Модуль IV. Основи проектування та розробки інформаційних систем

4.1. ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Інженерія програмного забезпечення – це область комп'ютерних наук, яка має справу з процесом побудови програмних систем, які є достатньо великими або складними, щоб потребувати розробки командою чи командами інженерів. Зазвичай такі системи програмного забезпечення існують у декількох версіях і використовуються продовж багатьох років. Протягом життя їх піддають багатьом змінам, щоб:

- виправити виявлені помилки;
- покращити особливості їх функціонування;
- додати нову функціональність або видалити існуючі, але вже непотрібні та застарілі, особливості роботи;
- пристосувати до використання в новому середовищі.

Існує очевидна різниця між *програмуванням* (у вузькому розумінні, тобто *кодуванням* програмної системи) і програмною інженерією. Термін "програміст" у цьому випадку використовується, щоб позначати індивідуума, який відповідає за деталі реалізації програмної системи: розробку та модифікацію алгоритмів і структур даних за допомогою специфічної мови програмування. *Інженер* програмного забезпечення також повинен мати значні навички та досвід програміста для того, щоб добре розуміти проблемні області роботи, цілі й завдання розробки програмного забезпечення. Але додатково інженери програмного забезпечення відповідають за управління проектом розробки програмної системи та за вирішення проблем:

- аналізу задачі розробки програмної системи;
- проектування;
- перевірки адекватності;
- тестування;
- виготовлення документації;
- обслуговування програмного забезпечення.

Програміст кодує закінчену в певному сенсі задачу та продукує відповідну завершену програму, тоді як інженер програмного забезпечення забезпечує специфікацію та розробку програмних

компонент, що комбінуватимуться з іншими компонентами, які, можливо, є написаними іншими інженерами програмного забезпечення, – для побудови закінченої цільової програмної системи.

Зазначимо, що програмний компонент, написаний одним індивідуумом, цілком може в подальшому існуванні підтримуватися та змінюватися зовсім іншою людиною – для виготовлення нових версій даної програмної системи, навіть через тривалий час потому, як перший з розробників залишить даний проект.

Таким чином, програмування – це, перш за все, особиста діяльність, тоді як інженерія програмного забезпечення є, по суті, діяльністю командною.

4.1.1. Необхідність програмної інженерії

У ранні часи розвитку інформаційно-комп'ютерних технологій самі користувачі розробляли програмне забезпечення. Користувачів було небагато, і вони були зазвичай вченими та інженерами, які допомагали будувати відповідні технічні засоби. Ранні комп'ютерні програми існували у формі машинної мови та мови асемблера. Звичайно, їх логічна складність була невисокою. Програмістами були інженери з електроніки, хто власне й будував комп'ютери та вимушений був займатися програмуванням як побічною діяльністю. Таким чином, продовж довгого часу інженери, які писали програми, одночасно були тими, хто їх виконував і виправляв проблеми, знайдені протягом виконання.

Комп'ютерні системи в той час були зосереджені на вирішенні проблем, пов'язаних із подоланням обчислювальної складності прикладних задач. Таким чином, основний акцент робився на досягненні практичних результатів, а зручні та елегантні користувацькі інтерфейси не цікавили розробників. За цих обставин не існувало нагальної потреби у структурованих незалежних від особистості методах для створення та підтримки програмного забезпечення.

З часом технічні засоби стали більш потужними та дешевими одночасно, і спрямування комп'ютерних програм також було змінено. Апаратні засоби стали вважатися більш загальним інструментом, а прикладні потреби інформаційних систем стали задовольнятися за допомогою програмного забезпечення, яке було написане із використанням мов високого рівня.

Загальний успіх галузі інформаційно-комп'ютерних технологій збільшив запити на створення нового програмного забезпечення для подальшої підтримки та розширення існуючих програмних систем. Це,

в свою чергу, призвело до підвищення попиту на програмістів. Люди, які займалися написанням програмного забезпечення, не були вже більше інженерами, досконало знайомими з внутрішнім наповненням комп'ютерної техніки. І, що дуже важливо, вони вже зазвичай не були людьми, які досконало знали конкретну предметну область та відповідні проблеми, яка повинні були вирішуватись за допомогою розробки програмного забезпечення.

На початку 1960-х років:

- Відбувся сплеск великої кількості невдалих проектів програмного забезпечення.
- Багато проектів програмного забезпечення були здані запізно, із перевищенням запланованого бюджету, і призвели до виробництва ненадійних програмних систем, ресурсоємних у подальшій підтримці.
- Багато проектів програмного забезпечення призвели до виробництва програмних систем, які не відповідали сформульованим замовником вимогам.
- Складність проектів програмного забезпечення зросла одночасно із зростанням складності апаратних засобів.
- Великі програмні системи були більш важкими та складними в подальшій підтримці.
- Попит на нове програмне забезпечення збільшувався швидше, ніж здатність генерувати нові програмні системи.

Всі згадані вище особливості галузі відомі, як "криза програмного забезпечення". Термін "інженерія програмного забезпечення" був введений на конференції в кінці 1960-х років – під час обговорення кризи програмного забезпечення. На конференції були озвучені думки та пропозиції розглядати розробку програмного забезпечення як інженерну проблему, що повинна вирішуватися відповідними розвиненими теоріями, методами та інструментами.

Як тільки потреба в розробці дисципліни інженерії програмного забезпечення була ідентифікована, 1970 роки побачили бурхливий розвиток інженерних принципів розробки програмного забезпечення. 1980 роки відзначені появою технологій автоматизації інженерії програмного забезпечення та зростанням програмних систем автоматизованої розробки програмного забезпечення (CASE, Computer Aided Software Engineering).

Подальші роки побачили збільшення акцентів на управлінні проектами розробки програмного забезпечення та використанні стандартів якості продукту та процесу розробки (наприклад ISO 9001).

Базове визначення функцій дисципліни інженерії програмного забезпечення було дано Ф. Бауером як “встановлення та використання надійних технічних принципів для економного вироблення програмного забезпечення, яке є надійним і ефективним для застосування на обчислювальній техніці”.

За іншим популярним визначенням, інженерія програмного забезпечення – це “практичне застосування наукового знання до проектування та конструювання комп’ютерних програм і пов’язаної з ними документації з точки зору їх розробки та обслуговування”.

Стандарт IEEE визначає інженерію програмного забезпечення як:

(1) Прикладення систематичного, дисциплінованого, вимірного підходу до розробки програмного забезпечення, операцій з ним та його обслуговування; тобто прикладення інженерних принципів до програмних систем;

(2) Дисципліна, яка вивчає підходи до (1).

Комбінуючи зазначені визначення, ми можемо сказати, що інженерія програмного забезпечення – це дисципліна, яка вивчає питання технології та управління систематичного виробництва і обслуговування програмних продуктів, які розвиваються та змінюються в часі й межах оцінки вартості.

Інженерія програмних систем як дисципліна забезпечує нас структурованими технічними засобами розвитку та підтримки програмного забезпечення. Вона надає методи для виконання типових завдань створення будь-якого програмного забезпечення: аналіз вимог, проектування системи, що відповідає таким вимогам, розробка програми, підтримка програмної системи, і т. ін. Інструментальні програмні засоби використовуються для того, щоб автоматизувати такі завдання або деякі етапи цих завдань.

4.1.2. Зв’язки з іншими дисциплінами

Інженерія програмного забезпечення є дисципліною, яка має свої особливості, але базується на таких дисциплінах:

- комп’ютерні науки;
- системний аналіз;
- економічна теорія;
- менеджмент;
- комунікації в менеджменті.

Комп’ютерні науки надають наукову основу інженерії програмного забезпечення (тою ж мірою, як, наприклад, радіотехніка залежить і базується на поняттях фізики).

Економічна наука забезпечує основу для оцінки ресурсів і контролю за рівнем витрат. Програмні системи повинні розвиватися і підтримуватись, що призводить до значної ролі економіки в інженерії програмного забезпечення.

Системний аналіз надає методи вивчення складних систем. Зазвичай програмне забезпечення є компонентом набагато складнішої системи. Наприклад програмне забезпечення на виробництві чи в керуванні складними технічними засобами є тільки одною з багатьох задіяних підсистем.

Менеджмент як наука управління забезпечує основу для керування проектом програмного забезпечення.

Розвинені здібності до *комунікації в менеджменті* (письмової, усної, міжособистісної) є критичними якостями для спеціалістів з інженерії програмного забезпечення, тому що технічні дії з виробництва програмного забезпечення відбуваються в межах організаційного контексту, і абсолютно необхідною при цьому є дійсно високий рівень взаємних зв'язків серед клієнтів, менеджерів, спеціалістів з інженерії програмного забезпечення, інженерів апаратних засобів та інших технічних працівників.

Отже інженерія програмних систем передбачає, що різноманітні концепції з комп'ютерних наук, системного аналізу, економічної теорії, менеджменту та комунікацій у менеджменті комбінуються в рамках вирішення технічної проблеми розробки програмного забезпечення.

4.1.3. Особливості інженерії програмного забезпечення

Програмне забезпечення – це різновид інформації, а не фізичне тіло. Воно не має маси, об'єму, кольору, аромату – жодних фізичних властивостей. Програмний код – це просто статичне зображення комп'ютерної програмної системи, і хоча ефекти, які продукує програма, є зазвичай помітними сторонньому спостерігачеві, сама програма безпосередньо – ні.

Проект програмного забезпечення можна порівняти із архітектурним проектом в умовах відсутньої гравітації: великий ступінь свободи є одночасно благословенням і прокляттям інженерії програмного забезпечення.

Програмне забезпечення фізично не старіє з часом, як це відбувається з технічними засобами. Програмні помилки є викликаними недоліками проектування або реалізації початкового задуму, а не старінням з часом програмної системи самої по собі. (Окремий випадок – умовне старіння програми через розвиток

навколишнього середовища, в якому така програмна система працює, тобто досягнення грані, за якою програма не відповідає умовам нормального функціонування в своєму довкіллі.)

Основний принцип управління великою системою є її декомпозиція на менші, краще керовані одиниці з добре проробленими зрозумілими інтерфейсами. Цей підхід "поділити та завоювати" є рутинною процедурою в інженерних дисциплінах. У продукуванні програмного забезпечення одиниці декомпозиції називаються *модулями*.

Модулі в програмному забезпеченні мають як інтерфейси для контролю, так і інтерфейси за даними (інформаційні з'єднання). Остання група інтерфейсів реалізує передачу параметрів між модулями, а також спільне володіння глобальними даними. Досить важко спроектувати складну систему програмного забезпечення таким чином, щоб всі контрольні та інформаційні інтерфейси серед модулів були явно визначені. Це, однак, є важливою передумовою надійності програмної системи, оскільки забезпечує відсутність побічних ефектів від взаємодії окремих модулів. Потреба вирішення проблеми проектування інтерфейсів в інженерії програмного забезпечення є тим більш нагальною, що кількість складових частин і зв'язків між модулями для випадку програмної системи є зазвичай набагато більшою, ніж в технічних системах, які мають матеріальне втілення.

У технічних дисциплінах інженер, забезпечений інструментами та завершеністю математичних побудов, щоб конкретизувати властивості продукту окремо від властивостей процесу проектування. Наприклад електротехнік покладається на відомі математичні рівняння, щоб перевірити, чи його проект не порушує вимоги за потужністю. У розробці ж програмного забезпечення, такі математичні інструменти не завжди добре розвинені. Часто типова розробка програмного забезпечення проводиться, покладаючись скоріше на досвід і практичні судження, замість математичних формул. Зазначимо, що досвід, розважливість, проникливість є дійсно необхідними передумовами успішного завершення проекту програмного забезпечення, але формальний аналіз є також надзвичайно важливим для практики розробки програмних систем.

Еволюція дисципліни інженерії програмного забезпечення визначила специфічну роль спеціаліста в цій галузі:

- Інженер програмного забезпечення повинен бути програмістом високої кваліфікації, мати досвід розробки структур даних і алгоритмів, добре розуміти одну або декілька мов програмування.

- Інженер програмного забезпечення повинен бути знайомий з декількома підходами до проектування, бути здатним транслювати нечітко сформульовані вимоги та бажання замовника в точні специфікації.
- Інженеру програмного забезпечення потрібна здатність вільно рухатися поміж декількох рівнів абстракції на різних стадіях проекту розробки програмної системи – від процедур і вимог деякої предметної галузі, через абстрактне представлення програмної системи, до конкретного програмного коду.
- Інженер програмного забезпечення повинен бути здатний побудувати та використовувати модель програмного додатку для того, щоб приймати обгрунтовані рішення щодо компромісів у розробці проекту. Модель використовується для отримання відповіді на запитання як щодо поведінки системи, так і щодо її продуктивності та якості.
- Інженерові програмного забезпечення потрібні розвинені навички комунікації. Йому також потрібна здатність планувати як власну роботу, так і роботу інших учасників проекту.

Таким чином, інженер програмного забезпечення має відповідати за багато речей одночасно. На практиці багато організацій ділять таку велику відповідальність серед декількох фахівців з різними посадами. Так, наприклад, системний аналітик у цьому випадку буде відповідальним лише за аналіз роботи системи.

4.1.4. Характеристики програмного продукту

Будь-яка добре спроектована програмна система повинна мати такі властивості:

- бути такою, що легко підтримується;
- надійність;
- ефективність;
- забезпечувати зручний користувацький інтерфейс.

Процес розробки програмної системи приречений бути серією компромісів між перерахованими атрибутами.

Кінцеве завдання інженерії програмного забезпечення – виробляти програмні продукти. Комп'ютерне програмне забезпечення, тобто продукт проведення інженерного проектування та реалізації, включає до свого складу:

- програмні коди, які виконуються в межах технічних засобів широкого діапазону (швидкодії, пам'яті тощо);
- документації в складі так званих "твердих" копій (тобто на матеріальному носії, папері) та електронних копій;

- даних різноманітного змісту та спрямування (числова форма, текст, аудіо- та відеоінформація, тощо).

Таким чином, програмні продукти – це програмні системи, які постачаються користувачеві разом з документацією та супровідними матеріалами, що описують спосіб встановлення, настройки й використання системи.

Програмні продукти загалом складаються з двох широких класів:

- *Загальна продукція*. Самостійні програмні системи, які виробляються організацією/фірмою з розробки програмного забезпечення та продаються на вільному ринку будь-якому клієнтові, який може купити їх.
- *Спеціалізована продукція*. Системи, які були замовлені специфічним клієнтом. Таке програмне забезпечення будується спеціально для конкретного клієнта деяким розробником.

До 1980-х років переважна більшість програмних продуктів, вироблених на продаж, були спеціалізованими програмами, націленими на виконання на великих машинах і виготовленими за технічними умовами користувача. Їх ціна була високою, тому що вся вартість розробки лягала на єдиного клієнта.

Після початку бурхливого розвитку персональних комп'ютерів ця ситуація кардинально змінюється. Тепер на ринку персональних комп'ютерів повністю превалює продукція неспеціалізованих програмних продуктів від таких компаній, як, наприклад, Microsoft. Вони зазвичай відносно дешеві, тому що вартість їх розробки рознесена по сотням або навіть тисячам різних клієнтів.

Однак все ще існує великий ринок для програмних систем спеціального призначення. Наприклад контроль апаратних засобів завжди вимагає вироблення деякого виду програмних систем спеціального призначення. А оскільки комп'ютери є вбудованими до різноманітних пристроїв, то існує досить значний попит на відповідні контролери.

Найістотніша різниця між спеціалізованими програмними продуктами та системами загального призначення, з точки зору інженерії програмного забезпечення, є те, що для останніх специфікації на розробку створюються внутрішнім чином, відділом маркетингу компанії. Цей погляд на майбутню функціональність програми відображає припущення, які системи будуть продаватися найкращим чином. Тому подібні програмні проекти є гнучкими та ненормативними.

Контрастом до цього стану речей є замовлені програми, в яких специфікації розробляються максимально чіткими та часто фіксуються контрактом між клієнтом і розробником. У таких системах гнучкість неможлива і часто непотрібна, зміни вносяться шляхом переговорів між клієнтом і розробником.

Подібно до звичайних технічних розробок продукування програмного забезпечення є не просто створенням кінцевого продукту, а процесом побудови системи шляхом, максимально ефективним з точки зору використаних коштів. За умови надання необмежених ресурсів, більшість проблем розробки програмного забезпечення ймовірно можуть бути вирішені. Але справжнім викликом для інженерів програмного забезпечення є вироблення кінцевого якісного продукту за умови наявності скінчених ресурсів, передбачених розкладом.

Атрибутами програмного продукту є його характеристики після встановлення та початку нормальної експлуатації. Вони є похідними від динамічної поведінки програмної продукції та способу її використання користувачем. Прикладами цих властивостей є: ефективність, експлуатаційна надійність, стійкість до помилок, здатність до перенесення на інші платформи тощо. Відносна важливість кожної з цих характеристик очевидно змінюється від системи до системи.

Наступна таблиця демонструє деякі важливі якісні атрибути добре спроектованого програмного забезпечення (ПЗ). Проведення оптимізації програмного продукту згідно з цим списком властивостей є досить важким, оскільки деякі риси виключають інші. Наприклад забезпечення кращого інтерфейса користувача, цілком можливо, буде зменшувати ефективність програмної системи. Співвідношення між вартістю та вдосконаленнями в кожній з цих властивостей не є лінійними і включають до себе суб'єктивні фактори, оскільки, перебуваючи в рамках складної системи, програмний продукт схильний бути об'єктом впливу такого погано передбачуваного фактора, як людина – кінцевий користувач.

Характеристика ПЗ	*Опис*
Експлуатаційна надійність	Програмна система повинна мати достатній ресурс для внесення до неї змін, які стали потрібними кінцевому користувачеві.
Здатність викликати довіру користувача	Цей атрибут програмного забезпечення включає до себе широкий діапазон характеристик, таких як надійність; безпечність для середовища, в якому працює ПЗ; захищеність від зовнішніх впливів тощо.
Ефективність	Програма не повинна безоглядно споживати системні ресурси, такі як пам'ять і робочий час процесора.
Зручність	Програмна система повинна мати зручний інтерфейс користувача, довідку та документацію.

4.1.5. Процес побудови програмного забезпечення

Організація процесу побудови програмного забезпечення стає важливою проблемою для компаній, які спеціалізуються на такій продукції. Тому характер цього процесу стає ще більш важливим як для постійних працівників, так і для довгострокових практикуючих фахівців та короткострокових консультантів в індустрії програмного забезпечення.

Процес може бути визначений, як метод виконання або створення чого-небудь. Розширюючи це визначення для специфічного випадку програмного забезпечення, ми можемо сказати, що процес побудови програмного забезпечення – це метод розробки або виробництва програмних систем.

В минулому процеси виробництва програмного забезпечення були надзвичайно залежними від конкретної індивідуальності виконавця. В загальному випадку такий підхід призводив до трьох основних проблем:

- Таке програмне забезпечення надзвичайно важко *підтримувати*. Уявімо собі, що розробник програмного забезпечення отримав травму та вибув із виробництва, а хтось інший приймає його частково завершену роботу. Без

загального підходу до програмної інженерії, схема подальших дій з відмітками про виконані та заплановані завдання або випадково могла існувати (на щастя), або не бути в наявності, постійно перебуваючи лише в голові розробника. Так чи інакше, працівник, який прийшов на заміну, повинен стартувати з чистого листа, оскільки, як би добре не працював попередник, але його методи роботи були індивідуальними, притаманними саме йому, – і часто нова людина не має жодних підказок, з чого продовжувати роботу. Цей процес розробки міг би стати врешті-решт прекрасним за своїм кінцевим результатом, але у всякому разі він залишається індивідуальним і неповторним окремим випадком, а не добре визначеним процесом.

- Дуже важко *точно виміряти якість* готового виробу згідно з деякою незалежною оцінкою. Якщо ми маємо двох розробників програмного забезпечення, кожен з яких працює згідно з їх власним процесом, визначаючи свої власні задачі вздовж свого шляху, ми не маємо ніякого об'єктивного методу для взаємного порівняння їх роботи, або – що є більш важливим – для порівняння з критеріями якості клієнтів.
- У такому випадку існують *великі накладні витрати*, оскільки кожен індивідуум вирішує власні задачі в ізоляції. Кращим підходом, звичайно, є попереднє вивчення досвіду інших розробників програмних систем, хто раніше вже ступив на цю дорогу.

4.1.6. Принципи, методи та інструменти

Принципи інженерії програмного забезпечення мають справу з *процесом* і завершальним *продуктом*. Правильний процес допоможе виробити вірний продукт, але бажана продукція також впливатиме на вибір того, який процес буде використаний. Традиційною проблемою в розробці програмного забезпечення є зосередження лише на процесі, або лише на продукті – без врахування іншого базового питання. Слід пам'ятати, що обидва вони однаково важливі.

Принципи інженерії програмного забезпечення є достатньо загальними, щоб бути застосованими всюди у процесі розробки програмних систем і керування програмними проектами. Їх, однак, недостатньо, щоб направляти розробку програмного забезпечення. Фактично вони – просто найбільш загальні й абстрактні твердження, які описують бажані властивості процесу розробки програмного забезпечення та кінцевої продукції. Але, щоб застосувати ці принципи,

інженера програмного забезпечення потрібно забезпечити відповідними методами та конкретними технічними прийомами, які допомагають впроваджувати бажані властивості в процеси та продукцію.

Методи програмної інженерії забезпечують технічні рецепти для побудови програмного забезпечення. Методи вирішують великий обсяг задач, серед яких: аналіз вимог, проектування, кодування, тестування і підтримка.

Технічні прийоми (техніки) програмного забезпечення є більш конкретними та механічними, ніж методи; часто вони також мають більш обмежену застосовність. Взагалі різниця між методом і технічним прийомом (технікою) не є гострою. Іноді методи і технічні прийоми інтегрують, щоб сформувати методологію. Задача методології – просувати певний підхід до вирішення проблеми, відбираючи заздалегідь належні методи та технічні прийоми.

Елемент	*Приклад*
Принципи	Модульність, абстрагування, поділ праці, розробка узагальнюючих підходів, поступове зростання і т. ін.
Методології	Модель водоспаду, еволюційна модель, модель трансформацій, спіральна модель і т. ін.
Методи	Різноманітні підходи до проектування, створення специфікацій, верифікації програмного забезпечення.
Технічні прийоми	Розбиття на модулі: зверху вниз, знизу догори, покрокове покращення і т. ін. Операційна специфікація: схема потоків даних, скінчений автомат, мережі Петрі й т. ін.

Інструментальні засоби програмної інженерії забезпечують автоматизовану або частково автоматизовану підтримку застосування методів, технічних прийомів і методологій. Коли інструменти об’єднані в систему для підтримки розробки програмного

забезпечення таким чином, що інформація, створена одним інструментом, може використовуватися іншим, це називають *системою автоматизованого проектування та створення програмного забезпечення* (Computer Aided Software Engineering, CASE). CASE об'єднують програмні та технічні засоби і базу даних (*репозиторій*, який містить важливу інформацію про аналіз, проектування, кодування та тестування) – з метою створення *середовища* програмної інженерії.

Оскільки технологія еволюціонує, то й інструментальні засоби для створення програмного забезпечення також еволюціонуватимуть. Трохи повільніше змінюються також методи і технічні прийоми – з тим, як збільшуються наші знання про особливості програмного забезпечення. Принципи є найбільш сталими елементами цієї системи, оскільки вони складають основу, на якій можуть бути відбудовані всі інші елементи.

4.1.7. Дискретні та неперервні дії програмної інженерії

Розробка програмного забезпечення часто розглядається як серія окремих дискретних дій (наприклад проектування, кодування, тестування), які приводять до готового виробу. Проте більш коректним є підхід, за яким якісне програмне забезпечення створюється не за допомогою множини *дискретних* процесів, а шляхом спілкування *безперервних* процесів, які направляють окремі дії ходу розробки програмного забезпечення.

Дійсно, більшість цих дій є безперервними, тобто такими, що виконуються впродовж всього процесу розробки програмного забезпечення. Деякі з цих дій, наприклад аналіз, проектування, кодування, є дискретними. Дискретні дії проводяться згідно з *життєвим циклом* розробки.

Вибір належного життєвого циклу є надзвичайно важливим для успіху всього проекту розробки програмного забезпечення. Наступна діаграма (рис. 4.1) показує взаємовідношення між дискретними та безперервними діями.

Аналіз	*Проектування*	*Кодування*	*Тестування*	*Підтримка*
План обслуговування				
Аналіз якості				
Контроль виробництва та перевірка адекватності				
Аналіз ризиків, керування при допущенні ризику				
Управління конфігурацією				
Управління проектом розробки програмного забезпечення				

Рис. 4.1. Приклади дискретних та безперервних дій

Верхній рядок наведеної таблиці складається з дискретних дій, кожна з яких має чіткий вхід і критерії виходу (хоча вони можуть бути такими, що повторюються). Нижчі рядки являють собою безперервні дії, які повинні продовжуватися весь час існування проекту розробки програмного забезпечення.

Отже дискретними діями при розробці програмного забезпечення вважають такі дії, які мають встановлений вхід і критерії виходу.

Наприклад аналіз вимог має вхідний критерій “Ми маємо проблему і хотіли мати рішення у вигляді програмного забезпечення”. Критерієм виходу є: “ми розуміємо вимоги, що ставить проблема, яку ми вирішуємо, і зібрали достатньо інформації для проектування програмного забезпечення, яке здатне вирішити поставлену проблему”.

Вхідним критерієм проекту є: “Чи достатньо зрозумілими є вимоги для того, щоб проектувати реальну систему?”, а критерієм виходу є: “Чи можна починати створення програмного коду, який відповідає спроектованим інтерфейсам і виконує заплановані проектом дії?”.

Зазвичай першою діяльністю у багатьох проектах з розробки програмного забезпечення є збір та аналіз вимог. Цей процес є частиною *розробки вимог*.

4.1.8. Вибір моделі життєвого циклу

Існує велика різноманітність моделей процесів розробки програмного забезпечення. Кожна модель являє собою спробу впорядкувати цю хаотичну по своїй суті діяльність.

Моделі процесу виробництва програмного забезпечення також відомі як *моделі життєвого циклу програмної системи*. Термін “життєвий цикл програми” використовується, щоб описати період часу, який починається із створення концепції системи програмного забезпечення і закінчується з відмовою від використання існуючої програмної системи.

Широкого розповсюдження набули такі моделі життєвого циклу програмної системи:

- модель водоспаду;
- модель створення прототипів;
- модель інкрементальної розробки;
- швидка розробка програмних систем (rapid application development model, RAD);
- модель трансформацій;
- спіральна модель.

Після збору й аналізу вимог наступний крок розробки програмного забезпечення є саме вибір життєвого циклу. Вибір належної моделі є дуже важливим кроком, оскільки в подальшому всі види діяльності відповідного процесу виробництва програмної системи є підпорядкованими такому вибору. Цей вибір є критично важливим для загального успіху проекту розробки програмного забезпечення, тому що він забезпечує структуру, послідовність та інтерфейси дискретних дій.

Організації, які заздалегідь обирають певну модель життєвого циклу до етапу аналізу проекту скоріше за все будуть виробляти слабке програмне забезпечення. Очікувані проблеми створення програмного забезпечення повинні бути проаналізовані до вибору відповідної методології. Тому для обрання життєвого циклу абсолютно необхідним є попередній збір і аналіз вимог до програмної системи.

Розглянемо далі деякі типові альтернативи життєвого циклу програмного забезпечення. Оскільки вибір життєвого циклу є настільки критичним для успіху проекту, його не вибирають просто завдяки індивідуальним смакам, симпатіям або виходячи з попереднього досвіду. Вірний вибір стає провідником і порадником, натомість слабкий вибір, зроблений наосліп, примушує в майбутньому втискувати власний виробничий процес в тісні рамки далекого від реальності шаблону.

Модель водоспаду

Модель водоспаду – перша опублікована модель процесу розробки програмного забезпечення – була розвинена з існуючих моделей розробки технічних систем. Спадання каскадом від одної стадії розробки до іншої дало назву моделі – “модель водоспаду”. Головні етапи даної моделі відповідають базовим діям загального процесу розробки технічних систем:

1. *Аналіз та специфікація вимог*. Загальні вимоги до сервісів, складових частин та мети програмної системи установлюються після

консультацій з користувачами (експертами). Їх детальне подальше визначення слугує специфікацією до розробки програми.

2. *Проектування програмної системи*. На даній стадії розробляється загальна архітектура програми. Встановлення й опис базових абстракцій програмної системи та відношень між ними також відбувається на цьому етапі.

3. *Реалізація та тестування підсистем програмного засобу*. На цій стадії програма розробляється згідно з проектом як множина окремих підсистем. Тестування передбачає перевірку відповідності поведінки кожного модуля специфікації.

4. *Інтеграція та тестування*. Окремі підсистеми збираються в єдину програму та проходять тестування як цілісна програмна система для перевірки відповідності специфікації вимог. Після тестування програма готова для передачі користувачеві.

5. *Експлуатація та підтримка*. Зазвичай ця стадія є найдовшою в життєвому циклі програмної системи. Система встановлюється та використовується на практиці. Підтримка включає в себе виправлення помилок, які не були виявлені на попередніх стадіях розробки, удосконалення реалізації окремих підсистем і розширення функціональності програми для задоволення нових вимог користувача.

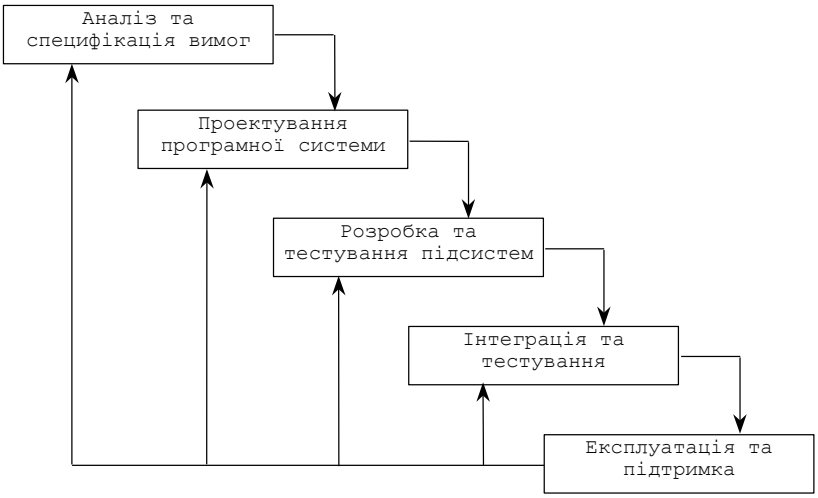

Рис. 4.2. Модель водоспаду з ітераціями

Модель водоспаду має такі *переваги*:

- Модель має чітко визначені фази з добре зрозумілими входами.
- Модель визначає послідовність конкретних технічних дій виробництва програмного забезпечення, які повинні приводити до якісного програмного продукту.

Головними *недоліками* цієї моделі є:

- Реальні проекти рідко буквально слідують за послідовним потоком дій, який пропонує модель.
- Модель припускає, що вимоги до програмного продукту чітко вказані на момент початку проекту. Модель не має ніякого вбудованого механізму, щоб управляти змінами до вимог, які були ідентифіковані з часом виконання проекту (через конкретні дії кодування, зворотний зв'язок з користувачем тощо).
- Модель перешкоджає залученню користувачів до розробки програмного забезпечення між фазами проектування та тестування програмної системи. Це певною мірою сприяє створенню лакун у виробництві та зашкоджує зародженню почуття "власності" у користувачів, оскільки модель не забезпечує активного залучення кінцевого споживача протягом проміжних фаз життєвого циклу програми.
- Для великих проектів користувачам доведеться чекати тривалий час до випуску програмної системи. Робоча версія програми не буде доступною, поки не завершиться остання фаза життєвого циклу. Вимоги до програмного забезпечення можуть змінюватись або навіть стати надмірними до того часу, коли кінцева програма буде створена.
- Припущення про те, що всі вимоги мають бути відомими вже на початку проекту розробки програмного забезпечення, можуть призвести до передчасних рішень, оскільки іноді важко точно оцінити ресурси в умовах обмеженої інформації. Також і для клієнта (або потенціального користувача) важко явно специфікувати всі вимоги та побажання. Однак модель водоспаду вимагає саме такого підходу до створення системи вимог, а це досить важко узгодити з природною невизначеністю, яка існує на початку всякого проекту розробки програмного забезпечення.

Незважаючи на свої добре відомі недоліки, модель водоспаду залишається популярною, тому що вона визначає чітку схему

технічних дій у процесі інженерії програмного забезпечення. На теперішній час розроблено багато інших моделей життєвого циклу, які покликані виключити відомі незручності моделі водоспаду.

Ітерації в моделі водоспаду

Така модель визнає факт, що групі розробників, можливо, доведеться відступати назад і виконувати деякі з дій попередньої фази, в міру того, як покращується розуміння проблеми з ходом виконання проекту.

V-модель

Модель водопаду прагне розглядати тестування як єдину фазу в життєвому циклі програмної системи. V-модель намагається надати більшу вагу тестуванню зв'язаних дій конструювання програмного продукту – шляхом поділу всього життєвого циклу на фазу "розробки" і фазу "тестування". У V-моделі робота фази тестування здійснюється паралельно з іншими фазами. Наприклад планування дій приймального тестування програмного продукту може відбуватися паралельно з фазою постановки вимог до програмної системи, а тестування програмної системи, як цілого, може не чекати на завершення тестування окремих модулів.

Перевагами V-моделі є той факт, що спеціальний наголос ставиться на тестуванні програмної системи. V-модель пов'язує дії з перевірки з відповідними діями специфікації та заохочує підготовку в життєвому циклі ранніх випробувальних планів.

Створення прототипів

Визнано, що визначення вимог до програмних систем у багатьох ситуаціях є дуже непростим завданням. У таких випадках виявлення відсутності або некоректності вимог в останній стадії є надзвичайно дорогим випадком у термінах затримок щодо доставки кінцевого продукту користувачеві. Також визнано, що проект розробки програмного забезпечення часто займає тривалий час, а запити клієнтів із часом змінюються. В таких ситуаціях кращим вибором є модель створення прототипів.

Прототипування починається із збору вимог до програмної системи. Розробники та клієнт зустрічаються та визначають загальні цілі виробництва програмного забезпечення, виділяють, які вимоги відомі, та окреслюють області, які повинні бути обов'язково уточнені згодом. Після того відбувається "швидке проектування".

Швидке проектування зосереджується на представленні тих аспектів програмного забезпечення, які будуть видимі клієнтові

(споживачеві), тобто на визначенні підходу до введення інформації та до вихідного формату. Швидке проектування призводить до розробки прототипу.

Після цього прототип оцінюється користувачем і застосовується для уточнення та покращення множини вимог до програмного забезпечення. Такі ітерації відбуваються, доки прототип не буде настроєний повністю для задоволення потреб клієнтів, в той же час надавши розробнику можливість краще зрозуміти, що саме йому потрібно зробити.

Є два види підходів до прототипування. Першим з них є *еволюційний підхід*. У цьому варіанті прототип побудований таким чином, що він може бути використаний на етапі розробки реального кінцевого програмного продукту. Головною вигодою від еволюційного підходу є те, що значна частина системи розвивається протягом етапу постановки вимог і проектування. Однак в той же час еволюційний підхід уповільнює сам процес визначення вимог, а також зменшує рівень уваги розробників до внутрішніх деталей, таких як стандарти кодування.

В іншому варіанті моделі прототипування, використання системи-прототипу закінчується одразу після того, як будуть чітко сформульовані всі вимоги до програмного засобу. Прототип не використовується для фактичної розробки кінцевої програмної системи. Типовою характеристикою такого підходу є те, що прототип конструюється за допомогою спеціальних програмних середовищ та інструментів – відмінних від інструментів і середовищ, які використовуються для побудови кінцевої програмної системи.

Методологія створення прототипів має переваги в наступних випадках:

- Існують проблеми формулювання користувачем своїх вимог через брак попереднього досвіду роботи в деякій галузі або взагалі в сфері комп’ютерної техніки.
- Очікується, що розроблювана система буде мати складний інтерфейс користувача.
- Існує спеціальне програмне інструментальне середовище, яке підтримує швидке створення прототипу.
- Запропонована система має складний алгоритм або вихідний формат, який має бути покращений.
- Існує добре налагоджена система зв’язку між розробником і користувачем.

Модель прототипування має деякі недоліки:

- Користувачі, які вже побачили робочий прототип, часто очікують дуже швидкої появи фактичної кінцевої системи.
- Якщо прототип не має достатньо привабливого вигляду, користувачі можуть бути розчаровані ним і можуть втратити інтерес до системи, яка розробляється.
- Прототипи розробляються поспішно, часто без оцінки всіх можливих варіантів або розуміння кінцевих результатів застосованих технічних рішень.

Покрокова модель (поступова розробка)

Для дуже великих програмних систем характерним є те, що користувачі зазвичай не хочуть чекати протягом років моменту, коли вони зможуть побачити завершену систему. Покрокова модель розбиває на окремі фази процес постачання користувачеві програмного продукту – після початкового планування інформаційної стратегії. Тут перший “крок” – це основний продукт (ядро) або найголовніша функціональність системи. Це передбачає задоволення основних вимог і подолання головних викликів, але й існування багатьох додаткових (відомих або ні) недороблених підсистем програми.

Основний продукт використовується клієнтом (або просто проходить стадію детального оцінювання). В результаті використання і/або оцінки, план розробки розвивається для отримання наступного “кроку”. План розробки стосується модифікацій основного продукту з метою кращої його відповідності потребам користувача та процесу побудови додаткових особливостей і функціональності. Процес повторюється після кожного наступного ”кроку”, поки не з’явиться повний завершений продукт.

Викладений еволюційний підхід є діалоговим інтерактивним за своєю природою; він передбачає тісний зв’язок з кінцевим користувачем. Але на відміну від методології створення прототипів покрокова модель зосереджується на неперервному постачанні *працездатного* продукту з кожним наступним етапом розробки.

Переваги покрокової моделі:

- Раннє постачання частин реальної кінцевої функціональності для використання клієнтом.
- Зворотний зв’язок від реального використання покрокових версій програмного продукту. Цей зворотний зв’язок може

використовуватися, щоб запобігти знайденим проблемам у наступних поставках.

- Особливості, що мають високий пріоритет, розробляються в першу чергу для початкових кроків постачання програмного продукту.
- У ході виконання проекту можлива гнучка зміна пріоритетів розробки.

Недоліки покрокової моделі:

- Повна вартість розробки є вищою.
- Повний період часу для виготовлення завершеного програмного продукту є більш довгим.
- Кропітке планування послідовності програм-кроків є критичною частиною методології. Неправильне планування може призвести до невдалого завершення проекту.

Методології швидкої розробки програмних систем (RAD)

Дана методологія є покроковою моделлю процесу розробки програмного забезпечення, яка передбачає надзвичайно короткий цикл побудови системи. Модель RAD є адаптацією простої лінійної послідовної покрокової моделі для випадку швидкісної розробки, забезпеченої з технічного боку існуванням базових програмних компонентів. Якщо вимоги до кінцевого продукту є добре зрозумілими, процес RAD надає можливість групі розробників створити замовлену програмну систему повної функціональності за надзвичайно короткий період, наприклад за 2–3 місяці.

Виконання програмного проекту, який використовує методологію RAD, вимагає створення окремих команд для управління кожним ідентифікованим програмним модулем і є допустимим тільки для проектів, де така модуляризація є значущою. Застосування швидкої розробки програм є доречним, якщо існує програмне середовище, що буде використане як інструмент швидкої розробки (наприклад пакет програм підтримки мови 4-го покоління), і якщо багатократне використання програмних компонентів є можливим у ньому.

Доцільний приклад області, в якій зручно застосовувати RAD, – це створення інформаційно-довідникових систем. У випадках, коли проектом передбачена інтенсивна взаємодія між програмними модулями або від кінцевого продукту вимагається висока продуктивність, застосування RAD є навряд чи доречним.

Типові дії застосування методології швидкої розробки складаються з чотирьох фаз:

- *Фаза планування вимог*: кінцеві користувачі та розробники формують об'єднані команди розробки програмної системи і беруть участь в координаційних нарадах, де вони розглядають методологію RAD і готуються для наступної фази. Основною метою даного етапу є розуміння, як кінцевими користувачами, так і розробниками, проблем, які вони намагаються разом вирішити, та границь побудови цільової програмної системи.
- *Фаза проектування*: цей етап складається із збору високорівневих і детальних вимог, за чим слідує моделювання інформаційної системи. На даному етапі повинен бути обраний програмний інструментарій розробки та прототипування. Далі розробляється робочий прототип, який аналізується та змінюється, поки команда розробників не дійде згоди щодо питання, чи прототип задовольняє вимоги користувача.
- *Фаза побудови системи*: розробники застосовують робочий прототип для створення версії кінцевої програмної системи. Це передбачає виконання таких процедур, як доповнення програми необхідними обчисленнями, інтерфейсами тощо; аналіз адекватності створеного продукту; розробка пам'яток користувача та іншої документації, необхідної для існування та підтримки системи.
- *Завершальна фаза*: дана фаза передбачає встановлення кінцевої програмної системи – після попереднього тестування на реальних даних.

Методологія RAD має такі переваги:

- Вона скорочує кількість часу, необхідного для побудови систем (наприклад баз даних) з розвиненим інтерфейсом користувача.
- Споживачі беруть участь в тестуванні та розробці програмної системи, що призводить до послідовного вдосконалення кінцевого продукту.
- RAD полегшує комунікацію, отже розробник може побудувати кращий програмний продукт.

Методологія RAD має такі недоліки:

- Важко досягти погодженості, гармонії в роботі та взаємодії програмних модулів системи, які були створені різними командами.
- Важко підтримувати стандарти проектування та кодування.

Модель трансформацій

У цій моделі неформальні вимоги користувача аналізуються та перетворюються на чіткі специфікації, використовуючи формальні методи. Як тільки вимоги повністю формалізовані, вони транслюються у кінцеву програмну систему.

Цей процес складається з двох головних стадій: *аналіз вимоги* і *оптимізація*. Аналіз вимог забезпечує виконання формальних вимог до кінцевого продукту, а оптимізація проводить подальше налагодження продуктивності – до моменту, поки не буде досягнуто результат, прийнятний за показниками ефективності.

Процес розробки програми із застосуванням методології трансформацій управляється інженером програмного забезпечення і може застосовувати переваги повторно використаних програмних компонентів. Повторно використані компоненти можуть врешті-решт прийняти форму окремих модулів, які входять до програмної системи. Перед тим, як застосовувати формальні перетворення специфікації в програмну систему, проводять аналіз адекватності – для тестування, чи всім вимогам і потребам кінцевого користувача задовольняє розробка.

Модель трансформацій має бути підтримана відповідними середовищами автоматизованої розробки програмних продуктів. Таке програмне середовище забезпечує інструменти для аналізу адекватності, викладення формальних вимог до системи, керування повторно використаними компонентами, проведення оптимізації та збереження історії розробки програмного забезпечення.

Спіральна модель

Одною з найбільш успішних методологій розробки програмних систем є спіральна модель. Вона побудована на розумінні того факту, що більшості організацій потрібно виконувати певні дії з життєвого циклу програмного продукту декілька разів, щоб вони врешті-решт призвели до кінцевого програмного продукту.

Таким чином, центральною частиною застосування спіральної моделі є те, що певні дії (наприклад аналіз ризиків, огляд програмних продуктів, планування) постійно повторюються на кожному наступному витку життєвого циклу системи. Це дозволяє розробникам працювати з покроковим приростом до кінцевого продукту, додаючи до конструювання програми на кожному етапі ще більше розуміння необхідної функціональності та очікувань користувача.

Головною характеристикою спіральної моделі є її циклічність. Кожен цикл спіралі складається з чотирьох стадій, кожна з яких представлена одним квадрантом у декартовій системі координат.

Радіус спіралі являє собою вартість, накопичену на даний момент процесом побудови програмної системи; розмір в кутовому вимірі являє собою ступінь розвитку процесу розробки програми.

- *Стадія 1* ідентифікує задачі досягнення бажаної ефективності продукту – в термінах властивостей, які продукт повинен набути. Іншою проблемою даної стадії є окреслення альтернатив розробки (наприклад які інструменти можуть бути застосовані для проектування, кодування тощо) та визначення обмежень на застосування окремих альтернатив.
- *Стадія 2* оцінює всі потенціальні зони ризику для альтернатив виконання проекту. Така оцінка ризиків може вимагати різноманітних видів діяльності (наприклад створення прототипів чи імітаційного моделювання).
- *Стадія 3* складається з розробки та аналізу адекватності продукту наступного рівня, за яким знову повинен слідувати етап аналізу ризиків.
- Нарешті, *стадія 4* складається з розгляду результатів стадій, які були пройдені до цих пір, і можливого планування наступного витку спіралі.

Спіральна модель має такі переваги:

- Аналіз ризиків є невід'ємним складником цієї моделі та виконується на кожному витку спіралі, що призводить до збільшення рівня впевненості в проекті.
- Модель є гнучкою та може бути прикладена для різноманітних ситуацій, в тому числі для багатократного використання програмних компонент і прототипування.
- Модель комбінує кращі особливості моделі водоспаду та методології створення прототипів.

Спіральна модель має такі недоліки:

- Модель є складною та може стати неадекватною для маленьких проектів з помірними ризиками.
- Модель вимагає значних можливостей проведення оцінки ризиків. Якщо головні ризики не були виявлені, існує висока вірогідність виникнення непростих проблем під час виконання проекту розробки програмного забезпечення.

Після збору та аналізу вимог щодо вибору конкретної моделі життєвого циклу, відбувається комбінація дискретних і безперервних процесів, які повинні призвести до створення кінцевого програмного

продукту. Для того, щоб зрозуміти кожну з цих дискретних і безперервних дій, розглянемо їх далі більш детально одну за одною.

4.1.9. Проектування

Фаза проектування є основою, ядром побудови програмного забезпечення. Протягом цієї фази, вимоги до системи переходять від свого представлення на папері або в думках розробників до існування у вигляді набору формалізованих характеристик певного заздалегідь визначеного об'єкта.

Процес проектування характеризується ітеративним покращенням розробки в міру того, як зображення проблеми переходить з думок розробника до більш формального представлення – із все більшим ступенем точності. Фундаментальною проблемою цього процесу є те, що проектувальники зобов'язані використовувати поточну інформацію для передбачення майбутнього стану системи, який може і не бути досягнений, якщо їх прогнози були невірними. Тобто кінцевий результат проектування повинен розглядатися ще до того, як повністю будуть зрозумілі засоби досягнення самого результату. Цей процес мислення характеризується поверненням від кінцевої бажаної мети назад до початкових умов і обмежень – з одночасним постійним порівнянням відповідності застосованих засобів і передбачених результатів.

Розробники програмного забезпечення одержують множину вимог як події або характеристики подій, а потім повинні запропонувати проект, який надасть змогу таким подіям із специфікованими характеристиками насправді трапитися. Досягнення результату проектування відбувається або шляхом випадкових наближень, або певним формалізованим шляхом, однак в обох випадках отриманий проект повинен бути предметом, що підлягає подальшій оцінці його якості.

Фаза проектування під час розробки нової (або підтримки існуючої) програмної системи може бути розділена на проектування системи та проектування програмних модулів. Діями, які відбуваються протягом цієї фази, можуть бути, наприклад:

- детальне розділення вимог до:
  - технічних засобів;
  - програмних засобів (можливо, до баз даних);
  - людських ресурсів;
  - інших елементів;
- визначення інтерфейсів модулів програмного забезпечення;
- інтерфейси користувача;

- створення планів тестування окремих модулів та інтегрованої програмної системи;
- проектування модулів.

Частина проектування, яка може бути названа проектуванням системи, відбувається там, де загальні ресурси розподіляються відповідно до початково сформульованих вимог. Такими ресурсами можуть бути: люди, програмні засоби, мережі, технічні засоби, або деякі інші елементи.

Розглянемо просту інформаційну систему, вимогою до якої є дозволити людям з багатьох місць постачати інформацію для кінцевої системи. Ми повинні зважати на такі питання розподілення ресурсів:

- чи ми дозволяємо кожному керувати потоком їх власної інформації, чи маємо призначити когось одного для збору та введення всіх даних;
- яким є очікуваний розмір даних, які нам потрібно зберігати;
- резервні копії системи;
- безпека даних і системи;
- обслуговування технічних засобів, програмного забезпечення, бази даних.

Фаза проектування системи повинна стати етапом визначення альтернатив розробки, обговорення вибору, ухвалення системних рішень.

Проектування програмного забезпечення потрібно проводити за схемою нарощення шарів і розбиття на модулі. Модульність – як в даних, так і в функціональності – дозволяє проводити незалежну розробку та тестування індивідуальних блоків вимог. Це дозволяє будувати систему у вигляді окремих перевірених блоків.

Добре впорядковане проектування програмного забезпечення може бути розділене на чотири головні частини:

- проектування структур даних;
- проектування архітектури;
- проектування інтерфейсів;
- проектування процедур (детальний розгляд компонент системи).

Проектування структур даних перетворює вимоги до даних у структури даних. Проектування архітектури визначає головні структури системи та їх відношення одна до одної. Проектування інтерфейсів виділяє, як окремі компоненти зв’язуються одна з одною та із зовнішнім світом. Проектування процедур перетворює структури даних і архітектуру в низькорівневі елементи програмного забезпечення (компоненти). Кожна з чотирьох дій проектування

призводить до кращого розуміння вимог до системи та послідовного вдосконалення кінцевого результату проектування.

Добре впорядковане середовище проектування збирає та/або структурує вимоги в ієрархічні проекти, побудовані на функціональному або об'єктному представленні. Двома загальними категоріями впорядкованих методів проектування є:

- функціональне проектування (розробка структури);
- об'єктно-орієнтоване проектування (розробка об'єктного представлення системи).

Ці методи використовують математичні та графічні нотації для того, щоб надати можливість людському розуму охопити та виразно уявити собі систему вимог, побудовану під час проектування. Використання цих методів вимагає навчання, досвіду та, дуже великою мірою, підтримки програмними інструментами.

Два згаданих способи проектування зосереджуються на різних елементах системи вимог для досягнення мети проектування.

Функціональне проектування зосереджується на алгоритмах (функціях, діях), які властиві даній системі вимог як первинний блок, з якого будується весь проект. Структури даних розглядаються як операційні параметри алгоритмів, а не як джерела інформації або пункти призначення інформації. Поєднання та взаємодія операцій проводиться з точки зору "що потрібно зробити", а не "який об'єкт повинен це зробити".

Глобальні пакети даних є звичайними у функціональному проектуванні, оскільки багато різних модулів діють на однакових елементах даних.

Об'єктно-орієнтоване проектування зосереджується на класах, об'єктах, або гравцях, які залучені та взаємодіють з вимогами як початкові блоки для побудови всієї кінцевої системи. Структура даних будується з окремих об'єктів та інформації, яку повинен мати кожен об'єкт для виконання його роботи. Подальше укрупнення, поєднання відбувається між об'єктами. Один об'єкт надає іншому лише ту інформацію, яка потрібна йому для успішного завершення роботи. Інтерфейси є критичною частиною об'єктно-орієнтованого проектування та є надзвичайно структурованими, оскільки локальні дані (конкретного об'єкта) є одним із базових принципів побудови системи.

Декомпозицію (розділення, розбиття) потрібно згадати як ключовий елемент високоякісного процесу проектування. Декомпозиція – це методика проектування шляхом максимальної ізоляції програмних компонентів.

Ця техніка надає можливості локалізувати помилкові умови в тих програмних компонентах, де відбулася помилка. Тому помилки будуть ідентифіковані та виправлені локально та швидко – з мінімальним руйнуванням нормального процесу функціонування проекту програмного забезпечення.

При строгому дотриманні принципу декомпозиції значно полегшеними є етапи тестування, перевірки адекватності та сертифікації програмних компонентів. Такі сертифіковані компоненти можуть буди використані в подальшому для більш ефективного створення програмної системи.

4.1.10. Кодування

Багато організацій роблять великий акцент саме на етапі кодування, тобто безпосереднього створення програмної системи в кодах визначеної машинної мови. Врешті-решт вони вважають, що саме кодування виявляє проблеми “реальних вимог” і визначає наскільки якісним буде кінцевий програмний продукт. Але фактично кодування повинно бути менш важливим, ніж аналіз вимог і проектування.

Етап фактичної розробки (в значенні кодування, програмування) програмної системи повинен зосереджуватись на точному перекладі результатів проектування в код обраної машинної мови. В реальному виробництві програмних продуктів часто фаза проектування не є повністю завершеною, коли вже починається кодування. У таких випадках дуже важливим для проектувальників і кодувальників є підтримка постійного зв’язку для того, щоб кожна конструктивна зміна в проекті знайшла своє впровадження у ході кодування.

Оскільки вимоги загалом змінюються, то зміни, звичайно, будуть відбуватись і в проекті (який управляє інтерфейсами), а отже і в кодуванні. Якісне планування керування конфігураціями та керування документуванням надає розробникам здатність швидко змінити їх код, щоб відобразити таку зміну вимог і всього проекту.

Головна увага протягом фази кодування повинна бути приділена питанням якості. Не слід намагатися додати якості програмного продукту під час тестування або протягом обслуговування. Натомість якісний код повинен будуватися з самого початку.

Вибір мови програмування

Вибір мови програмування відіграє надзвичайно важливу роль. Великі програмні системи є зазвичай довговічними, а обрана мова має велике значення в здатності підтримувати систему.

Сфера створення наукових програмних систем (включаючи вбудовані системи та системи реального часу виконання) зазвичай не мають дуже чіткого вибору конкретної мови програмування. Досить часто альтернативи зводяться до трьох-п'яти варіантів, в тому числі: C/C++, Ada, Pascal, Java.

Тут варто підкреслити, що мови C та C++ стали стандартом де-факто в багатьох галузях. Мова C сама по собі може бути легко перенесена на інші платформи, однак необхідні для її повноцінного функціонування бібліотеки є часто дуже відмінними для різних систем, що примушує програми на C змінювати свою поведінку при намаганні замінити технічні або програмні засоби її довкілля. C – надзвичайно стисла мова і її не завжди легко підтримувати.

Системи, які плануються для довгого життєвого циклу, повинні обов'язково розглядати проблеми обслуговування. Інтерфейси в C зазвичай мають бути покажчиками або скалярними значеннями. Ми не можемо передати масиви та записи як окремий ізольований тип даних, щоб зміцнити рівень абстракції. Натомість, ми вимушені використати покажчики на масиви та записи.

Оскільки існує поняття адреси процедури, то для неї неможливо заборонити чи обмежити спосіб звернення до визначеного набору даних. Ця нездатність підсилення абстракції проектування програмної системи (тобто застосування принципу декомпозиції, ізоляції компонент – обмеження дій, які можуть виконуватися із визначеною структурою даних) робить C досить зручною мовою для створення безпечного та підтримуваного коду.

C++ – це об'єктно-орієнтоване продовження C. Ця мова допускає всі засоби C, а також включає нову функціональність, що є інструментом побудови класів та ієрархій. Головна проблема C++ – це успадковані від C проблеми з важко зрозумілим виглядом коду та не легкою підтримкою коду. Проте C++ є надзвичайно широко поширеною мовою.

Якщо ваш проект буде розвиватися в C або C++, ви повинні весь час пам'ятати про необхідність вироблення зрозумілого коду. Це вимагає чіткого розділення функцій, модуляризації, дотримання стандартів оформлення програмного коду (зрозумілі імена змінних і функцій, форматування тексту пробільним символами, кропітке коментування тощо).

Мова програмування Ada (або Ada95) є наступником мови Ada83. Вона створювалась для програмних систем реального часу виконання та спеціалізованого програмного забезпечення нових технічних засобів, але швидко стала універсальною мовою. Ada проектувалося

для того, щоб підтримувати зрозумілість коду та бути легко підтримуваною, чого бракувало іншим мовам програмування. Ця мова містить в собі такі ж потужні об'єктно-орієнтовані інструменти, як і C++.

Ada має особливості, які сприяють якісній розробці програмного продукту: сильну типізацію, посилену абстрактність коду, розвинені засоби обробки помилок. Вона вважається чудовою мовою для реалізації критичних систем, які повинні мати довгий життєвий цикл.

Недоліки мови Ada мають два головних джерела. По-перше, більшість програмістів залишаються незнайомими з нею. По-друге, для коректного використання Ada її конструкції вимагають чіткого слідування технічним принципам якісного програмного забезпечення (наприклад ізоляція інформації, абстрагування, сильна типізація).

4.1.11. Тестування

Часто організації використовують тестування як шлях виділення та виправлення помилок. Хоча це дійсно є метою самого процесу тестування, такий набір функцій може бути розширений для підвищення якості проекту розробки програмного забезпечення.

Тестування потрібно також використовувати, щоб через зворотний зв'язок гарантувати якість внесення змін до початку самого процесу розробки програмної системи – таким чином, щоб запобігати появі помилок замість їх подальшого виправлення. Тому первинною метою тестування є перевірка, чи не існує загальних тенденцій до появи помилок або великої кількості однотипних помилок. Ця мета досягається не лише пошуком і виправленням помилок, але й шляхом простеження глибинних причин появи таких помилок.

Якщо корінна причина помилок – це проблема з некоректними або неповними вимогами до програмного продукту, слід використати цей зворотний зв'язок, щоб змінити сам процес збирання та аналізу вимог. Якщо причина багатьох помилок – це помилки кодування, слід змінити або підсилити процес огляду напрацьованого коду, розробити глибший перелік контрольних перевірок існуючого коду, або ввести додатковий нагляд за етапом безпосередньої реалізації.

Так чи інакше, для отримання максимальної окупності інвестицій від зусиль тестування потрібно:

- відстежувати глибинні причини помилок, які виявлені в системі;
- змінювати процес розробки програмного забезпечення з метою виключення або мінімізації причини помилок;
- вимірювати ефективність такого процесу модифікації.

Важливо пам'ятати, що дані програмної індустрії свідчать: тільки 50 % із всіх помилок створеної програмної системи знаходять протягом фази тестування. Решта помилок буде знайдена під час використання програмного забезпечення, що поставляється користувачеві. Для того, щоб гарантувати нормальні відносини з клієнтами та виробляти якісне програмне забезпечення, акценти етапу тестування повинні бути зроблені на використанні результатів цього етапу для виключення глибинних причин появи помилок. Використання тестування просто для виправлення помилок – це витрати часу та згублені можливості поліпшити повний процес створення програмного забезпечення.

Потрібно вірно розуміти перспективи тестування. Ця фаза не може довести кінцеву коректність програми; вона лише виявляє деякі її дефекти. Програмне забезпечення з багатьма помилками нічого не коштує, однак також нерезультативними очевидно є випробування програмної системи, які із самого початку нездатні виявити жодної помилки.

Одною загальною помилкою для великих проектів є зменшення кількості часу, відведеного на тестування з наближенням граничних строків виходу проекту в світ. На жаль, більшість проектів програмного забезпечення спізнюються. І в цьому випадку, як крайній засіб – після настання кінцевого строку виготовлення програмного засобу – багато менеджерів розглядають скорочення (або взагалі виключення) фази ретельного тестування системи, щоб прискорити виконання проекту.

Але, як згадано вище, фаза тестування ідентифікує, виправляє та виключає причини помилок. Очевидно, якщо ви зменшуєте випробувальний час, число залишених в результуючому коді помилок лише збільшується. Менш очевидним є те, що зменшення часу на тестування виключає можливість змінити сам процес створення програмного забезпечення для виключення причин помилок. Коли помилки, які пропущені на фазі тестування, будуть знайдені пізніше (наприклад, протягом обслуговування), можливість безболісно виключити їх глибинні причини є вже втраченою.

Зменшення часу на тестування тільки зменшує якість, але не зменшує час, необхідний в результаті для виробництва програмного забезпечення. Помилки все рівно залишаться, і вони все ще вимагатимуть виправлення – і якщо ця можливість була втрачена до передачі програмної системи користувачу, то закриття цієї проблеми просто пересувається на майбутнє.

Існує декілька видів тестування. Частіше за все, процес, який називають тестуванням, має називатися *блочним тестуванням* (*тестуванням елементів*).

Блочне тестування

Блочне тестування означає, що розробник перевіряє роботу єдиної одиниці коду (блоку), щоб перевірити, наскільки вона відповідає початковим вимогам. Блок зазвичай тестують із заглушками (допоміжного коду), з тим щоб дозволити проведення випробувань тільки для відібраних частин коду. Задача допоміжної частини коду – реалізувати емуляцію довкілля, в якому даний блок буде працювати в майбутньому, таким чином абстрагувавшись від існування інших програмних підсистем.

Для збільшення ефективності блочного тестування, випробування даної програмної одиниці потрібно проводити за допомогою сторонньої особи, а не того спеціаліста, хто розробляв цю частину системи. Розробники програмних модулів є зазвичай дуже близькими до коду та часто не можуть побачити очевидні помилки. Крім того, вони несвідомо сподіваються, що даний модуль не спіткає невдача під час тестування, що часто призводить до неприродної поведінки при виборі тестових прикладів і до зайвої психологічної напруги.

При всіх очевидних перевагах викладеного підходу, час і межі вартості розробки програмного засобу часто примушують розробників програмного модуля ігнорувати до певної межі тестування свого власного коду. Якщо цей негативний процес дійсно відбувається в певній організації, менеджерам слід переконатися, що розробники не просто механічно перебирають деякі тестові випадки, а чесно намагаються ідентифікувати свої помилки в програмному коді. Єдиним шляхом, який гарантує такий контроль, є аналіз видів випробувань, застосованих до програмних модулів.

Існує дві основні види стратегій блочного тестування: “*біла скриня*” і “*чорна скриня*”. Критичною вимогою до якісного процесу тестування є проведення обох видів випробування для кожного програмного модуля.

Тестування методом “біла скриня”

В цьому виді випробувань, тестувальник має доступ до фактичного програмного коду модуля. Особа, яка проводить тестування, може планувати свої випробування, попередньо досліджуючи код – ще до етапу підготовки тестових прикладів.

Типові випробування методом “біла скриня” можуть включати до свого набору (не обмежуючись лише наведеним списком):

- тестування окремих конструкцій;
- перевірку циклів;
- тестування покриття задачі накладеними умовами.

Хоча дійсно добре мати навчену команду незалежних тестувальників для організації процесу перевірки, розробник даної частини програмного коду також може допомагати протягом випробування методом “білої скрині”, оскільки має більші знання фактичного коду.

Випробування методом “білої скрині” необхідні, щоб бути певним, що код робить дії, які він повинен робити згідно специфікаціям. Крім того, такий спосіб тестування демонструє, що немає ніякого прихованого (випадкового або зловмисного) програмного коду для дій, які не були запитані від розробника.

Тестування методом “чорна скриня”

Цей вид тестування проводиться для того, щоб показати, як працює програмний модуль, коли він має справу з типовими вхідними даними. В певному сенсі, такі тести зосереджуються на функціональних вимогах до програмного забезпечення, яке обробляє типові порції інформації.

Існує багато типів випробувань методом “чорна скриня”. Головним для розуміння пунктом таких випробувань є те, що величезний розмір всіх можливих комбінацій наборів вхідних даних робить всеосяжне тестування методом “чорна скриня” неможливим. Методи обмеження кількості випробувальних випадків включають розділення всіх варіантів вхідних даних на класи еквівалентності (поділ входу на представницькі класи) та аналіз роботи підсистеми в граничних випадках (випробування на краях класів вхідних даних – на противагу до обрання типових значень певного класу даних).

Іншим важливим пунктом, який потрібно пам’ятати під час тестування методом “чорна скриня”, є те, що перевірка зумисне невірних вхідних наборів даних також є обов’язковою.

Явне та ясне розуміння того, які вхідні операції з даними проводить конкретний програмний модуль, може завдяки суб’єктивним факторам специфічного відношення розробника до власного коду, про які згадувалося в попередньому розділі, ускладнити або унеможливити якісне тестування методом “чорна скриня”.

Статичне тестування

Термін “статичне тестування” посилається на статичні характеристики коду – наприклад, аналіз того, чи всі змінні були оголошені, ініційовані та використані. Для більшості мов існують спеціалізовані інструменти, які виконують велику кількість подібних перевірок програмного коду. Використання інструментів статичного тестування повинно бути стандартизованим і має обов’язково бути застосовано в проекті розробки програмних засобів.

Динамічне тестування

Динамічне тестування посилається на випробування, які розглядають стани програмної системи протягом її типового сеансу роботи. По суті, це розгляд шляхів виконання програми та використання пам’яті протягом сеансу. Такий спосіб тестування слід виконувати ще на етапі блочного тестування, однак іноді проблеми інтеграції перешкоджають проведенню динамічного тестування до моменту об’єднання програмних модулів.

Інструменти динамічного тестування є надзвичайно спеціалізованими по відношенню до технічних засобів або (в меншій мірі) до мови програмування. Приклади типових звітів інструменту динамічного тестування включають перелік, які частини програми виконувались в деякий момент часу; як часто кожна частина програми виконувалась; які змінні отримали найбільшу кількість звернень; скільки системних викликів або бібліотечних функцій було застосовано протягом роботи програмної системи. Докладне дослідження таких питань допомагає визначити перелік “вузьких” місць програмної системи та суттєво підвищити її ефективність (в розумінні швидкодії та використання пам’яті).

Перевірка взаємодії та одночасного функціонування компонентів системи

Після того, як кожен програмний модуль був перевірений, необхідно перевірити також інтерфейси між ними. Акцентом протягом цієї фази тестування є розгляд всіх можливих взаємодій між різними модулями, в тому числі аналіз параметрів і глобальних змінних.

Це, можливо, найкритичніша частина фази тестування. Тоді як випробування модулів перевіряє лише індивідуальні властивості блоків, перевірка взаємодії та одночасного функціонування компонент системи зосереджується на здатності програмних одиниць разом скласти працездатну систему.

Оскільки складні системи зазвичай мають багато розробників, всеосяжна комунікація між окремими членами команди є занадто важкою. Крім того, велика кількість вимог і часті змін до вимог часто приводять до модулів, які розробляються в ізоляції.

Інтерфейси вважаються областями високого ризику невдачі, тому що іноді проект розробки програмної системи все ще не є повністю завершеним перед тим, як починається безпосереднє кодування. В той же час, розробка модулів, яка передує їх завершеному проектуванню, очевидно є потенційним джерелом неповних або некоректних інтерфейсів. Такі неповні або некоректні інтерфейси не будуть виявлені протягом випробування окремих програмних модулів. Але коли вони виявляються протягом перевірки взаємодії та одночасного функціонування компонентів системи, згадані модулі вже зазвичай не повинні змінюватись – натомість доведеться знову повернутися на стадію проектування, з тим, щоб виправити проблеми інтерфейсів, а потім внести зміни до програмних компонент, яких торкнулось таке перепланування.

Системне тестування

Виходячи з припущення, що блочне тестування та перевірка взаємодії та одночасного функціонування компонентів системи пройшли успішно, наступним кроком є об’єднання сукупності модулів у повну систему та аналіз її роботи як одного цілого. Метою цього виду випробування не є тестування роботи окремих модулів або інтерфейсів (що вже було перевірено на попередніх етапах), але підтвердження адекватності всієї системи, як такої, що відповідає поставленим до неї вимогам. Випробувальні тести для системної перевірки в ідеальному випадку повинні розроблятися ще протягом збору вимог до програмного засобу, і можуть бути використані для завершального аналізу протягом приймання готової програмної системи.

Якщо був провалений системний тест, це свідчить про наявність у проекті програмного забезпечення дуже серйозних помилок. Отже модулі та інтерфейси є цілком правильними, але повна система не в змозі задовольнити вимоги призначеної для користувача функціональності. Загалом це означає, що фаза аналізу вимог була некоректною, та система була написана для того, щоб задовольняти непрацездатним (або неповним) вимогам. Єдиний шлях виправлення таких помилок – це повернення на етап дослідження призначених для користувача потреб і продукування виправленого або переглянутого набору вимог. Відповідно це вимагає перепроектування системи, з

наступним доповненням, видаленням та зміною існуючого програмного коду.

Природною відповіддю на невдачі на стадії системного тестування є побудова командою розробників "швидкого рішення", тобто скороспілих змін до програмного коду – для того, щоб задовольнити потреби певного тестового прикладу. Хоча часом такі рішення дійсно мають право на існування, зазвичай це – шлях до невдачі. Система, де проблеми стадії формулювання вимог негайно виправляються в термінах програмного коду, часто призводять до так званих ланцюгових реакцій (ефекту снігової лавини).

Спочатку одна проста зміна викликає інші проблеми, які далі призводять до інших простих змін, які, в свою чергу, викликають більше проблем, поки кінець кінцем система має на собі стільки "латок", що більше не може підтримуватися або змінюватися без того, щоб потерпіти остаточний крах.

Отже наявність помилок у тестуванні системи вимагає повторного перегляду вимог, внесення модифікацій до проекту – і тільки тоді внесення модифікацій або доповнення до програмного коду.

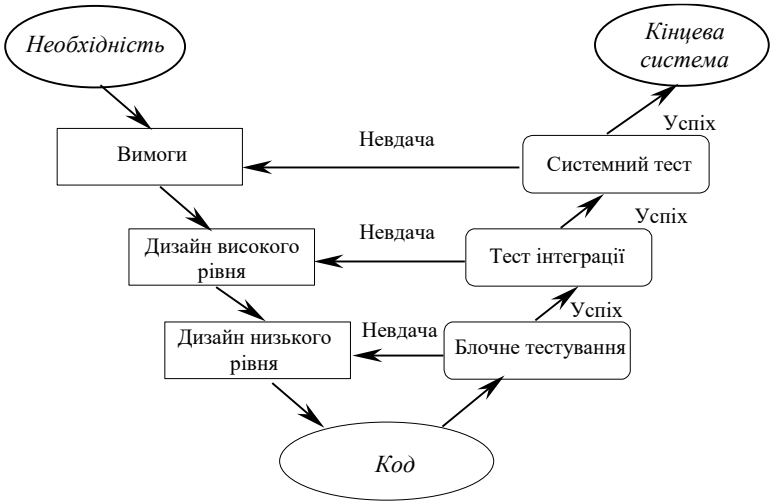

Рис. 4.3. Нейтралізація несправності протягом тестування

Рис. 4.3 показує взаємовідношення між помилками блочного тестування, помилками перевірки коректності проектування та помилками випробування всієї програмної системи. Ця діаграма пояснює ефект нейтралізації несправності. Вона також пояснює, чому

критичною потребою є завершення етапу формування вимог і проектування до початку кодування.

Інші види тестування

Тестування відновлення

Тестування відновлення є перевіркою системного рівня, яка вимушує систему терпіти аварію різними шляхами та перевіряє, чи належним чином відбувається відновлення роботи системи. Цей вид випробувань є абсолютно необхідним для систем реального часу виконання або систем, вбудованих до різноманітних технічних засобів, де потрібне автоматизоване відновлення системи. Задача цього виду тестування – це аналіз того, яким чином система переживає несподівані умови роботи.

Тестування безпеки

Тестування безпеки зосереджується на перевірці механізмів захисту, які вбудовані у програмну систему. Такі перевірки плануються для аналізу того, яким чином система забезпечена захистом від несанкціонованого проникнення та використання системи із зловмисною метою.

Тестування в граничних режимах

Тестування в граничних режимах визначає, в яких режимах роботи програмна систему терпітиме невдачу (тобто в яких умовах слід очікувати аварії). Типові умови використання (і межі робочих режимів) повинні бути вказані у вимогах до застосування системи, оскільки користувачі іноді хочуть знати, як сильно вони можуть навантажити систему перед тим, як вона припинить нормальне функціонування.

Для випадку програмного забезпечення, наприклад, ми можемо перевірити, скількох додаткових користувачів система може винести перед тим, як припинити нормальну роботу. Інші характеристики, які можуть допомогти провести тестування впливу навантаження на систему, включають кількість операцій за день, розмір вхідних даних, ненормальні умови (невдача мережі, енергетична невдача і т. ін.).

Приймальне тестування

Приймальне тестування – це формальне випробування, яке демонструє користувачеві, що програмна система відповідає всім його вказаним потребам. В якісному проекті розробки програмного забезпечення критерії для прийомного тестування будуть використані

як частина системного тесту. Якщо проект пройшов системний тест, але прийомне тестування виявило проблеми, то рівень тестування системи був неадекватним.

Тестування після змін проекту

Це не є фазою в повному розумінні слова, але скоріше технічним прийомом тестування. Випробування адекватності змін, внесених до проекту розробки програмного забезпечення, відбувається, коли будь-яка зміна відбувається в існуючій системі. Це – випробування, яке гарантує, що система все ще працює відповідно до специфікації – навіть після нових змін.

Після внесення змін, розробники повинні проводити такі тести з двома цілями. По-перше, потрібно впевнитися, що змінена частина системи все ще працює належним чином, і, по-друге, перевірити, чи коректно функціонує решта підсистем.

Якщо для даної програмної системи передбаченими є часті зміни та довгий життєвий цикл, то випробування адекватності внесених змін є критично важливою частиною роботи. Щоб мінімізувати час, потрібний для внесення коректних модифікацій, слід сконцентруватися на продукуванні якісної документації (починаючи з фази аналізу вимог до програмного продукту). Крім того, слід зосередитися на простому модульному проекті, який побудований на ізоляції інформації (обмеженні доступу до даних тільки тими модулями, які абсолютно вимагають такого доступу) та абстрагуванні (компоненти системи розбиті на модулі таких чином, який нагадує до певної міри представлення початкової задачі у вигляді фізичних або уявних частин реальної системи).

4.1.12. Обслуговування

Обслуговування часто вважають завершальною діяльністю в життєвому циклі програмного забезпечення. Насправді це – завершальна *дискретна* діяльність, а також діяльність, яка займає більш за все часу.

Цей етап гідний уваги, тому що зазвичай ваші клієнти спостерігають зусилля по обслуговуванню впродовж тривалого часу. На жаль, доволі часто (іноді, між 20 % і 50 %) із змінами програмного забезпечення в рамках обслуговування, до результуючої системи потрапляють додаткові помилки. Ця статистика робить надзвичайно важливим питання управління конфігураціями та контролю змін. Обслуговування – це важка робота, і нерозуміння цього факту само по собі є джерелом нових помилок.

В першу чергу якісне обслуговування вимагає управління конфігурацією та добре розвинену документацію. Зазвичай існує три види обслуговування:

- адаптивне обслуговування;
- обслуговування з метою покращення роботи системи;
- обслуговування з метою виправлення помилок (позапланове технічне обслуговування).

Адаптивне обслуговування

Термін "адаптивне обслуговування" посилається на зміну в програмному забезпеченні, яка має пристосувати його до навколишнього середовища, що змінюється.

Адаптивне обслуговування вимагає, щоб розробники відступили назад, до етапу формулювання вимог, ідентифікували ті вимоги, які були змінені навколишнім середовищем, визначили необхідні зміни проекту та реалізували ці зміни в програмному коді. Після того, як зміни внесені, необхідним є проведення повторних випробувань.

Після адаптивного обслуговування проведення повторного тестування часто вимагає більше часу, ніж у випадку оригінальної програмної системи. Це відбувається тому, що нова система містить багато частин оригінальної системи, але має і нові та змінені частини. Тестування повинне гарантувати, що оригінальні, нові та змінені частини коду все ще працюють, як цього вимагає користувач, а на необхідну функціональність оригінальної системи не впливають свіжі зміни до проекту. Проведення такого комплексу тестів є надзвичайно важким завданням. Для великих систем і серйозних змін, часто потрібні нові набори тестових прикладів та абсолютно нова серія випробувань.

Обслуговування для покращення роботи системи

Обслуговування для покращення роботи системи зводиться до внесення нових або виправлення старих характеристик, які збільшують ефективність програмної системи. У певному сенсі ця задача є дивною, оскільки ви змінюєте програмне забезпечення, яке вже є "правильним", причому змінюєте таким чином, що середній споживач не буде володіти інформацією про ці зміни, не помітить їх.

Ця задача є дуже специфічною, але часто вона є абсолютно необхідною. Протягом розробки програмного засобу, іноді через брак часу, приймається рішення скоротити процедуру розробки певної підсистеми – за рахунок зменшення оптимальності алгоритмів, якісного проектування бази даних тощо.

Тому є багато випадків, коли розробник розуміє, що в цій частині програмної системи він здатен примусити її працювати краще, швидше та ефективніше в розумінні пам'яті – але тільки тоді, коли він дістане таку можливість.

Обслуговування для покращення роботи системи хоч і має нижчий пріоритет, ніж інші зусилля обслуговування, але забезпечує для розробників можливість "зробити це правильно". Звичайно, слід контролювати такий процес, щоб не внести до системи негативних змін, перетворюючи робочу систему в таку, що не працює.

У випадках адаптивного і позапланового технічного обслуговування споживачі знають, що існує певна проблема. В обслуговуванні для покращення роботи системи жодної проблеми не існує, тому потрібно бути обережним і не створити нових проблем. Крім того, слід активно використовувати результати від статичних і динамічних тестів, щоб зміни для покращення не виконувались у тих частинах коду програми, які рідко використовуються.

В обслуговуванні для покращення роботи системи не повинно бути мотивів на зразок "покращити хоч щось". Як було зазначено раніше, обслуговування загалом викликає не меншу кількість помилок, ніж розробка системи. Тому в незалежності від того, якими малими є внесені зміни, розробники неодмінно повинні виконувати кропітке тестування. Також слід розуміти, що структуруючи неструктурний програмний код, ви не обов'язково покращуєте його. Для того, щоб дійсно поліпшити якість, цілком можливо, – доведеться повністю перепроектувати модуль і знову дослідити початковий набір вимог до програмної системи.

Обслуговування для виправлення помилок

Позапланове технічне обслуговування (для виправлення помилок) зводиться до процесу знаходження помилок у системі та реагування на них. Повідомлення про помилки може мати різне походження (від споживачів – аварійні повідомлення, або зсередини організації – наприклад внаслідок планового тестування).

Виправлення можуть, звичайно, змінюватись від простих до комплексних. Найголовніша частина обслуговування з виправлення помилок – це оновлення та підтримка документації таким чином, що майбутні модифікації до системи матимуть точну документацію. Управління конфігураціями також є надзвичайно важливим, тому що ця діяльність точно визначає, коли виправлена програма була випущена в світ, а також гарантує, що всі споживачі мають належну версію програмного забезпечення.

4.1.13. Безперервні дії із забезпечення якості

На противагу до дискретних дій, безперервні дії виконуються протягом всього життєвого циклу розробки програмного забезпечення.

Перша й найважливіша безперервна діяльність – це керування проектом розробки програмного забезпечення. Хоча ця діяльність зазвичай не розглядається як частина самого процесу побудови програми, управління проектом – це, мабуть, найкритичніша з усіх частина розробки.

Проекти програмного забезпечення зазвичай мають численних користувачів. Лінійка клієнтів може варіюватися від фактичних кінцевих користувачів – фізичних осіб, до організацій, які фінансують проект, урядових організацій тощо. Кожна організація, яка має справу з виробництвом програмних систем, зазвичай має чітко визначений статут (в термінах вимог, часу, витрат).

Успішний менеджер проектів розробки програмного забезпечення повинен мати здатність маніпулювати всіма цими (можливо, конфліктними або взаємовиключними) особливостями роботи і розробляти стратегічні плани на використання необхідних ресурсів (устаткування, час, гроші, персонал).

Управління списком вимог також можна вважати безперервною діяльністю. Причиною цього є те, що вимоги безупинно змінюються протягом розробки програмного забезпечення. Отже процес врахування вимог повинен миттєво реагувати на зміни, збирати нові вимоги, поширювати їх до відома всіх розробників, яких вони торкаються, і гарантувати, що вся документація підтримується адекватною та доступною для перегляду.

У певному сенсі всі безперервні дії можуть бути об’єднані під єдиним пунктом *аналіз/забезпечення якості*. Безперервні дії намагаються вплинути на якість виробництва програмного забезпечення та мають для цього механізми зворотного зв’язку. Хоча питання забезпечення якості ще будуть обговорюватися окремо, слід пам’ятати, що всі далі обговорювані дії можна назвати процесами збільшення якості. Це приводить нас до висновку, що проекти розробки програмного забезпечення, які зосереджуються лише на дискретних діях, нехтуючи діями безперервними, наражаються на велику небезпеку невдачі.

4.1.14. Управління конфігураціями

Управління конфігураціями – це надзвичайно складна діяльність, яка несе відповідальність за ряд різноманітних питань: від управління та оновлення документації до оцінки та ухвалення звітів про внесення

змін та існуючі помилки. Критично важливим є те, щоб управління конфігураціями відбувалося від самого початку проекту. Важливим є точне та своєчасне зберігання і управління вимогами до розробки програмного забезпечення. Великі проекти вимагають підтримки значної команди з управління конфігураціями, хоча цей вид специфічної діяльності сам по собі є не надто поширеним.

Найбільшою проблемою управління конфігураціями є прийняття рішень, в які моменти часу проводити синхронізацію проекту – виконувати розміщення всіх змін та доповнень до програмного коду (документації) під керівництвом процесу управління конфігураціями таким чином, щоб всі елементи проекту зберігали цілісність та керованість.

Очевидно такі дії не слід проводити надто часто, оскільки дуже ранні документи і програмна продукція на ранніх стадіях розвитку є невеликими за розміром та піддаються частим і суттєвим змінам. Проте якщо синхронізація проекту відбувається надто пізно, розробники ризикують невдачею у відстеженні змін і розумінні того, на які частини проекту ці зміни вплинули.

Для великих проектів значна кількість матеріалу, який повинен бути зафіксований, вимагає підтримки навченого персоналу, який організовує та категоризує матеріал таким чином, що на нього можна легко посилатися та поновлювати в потрібні моменти.

Іншою важливою функцією управління конфігураціями є створення відповідного контрольного органу, який розглядає зміни, аналізує їх, схвалює та управляє ними. Цей контрольний орган визначає, чи зміни взагалі потрібні, аналізує їх вартість і ефективність, і врешті приймає рішення. Коли надходять нові звіти про помилки або зміни в існуючій системі, повинен виконуватися такий самий процес ухвалення рішення. Якщо було ухвалене рішення не здійснювати зміну або не виправляти помилку, даний контролюючий орган повинен бути здатний привести цьому вичерпне пояснення. З іншого боку, як тільки рішення здійснювати зміну ухвалене, одразу ж повинен початися процес управління конфігураціями – для внесення змін у порядок виконання роботи.

Інше важливе завдання команди управління конфігурації – гарантувати те, що всі редакції програмного забезпечення існують як цілісні, вчасно оновлені, коректні версії. Якщо організація підтримує дуже багато редакцій програмного продукту на рік, і на їх підготовку витрачається багато часу, то необхідна робота з обслуговування стане меншою. З іншого боку, якщо організація випускає версії продукту дуже рідко, користувачі не одержать своєчасного виправлення помилок і внесення необхідних змін до програмної системи.

Отже, управління конфігураціями вимагає існування навченого персоналу, добре ознайомленого з існуючими проблемами. Якщо організація витрачає додатковий час на підготовку хорошої команди управління конфігураціями, то проводити подальший процес обслуговування буде набагато легше. Крім того, така організація набуває доброзичливості з боку своїх користувачів, оскільки команда управління конфігураціями зазвичай є головним підрозділом роботи з користувачем після випуску чергової версії програмної системи.

4.1.15. Перевірка якості та аналіз адекватності

Перевірка якості та аналіз адекватності програмного забезпечення – це велика та складна тема для обговорення. Згадані процеси складаються із всіх дій, які застосовуються для гарантування того, що програмне забезпечення працює належним чином і може бути підтриманим в майбутньому. Такі процеси зазвичай об'єднуються в один вид діяльності, який має назву: *підтримка якості програмного забезпечення*.

Різні дії включають у себе підтримку якості програмного забезпечення; в тому числі, наприклад, розглянуте вище тестування. Однак, перед тим, як ми зможемо формально обговорювати якість програмної системи, нам потрібно визначити, що саме ми розуміємо під терміном "якість"? Таке визначення неявно можна сформулювати шляхом уточнення поняття безперервної діяльності перевірки якості та аналізу адекватності.

1. *Перевірка якості*. Перевірка якості програмного забезпечення передбачає те, що програмне забезпечення виробляється відповідно до деякого визначеного процесу, який має чіткі пункти перевірки та зворотні зв'язки – щоб гарантувати його якість в процесі створення. Перевірка якості відповідає на запитання "Чи розробники *вірно* слідують процедурі створення програмного забезпечення?"

2. *Аналіз адекватності*. Аналіз адекватності відповідає на питання "Чи розробники створюють *вірне* програмного забезпечення?" Цей процес визначає, чи вироблене програмне забезпечення дійсно відповідає потребам споживача. Адже програмне забезпечення, яке загалом якісно побудоване, може залишитися непотрібним, якщо воно не відповідає призначеним для користувача вимогам.

Багато окремих дій сприяють проведенню підтримки якості програмного забезпечення. Кропітке тестування, наявність чіткого процесу розробки програмного забезпечення та воля його дотримуватися – ось приклади властивостей ходу виробництва програмної системи, які забезпечують підтримку якості.

Звичайно, перед тим, як слідувати процесу, ви повинні мати цей процес. Отож першим кроком діяльності щодо перевірки якості має стати встановлення процесу виробництва та зрозумілих критеріїв його дотримання. Якщо команда розробників наперед має думку, що вона не зможе виконати всі кроки встановленої процедури побудови програмного забезпечення, такий процес слід одразу ж змінити, щоб він не перешкоджав виробництву, а навпаки – працював на його користь.

Іншою важливою дією з підтримки якості програмного забезпечення є проведення частих та значущих оглядів на всіх стадіях процесу виробництва. Огляди можуть використовуватися для перевірки якості (із залученням при цьому тільки власного технічного персоналу) та для аналізу адекватності (з додатковим залученням користувачів або їх представників). Важливим фактором успішних і корисних оглядів є початок їх проведення одразу з фази аналізу вимог.

Огляди не повинні бути занадто загальними, абстрактними. Навпаки, оглядам потрібно бути детально структурованими відносно певного процесу. Щоб бути ефективними, огляди повинні застосовувати заздалегідь сформовані переліки важливих питань, які є відповідними до даного конкретного виду діяльності, що розглядається (проектування, кодування і т. ін.).

Перелік важливих питань повинен бути формалізований організацією та оновлюватися в міру необхідності окремими особами. Це дозволяє оглядам бути чіткими, корисними та не заформалізованими одночасно. Такий вид діяльності слугує зворотним зв'язком, який поліпшує повний процес розробки програмного забезпечення.

4.1.16. Аналіз і управління ризиками

Багато проектів розробки програмного забезпечення є невдалими через те, що їх планували неадекватно без розуміння наслідків альтернативних підходів до розробки. Призначення дій щодо аналізу та управління ризиками – це вкладення часу та грошей для того, щоб стати обізнаним про речі, які можуть впливати на хід розробки програмного забезпечення. Така діяльність має великий вплив на успішність проекту, оскільки хороше планування все ще залишається неадекватним, якщо проводиться без знань про існуючі ризики.

Необхідність проведення дій з аналізу та управління ризиками розуміють, на жаль, не завжди. Деякі організації просто бажають "тримати голову в піску". Інші організації не мають достатнього бюджету, щоб виконувати належне планування ризиків. І, нарешті, в

багатьох організаціях персонал неохоче висуває та ідентифікує ризики через суб'єктивні причини: побоювання, пов'язані з наслідками реалізації таких ризиків; небажання брати на себе відповідальність щодо контролю ситуації в умовах поінформованості про ризики тощо. Така тенденція не може вважатися нормальною, оскільки в такому виді людської діяльності, як бізнес, можливості прибувають саме від ризиків. Якщо ваша організація не бере на себе ніяких ризиків, то розробляючи продукцію, ви не створюєте інноваційні (чи навіть пануючі на ринку) продукти – ви лише використовуєте існуючі підходи, технології та ідеї, які стануть застарілими в момент виходу програмного забезпечення в світ.

Для успішного завершення проекту розробки програмних засобів як мінімум потрібно знати ризики високого рівня. Серед них, зазвичай:

- дефіцит кваліфікованого персоналу;
- складення нереального розкладу;
- нерозуміння вимог до програмного продукту;
- побудова поганого інтерфейса користувача;
- намагання витратити час на побудову надмірно складного інтерфейса користувача, який початково не потрібен клієнтам;
- безконтрольна зміна вимог до програмного продукту;
- дефіцит готових повторно використаних програмних компонент;
- задовгі діалоги всередині та зовні організації (велика тривалість відповіді);
- спроба перевищити можливості поточного рівня розвитку комп'ютерних технологій.

Кваліфіковане управління ризиками – це надзвичайно складна проблема. Якщо ваша організація не має людей, навчених в ідентифікації ризиків, документуванні ризиків, плануванні процедур зменшення ризиків, у загальному випадку потрібно або інвестувати кошти в навчання власного персоналу, або залучити сторонніх консультантів, які допоможуть довести проект розробки програмного забезпечення до успішного завершення.

4.1.17. Планування обслуговування програмних засобів

Планування обслуговування вимагає особливої уваги в таких діяльностях:

- проектування;
- кодування;
- документування.

Програмні системи, які мають бути легко підтримуваними, повинні добре проектуватися. Як і у випадку з якістю програмного продукту, здатність бути підтримуваною не може бути доданою до системи; натомість така здатність – це пряме віддзеркалення властивостей проекту розробки програмного засобу. Якісний проект повинен бути модульним; кожен програмний модуль повинен виконувати чітке завдання та мати всі необхідні для цього ресурси; від кожного модуля в системі повинно залежати настільки мало, як це тільки можливо з огляду підтримки успішного функціонування інших модулів. Такі директиви розробки потрібно також комбінувати з розвиненим використанням абстракцій і приховуванням інформації.

Структури даних повинні бути доступними тільки для тих модулів, яким для функціонування потрібен доступ до них, а з них – тільки модулі, які мають потреби модифікації, повинні бути здатними змінювати структури даних. Звичайно, для виконання таких умов і забезпечення належної якості роботи самих структур даних слід звернутися до використання мови програмування високого рівня, яка підтримує такі можливості.

Кодування – це інша область, в якій може бути досягнуто здатність програмної системи бути легко підтримуваною. Тут критичним для успіху є правило, яке говорить, що код створюється тільки один раз, але читається та змінюється після цього багаторазово. Кожна організація, яка працює над створенням програмних засобів, повинна підтримувати обов'язкові для застосування стандарти створення та оформлення програмного коду. Частий контроль створеного коду в ході виконання розробки програми допомагає досягати керованості системи шляхом створення коду зрозумілого, прозорого з точки зору намірів програміста. Крім того, необхідним є підтримка розвиненої поточної документації створеного програмного коду.

Можливо якісне документування – це найбільш критична частина успішного обслуговування програмних засобів. Після фактичного завершення розробки програмного засобу, проект зсуває акценти від створення до обслуговування програми, і персонал, який був зайнятий безпосередньою розробкою, зазвичай переходить до виконання інших обов'язків. Тому в момент, коли потрібне обслуговування продукту, в організації може й не бути людини, яка має досвід побудови даної програми. В таких умовах якісне документування є абсолютно необхідним для підтримки програмної системи.

Це означає, що персоналу, який займається обслуговуванням, доведеться шукати помилку та здійснювати її виправлення, не користуючись власними спогадами щодо деталей розробки та логіки

програмування, а застосовуючи лише існуючий програмний код і попередньо створену документацію. Тому дуже важливо, щоб проектна документація була об'єктом живим, динамічним, який точно відображає поточний стан справ всіх частин проекту розробки програмної системи (вимоги проектування, кодування, логічне обгрунтування вибору конкретної альтернативи в ході розробки, тестові приклади, результати тестування тощо).

Поняття ентропії передбачає те, що системи схиляються до хаосу, якщо до них не прикладають зусилля із забезпечення організації. Така концепція може бути також прикладеною і до програмного забезпечення. Якщо в організації не існує якісного та зрозумілого процесу підтримки створення програмної документації, то система існуючих документів неухильно сповзає до хаосу, який стає необоротним через певний час.

Для того, щоб документування мало корисні функції, повинен існувати набір стандартів оформлення документації проекту розробки програмного забезпечення. Потрібно також гарантувати, що цих стандартів будуть дотримуватися в ході обслуговування системи. Альтернативою є повне нерозуміння того, які частини програми були змінені, коли і навіщо, що врешті решт неминуче призведе до неможливості її обслуговування – оскільки в такому процесі обов'язково наступить момент, коли вартість нової розробки даної програмної системи стане дорівнювати вартості подальшої підтримки хаотичної системи, підданої масовим побічним ефектам.

У загальному випадку безперспективними є очікування того, що проектувальники та розробники стануть майстерними фахівцями побудови документації. Персонал з досвідом створення технічних текстів і їх підтримки є критично важливим для безперервного існування програмного забезпечення.

4.1.18. Управління проектом

Найголовнішою безперервною діяльністю виробництва програмного забезпечення є управління проектом. Це значно більше, ніж просто управління процесом створення технічної системи, така діяльність вимагає також якісного управління людьми та часом.

Так, наприклад, існує тенденція просувати програмістів (кодувальників) високої кваліфікації на посаду проектувальника. Кінець кінцем саме проектувальники несуть зазвичай безпосередню відповідальність за створення якісного програмного продукту. Тому для такого персоналу потрібним є додаткове навчання та набуття досвіду. Управління проектом вимагає від менеджера здатності

виділити такий персонал, забезпечити належний рівень поточного навчання, фактично керувати та організувати людей, вести їх до точно встановлених проектних розкладів.

У цьому процесі повинен існувати деякий вид механізму зворотного зв'язку, який дозволяє менеджеру проекту вимірювати пройдений шлях і давати точну оцінку досягненням розробки.

Для цього потрібно ввести та підтримувати *метрики*, які надають кількісну міру оцінки масштабів роботи та дозволяють виявляти проблеми – як в процесі виробництва програмного засобу, так і в самому продукті.

Метрики процесу виробництва надають можливість кількісної оцінки якості процесу розробки. Вимірювання таких речей, як рівень помилок (за модулями та в цілому) і час, витрачений на переробку існуючих компонент, є критично важливим для успіху проекту та його завершення згідно з розкладом. Вимірювання індивідуальних норм помилок є загалом безперспективним, оскільки персональні помилки можуть бути пов'язані з тим, що конкретна людина працює над більш складним завданням. Більш важливим є виявлення причин наявності помилок і подальше вдосконалення процесу виробництва програмного продукту – для їх виправлення.

Метрики конкретного проекту побудови програмного засобу використовують частіше для цілей планування. Якщо менеджер може точно уявити "розмір" поточного проекту, дані, отримані за час виконання попередніх проектів, можуть надати добру підказку щодо очікуваного часу завершення поточної розробки. Іншими важливими видами таких метрик є величини для визначення якості та повноти проектування та кодування програмного тексту.

Як і у випадку з іншими безперервними діями, діяльність з управління проектом є надзвичайно складною. Відповідна посада передбачає наявність навичок в галузях міжособистісних відносин, використання метрик процесу, планування розкладів тощо. Крім експертних знань, досвід і постійні тренування є іншими обов'язковими вимогами до менеджера проекту.