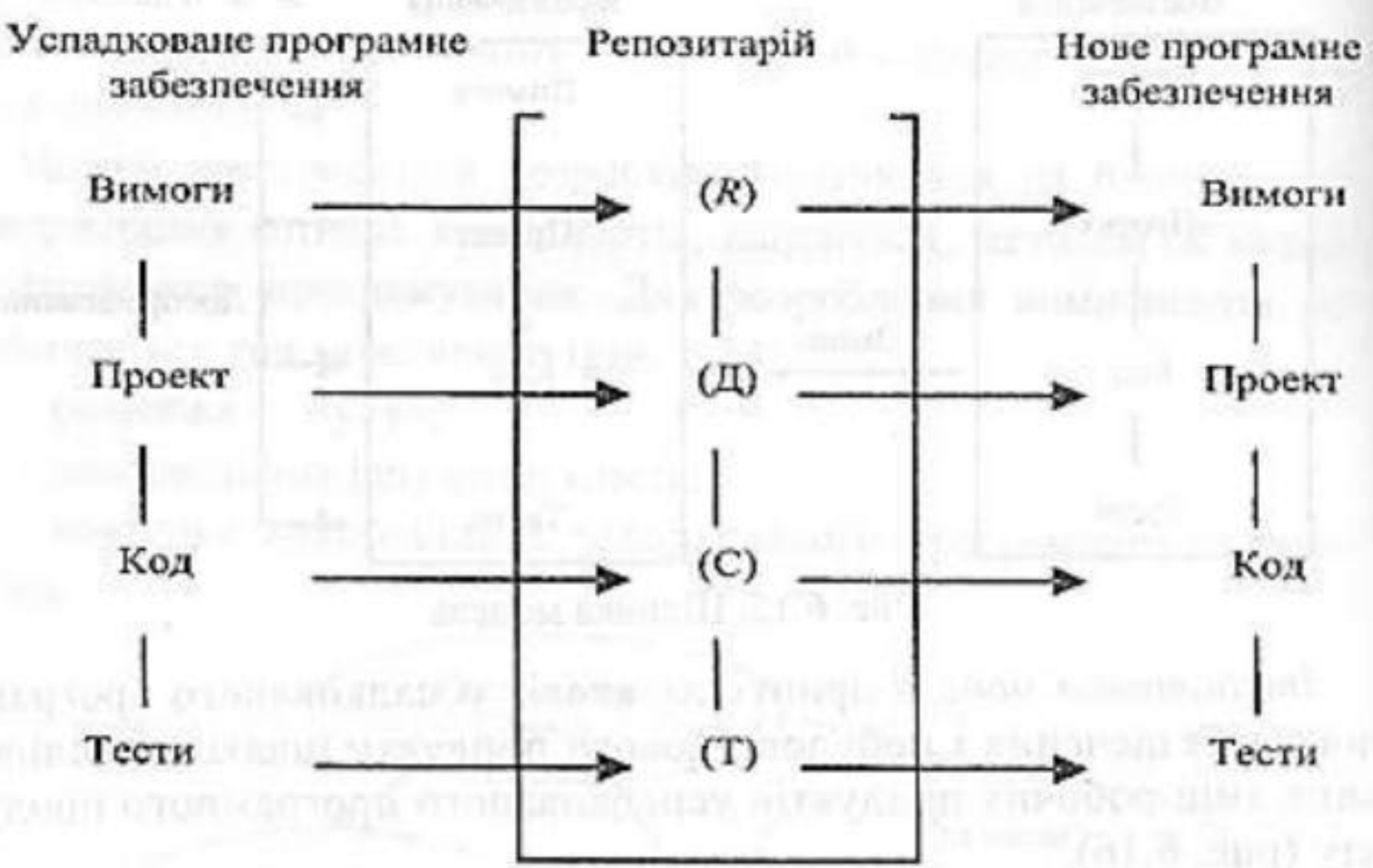

Специфікація вимог
Архітектурне проектування
Детальне проектування
Кодування
Інтеграція
Встановлення
Супровід
Інспекція вимог
Інспекція проекту
Перегляд проекту
Системне тестування
Інтегральне тестування
Тестування компонентів


Початкова розробка
Перша версія
Еволюційні зміни
Еволюція версія 1
Зміни супроводу
Супровід, версія 1
Еволюція нової версії
Виведення з експлуатації, версія 1
Знищення, версія 1
Еволюційні зміни
Еволюція, версія 2
Зміни супроводу
Супровід, версія 2
Еволюція нової версії
Виведення з експлуатації, версія 2
Знищення, версія 2
Еволюція, версія N


Специфікація класу
Розробка «з нуля»
Повторне використання
Оцінка існуючих класів
Реалізація
Реалізація
Тестування
Тестування
Супровід


Успадковане програмне забезпечення
Вимоги
Проект
Код
Тести
Зміни
Нове програмне забезпечення
Вимоги
Проект
Код
Тести
Доопрацювання


Успадковане програмне забезпечення
Вимоги
Проєкт
Код
Тести
Аналіз
Нове програмне забезпечення
Вимоги
Проєкт
Код
Тести


Успадковане програмне забезпечення
Репозитарій
Нове програмне забезпечення
Вимоги
(R)
Вимоги
Проект
(Д)
Проект
Код
(С)
Код
Тести
(Т)
Тести


Неформальна специфікація вимог
Формальна специфікація вимог
Засоби автоматичного виконання
Код програмного забезпечення