МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КІРОВОГРАДСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

МЕХАНІКО-ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Технологія проектування програмних систем

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт
(магістри)

з елементами кредитно – модульної
системи організації навчального процесу

для магістрів денної форми навчання за напрямом підготовки
8.050102.01 «Комп'ютерні системи та мережі»
8.050102.02 «Системне програмування»

укладачі:

Доцент Смірнов В.В.
Доцент Смірнова Н.В.

Кіровоград 2014

Технологія проектування програмних систем (магістри): Методичні вказівки до виконання лабораторних робіт для магістрів денної форми навчання за напрямом підготовки 8.050102.01 «Комп'ютерні системи та мережі» 8.050102.02 «Системне програмування» / Укл.: / Смірнов В.В., Смірнова Н.В. – Кіровоград: КНТУ, 2014. – 150 с.

Затверджено на засіданні кафедри ПЗ:
11 вересня 2013 р. протокол № 2;
17 вересня 2014 р., протокол № 4.

Укладач:
Смірнов Володимир Вікторович, к.т.н., доцент кафедри ПЗ,
Смірнова Наталія Володимирівна, к.т.н., доцент кафедри ПЗ,

Для магістрів денної форми навчання, які вивчають навчальну дисципліну "Технологія проектування програмних систем (магістри) навчання за напрямом підготовки 8.050102.01 «Комп'ютерні системи та мережі» 8.050102.02 «Системне програмування»

Лабораторні роботи розроблені на сучасному науково - методичному рівні. Відповідають вимогам до курсу практичного навчання студентів, у доступній формі викладені теоретичні відомості і описані практичні кроки оволодіння основами технології проектування програмних систем

ЗМІСТ

1. Опис навчальної дисципліни
“Технологія проектування програмних систем"

Дисципліна вивчає методи проектування локальних і розподілених програмних систем. Робоча програма складена для студентів денної форми навчання по напрямку підготовки 8.05010202 «Системне програмування».

Дисципліна відноситься до вибіркового блоку програми.

Ціль вивчення дисципліни

Основна мета курсу полягає в придбанні знань в області розробки і проектування сучасного програмного забезпечення та програмних систем, з використанням сучасних технологій.

Завдання вивчення дисципліни

- вивчення процесу розробки програмного забезпечення програмних систем;
- аналіз встановлення і специфікація вимог до програмної системи;
- поглиблений аналіз і обґрунтування основ проектування систем;
- проектування користувацького інтерфейсу;
- проектування баз даних, програм і транзакцій;
- тестування і керування змінами.

Предметом дисципліни є методи та алгоритми проектування і створення програмних систем.

Студент повинен знати і уміти після оволодіння дисципліни:

- володіти процесом розробки програмного забезпечення;
- проводити аналіз специфікації вимог до програмної системи;
- проводити поглиблений аналіз і обґрунтування методів проектування систем;
- уміти проектувати користувацький інтерфейс;
- володіти проектуванням баз даних, програм і транзакцій.

2. Тематика і зміст лекцій

№ теми	Тематика і зміст лекцій	Години
1	2	3
	Процес створення програмного забезпечення ***1. Моделі процесу створення ПЗ.*** *Каскадна модель. Еволюційна модель розробки. Формальна розробка систем. Розробка ПЗ на основі раніше створених компонентів* ***2. Ітераційні моделі розробки ПЗ.*** *Модель покрокової розробки. Спіральна модель розробки* ***3. Специфікація програмного забезпечення.*** ***4. Проектування і реалізація ПЗ.*** *Методи проектування. Програмування і налагодження* ***5. Атестація програмних систем.*** ***6. Еволюція програмних систем.*** ***7. Автоматизовані засоби розробки ПЗ.*** *Класифікація CASE-засобів*	2
2	**Керування проектами** ***1. Процеси керування*** ***2. Планування проекту.*** *План проекту. Контрольні оцінки етапів робіт* ***3. Графік робіт.*** *Тимчасові і мережні діаграми* ***4. Керування ризиками.*** *Визначення ризиків. Аналіз ризиків. Планування ризиків. Моніторинг ризиків.*	2
3	**Вимоги до програмного забезпечення** ***1. Функціональні і нефункціональні вимоги.*** *Функціональні вимоги. Нефункціональні вимоги. Вимоги предметної області.* ***2. Користувацькі вимоги.*** ***3. Системні вимоги.*** *Структурована мова специфікацій. Створення специфікацій за допомогою PDL. Специфікація інтерфейсів.* ***4. Документування системних вимог.***	2
4	**Розробка вимог** ***1. Аналіз здійсненності*** ***2. Формування і аналіз вимог.*** *Опорні точки зору*	2
5	**Прототипування програмних систем** ***1. Прототипування в процесі розробки ПЗ.*** *Еволюційне прототипування. Експериментальне прототипування.* ***2. Технології швидкого прототипування.*** *Застосування динамічних мов високого рівня. Програмування баз даних.* *Складання додатків з повторним використанням компонентів.* ***3. Прототипування користувацьких інтерфейсів.***	2
6	**Формальні специфікації ПЗ** ***1. Формальні специфікації в процесі розробки ПЗ*** ***2. Специфицирование інтерфейсів*** ***3. Специфікація поведінки систем***	2

№ теми	Тематика і зміст лекцій	Годинн
1	2	3
7	**<u>Архітектурне проектування</u>** ***1. Структурування системи.*** *Модель репозиторія. Модель клієнт/сервер.* *Модель абстрактної машини.* ***2. Моделі керування.*** *Централізоване керування. Системи, керовані подіями. Модульна декомпозиція.* ***3. Модульна декомпозиція.*** *Об'єктні моделі. Моделі потоків даних.* ***4. Проблемно-залежні архітектури.*** *Моделі класів систем. Базові архітектури.*	2
8	**<u>Проектування систем реального часу</u>** ***1. Проектування систем.*** *Моделювання систем реального часу. Програмування систем реального часу.* ***2. Керуючі програми.*** *Керування процесами.* ***3. Системи спостереження і керування.*** ***4. Системи збору даних.***	2
9	**<u>Проектування з повторним використанням компонентів</u>** ***1. Покомпонентна розробка.*** *Об'єктні структури додатків. Повторне використання комерційних програмних продуктів. Розробка повторно використовуваних компонентів* ***2. Сімейства додатків*** ***3. Проектні паттерни***	2
10	**<u>Проектування інтерфейсу користувача</u>** ***1. Принципи проектування інтерфейсів користувача*** ***2. Взаємодія з користувачем*** ***3. Представлення інформації.*** *Використання в інтерфейсах кольору* ***4. Засоби підтримки користувача.*** *Повідомлення про помилки. Проектування довідкової системи. Документація користувача.* ***5. Оцінювання інтерфейсу***	2
	Усього:	**20**

3. Лабораторні роботи по дисципліні

"Технологія проектування програмних систем"

№ заняття	Назва лабораторних робіт	Кіл. годин
1.	**№1** Процес створення програмного забезпечення	4
2.	**№2** Вимоги до програмного забезпечення	4
3.	**№3** Прототипування програмних систем	4
4.	**№4** Формальні специфікації ПЗ	4
5.	**№5** Проектування систем реального часу	4
	Усього годин	20

4. Шкала оцінювання

Шкала оцінювання: національна і ECTS

<table>
<tr><th rowspan="2">Сума балів за всі види навчальної діяльності</th><th rowspan="2">Оцінка ECTS</th><th colspan="2">Оцінка за національною шкалою</th></tr>
<tr><th>для іспиту, курсового проекту (роботи), практики</th><th>для заліку</th></tr>
<tr><td>90 – 100</td><td>A</td><td>відмінно</td><td rowspan="5">зараховане</td></tr>
<tr><td>82-89</td><td>B</td><td rowspan="2">добре</td></tr>
<tr><td>75-81</td><td>C</td></tr>
<tr><td>65-74</td><td>D</td><td rowspan="2">задовільно</td></tr>
<tr><td>60-64</td><td>E</td></tr>
<tr><td>35-59</td><td>FX</td><td>незадовільно з можливістю повторної здачі</td><td>не зараховане з можливістю повторної здачі</td></tr>
<tr><td>0-34</td><td>F</td><td>незадовільно з обов'язковим повторним вивченням дисципліни</td><td>не зараховане з обов'язковим повторним вивченням дисципліни</td></tr>
</table>

5. Оцінка успішності в балах при повному виконанні умов і графіку навчального процесу

Матеріал лекцій	Кількість лекцій	Бали за вивчення лекційного матеріалу	Лабораторні роботи				Науково дослідницька робота	Максимальна сума балів
			Теми лабораторних робіт	Годинник	Тестовий контроль	Виконання л.р 5б за 1 л.р.		
1. Процес створення програмного забезпечення 2. Керування проектами	2	4	**№1** Процес створення програмного забезпечення	4		5		9
3. Вимоги до програмного забезпечення 4. Розробка вимог	2	4	**№2** Вимоги до програмного забезпечення	4		5		9
5. Прототипування програмних систем	1	4	**№3** Прототипування програмних систем	4		5		9
6. Формальні специфікації ПЗ 7. Архітектурне проектування	2	4	**№4** Формальні специфікації ПЗ	4		5		9
8. Проектування систем реального часу 9. Проектування з повторним використанням компонентів 10. Проектування інтерфейсу користувача	3	4	**№5** Проектування систем реального часу	4	30	5	25	64
Усього:		20		20	30	25	25	100

Лабораторна робота № 1

Тема: Процес створення програмного забезпечення

Ціль - представити основні ідеї які лежать в основі процесу створення програмної забезпечення.

- знати основні концепції, що лежать в основі процесу створення ПЗ і моделей цього процесу;
- мати представлення про основні моделі процесу створення ПЗ і розуміти, коли яку з них використовувати;
- знати схему побудови моделей процесу формування вимог до ПЗ, його розробки тестування і модернізації;
- мати поняття про CASE-технології, призначені для підтримки процесу створення ПЗ.

Завдання:

1. Розробіть модель інтерактивної системи перегляду залізничних розкладів для пасажирів і модель процесу тестування програми, що виконується.

Питання:

1. Поясніть, чому програми, створювані відповідно до еволюційної моделі розробки, важкі для супроводу.
2. Поясніть, як каскадну модель і еволюційну модель із прототипуванням можна об'єднати зі спіральною моделлю розробки ПЗ.
3. Поясніть, чому в процесі визначення вимог необхідно розрізняти розробку користувацьких вимог і розробку системних вимог.
4. Назвіть п'ять основних компонентів будь-яких методів проектування. Які методи проектування ви знаєте? Опишіть їхні компоненти. Оцініте повноту цих методів.

5. Опишіть CASE-засоби, які можна використовувати у вашому робочому середовищі розробника, і класифікуйте їх по декільком параметрам (виконувана функція, підтримуваний процес, кількість підтримуваних процесів).

Короткі теоретичні відомості:

Процес створення програмного забезпечення – це безліч взаємозалежних процесів і результатів їх виконання, які ведуть до створення програмного продукту. Процес створення ПЗ може починатися з розробки програмної системи "з нуля", але частіше нове ПЗ розробляється на основі існуючих програмних систем шляхом їхньої модифікації.

Процес створення ПЗ, як і будь-яка інша інтелектуальна діяльність, заснований на людських судженнях і умовиводах, тобто є творчим. Внаслідок цього всі спроби автоматизувати процес створення ПЗ мають лише обмежений успіх. CASE-засоби можуть допомогти в реалізації деяких етапів процесу розробки ПЗ, але принаймні в найближчі кілька років не варто очікувати від них істотного просування в автоматизації тих етапів створення ПЗ, де суттєвий фактор творчого підходу до розробки ПЗ.

Одна із причин обмеженого застосування автоматизованих засобів до процесу створення ПЗ – величезне різноманіття видів діяльності, пов'язаних з розробкою програмних продуктів. Крім того, організації - розробники використовують різні підходи до розробки ПЗ. Також різняться характеристики і можливості створюваних систем, що вимагає особливої уваги до певних сторін процесу розробки. Тому навіть в одній організації при створенні різних програмних систем можуть використовуватися різні підходи і технології.

Незважаючи на те що спостерігається величезне різноманіття підходів, методів і технологій створення ПЗ, існують фундаментальні базові процеси, без реалізації яких не може обійтися жодна технологія розробки програмних продуктів. Перелічимо ці процеси.

1. *Розробка специфікації ПЗ.* Це фундаменти будь-якої програмної системи. Специфікація визначає всі функції і дії, які буде виконувати

розроблювальна система.

2. *Проектування і реалізація (виробництво) ПЗ.* Це процес безпосереднього створення ПЗ на основі специфікації.
3. *Атестація ПЗ.* Розроблене програмне забезпечення повинне бути атестоване на відповідність вимогам замовника.
4. *Еволюція ПЗ.* Будь-які програмні системи повинні модифікуватися відповідно до змін вимог замовника.

Хоча не існує “ідеального” процесу створення ПЗ, у багатьох організаціях-розробниках намагаються його вдосконалити, оскільки він може опиратися на застарілі технології і не включати кращих методів сучасної інженерії програмного забезпечення. Крім того, багато організацій постійно використовують ті самі технології (колись раніше добре себе зарекомендували) і їм також необхідні методи сучасної інженерії ПЗ.

Удосконалювати процес створення програмних систем можна різними шляхами. Наприклад, шляхом стандартизації, яка зменшить різнорідність використовуваних у даній організації технологій. Це, у свою чергу, приведе до вдосконалювання внутрішніх комунікацій у організації, зменшенню часу навчання персоналу і зробить економічно вигідним процес автоматизації розробок. Стандартизація зазвичай є першим кроком до впровадження нових методів і технологій інженерії ПЗ.

1. Моделі процесу створення ПЗ

Модель процесу створення програмного забезпечення – це загальне абстрактне представлення даного процесу. Кожна така модель представляє процес створення ПЗ в якомусь своєму “розрізу”, використовуючи тільки певну частину всієї інформації про процес. Тут представлені узагальнені моделі, засновані на архітектурному підході. У цьому випадку можна побачити всю структуру процесу створення ПЗ, абстрагуючись від приватних деталей окремих складових його етапів.

Ці узагальнені моделі не містять точного опису всіх стадій процесу створення ПЗ. Напроти, вони є корисними абстракціями, що допомагають “прикласти” різні

підходи і технології до процесу розробки. Крім того, мабуть, що процес створення великих систем не є єдиним, а складається з безлічі різних процесів, що ведуть до створення окремих частин великої системи.

Розглянемо наступні моделі створення програмного забезпечення.

1. *Каскадна модель.* Основні базові види діяльності, виконувані в процесі створення ПЗ (такі, як розробка специфікації, проектування і виробництво, атестація і модернізація ПЗ), представляються як окремі етапи цього процесу.
2. *Еволюційна модель розробки ПЗ.* Тут послідовно перемежовуються етапи формування вимог, розробки ПЗ і його атестації. Первісна програмна система швидко розробляється на основі деяких абстрактних загальних вимог. Потім вони уточнюються і деталізуються відповідно до вимог замовника. Далі система допрацьовується і атестується відповідно до нових уточнених вимог.
3. *Модель формальної розробки систем.* Заснована на розробці формальної математичної специфікації програмної системи і перетворенні цієї специфікації за допомогою спеціальних математичних методів у програми, що виконуються. Перевірка відповідностей специфікації і системних компонентів також виконується математичними методами.
4. *Модель розробки ПЗ на основі раніше створених компонентів.* Передбачає, що окремі складові частини програмної системи вже існують, тобто створені раніше. У цьому випадку технологічний процес створення ПЗ основну увагу приділяє інтеграції окремих компонентів у загальне ціле, а не їхньому створенню.

Каскадна і еволюційна моделі розробки широко використовуються на практиці. Модель формальної розробки систем успішно застосовувалася в багатьох проектах, але кількість організацій-розробників, що постійно використовують цей метод, невелика. Використання готових системних компонентів практикується повсюдно, але більшість організацій не дотримуються в точності моделі розробки ПЗ на основі раніше створених компонентів. Разом з тим цей метод повинен

одержати широке поширення в XXI сторіччі, оскільки складання систем з готових або раніше використаних компонентів значно прискорює розробку ПЗ.

Каскадна модель

Це перша модель процесу створення ПЗ, породжена моделями інших інженерних процесів. Вона показана на рис. 1.1. Цю модель також іноді називають моделлю життєвого циклу програмного забезпечення *(Життєвий цикл програмного забезпечення - це сукупність процесів, що протікають у період від моменту ухвалення рішення про створення ПЗ до його повного виводу з експлуатації. Таким чином, "життєвий цикл ПЗ є більш, широким поняттям, чим модель процесу створення ПЗ. Разом з тим каскадну модель можна розглядати як одну з моделей життєвого циклу ПЗ).*

Основні принципові етапи (стадії) цієї моделі відображають усі базові види діяльності, необхідні для створення ПЗ.

1. *Аналіз і формування вимог.* Шляхом консультацій із замовником ПЗ визначаються функціональні можливості, обмеження і мети створюваної програмної системи.
2. *Проектування системи і програмного забезпечення.* Процес проектування системи розбиває системні вимоги на вимоги, що пред'являються до апаратних засобів, і вимоги до програмного забезпечення системи. Розробляється загальна архітектура системи. Проектування ПЗ передбачає визначення і опис основних програмних компонентів і їх взаємозв'язків.
3. *Кодування і тестування програмних модулів.* На цій стадії архітектура ПЗ реалізується у вигляді безлічі програм або програмних модулів. Тестування кожного модуля включає перевірку його відповідності вимогам до даного модуля.
4. *Складання і тестування системи.* Окремі програми і програмні модулі інтегруються і тестуються у вигляді цілісної системи. Перевіряється, чи відповідає система своєї специфікації.
5. *Експлуатація і супровід системи.* Зазвичай (хоча і не завжди) це сама

тривала фаза життєвого циклу ПЗ. Система інсталюється, і починається період її експлуатації. Супровід системи включає виправлення помилок, які не були виявлені на більш ранніх етапах життєвого циклу, вдосконалювання системних компонентів і “підгонку” функціональних можливостей системи до нових вимог.

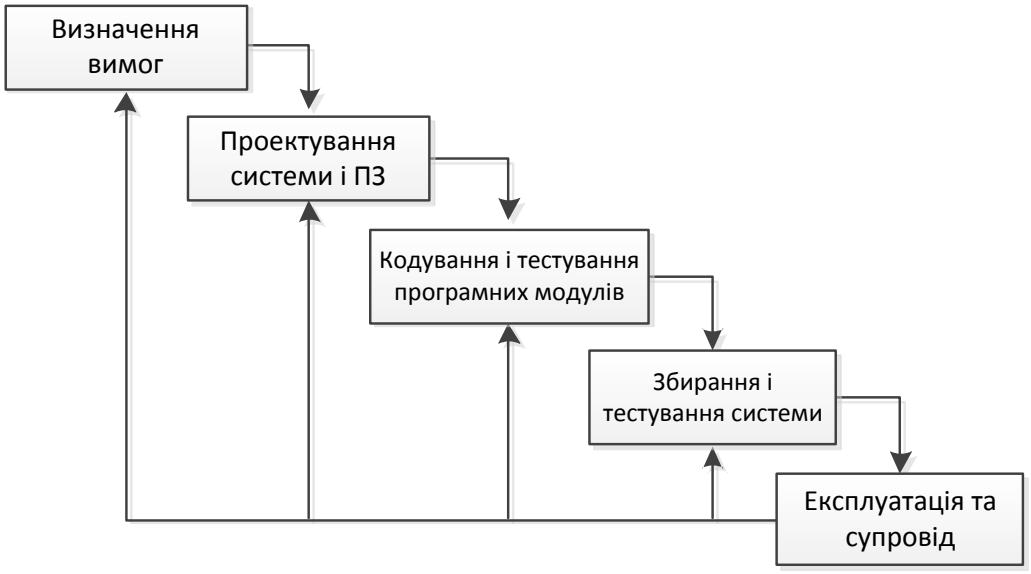

Рис. 1.1. Життєвий цикл програмного забезпечення

У принципі результат кожного етапу повинен затверджуватися документально (це як би сигнал про закінчення етапу). Тоді наступний етап не може початися до завершення попереднього. Однак на практиці етапи можуть перекриватися з постійним перетіканням інформації від одного етапу до іншого. Наприклад, на етапі проектування може виникнути необхідність уточнити системні вимоги або на етапі кодування можуть виявитися проблеми, які можна розв'язати лише на етапі проектування, і т.д.

Процес створення ПЗ не можна описати простою лінійною моделлю, тому що він неминуче містить послідовність повторюваних процесів.

Оскільки на кожному етапі проводяться певні роботи і оформляється супутня документація, повторення етапів приводить до повторних робіт і значним витратам. Тому після невеликого числа повторень зазвичай “заморожується” частина етапів створення ПЗ, наприклад етап визначення вимог, але триває виконання наступних етапів. Виникаючі проблеми, рішення яких вимагає повернення до “заморожених”

етапів, ігноруються або робляться спроби розв'язати їх програмно. “Заморожування” етапу визначення вимог може привести до того, що розроблена система не буде задовольняти всім вимогам замовника. Це також може привести до появи погано структурованої системи, якщо упущення етапу проектування виправляються тільки за допомогою програмістських хитрощів.

Останній етап життєвого циклу ПЗ (експлуатація і супровід) – це “повноцінне” використання програмної системи. На цьому етапі можуть виявитися помилки, допущені, наприклад, на першому етапі формування вимог. Можуть також виявитися помилки проектування і кодування, що може вимагати визначення нових функціональних можливостей системи. З іншого боку, система постійно повинна залишатися працездатною. Внесення необхідних змін у програмну систему може вимагати повторення деяких або навіть усіх етапів процесу створення ПЗ.

До недоліків каскадної моделі можна віднести негнучку розбивку процесу створення ПЗ на окремі фіксовані етапи. У цій моделі визначаючи систему рішення приймаються на ранніх етапах і потім їх важко скасувати або змінити , особливо це відноситься до формування системних вимог. Тому каскадна модель застосовується тоді, коли вимоги формалізовані досить чітко і коректно. Разом з тим каскадна модель добре відбиває практику створення ПЗ. Технології створення ПЗ, засновані на даній моделі, використовуються повсюдно, зокрема для розробки систем, що входять до складу великих інженерних проектів.

Еволюційна модель розробки

Ця модель заснована на наступній ідеї: розробляється первісна версія програмного продукту, яка передається на випробування користувачам, потім вона допрацьовується з урахуванням думки користувачів, виходить проміжна версія продукту, яка також проходить “випробування користувачем”, знову допрацьовується і так кілька раз, поки не буде отриманий необхідний програмний продукт (рис. 1.2). Відмітною рисою даної моделі є те, що процеси специфікування розробки і атестації ПЗ виконуються паралельно при постійному обміну інформацією між ними.

Розрізняють два підходи до реалізації еволюційного методу розробки.

1. *Підхід пробних розробок.* Тут велику роль відіграє постійна робота із замовником (або користувачами) для того, щоб визначити повну систему вимог до ПЗ, необхідну для розробки кінцевої версії продукту. У рамках цього підходу спочатку розробляються ті частини системи, які очевидні або добре специфіковані. Система еволюціонує (допрацьовується) шляхом додавання нових засобів у міру їх пропозиції замовником.
2. *Прототипування.* Тут метою процесу еволюційної розробки ПЗ є поетапне уточнення вимог замовника і, отже, одержання закінченої специфікації, що визначає розроблювальну систему. Прототип *(Під прототипом зазвичай розуміється діючий програмний модуль, що реалізує окремі функції створюваного ПЗ)* зазвичай будується для експериментування з тою частиною вимог замовника, які сформовані нечітко або із внутрішніми протиріччями.

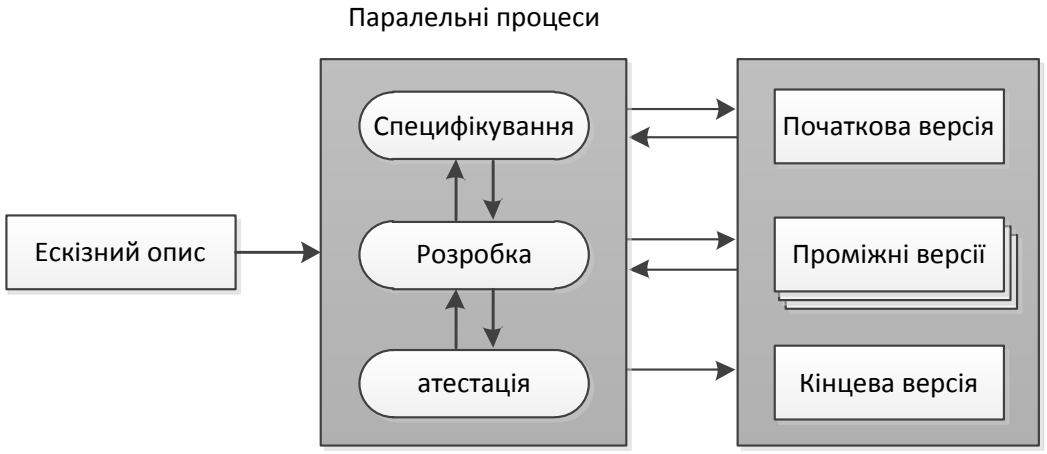

Рис. 1.2. Еволюційна модель розробки

Еволюційний підхід часто більш ефективний, чим підхід, побудований на основі каскадної моделі, особливо якщо вимоги замовника можуть мінятися в процесі розробки системи. Перевагою процесу створення ПЗ, побудованого на основі еволюційного підходу, є те, що тут специфікація може розроблятися поступово, у міру того як замовник (або користувачі) усвідомлює і сформулює ті завдання, які повинні вирішувати програмне забезпечення. Разом з тим даний підхід

має і деякі недоліки.

1. *Багато етапів процесу створення ПЗ не документовані.* Менеджерам проекту створення ПЗ необхідно регулярно документально відслідковувати виконання робіт. Але якщо система розробляється швидко, то економічно не вигідно документувати кожну версію системи.
2. *Система часто виходить погано структурованою.* Постійні зміни у вимогах приводять до помилок і недоглядам у структурі ПЗ. Згодом внесення змін у систему стає усе більш складним і витратним.
3. *Часто потрібні спеціальні засоби і технології розробки ПЗ.* Це викликано необхідністю швидкої розробки версій програмного продукту. Але, з іншого боку, це може привести до несумісності деяких застосовуваних засобів і технологій, що, у свою чергу, вимагає наявності в команді розробників фахівців високого рівня.

Еволюційний підхід найбільш прийнятний для розробки невеликих програмних систем (до 100 000 рядків коду) і систем середнього розміру (до 500 000 рядків коду) з відносно коротким строком життя. На більших довгоіснуючих системах занадто помітно проявляються недоліки цього підходу. Для таких систем рекомендується змішаний підхід до створення ПЗ, який увібрав би в себе кращі риси каскадної і еволюційної моделей розробки.

При такому змішаному підході для прояснення “темних місць” у системній специфікації можна використовувати прототипування. Частина системних компонентів, для яких повністю визначені вимоги, може створюватися на основі каскадної моделі. Інші системні компоненти, які важко піддаються специфікуванню, наприклад користувацький інтерфейс, можуть розроблятися з використанням прототипування.

Формальна розробка систем

Цей підхід до створення ПЗ має багато рис, схожих з каскадною моделлю, але побудований на основі формальних математичних перетворень системної специфікації в програму, що виконується. Процес створення програмного

забезпечення відповідно до цього підходу показаний на рис. 1.3. Тут для спрощення не зазначені зворотні зв'язки між етапами процесу.

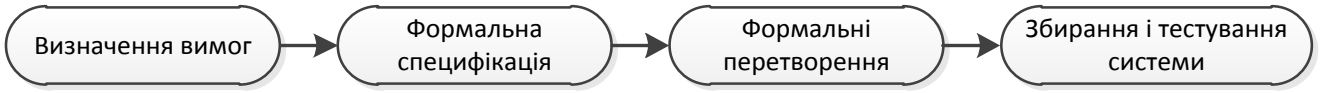

Рис. 1.3. Модель формальної розробки ПЗ

Між даним підходом і каскадною моделлю існують наступні кардинальні відмінності.

1. Тут специфікація системних вимог має вигляд деталізованої формальної специфікації, записаної за допомогою спеціальної математичної нотації.
2. Процеси проектування, написання програмного коду і тестування системних модулів заміняються процесом, у якому формальна специфікація шляхом послідовних формальних перетворень трансформується в програму, що виконується. Цей процес показаний на рис. 1.4.

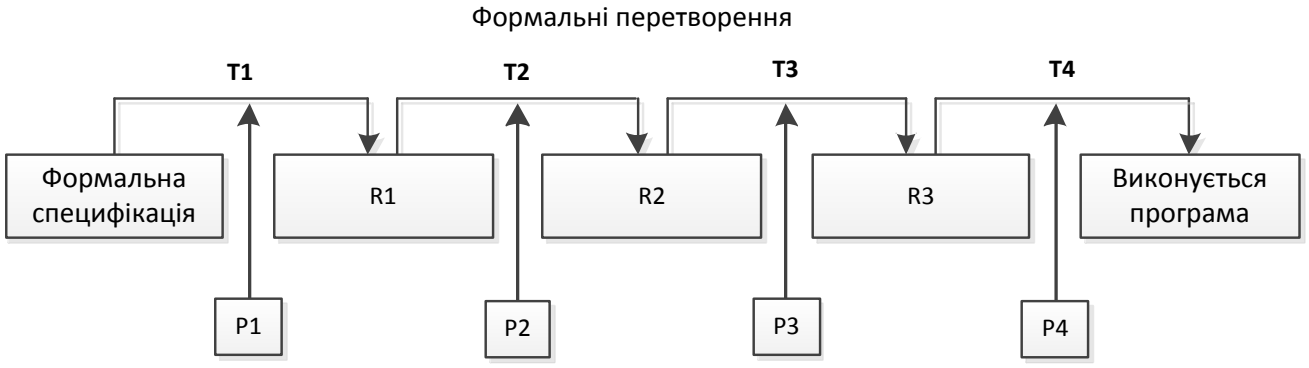

Рис. 1.4. Процес формальних перетворень

У процесі перетворення формальне математичне представлення системи послідовно і математично коректно трансформується в програмний код, поступово усе більш деталізований. Ці перетворення виконуються доти, поки всі позиції формальної специфікації не будуть трансформовані в еквівалентну програму. Перетворення виконуються математично коректно – тут не існує проблеми

перевірки відповідності специфікації і програми.

Перевага методу формальних перетворень, яка полягає в точній відповідності кінцевої програми специфікації, забезпечується тим, що дистанція між послідовними перетвореннями значно менше дистанції між специфікацією і програмою. Доказ коректності програмного коду для більших масштабованих систем зазвичай дуже довгостроковий, а часто просто не здійсненний. Щодо цього метод формальних перетворень, що полягає з послідовності невеликих формальних кроків, досить привабливий. Однак вибір для застосування відповідних формальних перетворень складний і неочевидний.

Найбільш відомим прикладом методу формальних перетворень є метод “чистої кімнати” (Cleanroom), розроблений компанією IBM. Цей метод передбачає покрокову розробку ПЗ, коли на кожному кроку застосовується формальні перетворення. Це дозволяє відмовитися від тестування окремих програмних модулів, а тестування всієї системи відбувається після її складання.

Як метод “чистої кімнати”, так і інші методи формальних перетворень ґрунтуються на B-методі. Усі ці методи мають декілька “уроджених” недоліків, а вартість розробки ПЗ за допомогою цих методів не набагато відрізняється від вартості розробок іншими методами. Методи формальних перетворень зазвичай застосовують для розробки систем, які повинні задовольняти строгим вимогам надійності, безвідмовності і безпеки, тому що вони гарантують відповідність створених систем їх специфікаціям.

Крім розробки зазначеного типу систем, методи формальних перетворень не знайшли широкого застосування, оскільки вимагають спеціальних знань і досвіду використання. Крім того, для більшості систем ці методи не дають істотного виграшу у вартості або якості в порівнянні з іншими методами розробки ПЗ. Основна причина полягає в тому, що функціонування більшості систем із труднощами піддається опису методом формальних специфікацій – при створенні більшості програмних систем більша частина зусиль розробників іде саме на створення специфікацій.

Розробка ПЗ на основі раніше створених компонентів

У більшості програмних проектів застосовується повторне використання деяких програмних модулів. Це зазвичай трапляється там, де розробники проекту знають про раніше створені програмні продукти, у складі яких є компоненти, що приблизно задовольняють вимогам розроблювальних компонентів. Ці компоненти модифікуються відповідно до нових вимог і потім включається до складу нової системи. В еволюційній моделі розробки для прискорення процесу створення ПЗ повторне використання раніше створених компонентів застосовується досить часто.

Неформальне рішення про повторне використання раніше створених програмних компонентів зазвичай приймається незалежно від загального процесу створення ПЗ. Разом з тим протягом декількох останніх років усе більш широко застосовується підхід до створення ПЗ, заснований саме на повторному використанню раніше створених програмних модулів.

Цей підхід заснований на наявності великої бази існуючих програмних компонентів, які можна інтегрувати в створювану нову систему. Часто такими компонентами є програмні продукти, що вільно продаються на ринку, які можна використовувати для виконання певних спеціальних функцій, таких як форматування тексту, числові обчислення і т.п. Загальна модель процесу розробки ПЗ з повторним використанням раніше створених компонентів показана на рис. 1.5.

У цьому підході початковий етап специфікування вимог і етап атестації такі ж, як і в інших моделях процесу створення ПЗ. А етапи, розташовані між ними, мають наступний сенс.

1. *Аналіз компонентів.* Маючи специфікацію вимог, на цьому етапі здійснюється пошук компонентів, які могли б задовольнити сформульованим вимогам. Зазвичай неможливо точно зіставити функції, реалізовані готовими компонентами, і функції, визначені специфікацією вимог.
2. *Модифікація вимог.* На цій стадії аналізуються вимоги з урахуванням інформації про компоненти, отримані на попередньому етапі. Вимоги модифікуються таким чином, щоб максимально використовувати

можливості відібраних компонентів. Якщо зміна вимог неможлива, повторно виконується аналіз компонентів для того, щоб знайти який-небудь альтернативне рішення.

3. *Проектування системи.* На даному етапі проектується структура системи або модифікується існуюча структура повторно використовуваної системи. Проектування повинне враховувати відібрані програмні компоненти і будувати структуру відповідно до їхніх функціональних можливостей. Якщо деякі готові програмні компоненти недоступні, проектується нове ПЗ.
4. *Розробка і складання системи.* Це етап безпосереднього створення системи. У рамках розглянутого підходу складання системи є скоріше частиною розробки системи, чим окремим етапом.

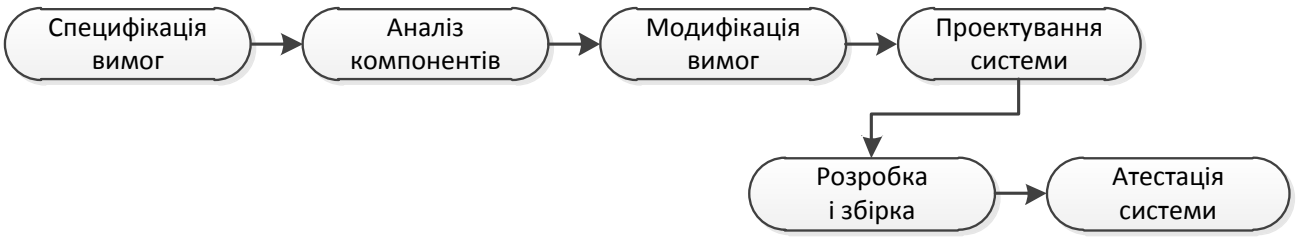

Рис. 1.5. Розробка ПЗ з повторним використанням раніше створених компонентів

Основні переваги описуваної моделі процесу розробки ПЗ з повторним використанням раніше створених компонентів полягають у тому, що скорочується кількість безпосередньо розроблювальних компонентів і зменшується загальна вартість створюваної системи. Разом з тим при використанні цього підходу неминучі компроміси при визначенні вимог; це може привести до того, що закінчена система не буде задовольняти всім вимогам замовника. Крім того, при проведенні модернізації системи (тобто при створенні її нової версії) відсутня можливість впливати на появу нових версій компонентів, використовуваних у системі, що значно ускладнює сам процес модернізації.

2. Ітераційні моделі розробки ПЗ

Описані моделі процесу створення ПЗ мають свої переваги і недоліки. При створенні великих систем, як правило, доводиться використовувати різні підходи до розробки різних частин системи, тобто в цілому до розробки системи застосовуються гібридні (змішані) моделі. Тому важливу роль відіграє можливість виконувати окремі процеси розробки підсистем і весь процес створення ПЗ ітераційно, коли у відповідь на зміни вимог повторно виконуються визначені етапи створення системи (найчастіше етапи проектування і кодування).

1. *Модель покрокової розробки,* де процеси специфікування вимог, проектування і написання коду розбиваються на послідовність невеликих кроків, які ведуть до створення ПЗ.
2. *Спіральна модель розробки,* у якій увесь процес створення ПЗ, від початкового ескізу системи до її кінцевої реалізації, розвертається по спіралі.

Істотною відмінністю ітераційних моделей є те, що тут процес розробки специфікації протікає паралельно з розробкою самої програмної системи. Більше того, у моделі покрокової розробки повну системну специфікацію можна одержати тільки після завершення самого останнього кроку процесу створення ПЗ. Очевидно, що такий підхід входить у суперечність із моделлю придбання ПЗ, коли повна системна специфікація є складовою частиною контракту на розробку системи. Тому, щоб застосовувати ітераційні моделі для розробки великих систем, які замовляються “серйозними” організаціями, наприклад державними агентствами, необхідно міняти форму контракту, на що такі організації йдуть з великими труднощами.

Модель покрокової розробки

У каскадній моделі створення ПЗ визначення вимог здійснюється разом із замовником до початку проектування системи, точно так само системна архітектура повинна бути створена до початку безпосередньої реалізації (кодування) системи.

Зміни у вимогах, зроблені на етапі написання коду, ведуть до необхідності виконання повторних робіт із проектування і кодуванню системи. Разом з тим до переваг каскадної моделі можна віднести простоту керування процесом створення ПЗ (у рамках даної моделі), а також наявність окремих етапів проектування і реалізації, що приводить до створення цілком працездатних систем, у яких враховані всі зміни в специфікації, зроблені вже під час самого процесу розробки ПЗ. На відміну від каскадної, в еволюційній моделі можна відкласти прийняття остаточних розв'язків про специфікацію і структурі системи, однак це може привести до створення погано структурованої системи, яка буде також важка в супроводі.

Модель покрокової розробки знаходиться десь між цими моделями і поєднує їх переваги. Ця модель (рис. 1.6) була запропонована Міллсом (Mills) як спроба зменшити кількість повторно виконуваних робіт у процесі створення ПЗ і збільшити для замовника часовий період остаточного ухвалення рішення про всі деталі системних вимог

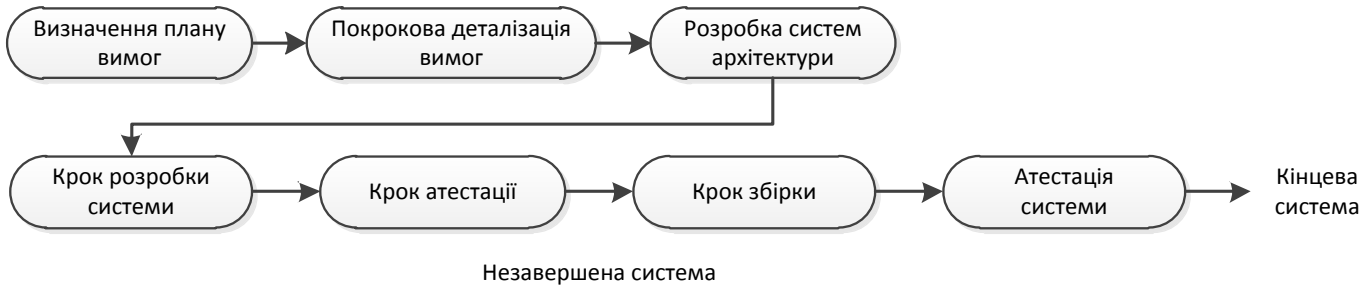

Рис. 1.6. Модель покрокової розробки

У процесі покрокової розробки замовник спочатку в загальних рисах визначає ті сервіси (функціональні можливості), які повинні бути присутнім у створюваної системи. При цьому встановлюються пріоритети, тобто визначається, які сервіси більш важливі, а які – менш. Також визначається кількість кроків розробки, причому на кожному кроці повинен бути отриманий системний компонент, що реалізує визначену підмножину системних функцій. Розподіл реалізації системних сервісів по кроках розробки залежить від їхніх пріоритетів. Сервіси з більш високими пріоритетами реалізуються першими.

Послідовність кроків розробки визначається заздалегідь до початку їх виконання. На перших кроках деталізуються вимоги для сервісів, потім для їхньої реалізації (на наступних кроках) використовується один з підходящих способів розробки ПЗ. У ході їх реалізації аналізуються і деталізуються вимоги для компонентів, які будуть розроблятися на більш пізніх кроках, причому зміна вимог для тих компонентів, які вже перебувають у процесі розробки, не допускається.

Після завершення кроку розробки одержуємо програмний компонент, який передається замовнику для інтегрування в підсистему, що реалізує певний системний сервіс. Замовник може експериментувати з готовими підсистемами і компонентами для того, щоб уточнити вимоги, пропоновані до наступних версій уже готових компонентів або до компонентів, розроблювальних на наступних кроках. По завершенню чергового кроку розробки отриманий компонент інтегрується з раніше зробленими компонентами; таким чином, після кожного кроку розробки система здобуває все більшу функціональну завершеність. Загальносистемні функції в цьому процесі можуть реалізуватися відразу або поступово, у міру розробки необхідних компонентів.

В описуваній моделі не передбачається, що на кожному кроці використовується той самий підхід до процесу розробки компонентів. Якщо створюваний компонент має добре розроблену специфікацію, то для його створення можна застосувати каскадну модель. Якщо ж вимоги визначені нечітко, можна використовувати еволюційну модель розробки.

Процес покрокової розробки має цілий ряд переваг.

1. Замовнику немає необхідності чекати повного завершення розробки системи, щоб одержати про неї представлення. Компоненти, отримані на перших кроках розробки, задовольняють найбільш критичним вимогам (тому що мають найбільший пріоритет) і їх можна оцінити на самій ранній стадії створення системи.
2. Замовник може використовувати компоненти, отримані на перших кроках розробки, як прототипи і провести з ними експерименти для уточнення вимог до тих компонентів, які будуть розроблятися пізніше.

3. Даний підхід зменшує ризик загальносистемних помилок. Хоча в розробці окремих компонентів можливі помилки, але ці компоненти повинні пройти відповідні тестування і атестацію, перш ніж їх передадуть замовнику.
4. Оскільки системні сервіси з високим пріоритетом розробляються першими, а всі наступні компоненти інтегруються з ними, неминуче виходить так, що найбільш важливі підсистеми зазнають більш ретельному всебічному тестуванню і перевірці. Це значно знижує ймовірність програмних помилок в особливо важливих частинах системи.

Разом з тим при реалізації покрокової розробки можуть виникнути певні проблеми. Компоненти, одержувані на кожному кроці розробки, мають відносно невеликий розмір (зазвичай не більш 20 000 рядків коду), але повинні реалізувати яку-небудь системну функцію. Відобразити безліч системних вимог до компонентів потрібного розміру досить складно. Більше того, багато систем повинні мати набір базових системних властивостей, які реалізуються спільно різними частинами системи. Оскільки вимоги детально не визначені доти, поки не будуть розроблені всі компоненти, буває досить складно розподілити загальносистемні функції по компонентах.

У цей час запропонований метод так званого екстремального програмування (extreme programming), який усуває деякі недоліки методу покрокової розробки. Цей метод заснований на покроковій розробці малих програмних компонентів, що реалізують невеликі функціональні вимоги, постійнім залученням замовника в процес розробки і знеособленім програмуванні.

Спіральна модель розробки

Спіральна модель процесу створення програмного забезпечення (рис. 1.7) у цей час одержала широку відомість та популярність. На відміну від розглянутих раніше моделей, де процес створення ПЗ представлений у вигляді послідовності окремих процесів з можливим зворотним зв'язком між ними, тут процес розробки представлений у вигляді спіралі. Кожний виток спіралі відповідає однієї стадії (ітерації) процесу створення ПЗ. Так, самий внутрішній виток спіралі відповідає

стадії ухвалення рішення про створення ПЗ, на наступному витку визначаються системні вимоги, далі випливає стадія (виток спіралі) проектування системи і т.д.

Кожний виток спіралі розбитий на чотири сектори.

1. *Визначення цілей.* Визначаються мети кожної ітерації проекту. Крім того, установлюються обмеження на процес створення ПЗ і на сам програмний продукт, уточнюються плани виробництва компонентів. Визначаються проектні ризики (наприклад, ризик перевищення строків або ризик перевищення вартості проекту). Залежно від "виявлених" ризиків, можуть плануватися альтернативні стратегії розробки ПЗ.
2. *Оцінка і дозвіл ризиків.* Для кожного певного проектного ризику проводиться його детальний аналіз. Плануються заходи для зменшення (дозволу) ризиків. Наприклад, якщо існує ризик, що системні вимоги визначені невірно, планується розробити прототип системи.
3. *Розробка і тестування.* Після оцінки ризиків вибирається модель процесу створення системи. Наприклад, якщо домінують ризики, пов'язані з розробкою інтерфейсів, найбільш підходящою буде еволюційна модель розробки ПЗ із прототипуванням. Якщо основні ризики пов'язані з відповідністю системи і специфікації, швидше за все, слід застосувати модель формальних перетворень. Каскадна модель може бути застосована в тому випадку, якщо основні ризики визначені як помилки, які можуть виявитися на етапі складання системи.
4. *Планування.* Тут переглядається проект і приймається рішення про те, чи починати наступний виток спіралі. Якщо приймається рішення про продовження проекту, розробляється план на наступну стадію проекту.

Істотна відмінність спіральної моделі від інших моделей процесу створення ПЗ полягає в точному визначенню і оцінюванні ризиків. Якщо говорити неформально, то ризик – це ті неприємності, які можуть трапитися в процесі розробки системи. Наприклад, якщо при написанні програмного коду використовується нова мова програмування, то ризик може полягати в тому, що компілятор цієї мови може бути ненадійним або що результуючий код може бути не

досить ефективним. Ризики можуть також полягати в перевищенні строків або вартості проекту. Таким чином, зменшення (дозвіл) ризиків - важливий елемент керування системним проектом.

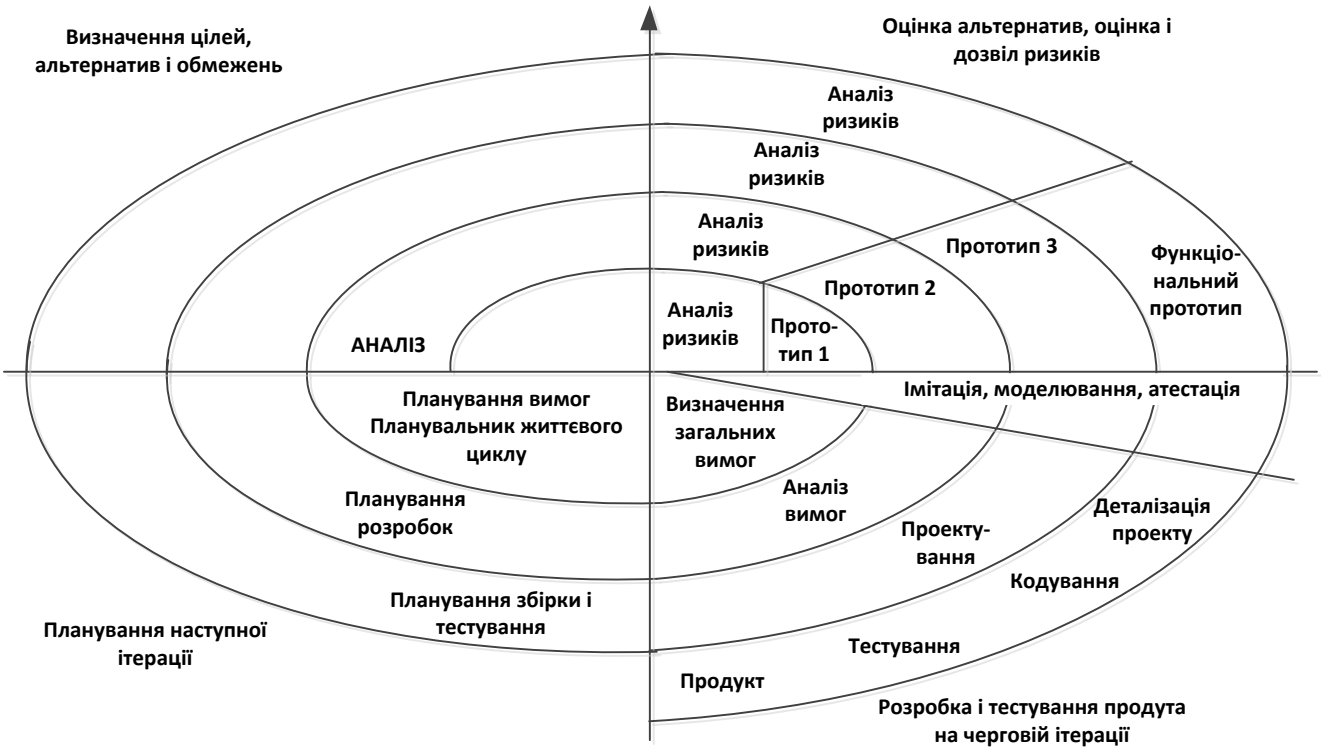

Рис. 1.7. Спіральна модель створення ПЗ

Перша ітерація створення ПЗ в спіральній моделі починається з ретельного пророоблення системних показників (цілей системи), таких як експлуатаційні показники і функціональні можливості системи. Зазвичай, альтернативних шляхів досягнення цих показників або цілей можна сформувати нескінченно багато. Але кожна альтернатива повинна оцінювати вартість досягнення кожної сформульованої мети. Результати аналізу можливих альтернатив служать джерелом оцінки проектного ризику. Це відбувається на наступній стадії спіральної моделі, де для оцінки ризиків використовуються більш детальний аналіз альтернатив, прототипування, імітаційне моделювання і т.п. З урахуванням отриманих оцінок ризиків вибирається той або інший підхід до розробки системних компонентів, далі він реалізується, потім здійснюється планування наступного етапу процесу

створення ПЗ.

У спіральній моделі немає фіксованих етапів, таких як розробка специфікації або проектування. Ця модель може містити в собі будь-які інші моделі розробки систем. Наприклад, на одному витку спіралі може використовуватися прототипування для більш чіткого визначення вимог (і, отже, для зменшення відповідних ризиків). Але на наступному витку може застосовуватися каскадна модель. Якщо вимоги чітко сформульовані, може застосовуватися метод формальних перетворень.

3. Специфікація програмного забезпечення

Основні базові процеси створення ПЗ: формування специфікації, розробка, атестація і модернізація програмних систем. Перший із цих процесів, формування специфікації, призначений для визначення сервісів, які буде мати проектоване ПЗ, а також обмежень, що накладаються на функціональні можливості і розробку програмної системи. Цей процес у цей час зазвичай називають “розробка вимог” (requirements engineering). Розробка вимог часто є критичним етапом у створенні ПЗ, оскільки помилки, допущені на цьому етапі, ведуть до виникнення проблем на етапах проектування і розробки.

Схема процесу розробки вимог показана на рис. 1.8. Результатом його виконання є розробка документації, що формалізує вимоги, які пред'являються до системи, тобто створення системної специфікації. У цій документації вимоги зазвичай представлені на двох рівнях деталізації. На самому верхньому рівні представлені вимоги, обумовлені кінцевими користувачами або замовниками ПЗ; але для розробників необхідна більш деталізована системна специфікація.

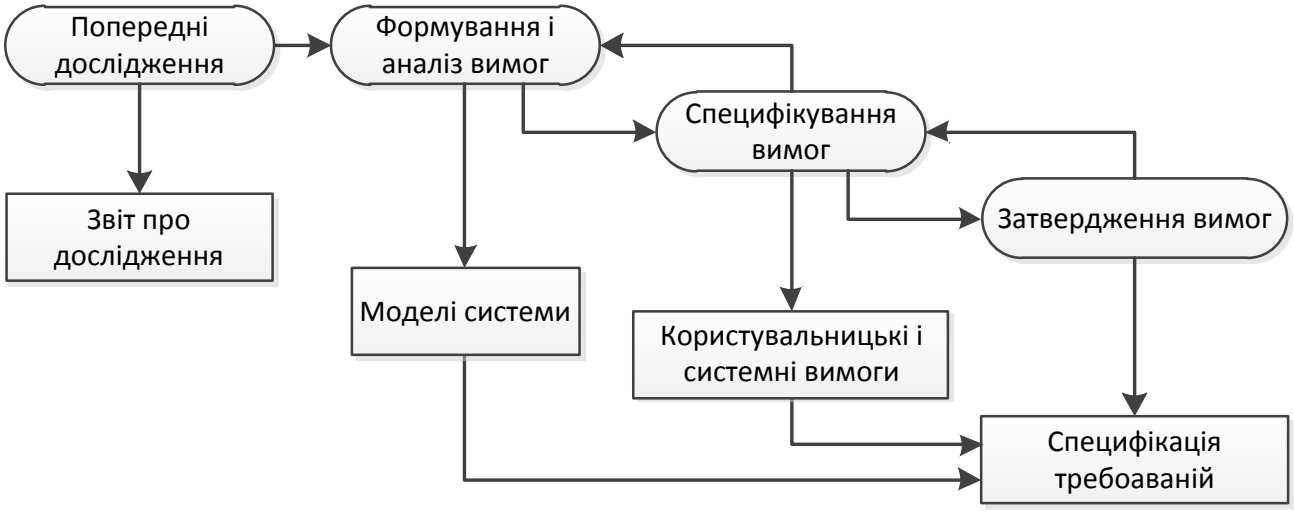

Рис. 1.8. Процес розробки вимог

Процес розробки вимог включає чотири основні етапи.

1. *Попередні дослідження.* Оцінюється ступінь задоволеності користувачів існуючими програмними продуктами і апаратними засобами, а також економічна ефективність майбутньої системи і можливість укластися в існуючі бюджетні обмеження при її розробці. Цей етап повинен бути по можливості коротким і дешевим.
2. *Формування і аналіз вимог.* Формуються системні вимоги шляхом вивчення існуючих аналогічних систем, обговорення майбутньої системи з потенційними користувачами і замовниками, аналізу завдань, які повинна вирішувати система, і т.п. Цей етап може включати розробку декількох моделей системи і її прототипів, що допомагає сформувати функціональні вимоги до системи.
3. *Специфікування вимог.* Здійснюється переклад усієї сукупності інформації, зібраної на попередньому етапі, у документ, що визначає безліч вимог. Цей документ зазвичай містить два типи вимог:
    - користувацькі – узагальнені представлення замовників і кінцевих користувачів про систему;
    - системні – детальний опис функціональних показників системи.
4. *Твердження вимог.* Перевіряється здійсненність, погодженість і повнота безлічі вимог. У процесі формування обмежень неминуче виникнення яких-небудь помилок. На цьому етапі вони повинні бути по можливості виявлені

і усунуті.

Зазвичай, процес розробки вимог важко укласти в описану послідовність етапів. Наприклад, аналіз вимог виконується протягом усього процесу їх розробки, тому внесення нових або зміна вже сформульованих вимог можливо на будь-якому етапі. Як правило, етапи розробки вимог перекриваються в часі.

4. Проектування і реалізація ПЗ

Реалізація програмного забезпечення – це процес перекладу системної специфікації в працездатну систему. Етап реалізації завжди включає процеси проектування і програмування, але якщо для розробки ПЗ застосовується еволюційний підхід, етап реалізації також може включати процес внесення змін у системну специфікацію.

На етапі проектування ПЗ визначається його структура, дані, які є частиною системи, інтерфейси взаємодії системних компонентів і іноді використовувані алгоритми. Проектувальники відразу ніколи не одержують закінчений результат – процес проектування зазвичай проходить через розробку декількох проміжних версій ПЗ. Проектування передбачає послідовну формалізацію і деталізацію створюваного ПЗ з можливістю внесення змін у рішення, прийняті на більш ранніх стадіях проектування.

Процес проектування може включати розробку декількох моделей системи різних рівнів узагальнення. Оскільки проектування – це процес декомпозиції, такі моделі допомагають виявити помилки, допущені на ранніх стадіях проектування, а отже, дозволяють внести зміни в раніше створені моделі. На рис. 1.9 показана схема процесу проектування ПЗ із вказівкою результату кожного етапу проектування. Ця схема побудована в припущенні, що всі етапи процесу проектування виконуються послідовно. На практиці ці етапи перекриваються внаслідок неминучих зворотних зв'язків від одного етапу до попереднього і повторного виконання деяких проектних робіт.

Результатом кожного етапу проектування є специфікація, необхідна для виконання наступного етапу. Ця специфікація може бути абстрактною і

формальною, тобто такою, яка необхідна для деталізації системних вимог; але вона може бути і частиною розроблювальної системи. Тому що процес проектування безперервний, специфікації поступово стають усе більш деталізованими. Кінцевими результатами процесу проектування є точні специфікації на алгоритми і структури даних, які будуть реалізовані на наступному етапі створення ПЗ.

Нижче перераховані окремі етапи процесу проектування.

1. *Архітектурне проектування.* Визначаються і документуються підсистеми і взаємозв'язки між ними.
2. *Узагальнена специфікація.* Для кожної підсистеми розробляється узагальнена специфікація на її сервіси і обмеження.
3. *Проектування інтерфейсів.* Для кожної підсистеми визначається і документується її інтерфейс. Специфікації на ці інтерфейси повинні бути точно вираженими і однозначними, щоб використання підсистем не вимагало знань про те, як вони реалізують свої функції. На цьому етапі можна застосувати методи формальних специфікацій.
4. *Компонентне проектування.* Проводиться розподіл системних функцій (сервісів) по різних компонентах і їх інтерфейсам.
5. *Проектування структур даних.* Детально розробляються структури даних, необхідні для реалізації програмної системи.
6. *Проектування алгоритмів.* Детально розробляються алгоритми, призначені для реалізації системних сервісів.

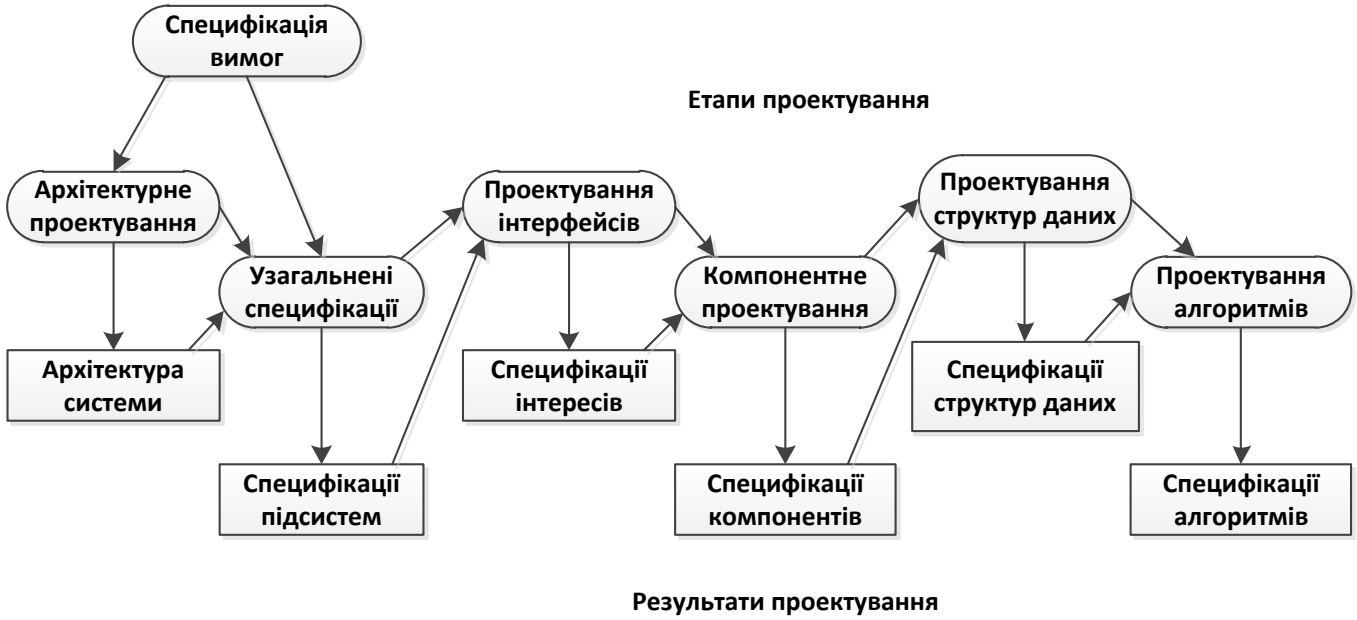

Рис. 1.9. Узагальнена схема процесу проектування

Описана схема процесу проектування є досить загальною і на практиці може (і повинна) адаптуватися відносно до розробки конкретного програмного продукту. Наприклад, два останні етапи, проектування структур даних і алгоритмів, можуть бути як складовими частинами процесу проектування, так і входити в процес реалізації ПЗ. Якщо для створення програмної системи використовуються деякі вже готові компоненти, це може накласти обмеження на архітектуру системи і інтерфейси системних модулів. Це означає, що кількість компонентів, що вимагають проектування, значно зменшиться. Якщо в процесі проектування використовується метод проб і помилок, то системні інтерфейси можуть розроблятися після визначення структур даних.

Методи проектування

У багатьох проектах розробки ПЗ процес проектування виконується за допомогою *спеціально підібраних* методів. Відштовхуючись від безлічі вимог, зазвичай записаних природньою мовою, спочатку виконується неформальне проектування. Коментарі до програмного коду і проміжні специфікації можуть змінюватися в процесі реалізації системи. Після завершення стадії реалізації (тобто програмування і налагодження системи) у проектну документацію також вносяться

зміни, покликані усунути помилки і неповноту опису системи в первісній специфікації.

Найбільш розробленим підходом до проектування ПЗ мають так звані структурні методи, які пропонують безліч формалізованих нотацій і нормативних інструкцій для проектування програмних продуктів. Як приклад цих методів можна назвати структурне проектування, структурний аналіз систем, розробку систем Джексона (Jackson), а також різноманітні методи, засновані на об'єктно-орієнтованому підході.

Застосування структурних методів зазвичай приводить до створення графічних моделей системи і великому обсягу проектної документації. CASE-засоби призначені для підтримки саме таких методів. Структурні методи успішно застосовувалися в багатьох програмних проектах. Вони значно знижують вартість розробки, оскільки використовують стандартні нотації для одержання стандартної проектної документації. Ні про один з цих методів не можна сказати, що він краще або гірше інших. Успішне або неуспішне застосування того або іншого методу часто залежить від типу розроблювального ПЗ.

Кожний структурний метод включає такі компоненти, як модель процесу проектування, стандартизовані нотації для представлення структури системи, формати звітів, правила і нормативні вказівки по проектуванню. Хоча розроблена велика кількість таких методів, вони мають щось загальне. Структурні методи підтримують усі або, принаймні деякі з перерахованих нижче моделей систем.

1. Модель потоків даних, де система моделюється у вигляді потоку даних, преутворених у цій системі.
2. Модель “ сутність-зв'язок”, яка застосовується для опису сутностей (об'єктів програмної системи) і зв'язків між ними. Ця модель часто використовується при проектуванні структур баз даних.
3. Структурна модель, призначена для документування системних компонентів і їх взаємозв'язків.
4. Об'єктно-орієнтовані методи, за допомогою яких одержують ієрархічну модель системи, моделі статичних і динамічних відносин між об'єктами і модель взаємодії об'єктів під час роботи системи.

Деякі структурні методи доповнюються іншими системними моделями, такими як діаграми переходів (з одного стану в інший) або сценарії життя сутностей, які показують послідовність перетворень для кожної сутності. Багато методів передбачають наявність централізованих сховищ (репозиторіїв) для системної інформації або словників використовуваних даних.

На практиці методи представляють нормативні керівництва неформально, так що різні проектувальники можуть реалізувати різні шляхи проектування. Фактично ці “методи” є набором стандартних нотацій і просто відображають успішну практику проектування. Дотримуючись цих методів і їх нормативним інструкціям, можна прийти до раціонального і розумного процесу проектування. Разом з тим творчість проектувальників повинна виявитися в способі декомпозиції системи, що адекватно відображає системні вимоги. З іншого боку, проведені дослідження праці проектувальників показали, що найчастіше вони просто сліпо додержуються цих методів. Та і самі методи вони вибирають залежно від приватних обставин, а не відповідно до їхніх переваг або недоліків.

Програмування і налагодження

Процес програмування (написання програмного коду, кодування) зазвичай випливає безпосередньо за процесом проектування. Але для деяких класів програм, наприклад критичних по надійності систем, остання стадія проектування (детальне проектування) і початок кодування можуть перекриватися. У процесі проектування можуть використовуватися CASE-засоби, які дозволяють одержати кістякову програму. Така програма містить код для визначення і реалізації інтерфейсів, і в багатьох випадках програмісту залишається тільки додати код, що реалізує деякі деталі функціонування програмного компонента.

Програмування – індивідуальний процес, тут не існує загальних правил, яким необхідно випливати при написанні програмного коду. Деякі програмісти починають кодування з компонентів, які вони добре розуміють, залишаючи наостанку кодування компонентів, які є для них “темними”. Інші застосовують протилежний підхід, залишаючи прості для них компоненти на потім.

Зазвичай програмісти самі тестують написаний ними програмний код для виявлення можливих помилок і програмних дефектів. Цей процес називається *налагодженням* програми. У принципі тестування і налагодження є різними процесами. При тестуванні встановлюється наявність програмних помилок. У ході налагодження встановлюється місце розташування помилок, потім вони усуваються. На рис. 1.10 показаний можливий процес налагодження програми. Налагодження може бути частиною як процесу розробки, так і процесу тестування ПЗ.

Програміст, який проводить налагодження повинен згенерувати такі режими роботи системи, які допоможуть виявити програмні помилки по аномальній поведінці системи. Локалізація помилок може вимагати проведення ручного трасування коду програми. У процесі тестування і налагодження можуть допомогти налагоджувальні засоби, що показують значення програмних змінних і виконуючі трасування операторів, що виконуються.

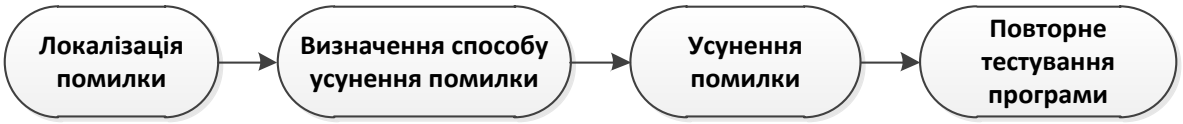

Рис. 1.10. Процес налагодження

5. Атестація програмних систем

Атестація ПЗ, або більш узагальнено – верифікація і атестація, призначено показати відповідність системи її специфікації, а також очікуванням і вимогам замовника і користувачів. До процесу атестації також можна віднести елементи контролю, такі як інспекція і оцінювання, які виконуються на кожному етапі створення ПЗ – від формування загальних вимог до кодування програм. Але все-таки основні дії по атестації виконуються після завершення реалізації на етапі тестування закінченої системи.

За винятком невеликих програм, програмні системи неможливо протестувати як єдиний цільний програмний елемент. Великі системи будуються на основі підсистем, які, у свою чергу, будуються з модулів, модулі ж компонуються із програм-процедур і програм-функцій. Для таких систем процес тестування

виконується поступово, у міру реалізації системи.

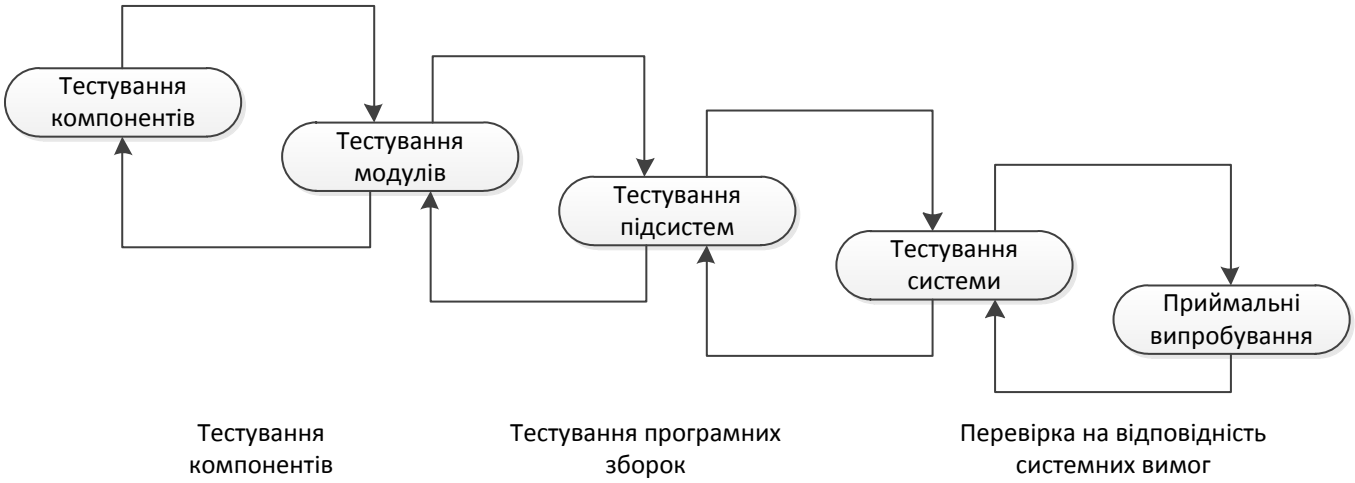

Рис. 1.11. Процес тестування

На рис. 1.11 показаний п'ятиетапний процес тестування, де спочатку тестуються окремі програмні компоненти і підсистеми, потім зібрана система і нарешті система з даними, наданими замовником. В ідеалі помилки в програмних компонентах повинні виявлятися і виправлятися ще в процесі їх кодування, а помилки і недогляду в інтерфейсах – під час складання системи. Але, оскільки після виявлення будь-яких програмних помилок необхідно виконати налагодження програми, це приводить до необхідності повторення деяких етапів тестування. Наприклад, якщо програмна помилка виявилася на етапі складання системи, необхідно повторити процес тестування того програмного компонента, у якому виявлена ця помилка. Тому процес тестування ітераційний, зі зворотною передачею інформації з наступних етапів на попередні.

Процес тестування складається з декількох етапів.

1. *Тестування компонентів.* Тестуються окремі компоненти для перевірки правильності їх функціонування. Кожний компонент тестується незалежно від інших.
2. *Тестування модулів.* Програмний модуль – це сукупність залежних компонентів, таких як опис класу об'єктів, декларування абстрактних типів

даних і набір процедур і функцій. Кожний модуль тестується незалежно від інших системних модулів.

3. *Тестування підсистем.* Тестуються набори модулів, які складають окремі підсистеми. Основна проблема, яка часто проявляється на цьому етапі – непогодженість модульних інтерфейсів. Тому при тестуванні підсистем основна увага приділяється виявленню помилок у модульних інтерфейсах шляхом прогону їх через усі можливі режими.
4. *Тестування системи.* З підсистем збирається кінцева система. На цьому етапі основна увага приділяється сумісності інтерфейсів підсистем і виявленню програмних помилок, які проявляються у вигляді непередбаченої взаємодії між підсистемами. Тут також проводиться атестація системи, тобто перевіряється відповідність системної специфікації її функціональних і нефункціональних показників, а також оцінюються інтеграційні характеристики системи.
5. *Приймальні випробування.* Це кінцевий етап процесу тестування, після якого система приймається до експлуатації. Тут система тестується із залученням даних, що надаються замовником системи, а не на основі тестових даних, як було на попередньому етапі. На цьому етапі можуть виявитися помилки, допущені ще на етапі визначення системних вимог, оскільки випробування з реальними даними можуть дати інший результат, чим тестування зі спеціально підібраними тестовими даними. Приймальні випробування можуть також виявити інші проблеми в системних вимогах, якщо реальні системні характеристики не відповідають потребам замовника або система функціонує непередбаченим образом.

Тестування програмних компонентів і модулів зазвичай виконується тим програмістом, який їх розробляв. Програмісти мають власні набори тестових даних і тестують програмний код поступово, у міру його створення. Такий підхід до тестування окремих компонентів і модулів цілком виправданий, оскільки ніхто краще програміста, що розробив програмний компонент, його не знає, і тому він може підібрати найкращі тестові дані. Тестування програмних елементів можна розглядати як частину процесу їх створення, тому ми маємо право очікувати точної відповідності цих елементів і їх специфікацій.

Останні етапи тестування виконуються в процесі складання системи, до якого залучається декілька програмістів. Тому ці роботи повинні бути сплановані заздалегідь. Якщо тестування виконує незалежна команда випробувачів, плани проведення тестування повинні бути погоджені з етапами розробки специфікації і проектування. На рис. 1.12 показано, як плани тестування можуть бути пов'язані з іншими процесами розробки ПЗ.

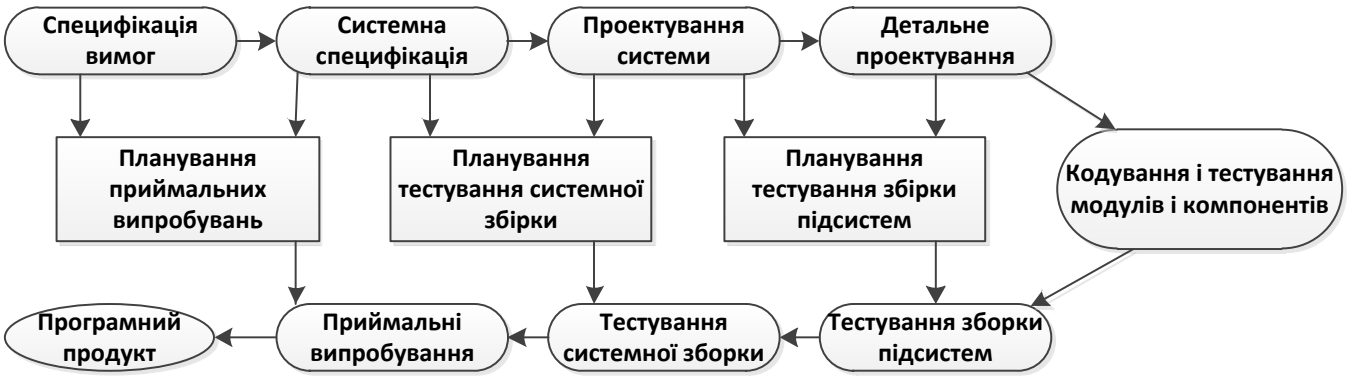

Рис. 1.12. Етапи тестування в процесі розробки ПЗ

Приймальні випробування іноді називають *альфа-тестуванням.* Зроблені на замовлення системи призначені для одного замовника. Для таких систем процес альфа-тестування триває доти, поки розробники і замовник не впевняться в тому, що розроблена система повністю відповідає системним вимогам.

Якщо система розробляється для продажу на ринку програмних продуктів, використовується так зване *бета-тестування.* Для бета-тестування система розсилається великій кількості потенційних користувачів і замовників. Вони відсилають розробникам звіти про виявлені проблеми в експлуатації системи. Бета-тестування дозволяє перевірити систему в реальних умовах експлуатації і знайти помилки, пропущені розробниками. Після одержання звітів про випробування система модернізується і знову передається на бета-тестування або відразу надходить у продаж.

6. Еволюція програмних систем

Одна з основних причин того, що в цей час у великі складні системи усе ширше впроваджується програмне забезпечення, полягає в гнучкості програмних систем. Після ухвалення рішення про розробку і виробництво апаратних компонентів системи внесення в них змін стає досить дорогим. З іншого боку, у програмне забезпечення можна вносити зміни протягом усього процесу розробки системи. Ці зміни також можуть бути вкрай дорогими, але все-таки вони значно дешевше змін в апаратному устаткуванні системи.

Історично склалося так, що існує чітка “демаркаційна лінія” між процесом розробки системи і процесом її вдосконалювання, точніше, процесом супроводу системи. Розробка системи розглядається як творчий процес, починаючи з етапу вироблення загальної концепції системи і закінчуючи одержанням працюючого програмного продукту. Супровід системи – це внесення змін у систему, яка вже знаходиться в експлуатації. І хоча вартість супроводу може в кілька раз перевищувати вартість розробки, однаково процес супроводу вважається менш творчим і відповідальним, чим процес первісного створення системи.

У цей час згадана демаркаційна лінія між процесами розробки і супроводу поступово стирається. Тільки деякі знову створені програмні системи можна назвати повністю новими. Тому має сенс розглядати процес супроводу як безперервне продовження процесу розробки. Замість двох окремих процесів раціонально прийняти еволюційний підхід інженерії програмного забезпечення, де програмні продукти протягом свого життєвого циклу безупинно змінюються (еволюціонують) у відповідь на зміни в системних вимогах і потребах користувачів. Схема цього еволюційного процесу програмних систем показана на рис. 1.13.

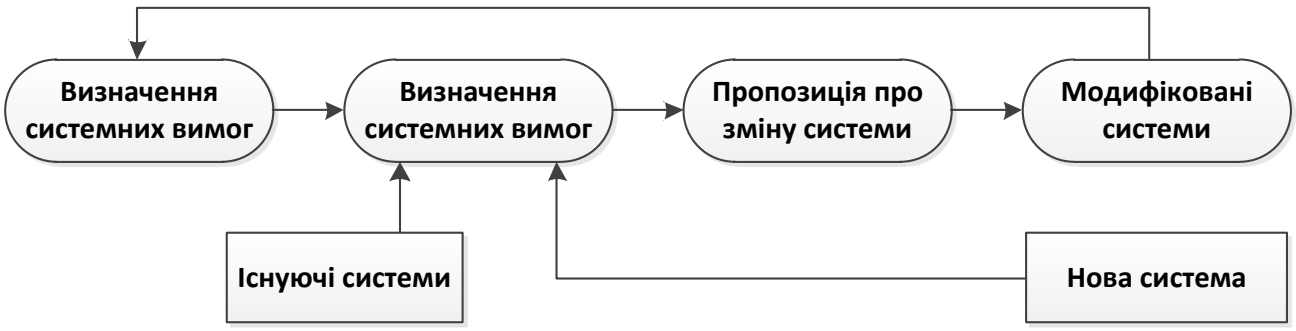

Рис. 1.13. Еволюція систем

7. Автоматизовані засоби розробки ПЗ

Абревіатура CASE (Computer-Aided Software Engineering - автоматизована розробка ПЗ) позначає спеціальний тип програмного забезпечення, призначеного для підтримки таких процесів створення ПЗ, як розробка вимог, проектування, кодування і тестування програм. Тому до CASE-засобів відносяться редактори проектів, словники даних, компілятори, відладчики, засоби побудови систем і т.п.

CASE-технології пропонують підтримку процесу створення ПЗ шляхом автоматизації деяких етапів розробки, а також створення і надання інформації, необхідної для розробки.

Приведемо приклади тих процесів, які можна автоматизувати за допомогою CASE-засобів.

1. Розробка графічних моделей системи на етапах створення специфікації і проектування.
2. Проектування структури ПЗ з використанням словників даних, що зберігають інформацію про об'єкти структури і зв'язкаи між ними.
3. Генерування користувацьких інтерфейсів на основі графічного опису інтерфейсу, створюваного в діалоговому режимі.
4. Налагодження програм на основі інформації, одержуваної в ході виконання програми.

5. Автоматична трансляція програм, написаних на застарілих мовах програмування (наприклад, COBOL), у програми, написані на сучасних мовах.

В даний час відповідні CASE-технології існують для більшості процесів, виконуваних у ході розробки ПЗ. Це веде до певного поліпшення якості створюваних програм і підвищенню продуктивності праці розробників програмного забезпечення. Разом з тим ці досягнення значно уступають тем очікуванням, які були присутні при зародженні CASE-технологій. Тоді вважалося, що варто тільки впровадити CASE-засоби – і можна одержати досить значне підвищення і якості програм, і продуктивності праці. Фактично це підвищення становить приблизно 40%. Хоча і це підвищення досить значне, CASE-технології не зробили революції в інженерії програмного забезпечення, як очікувалося.

Розширення застосування CASE-технологій обмежують два фактори.

1. Створення ПЗ, особливо етап проектування, багато в чому є творчим процесом. Існуючі CASE-засоби автоматизують рутинні процеси, спроби залучити їх до розв'язку інтелектуальних і творчих завдань проектування особливим успіхом не увінчалися.
2. У багатьох організаціях-розробниках створення ПЗ – результат роботи команди фахівців із програмного забезпечення. При цьому багато часі витрачається на “порожнє” спілкування між членами команди розробників. У цій ситуації CASE-технології не можуть запропонувати нічого такого, що здатне підвищити продуктивність праці розробників.

Класифікація CASE-засобів

Класифікація CASE-засобів допомагає зрозуміти їхні основні типи і роль, яку вони відіграють у підтримці процесів створення програмного забезпечення. Існує кілька різних класифікацій CASE-засобів, і кожна пропонує свій погляд на ці програмні продукти. Розглянемо наступні класифікації.

1. Класифікація *по виконуваних функціях.*

2. Класифікація *по типах процесів розробки,* які вони підтримують.
3. Класифікація *по категоріях,* де CASE-засоби класифікуються по ступеню інтеграції програмних модулів, що підтримують різні процеси розробки.

У табл. 1.1 представлена класифікація по виконуваних функціях із прикладами відповідних CASE-засобів. Це неповний список типів CASE-засобів, зокрема тут не представлені засоби підтримки повторного використання програмних компонентів.

Таблиця 1.1. Класифікація CASE-засобів по виконуваних функціях

Тип CASE-засоби	**Приклади**
Засоби планування	Засоби системи PERT *(PERT (Program Evaluation and Review Technique) - відома система планування і керівництва розробками програмних систем),* засоби оцінювання, електронні таблиці
Засоби редагування	Текстові редактори, редактори діаграм, тестові процесори
Засоби керування змінами	Засоби оперативного контролю над вимогами, системи керування змінами
Засоби керування конфігурацією *(Конфігурацією ПЗ називається сукупність його функціональних характеристик і фізичних показників, зафіксована в системній специфікації.)*	Системи керування версіями ПЗ, засоби побудови систем
Засоби прототипування	Мови програмування найвищого рівня, генератори користувацьких інтерфейсів
Засоби, орієнтовані на підтримку певних методів	Редактори системних структур, словники даних, генератори програмного коду
Засоби, орієнтовані на певні мови	Компілятори, інтерпретатори

програмування	
Засоби аналізу програм	Генератори перехресних посилань, статичні і динамічні аналізатори програм
Засоби тестування	Генератори тестових даних, компаратори файлів *(Компаратори - спеціальні програми порівняння яких-небудь об'єктів. У цьому випадку маються на увазі програми порівняння файлів, що містять програмний код.)*
Засоби налагодження	Інтерактивні засоби налагодження
Засоби документування	Програми розмітки сторінок, редактори зображень, генератори звітів
Засоби модернізації ПЗ	Системи створення перехресних посилань, системи модернізації програм

У табл. 1.2 представлена інша класифікація CASE-засобів. Класифікація по типах показує, які процеси створення ПЗ підтримуються тими або іншими CASE-засобами. Засоби планування і оцінювання, редагування текстів, підготовки документації і керування конфігурацією можна використовувати на всіх етапах розробки ПЗ.

Таблиця 1.2. Класифікація CASE-засобів по типах підтримуваних ними процесів розробки

Засоби модернізації ПЗ			•	
Засоби тестування			•	•
Засоби налагодження			•	•
Засоби аналізу програм			•	•
Засоби, орієнтовані на певні мови програмування		•	•	

Засоби, орієнтовані на підтримку певних методів	•	•		
Засоби прототипування	•			•
Засоби керування конфігурацією		•	•	
Засоби керування змінами	•	•	•	•
Засоби документування	•	•	•	•
Засоби редагування	•	•	•	•
Засоби планування	•	•	•	•
	Специфіку-	Проекту-	Реалізація	Атестація

Інша класифікація CASE-засобів будується на основі широти охвату процесів розробки ПЗ, підтримуваних даним засобом. Запропонована класифікація, що містить наступні три категорії *(У літературі по CASE-технологіям можна зустріти і іншу класифікацію CASE-засобів по категоріях: допоміжні програми (tools), інструментальні пакети розробника (toolkits) і автоматизовані робочі місця розробника (workbenches).*

1. *Допоміжні програми* (tools) підтримують окремі процеси розробки ПЗ, такі як перевірка несуперечності архітектури системи, компіляція програм, порівняння результатів тестів і т.п. Допоміжні програми можуть бути універсальними функціонально-закінченими засобами (наприклад, текстової процесор) або можуть входити до складу інструментальних засобів.
2. *Інструментальні засоби* (workbenches) підтримують визначені процеси розробки ПЗ, наприклад створення специфікації, проектування і т.д. Зазвичай інструментальні засоби є набором допоміжних програм, які в більшому або меншому ступені інтегровані.
3. *Робочі середовища розробника* (environments) підтримують усі або більшість процесів розробки ПЗ. Робочі середовища зазвичай включають кілька різних інтегрованих інструментальних засобів.

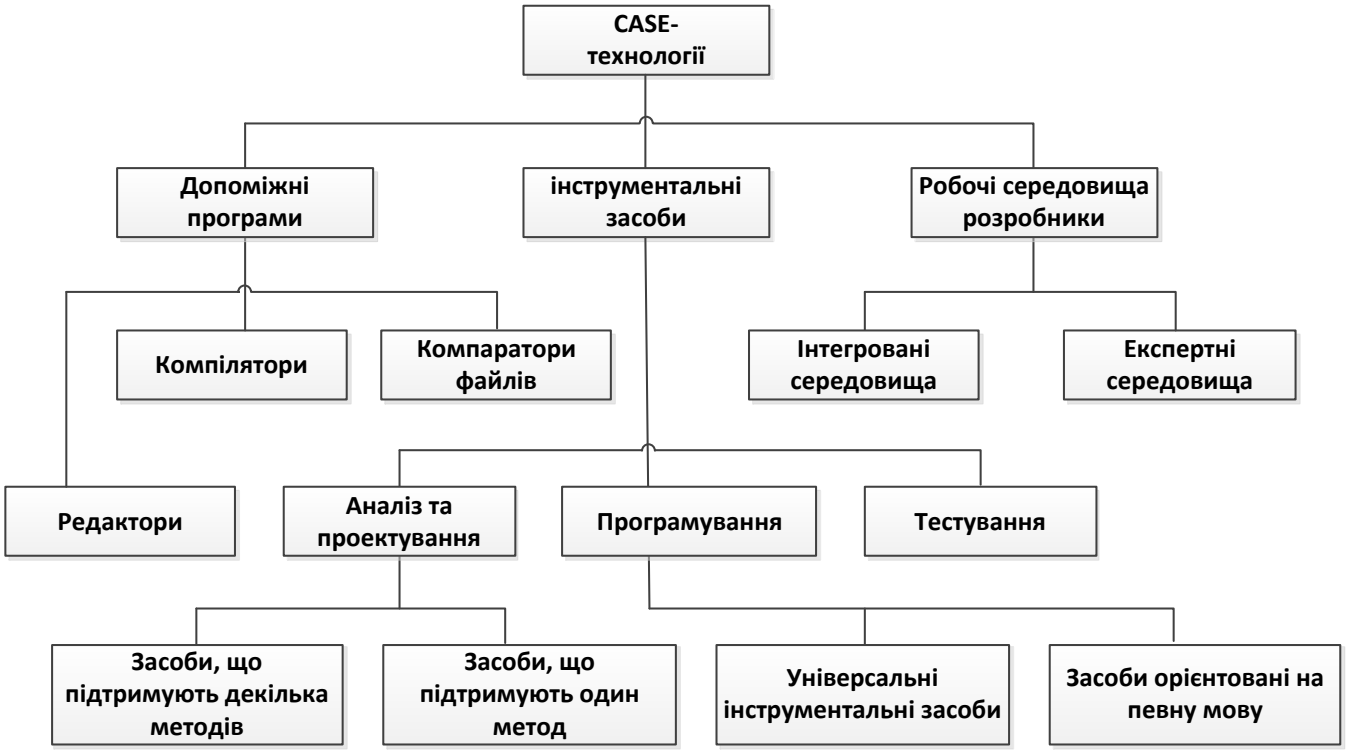

Рис. 1.14. Класифікація CASE-засобів по категоріях

На рис. 1.14 схематично представлена класифікація по категоріях із прикладами CASE-засобів різних категорій. Зрозуміло, на одній схемі неможливо показати всі типи допоміжних програм і інструментальних засобів, багато хто з них тут не представлені.

Необхідні окремі допоміжні програми вибираються розробником ПЗ зазвичай за своїм розсудом. Інструментальні засоби, як правило, підтримують визначені методи розробки відповідно до деякої моделі процесу створення ПЗ і містять набори правил і нормативних вказівок, якими слід керуватися в процесі розробки. Робочі середовища розробника можна розділити на інтегровані і експертні. Інтегровані робочі середовища надають інфраструктуру підтримки для даних, керування і інтеграції системних представлень. Експертні робочі середовища більш інтелектуальні. Вони включають базу знань про процеси створення ПЗ і механізм, який відповідно до обраної моделі процесу створення ПЗ пропонує розробнику для застосування ті або інші допоміжні програми і інструментальні засоби.

На практиці границі між CASE-засобами різних категорій розмиті. Допоміжну програму можна придбати як окремий продукт, але вона може використовуватися для підтримки різних процесів розробки. Наприклад, більшість текстових

процесорів у цей час мають у своєму розпорядженні вбудованих редактори діаграм; або інструментальні CASE-засоби для проектування все частіше пропонують підтримку процесам програмування і тестування, тим самим наближаючись до робочих середовищ. Тому не завжди можна легко позиціонувати який-небудь CASE-продукт по категоріях відповідно до цієї класифікації. Разом з тим класифікація по категоріях корисна для розуміння того, наскільки широкий діапазон процесів розробки, які можуть бути підтримані тем або іншим Case-засобом.

Лабораторна робота № 2

Тема: Вимоги до програмного забезпечення

Ціль – дати основні поняття про вимоги, пропоновані до програмних систем, і показати різні способи представлення цих вимог.

- мати поняття про концепції користувацьких і системних вимог і знати, чому для запису цих вимог використовуються різні способи;
- розуміти відмінності між функціональними і нефункціональними вимогами;
- освоїти два методи опису системних вимог: заснований на структурованій природній мові і заснований мовою програмування;
- знати стандарти документування вимог до програмного забезпечення.

Завдання:

1. Автоматизована система продажу залізничних квитків. Користувач указує пункт призначення, вставляє кредитну картку і вводить особистий ідентифікаційний номер. Система видає проїзний квиток і знімає із кредитної картки суму, рівну вартості проїзду в зазначений пункт. Коли користувач натискає початкову кнопку, відображається меню можливих пунктів призначення із вказівкою вартості проїзду до кожного пункту. Після вибору пункту призначення користувачеві пропонується вставити кредитну картку. Потім перевіряється кредитна картка, після чого користувачеві пропонується ввести особистий ідентифікаційний номер. По закінченню операцій із кредитною карткою видається проїзний квиток.

2. Перепишіть вимогу попереднього пункту, використовуючи структурний підхід.

3. Запишіть системні вимоги для системи продажу квитків за допомогою мови, заснованого мовою програмування Java. Зверніть увагу на обробку помилок користувача.

Питання:

1. Опишіть три типи нефункціональних вимог, які можуть мати місце в програмних системах. Приведіть приклади кожного типу вимог.
2. Запишіть нефункціональні вимоги для описаної вище автоматизованої системи продажу квитків системи, що характеризують надійність, і час її відповіді.
3. Якими властивостями повинна володіти мова програмування, щоб її можна було застосувати для написання специфікацій інтерфейсів? У цьому аспекті розглянете можливості мов C, Java.
4. Поясніть, як у специфікації системних вимог можна простежити взаємозв'язок між функціональними і нефункціональними вимогами.

Короткі теоретичні відомості:

Проблеми, які доводиться вирішувати фахівцям у процесі створення програмного забезпечення, зазвичай дуже складні. Природа цих проблем не завжди ясна, особливо якщо розроблювальна програмна система інноваційна. Зокрема, важко чітко описати ті дії, які повинна виконувати система. Опис функціональних можливостей і обмежень, що накладаються на програмну систему, називається *вимогами* до цієї системи, а сам процес формування, аналізу, документування і перевірки цих функціональних можливостей і обмежень – *розробкою вимог* (requirements engineering).

Термін *вимоги* (до програмної системи) може трактуватися по-різному. У деяких випадках під вимогами розуміються високорівневі узагальнені твердження про функціональні можливості і обмеженнях системи. Інша крайня ситуація – деталізований математичний формальний опис системних функцій. Девіс (Davis) так пояснює причини цих відмінностей.

Якщо компанія прагне виграти контракт на розробку великого програмного проекту, вона змушена, поки рішення не прийняте, представляти вимоги в самому узагальненому вигляді, щоб, з одного боку, задовольнити вимоги замовника, а з іншого – мати можливість для маневру при конкуренції з іншими компаніями-розробниками. Після того як контракт виграний, компанія повинна представити замовнику більш докладний опис системи із вказівкою всіх виконуваних нею функцій. В обох ситуаціях надаються документи, які називаються *документованими вимогами* до системи.

Деякі проблеми, що виникають у процесі розробки вимог, породжені відсутністю чіткого розуміння відмінності між цими різними рівнями вимог. Щоб розрізнити вимоги різних рівнів, тут використовуються терміни *користувацькі вимоги* (user requirements) для позначення високорівневих узагальнених вимог і *системні вимоги* (system requirements) для деталізованого опису виконуваних системою функцій. Крім вимог цих двох рівнів, застосовується ще більш деталізований опис системи – *проектна системна специфікація* (software design specification), яка може служити мостом між етапом розробки вимог і етапом проектування системи. Три перераховані види вимог можна визначити в такий спосіб.

1. *Користувацькі вимоги* – опис природньою мовою (плюс діаграми) функцій, виконуваних системою, і обмежень, що накладаються на неї.
2. *Системні вимоги* – деталізований опис системних функцій і обмежень, які іноді називають функціональною специфікацією. Вона є основою для висновку контракту між покупцем системи і розробниками ПЗ.
3. *Проектна системна специфікація* – узагальнений опис структури програмної системи, який буде основою для більш деталізованого проектування системи і її наступної реалізації. Ця специфікація доповнює і деталізує специфікацію системних вимог.

Відмінність між користувацькими і системними вимогами показані в прикладі, представленому в табл. 3.1. Тут показано, як користувацькі вимоги можуть бути перетворені в системні.

Таблиця 3.1. Користувацькі і системні вимоги

Користувацькі вимоги
1. ПЗ повинне надати засіб доступу до зовнішніх файлів, створених в інших програмах.
Специфікація системних вимог
1.1 Користувач повинен мати можливість визначати тип зовнішніх файлів.
1.2 Для кожного типу зовнішнього файлу повинен бути відповідний засіб, застосовнийе до цього типу файлів.
1.3 Зовнішній файл кожного типу повинен бути представлений відповідною піктограмою на дисплеї користувача.
1.4 Користувачеві повинна бути надана можливість самому визначати піктограму для кожного типу зовнішніх файлів.
1.5 При виборі користувачем піктограми, що представляє зовнішній файл, до цього файлу повинен бути застосований засіб, асоційований із зовнішніми файлами даного типу.

Користувацькі вимоги пишуться для замовника ПЗ і для особи, що містить контракт на розробку програмної системи, причому вони можуть не мати детальних технічних знань по розроблювальній системі (рис. 3.1). Специфікація системних вимог призначена для керівного технічного складу компанії-розробника і для менеджерів проекту. Вона також необхідна замовникові ПЗ і субпідрядникам по розробці. Ці обидва документа також призначені для кінцевих користувачів програмної системи. Нарешті, проектна системна специфікація є документом, який орієнтований на розробників ПЗ.

Рис. 3.1. Різні типи специфікацій вимог і їх читачі

1. Функціональні і нефункціональні вимоги

Вимоги до програмної системи часто класифікуються як функціональні, нефункціональні і вимоги предметної області.

1. *Функціональні вимоги.* Це перелік сервісів, які повинна виконувати система, причому повинно бути зазначене, як система реагує на ті або інші вхідні дані, як вона поводиться в певних ситуаціях і т.д. У деяких випадках вказується, що система не повинна робити.
2. *Нефункціональні вимоги.* Описують характеристики системи і її оточення, а не поведінка системи. Тут також може бути наведений перелік обмежень, що накладаються на дії і функції, виконувані системою. Вони включають тимчасові обмеження, обмеження на процес розробки системи, стандарти і т.д.
3. *Вимоги предметної області.* Характеризують ту предметну область, де буде експлуатуватися система. Ці вимоги можуть бути функціональними і нефункціональними.

У дійсності чіткої границі між цими типами вимог не існує. Наприклад,

користувацькі вимоги, що стосуються безпеки системи, можна віднести до нефункціональних. Однак при більш детальному розгляді таку вимогу можна віднести до функціональних, оскільки вона породжує необхідність включення в систему засоби авторизації користувача. Тому, розглядаючи далі ці види вимог, ми повинні завжди пам'ятати, що дана класифікація в значній мірі штучна.

Функціональні вимоги

Ці вимоги описують поведінку системи і сервіси (функції), які вона виконує, і залежать від типу розроблювальної системи і від потреб користувачів. Якщо функціональні вимоги оформлені як користувацькі, вони, як правило, описують системи в узагальненому вигляді. На противагу цьому функціональні вимоги, оформлені як системні, описують систему максимально докладно, включаючи її вхідні і вихідні дані, виключення і т.д.

Функціональні вимоги для програмних систем можуть бути описані різними способами. Розглянемо для прикладу функціональні вимоги до бібліотечної системи університету, призначеної для замовлення книг і документів з інших бібліотек.

1. Користувач повинен мати можливість проводити пошук необхідних йому книг і документів або по всій безлічі доступних каталожних баз даних або за певною їх підмножиною.
2. Система повинна надавати користувачеві підходящий засіб перегляду бібліотечних документів.
3. Кожне замовлення повинен бути забезпечений унікальним ідентифікатором (ORDER_ID), який копіюється у формуляр користувача для постійного зберігання.

Ці функціональні користувацькі вимоги визначають властивості, які повинна мати система. Вони взяті з документа, що містить користувацькі вимоги, і показують, що функціональні вимоги можуть бути описані з різним рівнем деталізації (перша і третя вимоги).

Багато проблем, що виникають при розробці систем, пов'язані з неточністю і “розмитістю” специфікації вимог. Звичайно, розробники інтерпретують вимоги, що

допускають двояке тлумачення, так, щоб систему було простіше реалізувати. Але це тлумачення може не збігатися з очікуваннями замовника. Така ситуація приводить до розробки нових вимог і внесенню змін у систему. Це, у свою чергу, веде до затримки здачі готової системи і її подорожчанню.

Розглянемо другу вимогу до бібліотечної системи з наведеного вище списку і звернемо увагу на вираження “підходящий засіб перегляду документів”. Бібліотечна система може надавати документи в широкому спектрі форматів. У вимозі мається на увазі, що система повинна надати засоби для перегляду документів у будь-якому форматі. Але оскільки ця умова чітко не виписана, розробники у випадку дефіциту часу можуть використовувати простий засіб для перегляду текстових документів і наполягати на тому, що саме таке рішення випливає з даного вимоги.

У принципі специфікація функціональних вимог повинна бути комплексною і несуперечливою. Комплексність має на увазі опис (визначення) усіх системних сервісів. Несуперечність означає відсутність несумісних і взаємовиключних визначень сервісів. На практиці для великих і складних систем украй важко розробити комплексну і несуперечливу специфікацію функціональних вимог. Причина криється частково в складності самої розроблювальної системи, а частково – у неузгоджених опорних точках зору на те, що повинна робити система. Ця непогодженість може не виявитися на етапі первісного формулювання вимог – для її виявлення необхідний більш глибокий аналіз специфікації. Коли непогодженість системних функцій виявиться на якому-небудь етапі життєвого циклу програми, у системну специфікацію прийде внести відповідні зміни.

Нефункціональні вимоги

Як випливає з назви, нефункціональні вимоги не зв'язані безпосередньо з функціями, виконуваними системою. Вони пов'язані з такими інтеграційними властивостями системи, як надійність, час відповіді або розмір системи. Крім того, нефункціональні вимоги можуть визначати обмеження на систему, наприклад на пропускну здатність пристроїв введення-виведення, або формати даних, використовуваних у системному інтерфейсі.

Багато нефункціональних вимог відносяться до системи в цілому, а не до

окремих її засобів. Це означає, що вони більш значимі і критичні, чим окремі функціональні вимоги. Помилка, допущена у функціональній вимозі, може знизити якість системи, помилка в нефункціональних вимогах може зробити систему непрацездатною.

Разом з тим нефункціональні вимоги можуть відноситись не тільки до самої програмної системи: одні можуть відноситись до технологічного процесу створення ПЗ, інші – містити перелік стандартів якості, що накладаються на процес розробки. Крім того, у специфікації нефункціональних вимог може бути зазначене, що проектування системи повинне виконуватися тільки певними CASE-засобами, і наведений опис процесу проектування, якому необхідно випливати.

Нефункціональні вимоги відображають користувацькі потреби: при цьому вони ґрунтуються на бюджетних обмеженнях, ураховують організаційні можливості компанії-розробника і можливість взаємодії розроблювальної системи з іншими програмними і обчислювальними системами, а також такі зовнішні фактори, як правила техніки безпеки, законодавство про захист інтелектуальної власності і т.п. На рис. 3.2 показана класифікація нефункціональних вимог.

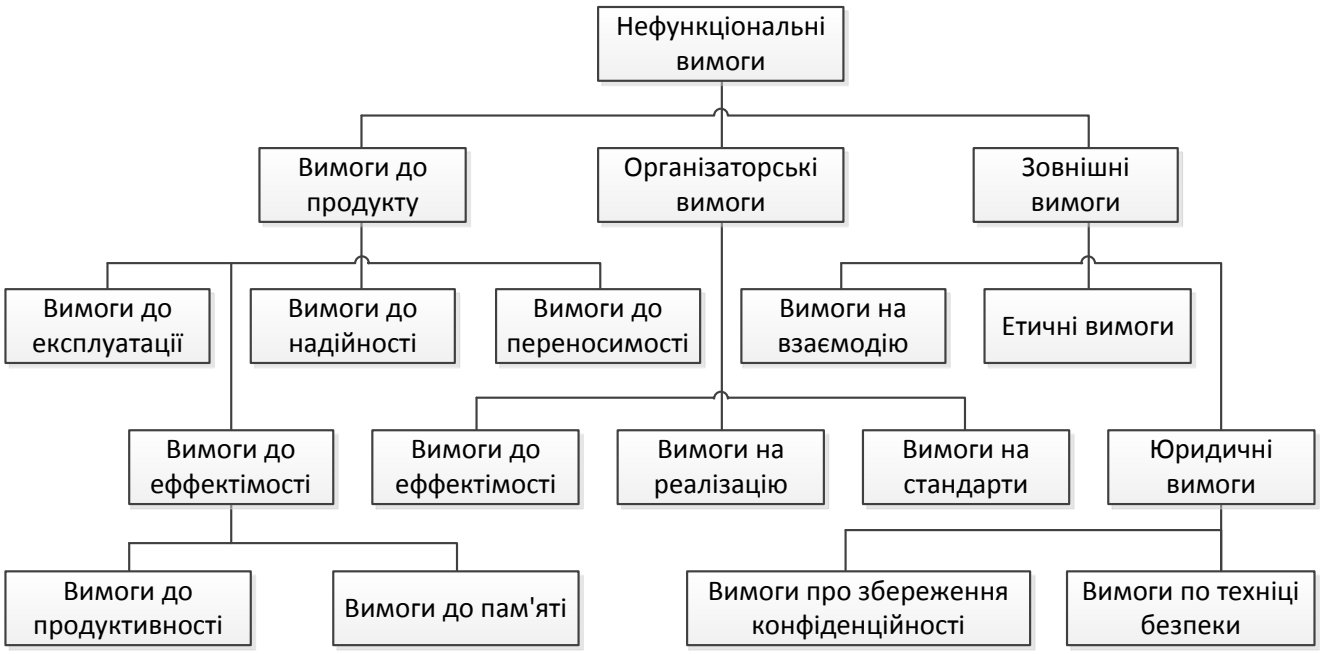

Рис. 3.2. Типи нефункціональних вимог

Усі нефункціональні вимоги, показані на рис. 3.2, можна розбити на три великі групи.

1. *Вимоги до продукту.* Описують експлуатаційні властивості програмного продукту. Сюди відносяться вимоги до продуктивності системи, обсягу необхідної пам'яті, надійності (визначає частоту можливих збоїв у системі), переносимості системи на різні комп'ютерні платформи і зручності експлуатації.
2. *Організаційні вимоги.* Відображають політику і організаційні процедури замовника і розробника ПЗ. Вони включають стандарти розробки програмного продукту, вимоги до реалізації ПЗ (тобто до мови програмування і методам проектування), вихідні вимоги, які визначають строки виготовлення програмного продукту документацію, що і супроводжує.
3. *Зовнішні вимоги.* Ураховують фактори, зовнішні відносно розроблювальної системи і процесу її розробки. Вони включають вимоги, що визначають взаємодію даної системи з іншими системами, юридичні вимоги, слідування яким гарантує, що система буде розроблятися і функціонувати в рамках існуючого законодавства, а також етичні вимоги. Останні повинні гарантувати, що система буде прийнятною для користувачів або замовника.

Приклад (*3.1*) вимог до продукту, організаційних і зовнішніх вимог. Вимоги до продукту зв'язані із середовищем програмування APSE для мови ADA. Це обмежує свободу проектувальника системи у виборі символів – можна використовувати тільки символи з користувацького інтерфейсу APSE. Організаційні вимоги вказують, що система повинна розроблятися згідно із внутрішнім стандартом компанії на розробку ПЗ, що має код XYZCo-SP-STAN-95. Зовнішні вимоги випливають із необхідності дотримання законодавства про збереження конфіденційності. Внаслідок цього системні оператори не будуть мати доступ до тих даних, які їм не потрібні для роботи із системою.

Приклад (3.1.) нефункціональних вимог

Вимоги до продукту

Усі взаємодії між інтерфейсом APSE і користувачем здійснюються на основі стандартної безлічі символів мови ADA.

Організаційні вимоги

Розробка системи і створення супутньої документації виконуються на основі стандарту XYZCo-SP-STAN-95.

Зовнішні вимоги

Система не повинна розкривати конфіденційної інформації про замовника системи, крім його імені, а також телефонного номера системних операторів.

Основна проблема нефункціональних вимог полягає в тому, що їх виконання важко перевірити. Часто вони пишуться для того, щоб відобразити загальні цілі замовника системи, такі, як простота експлуатації, можливість відновлення після збоїв або швидка відповідь на запити користувача. Реалізація подібних вимог може виявитися складною для системних розробників, оскільки вони нечітко сформульовані і відкривають простір для різних тлумачень. Подібну ситуацію ілюструє приклад, наведений в прикладі 3.2. Тут одним з основних показників (цілей) системи зазначена простота експлуатації, що у вигляді нефункціональних вимог можна виразити різними способами. У цьому випадку вимога сформульована так, що її можна перевірити.

Приклад 3.2. Системні цілі і перевірка вимог. Системна мета

Система повинна бути простою в експлуатації для досвідченого оператора і зводити кількість його помилок до мінімуму.

Нефункціональна вимога, що перевіряється

Досвідченому операторові повинні бути доступні всі системні функції після

двох годин навчання роботі з даною системою. Після такого навчання середнє число помилок оператора не повинне перевищувати двох за робочий день.

В ідеалі нефункціональні вимоги повинні виражатися через кількісні показники, які можна об'єктивно виміряти. У табл. 3.2 наведені показники, за допомогою яких можна специфікувати нефункціональні системні властивості.

На практиці виразити нефункціональні вимоги за допомогою кількісних показників досить важко. Часто замовник ПЗ не може оформити своє бачення майбутньої системи за допомогою вимог, виражених кількісними показниками. Або деякі системні вимоги, наприклад зручність супроводу, взагалі не можна виразити через кількісні показники. Крім того, витрати на об'єктивний вимір кількісних нефункціональних вимог можуть виявитися вкрай високими. Тому часто документ, що специфікує вимоги до системи, містить опис системних цілей разом із чітко сформульованими вимогами. Ці системні цілі корисні, оскільки відображають представлення (і пріоритети) замовника про майбутню систему. Разом з тим замовник повинен розуміти, що його системні цілі можуть трактуватися різними способами і їх неможливо об'єктивно проконтролювати.

Таблиця 3.2. Кількісні показники для нефункціональних вимог

Показник	**Одиниці виміру**
Швидкість	Кількість виконаних транзакцій у секунду; час реакції на дії користувача; час відновлення екрана
Розмір	Кілобайти; кількість модулів пам'яті
Простота експлуатації	Час навчання персоналу; кількість статей у довідковій системі
Надійність	Середня тривалість часу між двома послідовними проявами помилок у системі; імовірність виходу системи з ладу;

	коефіцієнт готовності системи
Стійкість до збоїв	Час відновлення системи після збою; відсоток подій, що приводять до збоїв; імовірність псування даних при збоях
Переносимість	Відсоток машинно-залежних операторів; кількість машинно-залежних підсистем

Нефункціональні вимоги часто вступають у конфлікт із іншими вимогами, пропонованими системі. Наприклад, відповідно до одного із системних вимог розмір системи не повинен перевищувати, наприклад 4 Мбайт, оскільки вона повинна повністю поміститися в постійний запам'ятовувальний пристрій обмеженої ємності. Інша вимога зобов'язує використовувати для написання системи мову програмування ADA, яка часто застосовується для створення критичних систем реального часу. Але, допустимо, відкомпільована системна програма, написана мовою ADA, займає більше 4 Мбайт. Отже, одночасне виконання цих вимог неможливо. У цій ситуації слід відмовитися від одної з вимог. Можна або застосувати іншу мова програмування, або збільшити обсяг пам'яті, що виділяється для системи.

У принципі функціональні і нефункціональні вимоги в документі, що описує вимоги до системи, повинні бути рознесені по різних розділах. Але на практиці цю умову виконати непросто. Якщо нефункціональні вимоги помістити окремо від функціональних, буде важко простежити взаємозв'язки між ними. Якщо всі вимоги зібрані в одному списку, складно провести аналіз функціональних і нефункціональних вимог окремо і визначити вимоги, що відносяться до системи в цілому. Вид представлення вимог в одному документі також суттєво залежить від типу специфіцируємої системи. Але в кожному разі повинні бути виділені вимоги, що описують інтеграційні властивості системи. Для цього їх можна помістити в окремий розділ або яким-небудь іншим способом відокремити від інших вимог.

Вимоги предметної області

Ці вимоги відображають умови, у яких буде експлуатуватися програмна система. Вони можуть бути представлені у вигляді нових функціональних вимог, у вигляді обмежень на вже сформульовані функціональні вимоги або у вигляді вказівок, як система повинна виконувати обчислення. Ці вимоги дуже важливі, оскільки відображають ту предметну область, де буде використовуватися дана система. Невиконання вимог предметної області може привести до виходу системи з ладу.

Як приклад розглянемо вимоги до бібліотечної системи.

1. Стандартний користувацький інтерфейс, що надає доступ до всіх бібліотечних баз даних, повинен ґрунтуватися на стандарті Z39.50.
2. Для забезпечення авторських прав деякі документи повинні бути вилучені із системи відразу після одержання. Для цього, залежно від бажання користувача, ці документи можуть бути роздруковані або на локальному системному сервері, або на мережному принтері.

Перша вимога є обмеженням на системну функціональну вимогу. Воно вказує, що користувацький інтерфейс до баз даних повинен бути реалізований згідно з відповідним до бібліотечного стандарту. Друга вимога є зовнішньою і спрямована на виконання закону про авторські права, застосовуваного до бібліотечних матеріалів. Із цієї вимоги випливає, що система повинна мати засіб “видалити_на_друк”, що застосовується автоматично для деяких типів бібліотечних документів.

В прикладі 3.3 наведений приклад вимоги предметної області, що вказує, як повинні виконуватися обчислення. Воно взяте зі специфікації системи автоматичного гальмування поїзда. Ця система повинна автоматично зупиняти поїзд на червоний сигнал семафора. Дана вимога вказує спосіб обчислення швидкості поїзда при гальмуванні. Тут використана термінологія, застосовувана при розрахунках швидкостей поїзда. Щоб розібратися в ній, необхідні відповідні знання про системи керування поїздами і їх характеристиках.

Приклад 3.3. Приклад вимог предметної області

Гальмування поїзда обчислюється по формулі

$D_{поїзд} = D_{керування} + D_{градієнт}$

де $D_{градієнт}$ *рівний 9.81* $м \cdot с^2$ ** компенсуючий градієнт/альфа. Значення 9.81* $м \cdot с^2$*/альфа відомо для всіх типів поїздів.*

Наведений приклад показує основну проблему, пов'язану з вимогами предметної області. Вимоги цього типу використовують мову і позначення, властиві даної предметної області, що ускладнює їхнє розуміння розробниками ПЗ. Внаслідок цієї вимоги предметної області не завжди виконуються так, як мається на увазі замовниками програмної системи.

2. Користувацькі вимоги

Користувацькі вимоги до системи повинні описувати функціональні і нефункціональні системні вимоги так, щоб вони були зрозумілі навіть користувачеві, що не має спеціальних технічних знань. Ці вимоги повинні визначати тільки зовнішню поведінку системи, уникаючи по можливості визначення структурних характеристик системи. Користувацькі вимоги повинні бути написані природньою мовою з використанням простих таблиць, а також наочних і зрозумілих діаграм.

Разом з тим при описі вимог природньою мовою можуть виникнути різні проблеми.

1. *Відсутність чіткості викладу.* Іноді нелегко викласти яку-небудь думку природньою мовою чітко і недвозначно, не зробивши при цьому текст багатослівним і важкочитаним.
2. *Змішання вимог.* У користувацьких вимогах відсутній чіткий поділ на функціональні і нефункціональні вимоги, на системні цілі і проектну інформацію.

3. *Об'єднання вимог.* Кілька різних вимог до системи можуть описуватися як єдина користувацька вимога.

У якості ілюстрації до описаних проблем розглянемо вимогу до середовища програмування мовою ADA, представлену в прикладі 3.4. Ця вимога містить опис як загального плану, так і деталізований. З інформації, що втримується в описі загального плану, випливає, що засоби керування конфігурацією є складовою частиною інтерфейсу APSE, у той час як з більш деталізованого опису випливає, що засоби керування конфігурацією повинні надавати доступ до об'єктів, що входять до складу груп, без вказівки їх повних імен. Цю інформацію краще помістити в специфікацію системних вимог.

Приклад 3.4. Вимога до бази даних для середовища програмування Ada

4.A.5. База даних повинна підтримувати генерацію і керування конфігурацією об'єктів; у базі даних згруповані об'єкти можуть виступати у вигляді окремих об'єктів. Засіб керування конфігурацією повинен надати можливість доступу до об'єктів, що входять до складу груп, за допомогою їх неповних імен.

У документі, що містить вимоги до системи, бажано відокремлювати користувацькі вимоги від більш деталізованих системних вимог. Інакше непідготовлений читач користувацьких вимог може “потонути” у технічних подробицях, розуміння яких вимагає певних професійних знань. Приклад змішання різних вимог демонструє приклад 3.5. Він представляє вимогу до CASE-засобів для редагування схем структур програмних систем. У цій вимозі мова йде про можливість відображення або приховання сітки, що допомагає користувачеві точно позиціонувати структурні елементи схеми.

Приклад 3.5. Приклад користувацької вимоги

Для тонного позиціонування структурних елементів схеми користувач може відобразити на екрані сітку, параметри якої можуть задаватися (у сантиметрах

або дюймах) за допомогою спеціальної опції на панелі керування. За замовчуванням сітка не відображається. Сітка може бути виведена на екран або схована в будь-який момент сесії редагування, також у будь-який момент є можливість переходу із сантиметрів на дюйми і назад. Крок сітки повинен підганятися під розмір схеми.

У цій вимозі переплітається не менш трьох різних вимог.

1. Концептуальна функціональна вимога: система редагування повинна мати у своєму розпорядженні можливість відображення сітки. Це основна причина появи даного вимоги.
2. Нефункціональна вимога, що дає докладну інформацію про те, у яких одиницях буде вимірятися крок сітки (сантиметри або дюйми).
3. Нефункціональна користувацька вимога, що відноситься до інтерфейсу: як користувач може відобразити або сховати сітку.

Описана вимога містить і іншу інформацію, зокрема необхідну для ініціалізації системи. У вимозі сказано, що за замовчуванням сітка відключена. Разом з тим нічого не сказане про те, які одиниці виміру обрані за замовчуванням. Далі сказано, що користувач може перемикатися між сантиметрами і дюймами, але не сказано, що він може міняти крок сітки.

Коли користувацька вимога містить так багато інформації, це ускладнює його розуміння і обмежує волю розробника в пошуку розв'язку завдання, поставленої у вимозі. Користувацькі вимоги повинні просто описувати основні можливості системи. В приклада 3.6 показана користувацька вимога, де сфокусована увага тільки на самому засобі відображення сітки без деталізації його властивостей.

Приклад 3.6. Опис засобу відображення сітки

Засіб відображення сітки

Редактор повинен мати засіб виводу на екран сітки, яка складається з паралельних горизонтальних і вертикальних ліній і повинна відображатися у вигляді фону на екрані редактора. *Сітка – пасивний елемент, що полегшує*

вирівнювання користувачем структурних елементів схем.

Обґрунтування. *Сітка повинна допомогти користувачеві створити акуратну схему із правильно розміщеними елементами. Хоча активна сітка (така, у якій елементи "прив'язуються" до вузлів сітки) також може бути корисною, вона не забезпечує точного позиціонування елементів. Користувач визначить положення елементів схеми краще, чим це зробить автоматизований засіб.*

Специфікація: *Екліпс/АРМ/Засоби/DE/FS.*

Зверніть увагу на обґрунтування вимоги. Воно допомагає розробникам зрозуміти, чому ця вимога включена в специфікацію і у якому ступені воно може змінитися в майбутньому. Наприклад, в обґрунтуванні вимоги на засіб відображення сітки сказано, що також може бути корисною активна сітка, де елементи схеми автоматично "прив'язуються" до вузлів сітки. Однак тут перевага свідомо віддана пасивній сітці. Якщо надалі виникне необхідність внести зміни в дану вимогу, то із цього обґрунтування буде видно, що варіант пасивної сітки обраний навмисно, а не з'явився на етапі реалізації системи.

Наступний приклад вимоги, що також відноситься до системи редагування схеми структури ПЗ, показаний в прикладі 3.7. Це деталізована специфікація системної функції. У цьому випадку вимога включає список дій, які повинен виконати користувач для реалізації даної функції; іноді необхідно записати подібну послідовність дій, оскільки деякі функції повинні виконуватися тільки строго визначеним способом. Інформація про те, як реалізується дана функція, у цій вимозі відсутня.

Приклад 3.7. Користувацька вимога по створенню структурних елементів схеми

1. ***Додавання структурних елементів у схему***
2. ***Редактор повинен мати засіб, що надає користувачеві можливість додавати в схему нові структурні елементи обраного типу***
3. *Послідовність дій користувача для додавання в схему нового структурного елемента.*
   - *Користувач вибирає тип елемента, що додається.*
   - *Користувач поміщає курсор у потрібну позицію на схемі і указує, яким символом буде відображатися новий елемент.*
   - *Користувач переміщає символ елемента в кінцеву позицію.*

Обґрунтування. *Такий підхід до реалізації функції додавання нових структурних елементів надає користувачеві безпосередній контроль над вибором типу елемента і його позиціонуванням на схемі.*

Специфікація: *Екліпс/АРМ/Засоби/DE/FS.*

Щоб звести до мінімуму неясності при написанні користувацьких вимог, рекомендовано дотримуватися наведених нижче правил.

1. Розробіть стандартну форму для запису користувацьких вимог і неухильно її дотримуйтеся. Стандартна форма запису зменшує неясності у формулюванні вимог і дозволяє легко їх перевірити. Рекомендовано включати у форму запису вимоги не тільки саме його формулювання, але його обґрунтування і посилання на більш деталізовану специфікацію вимог.
2. Робіть відмінність між обов'язковими і описовими вимогами, як показано в прикладі 3.7. Тут обов'язковою вимогою є наявність засобу додавання нових структурних елементів, описовим – опис послідовності дій користувача. Описова вимога не є абсолютно необхідною для реалізації даної користувацької вимоги і при необхідності може бути змінена.
3. Використовуйте різні накреслення шрифту (напівжирне і курсив) для

виділення ключових частин вимоги.

4. Уникайте по можливості комп'ютерного жаргону. Це не виключає використання технічних термінів тієї предметної області, для якої розробляється програмне забезпечення.

3. Системні вимоги

Системні вимоги – це більш деталізований опис користувацьких вимог. Вони зазвичай є основою для укладення контракту на розробку програмної системи і тому повинні представляти максимально повну специфікацію системи в цілому. Системні вимоги також використовуються в якості відправної точки на етапі проектування системи.

Специфікація системних вимог може будуватися на основі різних системних моделей, таких, як об'єктна модель або модель потоків даних. Різні моделі, використовувані при розробці специфікації системних вимог.

У принципі системні вимоги визначають, що повинна робити система, не показуючи при цьому механізму її реалізації. Але, з іншого боку, для повного опису системи потрібна деталізована інформація про неї, яка по можливості повинна включати всю інформацію про системну архітектуру. На те існує ряд причин.

1. Первісна архітектура системи допомагає структурувати специфікацію вимог. Системні вимоги повинні описувати підсистеми, з яких складається розроблювальна система.
2. У більшості випадків розроблювальна система повинна взаємодіяти із уже існуючими системами. Це накладає певні обмеження на архітектуру нової системи.
3. У якості зовнішньої системної вимоги може виступати умова використання для розроблювальної системи спеціальної архітектури.

Специфікації системних вимог часто пишуться природньою мовою. Використання природньої мови може породити певні проблеми при написанні деталізованої специфікації. Застосування природньої мови має на увазі, що ті, хто пише специфікацію, і ті, хто її читає, ті самі слова і вираження розуміють однаково.

Однак насправді це не так, оскільки природній мові властива певна розмитість понять. Внаслідок цього та сама вимога може трактуватися різними людьми по-різному.

Щоб уникнути подібних проблем, розроблені методи опису вимог, які структурують специфікацію і зменшують розмитість визначень. Ці методи представлені в табл. 3.3. Крім цього, розроблені інші підходи, наприклад спеціальні мови опису вимог, які використовуються відносно рідко.

Таблиця 3.3. Способи запису специфікацій вимог

Система запису	**Опис**
Структурована природня мова	Використання стандартних форм і шаблонів для написання специфікації
Мови опису програм	Використання спеціальних структурованих мов, подібних мовам програмування, де специфікація вимог будується на основі обраної операційної моделі системи
Графічні нотації	Графічна мова, що використовує для опису функціональних вимог діаграми і блок-схеми, доповнені текстовими поясненнями. Найбільш відомий приклад такої графічної мови – діаграми структурного аналізу і проектування ПЗ (SADT).
Математичні специфікації	Це системи нотацій, засновані на математичних концепціях, таких, як теорія кінцевих автоматів або теорія множин. Це формалізований однозначно і позбавлений двозначності запис системних вимог. Однак багато замовників ПЗ не розуміють формальних специфікацій, внаслідок чого виникають певні проблеми при укладення контрактів на розробку програмних продуктів.

Структурована мова специфікацій

Це скорочена форма природньої мови, призначена для написання специфікації вимог. Перевагою такого підходу до написання специфікацій є те, що він зберігає виразність і зрозумілість природньої мови і разом з тим формалізує опис вимог. Структурованість мови проявляється у використанні спеціальної термінології, а також шаблонів для опису системних вимог. Структурована мова може включати мовні конструкції, узяті з мов програмування.

Для написання специфікації вимог для програмної системи керування польотами розроблені спеціальні форми, що допомагають описати вхідні і вихідні дані і функції системи.

Для опису системних вимог часто розробляються спеціальні форми і шаблони. Вони повинні враховувати, на основі чого будується специфікація: на основі об'єктів, керованих системою, на основі функцій, виконуваних системою, або на основі подій, оброблюваних системою. Приклад форми для специфікації показано в прикладі 3.8. Це більш деталізований опис функції створення структурних елементів для системи редагування програмних архітектур, описаної в прикладі 3.7.

Приклад 3.8. Специфікація системної вимоги, що використовує стандартну форму

Еклiпс/АРМ/Засоби/DE/RD/3.5.1

Функція. *Додавання структурних елементів у схему.*

Опис. *Додавання структурних елементів в існуючу схему системної архітектури. Користувач вибирає тип структурного елемента і його місце розташування. Після вставки в схему структурний елемент стає виділеним (поточним структурним елементом). Користувач визначає місце розташування елемента шляхом переміщення курсору по області схеми.*

Вхідні дані. *Тип елемента, позиція елемента, ідентифікатор схеми.*

Джерела вхідних даних. *Тип елемента і позиція елемента задаються користувачем, ідентифікатор схеми отриманий з бази даних проекту.*

Вихідні дані. *Ідентифікатор схеми.*

Пункт призначення. *База даних проекту. Ідентифікатор схеми міститься в базу даних проекту по завершенню виконання даної функції.*

Для виконання функції потрібна *схема, визначена вхідним ідентифікатором схеми.* ***Передумова.*** *Схема відкрита і відображається на екрані користувача.*

Постумови. *Схема, за винятком вставки нового структурного елемента, не змінюється.*

Побічні ефекти. *Немає.*

Специфікація: *Єкліпс/АРМ/Засоби/DE/RD/3.5.1*

Стандартні форми, використовувані для специфікування функціональних вимог, повинні містити наступну інформацію.

1. Опис функції або об'єкта.
2. Опис вхідних даних і їх джерела.
3. Опис вихідних даних із вказівкою пункту їх призначення.
4. Вказівка, що необхідна для виконання функції.
5. Якщо це специфікація функції, необхідний опис попередніх умов (передумов), які повинні виконуватися перед викликом функції, і опис заключної умови (постумов), яка має бути виконане після завершення виконання функції.
6. Опис побічних ефектів (якщо вони є).

Використання структурованої мови знімає деякі проблеми, властиві специфікаціям, написаною природньою мовою, оскільки знижує “варіабельність” специфікації і більш ефективно її структурує. Разом з тим деяка “розмитість” визначень і описів у специфікації залишається. Альтернативою використанню структурованої природньої мови може служити спеціальна мова опису специфікацій, яка повністю знімає проблему нечіткості опису вимог. Але з іншого боку, неспеціаліст знайде таку специфікацію важкою для читання і розуміння.

Створення специфікацій за допомогою PDL

Для зменшення властивої природній мові нечіткості понять при описі системних вимог використовується спеціальна мова опису програм (program description language – PDL). Ця мова подібна таким мовам програмування, як Java і Ada, але більш абстрактна. Перевагою застосування PDL для створення специфікації є те, що в такій специфікації можна перевірити синтаксис і семантику існуючими програмними засобами. Ці перевірки дозволяють вилучити помилки і непогодженість в описі вимог.

Застосування PDL приводить до дуже докладних і деталізованих специфікацій, які іноді просто неможливо ввести в заключний документ із описом системних вимог. Рекомендовано використовувати PDL у наступних ситуаціях.

1. Якщо описувана операція складається з послідовності простих дій і важливий порядок їх виконання. Описати такі послідовності дій природньою мовою часом важко, оскільки їх виконання може супроводжуватися вкладеними умовами або вони можуть повторюватися циклічно. Цієї ситуації відповідає специфікація системи обслуговування банкоматів, наведена в лістингу 3.1.
2. Якщо необхідно специфікувати апаратні або програмні інтерфейси, тому що практично у всіх специфікаціях системних вимог доводиться описувати інтерфейси.

Лістинг 3.1. Специфікація системи керування банкоматами

```
class ATM {
// декларативна частина
public static void main (String args[]) throws Invalidcard{
   try {
         thiscard.read ();
         //може породити виняткову ситуацію Invalidcard
         pin = Keypad.readpin ();
         attempts = 1;
while (!thiscard.pin.eguals (pin) & attempts < 4)
   { pin = Keypad.readpin();
     Attempts = attempts + 1;
   }
   if ( !thiscard.pin.eguals (pin))
         throws new Invalidcard ("Неправильний PIN-код") ;
thisbalance = thiscard.getbalance();
do ( Screen.promt ("Будь ласка, виберіть сервіс");
```

```
        service = Screen.touchkey ();
        switch (service) {
               CASE Services.withdrawalwithreceipt:
                    receiptrequired = true;
              CASE Services.withdrawalnoreceipt:
                    amount = Keypad.readamount ();
               if (amount > thisbalance)
               { Screen.printmsg ("Рахунок перевищений");
                  break;
               }
               Dispenser.deliver (amount);
               newbalance = thisbalance - amount;
               if (receiptrequired)
                  Receipt.print (amount, newbalance);
               break;
           //далі опис інших сервісів
           default: break ;
        }
      }
         while (service != Services.quit);
         thiscard.returntouser ("Будь ласка, заберіть картку");
   }
      catch (Invalidcard e )
      { Screen.printmsg ("Недійсна картка або PIN-код") ;
   }
      //далі обробка інших виключень
    }//main ()
}//ATM
```

Разом з тим підхід до побудови специфікацій, заснований на PDL, має свої недоліки.

1. PDL, використовуваний для написання специфікації, може не мати достатніх засобів для опису всіх системних функцій.
2. Специфікації, створені за допомогою PDL, зрозумілі тільки людям, що мають певні знання мов програмування.
3. Системна архітектура, отримана на основі такої специфікації, не є системною моделлю, яка допомогла б користувачеві розібратися в структурі системи.

Найбільш ефективним способом розробки специфікацій є комбінація підходу, заснованого на PDL, з використанням структурованої природньої мови. У цьому випадку формалізовані записи природньою мовою використовуються для опису системи в цілому, а PDL – для опису послідовностей керуючих дій або для деталізованого опису інтерфейсів.

Специфікація інтерфейсів

Переважна більшість розроблювальних програмних систем повинні взаємодіяти з іншими системами, що вже існують. Для того щоб нова система могла працювати разом з іншими системами, необхідно специфікувати інтерфейси цих систем. Такі специфікації створюються на ранньому етапі розробки систем і включаються (зазвичай у вигляді додатка) у специфікацію системних вимог.

Розрізняють три типи специфіцікуємих інтерфейсів.

1. Процедурні інтерфейси, коли існуючі підсистеми пропонують набір сервісів, доступних за допомогою інтерфейсної процедури, що викликається.
2. Структури (інтерфейсні формати) даних, які пересилаються від однієї підсистеми до іншої. У цьому випадку може використовуватися PDL, заснований мовою програмування Java, який дозволяє описати класи даних з відповідними полями, що формують структуру даних.
3. Спеціальні представлення даних, наприклад у вигляді впорядкованої послідовності двійкових розрядів. Мова Java не підтримує такі детальні описи даних не рекомендується в такому разі використовувати PDL, заснований мовою Java.

Формальні нотації дозволяють описати інтерфейси чітко і недвозначно. Однак такі специфікації зрозумілі тільки фахівцям, що володіють певними знаннями. Формальні нотації рідко використовуються на практиці, хоча, вони ідеально підходять для створення специфікацій інтерфейсів. Менш формальні специфікації інтерфейсів, отримані за допомогою PDL, є компромісом між зрозумілістю і точністю опису інтерфейсів.

У лістингу 3.2 представлений приклад опису інтерфейсу першого типу (з перерахованих вище). Цей опис процедурного інтерфейсу сервера друку. Він управляє чергами на друк, що сдійснюється декількома принтерами. Користувачі можуть перевіряти чергу, відповідну до будь-якого принтера, і видаляти свої файли із черги. Вони також можуть перемикати свої файли з одного принтера на інший.

Лістинг 3.2. Опис інтерфейсу сервера друку за допомогою PDL

```
interface Printserver {
      // визначення абстрактного сервера друку
      // потрібно: інтерфейс Printer, інтерфейс Printdoc
      // надає функції: initialize (ініціалізація),
      // print (друк),
      // displayprintqueue (відображення черги на друк),
      // cancelprintjob (видалення файлу із черги),
      // switchprinter (перемикання між принтерами)

         void initialize (Printer p);
         void print (Printer p, PrintDoc d );
         void displayPrintQueue (Printer p );
         void cancelprintjob (Printer p, Printdoc d);
         void switchprinter (Printer printrinter p2,     Printdoc d);
      }//Printserver
```

Специфікація, наведена в лістингу 3.2 – абстрактна модель сервера друку без деталізації інтерфейсних частковостей. Інтерфейсні функції можна описати за допомогою структурованої природньої мови (як показано в прикладі 3.6), за допомогою PDL, заснованого мовою програмування Java (лістинг 3.1), або за допомогою формальних нотацій.

4. *Документування системних вимог*

Документ, що містить вимоги, також називаний специфікацією системних вимог – це офіційне приписання для розробників програмної системи. Він містить користувацькі вимоги і деталізований опис системних вимог. У деяких випадках користувацькі і системні вимоги можуть не різнитися, виступаючи спільно у вигляді однорідного опису системи. В інших випадках користувацькі вимоги приводяться у введенні документа-специфікації. Якщо загальна кількість вимог велика, деталізовані системні вимоги можуть бути представлені у вигляді окремого документа.

Системну специфікацію читає безліч різних людей, починаючи від вищого керівництва компанії – замовника системи і закінчуючи рядовим розробником системи. На рис. 3.3, показані категорії читачів специфікації.

Рис. 3.3. Читачі системної специфікації

Хенінгер (Heninger) сформулював шість умов, яким повинна відповідати специфікація програмної системи.

- Описувати тільки зовнішню поведінка системи.
- Указувати обмеження, що накладаються на процес реалізації системи.
- Передбачати можливість внесення змін у специфікацію.
- Служити довідковим засобом у процесі супроводу системи.
- Відображати весь життєвий цикл системи.
- Передбачати реакцію системи і групи супроводу на непередбачені (позаштатні) ситуації.

Хоча із часу написання цих умов пройшло більш 20 років, вони не втратили своєї актуальності і є гарною підмогою при розробці специфікацій. Разом з тим часом складно або навіть неможливо описати систему тільки в термінах зовнішньої поведінки, тобто тільки за допомогою функцій, які вона повинна виконувати,

оскільки в специфікації також повинні бути відбиті обмеження, що накладаються оточенням уже існуючих систем. Інша умова – охват усього життєвого циклу системи – приймається всіма розробниками, але рідко виконується на практиці.

Багато організацій, такі, як Міністерство оборони США і Інститут інженерів по електротехніці і радіоелектроніці IEEE, розробили власні стандарти документування специфікацій. Найбільш відомий стандарт розроблений IEEE і називається IEEE/ANSI 830-1993 [IEEE, 1993]. Даний стандарт передбачає наступну структуру специфікації.

1. Введення
   - Мети документа
   - Призначення програмного продукту
   - Визначення, акроніми і абревіатури
   - Список літератури і інших джерел
   - Огляд специфікації

2. Загальний опис
   - Опис програмного продукту
   - Функції програмного продукту
   - Користувацькі характеристики
   - Загальні обмеження
   - Обґрунтування, припущення і допущення

3. Специфікація вимог охоплює функціональні, нефункціональні і інтерфейсні вимоги. Це найбільш значима частина документа, але внаслідок украй широкого діапазону можливих вимог, пропонованих програмним системам, у стандарті не визначена структура цього розділу. Тут можуть бути документовані зовнішні інтерфейси, описані функціональні можливості системи, наведені вимоги, що визначають логічну структуру баз даних, обмеження, що накладаються на структуру системи, описані інтеграційні властивості системи або її якісні характеристики.

4. **Додатки**
5. **Покажчики**

Хоча стандарт IEEE не ідеальний, він може служити відправною точкою при написанні специфікації. Зазвичай, при її написанні необхідно також ураховувати стандарти, прийняті в організації – розробники ПЗ. У табл. 3.4 описані можливі розділи специфікації, побудованої на підставі стандарту IEEE.

Розділ	**Опис**
Передмова	Тут визначається коло осіб, на яких розрахований даний документ. Описуються попередні версії розроблювального програмного продукту, а також зміни, внесені в кожну версію. Дається обґрунтування для створення нової версії продукту
Введення	Тут більш розгорнуто обґрунтовується необхідність створення системи. Коротко перелічуються системні функції і пояснюється, як система буде працювати разом з іншими системами. Повинно бути показано, як розробка системи "вписується" у загальну бізнес-стратегію компанії, що замовляє програмний продукт
Глосарій	Дається опис технічних термінів, використовуваних у документі. Тут не робиться яких-небудь припущень про рівень знань або практичному досвіді читача документа
Користувацькі вимоги	Описуються сервіси, надавані користувачам, і нефункціональні системні вимоги. Цей опис може бути зроблений природньою мовою з використанням діаграм, блок-схем і інших форм запису, зрозумілих замовникові програмної системи. Тут також повинні бути наведені стандарти на програмний продукт і процес його розробки

Системна архітектура	Тут приводиться високорівневе представлення можливої системної архітектури із вказівкою, як розподілені системні функції по компонентах системи. Обов'язково повинні бути виділені повторно використовувані (тобто вже існуючі) компоненти
Системні вимоги	Докладно описуються функціональні і нефункціональні вимоги. Якщо необхідно, нефункціональні вимоги доповнюються описом інтерфейсів інших систем
Системні моделі	Тут представлено кілька системних моделей, що показують взаємовідношення між системними компонентами і між системою і її оточенням. Це можуть бути об'єктні моделі, моделі потоків даних або моделі даних
Еволюція системи	Приводяться основні припущення і допущення, на яких базується система, а також очікувані (прогнозовані) зміни в апаратних засобах, у потребах користувачів і т.п.
Додатки	Тут приводиться спеціалізована інформація, що відноситься до розроблювальної системи, наприклад опис апаратних засобів або бази даних, з якими повинна працювати система. При описі апаратних засобів необхідно показати мінімальну і оптимальну конфігурації, при яких може працювати програмна система. Опис бази даних повинен відображати логічну структуру даних, з якими буде працювати система, і відносини між ними
Покажчики	У документі можливе використання різних покажчиків. Це може бути звичайний алфавітний покажчик, покажчик діаграм або покажчик системних функцій

Зазвичай, інформація, що включається в специфікацію, залежить від типу розроблювального програмного забезпечення і від обраної технології розробки. Наприклад, якщо при розробці буде використовуватися еволюційний підхід, багато розділів специфікації, перераховані в табл. 3.4, будуть зайві. Якщо ж

розроблювальне ПЗ є тільки частиною великої системи, що полягає із взаємодіючих апаратних і програмних систем, тоді по необхідності вимоги будуть дуже докладними і деталізованими. У цьому випадку специфікація може мати значний обсяг і буде містити більше розділів, чим перераховане в табл. 3.4. Для більших документів необхідно робити зміст і докладні покажчики для пошуку необхідної інформації.

Лабораторна робота № 3

Тема: Прототипування програмних систем

Ціль – показати, як прототипування використовується в процесі розробки програмного забезпечення, і описати різні підходи до розробки прототипів.

- зрозуміти роль прототипування в процесі розробки ПЗ;
- знати відмінність між еволюційним і експериментальним прототипуванням;
- освоїти три методи розробки прототипів: з використанням мов програмування високого рівня, на основі баз даних і з повторним використанням програмних компонентів;
- зрозуміти, чому прототипування – найбільш ефективна і зручна технологія проектування і розробки користувацьких інтерфейсів.

Завдання:

1. Спроектуйте програмну систему перетворення вимог у формальну специфікацію. Прокоментуйте переваги і недоліки наступних стратегій розробки такої системи.

Питання:

1. Поясніть, чому для розробки великих систем рекомендується експериментальне прототипування.
2. У яких обставинах ви рекомендували б прототипування як засіб обґрунтування системних вимог?
3. Опишіть труднощі, які можуть виникнути при прототипуванні вбудованих комп'ютерних систем реального часу.
4. Як найбільше ефективно визначити придатні для повторного використання компонента?
5. Які переваги і недоліки використання механізму OLE для швидкої

розробки додатків?

Короткі теоретичні відомості:

Замовникам програмного забезпечення і кінцевим користувачам зазвичай складно чітко сформулювати вимоги до розроблювальної програмної системи. Важко передбачити, як система буде впливати на трудовий процес, як вона буде взаємодіяти з іншими системами і які операції, виконувані користувачами, необхідно автоматизувати. Ретельний аналіз вимог допомагає зменшити невизначеність щодо того, що система повинна робити. Однак реально перевірити вимоги, перш ніж їх затвердити, практично неможливо. У цій ситуації може допомогти прототип системи.

Прототип є початковою версією програмної системи, яка використовується для демонстрації концепцій, закладених у системі, перевірки варіантів вимог, а також пошуку проблем, які можуть виникнути як у ході розробки, так і при експлуатації системи, і можливих варіантів їх розв'язку. Дуже важлива швидка розробка прототипу системи, щоб користувачі могли почати експериментувати з ним якомога раніше.

Прототип ПЗ допомагає на двох етапах процесу розробки системних вимог.

1. ***Постановка вимог.*** Користувачі можуть експериментувати із системними прототипами, що дозволяє їм перевіряти, як буде працювати система. Користувачі одержують нові ідеї для постановки вимог, можуть визначити сильні і слабкі сторони ПЗ. У результаті можуть сформуватися нові вимоги.
2. ***Перевірка вимог.*** Прототип дозволяє виявити помилки і недогляд в раніше прийнятих вимогах. Наприклад, системні функції, визначені у вимогах, можуть бути корисними і потрібними (з погляду користувача). Однак у процесі застосування цих функцій разом з іншими функціями користувачі можуть змінити первісну думка про них. У результаті вимоги до системи зміняться, відбиваючи змінене розуміння користувачами системних функцій.

Прототипування можна використовувати при аналізі ризиків і на

початковому етапі розробки планів керування програмним проектом. Основною небезпекою при розробці ПЗ є помилки і недогляд у вимогах. Витрати на усунення помилок у вимогах на більш пізніх стадіях процесу розробки можуть бути дуже високими. Експерименти показують, що прототипування зменшує число проблем, пов'язаних з розробкою вимог. Крім того, прототипування зменшує загальну вартість розробки системи. Із цих причин воно часто використовується в процесі розробки вимог.

Однак відмінність між прототипуванням, як окремим етапом процесу розробки ПЗ, і розробкою основної програмної системи неочевидно. У цей час багато систем розробляються з використанням еволюційного підходу, коли швидко створюється первісна версія системи, яка потім поступово змінюється до її остаточного варіанта. При цьому часто використовуються методи швидкої розробки додатків, які також можна використовувати при створенні прототипів.

Поряд з тим що прототипи допомагають формувати вимоги, вони мають і інші переваги.

1. Різне тлумачення вимог розробниками ПЗ і користувачами можна виявити при демонстрації діючого прототипу системи.
2. У процесі створення прототипу розробники можуть виявити неповні або неузгоджені вимоги.
3. Працюючи, хоча і обмежено, у вигляді прототипу, система може продемонструвати свої слабкі і сильні сторони.
4. Прототип може бути основою для написання специфікації високоякісної системи. Розробка прототипу зазвичай веде до поліпшення специфікації системи.

Діючий прототип може також використовуватися для інших цілей.

1. ***Навчання користувача.*** Прототип системи можна використовувати для навчання персоналу перед поставкою остаточного варіанта системи.
2. ***Тестування системи.*** Прототипи дозволяють “прокручувати ” тести. Той самий тест запускається на прототипі і на системі. Якщо виходять однакові результати, це означає, що тест не виявив дефектів у системі. Якщо результати відрізняються, то необхідно досліджувати причини відмінності,

що дозволяє виявити можливі помилки в системі.

Ефективність застосування прототипів при розробці ПЗ полягає в наступному.

1. Поліпшуються експлуатаційні якості системи.
2. Система більше відповідає потребам користувачів.
3. Системна архітектура стає більш досконалою.
4. Супровід системи спрощується і стає більш зручним.
5. Скорочуються витрати на розробку системи.
6. Поліпшення експлуатаційних якостей системи і збільшення відповідності системи потребам користувача не вимагають збільшення загальної вартості розробки системи. Прототипування зазвичай підвищує вартість початкових етапів розробки ПЗ, але знижує витрати на більш пізніх етапах.

Модель процесу розробки прототипу показана на рис. 5.1. На першому етапі даного процесу визначаються призначення прототипу і ціль прототипування. Метою може бути розробка макета користувацького інтерфейсу, перевірка функціональних системних вимог або демонстрація реалізованості системи для керівництва. Той самий прототип не може служити одночасно всім цілям. Якщо мети визначені неточно, функції прототипу можуть бути сприйняті невірно.

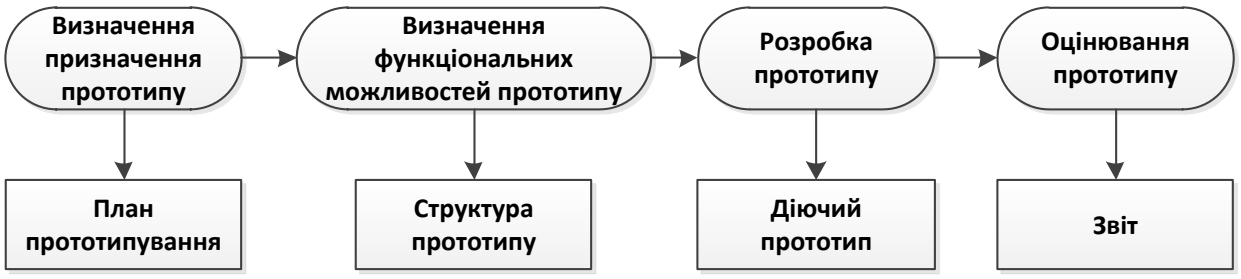

Рис. 5.1. Процес розробки прототипу

На наступному етапі процесу розробки прототипу визначаються його функціональні можливості, тобто приймається рішення про те, які властивості системи повинен відбивати прототип, а які (що, можливо, більш важливо) – ні. Для зменшення витрат на створення прототипу можна виключити деякі системні функції. Наприклад, можна послабити тимчасові характеристики і вимоги до використання пам'яті. Засоби керування і обробки помилок можуть ігноруватися або бути елементарними, якщо, зазвичай, метою прототипування не є модель інтерфейсу користувача. Також можуть бути знижені вимоги до надійності і якості програм.

Заключний етап процесу прототипування – оцінювання створеного прототипу.

1. Прототипування в процесі розробки ПЗ

Як ми вже відзначали, кінцевим користувачам важко представити, як вони будуть використовувати нову систему ПЗ в повсякденній роботі. Якщо система більша і складна, то це неможливо зробити, перш ніж система буде створена і введена в експлуатацію.

Один зі способів подолання цих труднощів полягає у використанні еволюційного методу розробки систем. Це означає, що користувачеві надається незавершена система, яка потім змінюється і доповнюється доти, поки не стануть ясні всі вимоги користувача. У якості альтернативи можна побудувати “експериментальний” прототип, який допоможе проаналізувати і перевірити вимоги. Після цього створюється система. На рис. 5.2 показано обидва підходи до використання прототипів.

Рис. 5.2. Еволюційне і експериментальне прототипування

Еволюційне прототипування починається з побудови відносно простої системи, яка реалізує найбільш важливі вимоги користувача. У міру виявлення нових вимог прототип змінюється і доповнюється. В остаточному підсумку він стає тою системою, яка потрібна. У цьому процесі не використовується детальна системна специфікація, у багатьох випадках немає навіть формального документа із системними вимогами. У цей час еволюційне прототипування є звичайною технологією розробки програмних систем, яка широко використовується при розробці Web-вузлів і додатків електронної комерції.

На противагу еволюційному підходу метод експериментального прототипування призначений для розробки і уточнення системної специфікації. Прототип створюється, оцінюється і модифікується. Дані оцінювання прототипу використовуються для подальшої деталізації специфікації. Коли системні вимоги сформовані, прототип більше не потрібний.

Існує відмінність між цілями еволюційного і експериментального прототипування.

- Метою еволюційного прототипування є поставка працюючої системи кінцевому користувачеві, це означає, що необхідно почати створення системи, що реалізує вимоги користувача, які найбільш зрозумілі і які мають найвищий пріоритет. Вимоги з більш низьким пріоритетом і нечіткі вимоги реалізуються по запитах користувачів.
- Метою експериментального прототипування є перевірка і формування системних вимог. Тут спочатку створюється прототип, що реалізує ті вимоги, які сформульовані нечітко і з якими необхідно “розібратися”. Вимоги, які сформульовані чітко і зрозуміло, не мають потреби в прототипуванні.

Інша важлива відмінність між цими підходами стосується керування якістю розроблювальної системи. Експериментальні прототипи мають дуже короткий строк життя. Вони швидко міняються і для них висока експлуатаційна надійність не потрібна. Для експериментального прототипу допускається знижена ефективність і безвідмовність. оскільки прототип повинен виконати тільки свою

основну функцію – допомогти в розумінні вимог.

На противагу цьому прототипи, які еволюціонують у закінчену систему, повинні бути розроблені з такими ж стандартами якості, що і будь-яке інше програмне забезпечення. Вони повинні мати стійку структуру і високу експлуатаційну надійність. Вони повинні бути безвідмовні, ефективні і відповідати відповідним до стандартів.

Еволюційне прототипування

В основі еволюційного прототипування лежить ідея розробки первісної версії системи, демонстрації її користувачам і наступної модифікації аж до одержання системи, що відповідає всім вимогам (рис. 5.3). Такий підхід спочатку використовувався для розробки систем, які важко або неможливо специфікувати (наприклад, систем штучного інтелекту). У цей час він стає основною методикою при розробці програмних систем. Еволюційне прототипування має багато спільного з методами швидкої розробки додатків і часто входить у ці методи як їх складова частина

Рис. 5.3. Еволюційне прототипування

Цей метод прототипування має дві основні переваги.

1. ***Прискорення розробки системи.*** Як вказувалося у введенні, сучасні темпи змін у діловій сфері вимагають швидких змін програмного забезпечення. У деяких випадках швидка поставка ПЗ, зручність і

простота його використання більш важливі, чим повний спектр функціональних можливостей системи або довгострокові можливості її супроводу.

2. ***Взаємодія користувача із системою.*** Участь користувачів у процесі розробки означає, що в системі більш повно будуть враховані користувацькі вимоги.

Між окремими методами швидкої розробки ПЗ існують відмінності, але всі вони мають деякі загальні властивості.

1. Етапи розробки технічних вимог, проектування і реалізації перемежовуються. Не існує детальної системної специфікації, проектна документація зазвичай залежить від інструментальних засобів, використовуваних для реалізації системи. Користувацькі вимоги визначають тільки найбільш важливі характеристики системи.
2. Система розробляється покроково. Кінцеві користувачі і інші особи, що формують вимоги, беруть участь на кожному кроці проектування і оцінювання нової версії системи. Вони можуть пропонувати зміни і нові вимоги, які будуть реалізовані в наступній версії системи.
3. Застосування методів швидкої розробки систем. Вони можуть використовувати інструментальні CASE-засоби і мови четвертого покоління.
4. Користувацький інтерфейс системи зазвичай створюється з використанням інтерактивних систем розробки, які дозволяють швидко спроектувати і створити інтерфейс.

Еволюційне прототипування і методи, засновані на використанні детальної системної специфікації, відрізняються підходами до верифікації і атестації систем. Верифікація – процес перевірки системи на відповідність специфікації. Оскільки для прототипу не створюється докладної специфікації, його верифікація неможлива.

Атестація системи повинна показати, що програма відповідає тем цілям,

для яких вона створювалася. Атестацію також важко провести без детальної специфікації, оскільки немає чітких формулювань цілей. Кінцеві користувачі, що брати участь у процесі розробки, можуть бути задоволені системою, у той час як інші користувачі – незадоволені, оскільки система не повністю відповідає тем цілям, які вони неявно перед нею поставили.

Верифікацію і атестацію системи, розробленої з використанням еволюційного прототипування, можна здійснити, якщо вона в достатньому ступені відповідає поставленій меті і своєму призначенню. Цю відповідність, зазвичай, не можна виміряти, можна зробити лише суб'єктивні оцінки. Такий підхід, як буде показано нижче, може породити проблеми, якщо програмна система створюється сторонніми організаціями-розробниками.

Існує три основні проблеми еволюційного прототипування, які необхідно враховувати, особливо при розробці великих систем із тривалим строком життєвого циклу.

1. ***Проблеми керування.*** Структура керування розробкою програмних систем будується відповідно до затвердженої моделі процесу створення ПЗ, де для оцінювання чергового етапу розробки використовуються спеціальні контрольні проектні елементи. Прототипи еволюціонують настільки швидко, що створювати контрольні елементи стає нерентабельно. Крім того, швидка розробка прототипу може зажадати застосування нових технологій. У цьому випадку може виникнути необхідність залучення фахівців з більш високою кваліфікацією.

2. ***Проблеми супроводу системи.*** Через безперервні зміни в прототипах змінюється також структура системи. Це означає, що система буде важка для розуміння всім, крім первісних розробників. Крім того, може застаріти спеціальна технологія швидкої розробки, яка використовувалася при створенні прототипів. Тому можуть виникнути труднощі при пошуку людей, які мають знання, необхідні для супроводу системи.

3. ***Проблеми укладення контрактів.*** Зазвичай контракт на розробку систем між замовником і розробниками ПЗ ґрунтується на системній специфікації. При відсутності такий важко скласти контракт на розробку системи. Для замовника може бути невигідний контракт, по якому доводиться просто платити розробникам за час, витрачений на розробку проекту; також малоймовірно, що розробники погодяться на контракт із фіксованою ціною, оскільки вони не можуть передбачити всі прототипи, які буде потрібно створити в процесі розробки системи.

Із цих проблем випливає, що замовники повинні розуміти, наскільки ефективно еволюційне прототипування в якості методу розробки ПЗ. Цей метод дозволяє швидко створювати системи малого і середнього розміру, при цьому вартість розробки знижується, а якість підвищується. Якщо до процесу розробки залучаються кінцеві користувачі, то, імовірно, система буде відповідати їхнім реальним потребам. Однак організації-розробники, що використовують цей метод, повинні враховувати, що життєвий цикл таких систем буде відносно короткий. При зростанні проблем із супроводом систему необхідно замінити або повністю переписати. Для великих систем, коли до розробки залучаються субпідрядники, на перший план виходять проблеми керування еволюційним прототипуванням. У цьому випадку краще застосовувати експериментальне прототипування.

Покрокова розробка (рис. 5.4) дозволяє уникнути деяких проблем, характерних для еволюційного прототипування. Загальна архітектура системи, визначена на ранньому етапі її розробки, виступає в ролі системного каркасу. Компоненти системи розробляються покроково, потім включаються в цей каркас. Якщо компоненти атестовані і включені в каркас, ні архітектура, ні компоненти вже не міняються, за винятком випадку, коли виявляються помилки.

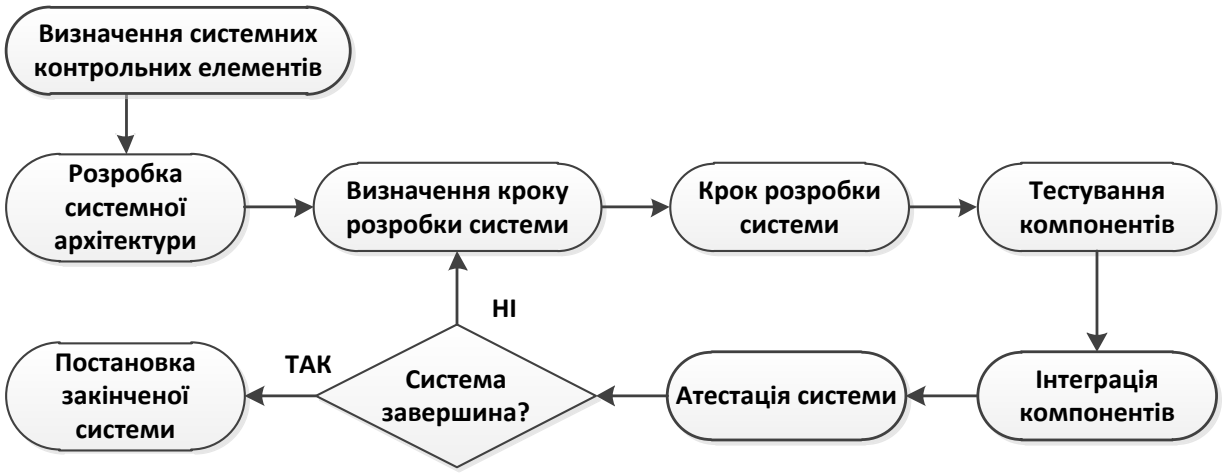

Рис. 5.4. Покроковий процес розробки

Процес покрокової розробки більш керований, чим еволюційне прототипування, оскільки додержується звичайним стандартам розробки ПЗ. Тут плани і документація створюються для кожного кроку розробки системи, що зменшує кількість помилок. Як тільки системні компоненти інтегровані в каркас, їх інтерфейси більше не змінюються.

Експериментальне прототипування

Модель процесу розробки ПЗ, заснована на експериментальному прототипуванні, показана на рис. 5.5. У цій моделі розширений етап аналізу вимог з метою зменшення загальних витрат на розробку. Основне призначення прототипу – зробити зрозумілими вимоги і надати додаткову інформацію для оцінки ризиків. Після цього прототип більше не використовується і не бере участь в подальшому процесі розробки системи.

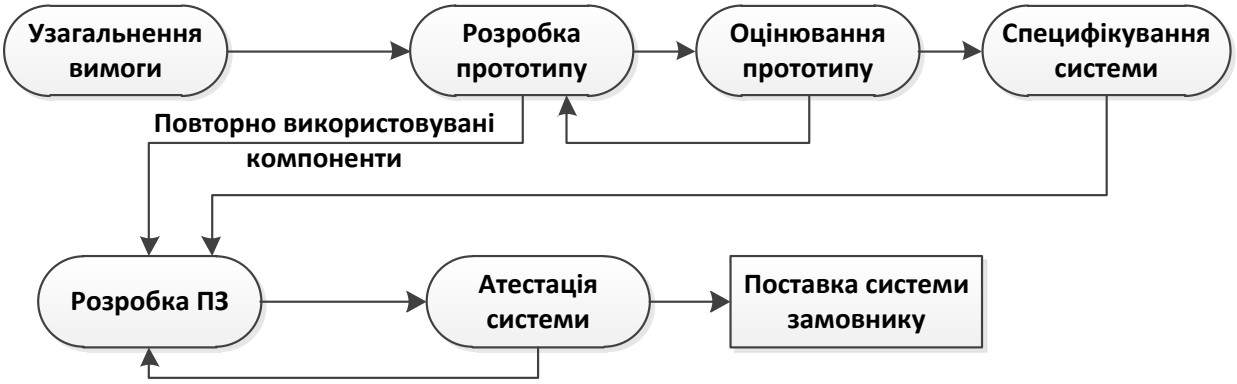

Рис. 5.5. Розробка ПЗ з використанням експериментальних прототипів

Цей метод прототипування зазвичай використовується для розробки апаратних систем. Перш ніж буде почата дорога розробка системи, створюється макет (прототип), який використовується для перевірки структури системи. Електронний макет системи створюється з використанням готових компонентів, що дозволяє розробити версію системи до того, як будуть вкладені грошові кошти в розробку спеціалізованих інтегральних схем.

Експериментальний прототип програмних систем зазвичай не використовується для перевірки архітектури системи, він допомагає розробити системні вимоги. Прототип часто зовсім не схожий на кінцеву систему. Система розробляється по можливості швидко, тому для прискорення формування вимог використовується спрощений прототип системи. Б експериментальний прототип закладаються тільки обов'язкові системні функції, стандарти якості для прототипу можуть бути знижені, критерії ефективності ігноруються. Мова програмування прототипу часто відрізняється від мови програмування, на якій буде створюватися остаточний варіант системи.

У моделі процесу розробки ПЗ, показаної на рис. 5.5, передбачається, що прототип розробляється виходячи з узагальнених системних вимог, далі над прототипом проводяться експерименти і він змінюється доти, поки його функціональні можливості не задовольнять замовника. Після цього на основі прототипу деталізуються системні вимоги, реалізується звичайна для організації-розробника технологія розробки ПЗ і система доводить до остаточної версії. Деякі компоненти прототипу можуть використовуватися в системі, тому вартість розробки може бути знижена.

В описуваній моделі розробки ПЗ основна проблема полягає в тому, що експериментальний прототип може не відповідати кінцевій системі, що поставляється замовникові. Фахівець, тестуючий прототип, може мати власні інтереси до системи, не типові для її користувачів. Час тестування прототипу може бути недостатнім для його повного оцінювання. Якщо прототип працює повільно, експерти можуть внести в нього зміни, які прискорюють роботу, але не будуть збігатися із засобами прискорення роботи кінцевої системи.

Розробники іноді зазнають тиску менеджерів для прискорення роботи над прототипом, особливо якщо намічається затримка в поставці остаточної версії системи. Зазвичай таке “прискорення” породжує ряд проблем.

1. Неможливо швидко настроїти прототип для виконання таких нефункціональних вимог, як продуктивність, захищеність, стійкість до збоїв і безвідмовність, які ігнорувалися під час розробки прототипу.
2. Часті зміни під час розробки неминуче приводять до того, що прототип погано документований. Для розробки прототипу використовується тільки специфікація системної архітектури. Цього недостатньо для довгочасного супроводу системи.
3. Зміни, зроблені під час розробки прототипу можуть порушити архітектуру системи. Її обслуговування буде складним і дорогим.
4. У процесі розробки прототипу послабляються стандарти якості.

Щоб бути корисними в процесі розробки вимог, експериментальні прототипи не обов'язково повинні виконувати роль реальних макетів систем. Паперові форми, що імітують користувацькі інтерфейси систем, показали свою ефективність при формуванні вимог користувача, в уточненні проекту інтерфейсу і при створенні сценаріїв роботи кінцевого користувача. Вони дуже дешеві в розробці і можуть бути створені за кілька днів. Розширенням цієї методики є макет користувацького інтерфейсу “Wizard of Oz” (Чарівник країни Оз). Користувачі взаємодіють із цим інтерфейсом, але їх запити спрямовані до фахівця, який інтерпретує їх і імітує відповідну реакцію.

2. Технології швидкого прототипування

Ці технології розраховані головним чином на забезпечення швидкої розробки прототипів, а не на такі їх системні характеристики, як продуктивність, зручність експлуатації або безвідмовність. Існує три основні методи швидкої розробки прототипів.

1. Розробка із застосуванням динамічних мов високого рівня.
2. Використання мов програмування баз даних.
3. Складання додатків з повторним використанням компонентів.

Для зручності ці методи описані в окремих розділах. Але на практиці вони часто спільно використовуються при розробці прототипів систем. Наприклад, мова програмування баз даних може застосовуватися для добування даних з їхньою наступною обробкою за допомогою повторно використовуваних компонентів. Інтерфейс користувача системи можна розробити, використовуючи візуальне програмування.

У цей час розробка прототипів зазвичай опирається на набір інструментів, що підтримують принаймні два із цих методів. Наприклад, система Smalltalk VisualWorks підтримує мову дуже високого рівня і забезпечує повторне використання компонентів. Пакет Lotus Notes включає підтримку програмування баз даних за допомогою мови високого рівня і повторне використання компонентів, які можуть забезпечити операції над базою даних.

Більшість систем прототипування сьогодні підтримують візуальне програмування, при якому деякі частини або весь прототип розробляються в інтерактивному режимі. Замість послідовного написання програм розробник прототипу віддає перевагу працювати із графічними піктограмами, що представляють функції, дані або компоненти інтерфейсу користувача сценаріями, що і відповідають, керування цими піктограмами. Програма, готова до виконання, генерується автоматично з візуального представлення системи. Це спрощує розробку програми і зменшує витрати на прототипування.

Застосування динамічних мов високого рівня

Динамічні мови високого рівня – це мови програмування, які мають потужні засоби контролю даних під час виконання програми. Вони спрощують розробку програм, тому що зменшують число проблем, пов'язаних з розподілом пам'яті і керуванням нею. Такі мови мають засоби, які зазвичай повинні бути побудовані з більш примітивних конструкцій у мовах, подібних Ada або C. Приклади мов дуже високого рівня - Lisp (заснований на структурах списків), Prolog (заснований на алгебрі логіки) і Smalltalk (заснований на об'єктах).

Донедавна динамічні мови високого рівня широко не використовувалися для розробки великих систем, оскільки вони потребують ґрунтовних засобів динамічної підтримки. Ці засоби збільшували обсяг необхідної пам'яті і зменшували швидкість виконання програм, написаних на цих мовах. Однак зростання потужності і зниження вартості комп'ютерного обладнання зробило ці фактори не настільки істотними.

Таким чином, для багатьох ділових додатків ці мови можуть замінити такі традиційні мови програмування, як C, COBOL і Ada. Мова Java, безсумнівно, є основною мовою розробки, що має коріння в мові C++, але із включенням багатьох засобів мови Smalltalk на зразок платформної незалежності і автоматичного керування пам'яттю. Мова Java поєднує в собі багато переваг мов високого рівня, сполучаючи це з точністю і можливістю оптимізації виконання, зазвичай пропонованої мовами третього покоління. У мові Java багато компонентів, доступних для повторного використання, усе це робить його підходящим для еволюційного прототипування.

У табл. 5.1 представлені динамічні мови, які найбільше використовуються при розробці прототипів. При виборі мови для написання прототипу необхідно відповісти на низку питань.

1. ***Який тип розроблювального додатка***? Як показано в табл. 5.1, для кожного типу додатка можна застосувати кілька різних мов. Якщо необхідний прототип додатка, який обробляє даних природньою мовою, то мови Lisp або Prolog більш підходять, чим Java або Smalltalk.

2. ***Який тип взаємодії з користувачем?*** Різні мови забезпечують різні типи взаємодії з користувачем. Деякі мови, такі як Smalltalk і Java, добре інтегруються з Web-броузерами, у той час як мова Prolog найкраще підходить для розробки текстових інтерфейсів.
3. ***Яке робоче середовище забезпечує мова?*** Розвинене робоче середовище підтримки мови зі своїми інструментальними засобами і легким доступом до повторно використовуваних компонентів спрощує процес розробки прототипу.

Таблиця 5.1. Мови високого рівня, використовувані при прототипуванні

Мова	**Тип мови**	**Тип додатка**
Smalltalk	Об'єктно-орієнтований	Інтерактивні системи
Java	Об'єктно-орієнтований	Інтерактивні системи
Prolog	Логічний	Системи обробки символьної інформації
Lisp	Заснований на списках інформації	Системи обробки символьної

Динамічні мови високого рівня для створення прототипу можна використовувати спільно, коли різні частини прототипу програмуються на різних мовах.

Не існує ідеальної мови для прототипування великих систем, оскільки зазвичай різні частини системи різнотипні. Переваги багатомовного підходу в тому, що для створення кожного компонента можна підібрати найбільш підходящу мову і у такий спосіб прискорити розробку прототипу. Недолік такого підходу в тому, що важко розробити комунікаційні зв'язки для компонентів, написаних на різнорідних мовах.

Еволюційна розробка в цей час є стандартною методикою для створення бізнес-додатків малого і середнього розміру. Більшість бізнес-додатків містять у собі систему керування базою даних і обробку даних, що перебувають у ній.

Для підтримки розробки таких додатків усі комерційні системи керування базами даних мають внутрішні засоби програмування. Програмування баз даних виконується на основі спеціалізованих мов, які мають вбудовану базу знань і засобів, необхідні для роботи з базами даних. Робоче середовище підтримки мови забезпечує інструментальні засоби для створення користувацьких інтерфейсів, числових обчислень і звітів. Термін ***мова четвертого покоління*** застосовується як до самої мови програмування баз даних, так і до його робочого середовища.

Мови четвертого покоління успішно застосовуються на практиці, оскільки більшість сучасних додатків тією чи іншою мірою займаються обробкою інформації, укладеної в базах даних. Основні операції, виконувані такими додатками – це модифікація бази даних і створення звітів на основі інформації, витягнутої з бази даних. Зазвичай для введення і виведення даних використовуються стандартні форми. Мови четвертого покоління мають засоби для створення інтерактивних додатків, що дозволяють користувачам вносити зміни в базу даних. Користувацький інтерфейс зазвичай складається з набору стандартних форм або електронної таблиці.

Зазвичай робоче середовище мов четвертого покоління включає наступні інструментальні засоби (рис. 5.6).

1. У якості мови програмування баз даних (точніше, мови запитів до бази даних) зазвичай використовується SQL.
2. Генератор інтерфейсів використовується для створення форм введення і відображення даних.
3. Електронна таблиця застосовується для аналізу даних і виконання різних дій над числовою інформацією.
4. Генератор звітів призначений для створення звітів на основі інформації, яка міститься в базі даних.

Рис. 5.6. Компоненти мови четвертого покоління

Більшість бізнес-додатків передбачають структуровані форми для введення і виведення даних, тому мови четвертого покоління забезпечують потужні засоби для визначення екранних форм і створення звітів. Екранні форми часто визначаються як ряд взаємозалежних форм, тому система, що генерує екрани, повинна забезпечувати наступне.

1. ***Інтерактивне визначення форм***, коли розробник визначає поля введення і їх організацію.
2. ***Зв'язування форм,*** коли розробник задає певні дані, введення яких викликає відображення подальших форм:
3. ***Перевірки вхідних даних,*** коли розробник при формуванні полів форм визначає дозволений діапазон вхідних величин.

У цей час більшістю мов четвертого покоління підтримується розробка інтерфейсів баз даних, заснованих на Web-броузерах. Вони роблять базу даних доступною за допомогою Internet. Це знижує вартість навчання і програмного забезпечення і дозволяє зовнішнім користувачам мати доступ до бази даних. Однак обмеження протоколів Internet і повільний перегляд Web-сторінок роблять цей метод не підходящим для систем, у яких потрібна швидка взаємодія з користувачем.

Методи, засновані на мовах четвертого покоління, можуть використовуватися для еволюційного прототипування або для генерування

“одноразового” прототипу системи. Структура, яку CASE-засоби накладають на розроблювальний додаток і супутню документацію, визначає більш зручний супровід прототипів, чим пропонують прототипи, розроблені вручну. CASE-засоби можуть генерувати код SQL або код мовою нижчого рівня, наприклад COBOL.

Хоча мови четвертого покоління підходять для розробки прототипів, все-таки вони мають ряд недоліків, що проявляються при розробці систем. Програми, написані на мовах четвертого покоління, як правило, виконуються повільніше подібних програм, написаних на звичайних мовах програмування, і вимагають набагато більше пам'яті

Незважаючи на те що застосування мов четвертого покоління знижує вартість розробки систем, загальна сума витрат за повний життєвий цикл таких систем поки не ясна. Їхні програми зазвичай погано структуровані і важкі в супроводі. Специфічні проблеми можуть виникати при модифікації подібних систем. У цей час мови четвертого покоління не стандартизовані і не уніфіковані, тому при модифікації систем, швидше за все, прийдеться переписати програми, оскільки мова, на якій вони написані, застаріє.

Складання додатків з повторним використанням компонентів

Час, необхідний для розробки системи, можна зменшити, якщо багато частин такої системи будуть використані неодноразово. Для швидкої побудови прототипу необхідно мати набір компонентів, придатних для повторного використання, і механізм складання системи із цих компонентів. Цей підхід показаний на рис. 5.7.

Рис. 5.7. Складання повторно використовуваних компонентів

Прототипування з повторно використовуваними компонентами застосовується при розробці вимог, зазвичай, якщо є підходящі компоненти. Якщо підходящих компонентів нема, то для реалізації деяких вимог буде необхідний компромісний підхід. Функціональні можливості доступних компонентів можуть не точно відповідати користувацьким вимогам. Але, з іншого боку, ці вимоги зазвичай досить гнучкі, тому в багатьох випадках можливе створення прототипу.

Розробку прототипу з повторним використанням компонентів можна реалізувати на двох рівнях.

1. *Рівень додатка*, коли цілі прикладні системи інтегруються із прототипом так, щоб були об'єднані їхні функціональні можливості. Наприклад, якщо прототипу потрібні засоби обробки тексту, то це можна забезпечити шляхом інтеграції в прототип стандартною системою текстового процесора. Відзначимо, що додатки Microsoft Office підтримують інтеграцію зі сторонніми системами.
2. *Рівень компонентів*, коли окремі компоненти поєднуються усередині структури, що реалізує систему. Така структура може бути створена за допомогою одною з мов опису сценаріїв, таких, як Visual Basic, TCL/TK, Python або Perl. У якості альтернативи можуть застосовуватися такі системи, як CORBA, DCOM або javaBeans.

Повторно використовувані додатки дають доступ до всіх своїх функціональних можливостей. Якщо, крім того, додаток забезпечує створення сценаріїв або засоби автоматизації (наприклад, макроси Excel), вони також можуть використовуватися для розширення функціональних можливостей прототипу.

Для розуміння цього методу розробки прототипу корисний складений документ, який являє собою схему обробки даних прототипом і який можна розглядати як контейнер для декількох різних об'єктів. Ці об'єкти містять різні типи даних (такі, як таблиця, діаграма, форма), які можуть оброблятися різними додатками.

На рис. 5.8 представлений складений документ для прототипу системи, що включає текстові елементи, елементи електронної таблиці і звукові файли. Текстові елементи обробляються текстовим процесором, таблиці – електронною таблицею, а звукові файли – аудіопрогравачем. Коли користувач системи звертається до об'єкта певного типу, викликається пов'язаний з ним додаток.

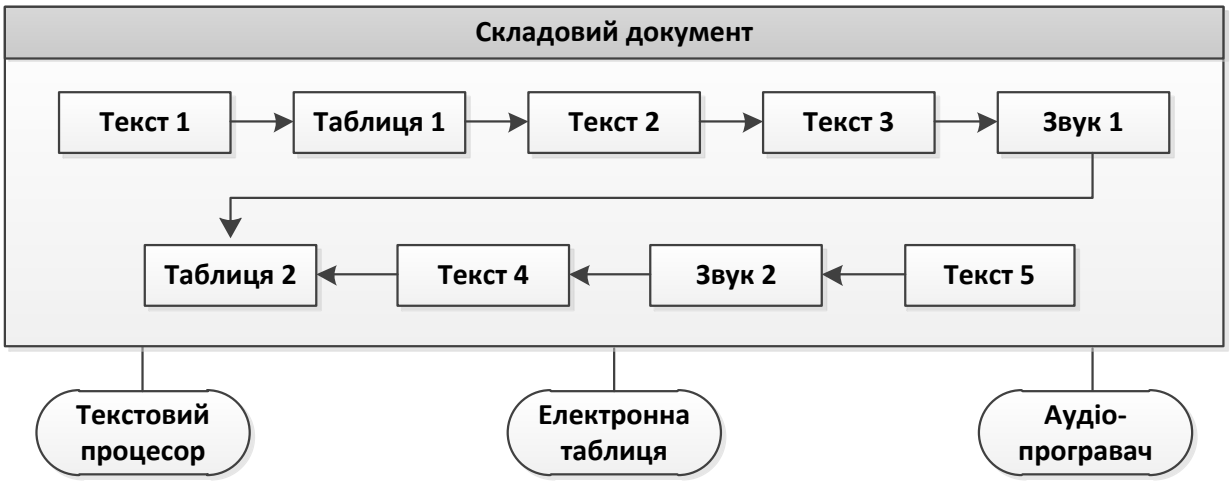

Рис. 5.8. Зв'язування додатків за допомогою складеного документа

Розглянемо прототип системи, що підтримує керування розробкою вимог. Для цієї системи необхідні засоби фіксації вимог, їх зберігання, створення звітів, пошуку залежностей між вимогами і керування цими залежностями за допомогою матриць оперативного контролю. У прототипі повинна бути база даних (для зберігання вимог), текстовий процесор (для введення вимог і створення звітів), електронна таблиця (для керування матрицями контролю) і спеціально написана програма для пошуку залежностей між вимогами.

Основна перевага описуваного підходу до прототипування полягає в тому, що багато функціональних засобів прототипу можна реалізувати швидко і дешево. Якщо користувачі, тестуючі прототип, знайомі з додатками, інтегрованими в прототип, їм немає необхідності вчитися використовувати нові засоби прототипу. Проблеми при роботі із прототипом можуть виникнути тільки при перемиканні з одного додатка на інший. Але це в значній мірі залежить від використовуваної операційної системи. Для організації перемикання між додатками найбільше широко використовується механізм зв'язування і

впровадження об'єктів OLE від Microsoft.

Не завжди можливо або зручно використовувати цілі додатки. Для створення прототипів можна використовувати більш “тонкі” компоненти. Це можуть бути окремі функції або об'єкти, які виконують спеціальні дії, наприклад сортування, пошук, відображення даних і т.д. Прототипування починається з визначення загальної структури прототипу, потім компоненти інтегруються відповідно до цієї структури. Якщо немає компонентів, що виконують функції, що вимагаються, замість них розробляються окремі програми, які в майбутньому також можна повторно використовувати.

Візуальні системи розробки додатків, подібні Visual Basic, підтримують повторне використання компонентів. Розробники будують систему в інтерактивному режимі, визначаючи інтерфейс у вигляді набору екранних форм, полів, кнопок і меню. Ці елементи інтерфейсу іменовані, і кожний з них зв'язаний зі сценарієм обробки подій і даних. Ці сценарії можуть викликати повторно використовувані програмні компоненти.

На рис. 5.9 показаний екран додатка, що містить меню (у верхній частині), поля введення (білі області ліворуч на екрані), поля виводу і кнопки, представлені округленими прямокутниками праворуч на екрані. Після розташування цих графічних елементів на екрані розробник визначає, які готові компоненти будуть пов'язані з ними, або пише програми, що виконують необхідні дії. На рис. 5.9 показані компоненти, пов'язані з деякими відображуваними елементами екрана.

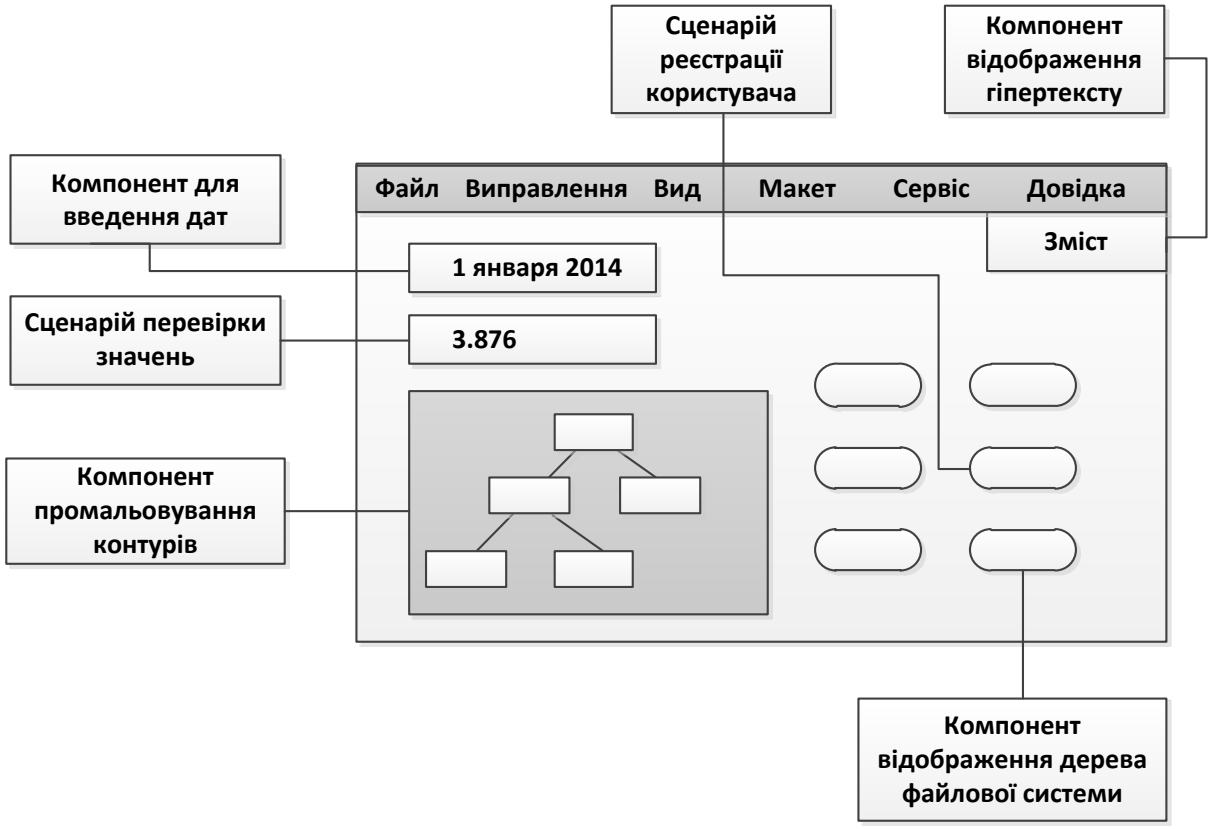

Рис. 5.9. Візуальне програмування з повторним використанням компонентів

Visual Basic – приклад сімейства мов, названих мовами сценаріїв. Мови сценаріїв – нетипові мови високого рівня, розроблені для інтеграції компонентів у єдину систему. Раннім прикладом мови сценаріїв може служити оболонка Unix, с тих пор були створені інші могутніші мови створення сценаріїв. Ці мови мають керуючі структури і графічні інструментальні засоби, що радикально зменшують час розробки систем.

Описаний підхід до розробки систем надає можливість швидко розробити відносно малі і прості додатки, які можуть бути побудовані однією особою або невеликим колективом розробників. Для великих систем, які повинні розроблятися великими колективами, подібний підхід організувати більш складно, оскільки не існує явної архітектури системи і часто є складні залежності між різними компонентами системи. У цьому причина труднощів при внесенні змін у систему. Крім того, обмежений набір компонентів, тому буває важко реалізувати нестандартні користувацькі інтерфейси. Загальний покомпонентний метод розробки є більш підходящим для більших програмних систем.

3. Прототипування користувацьких інтерфейсів

Графічні інтерфейси користувача в цей час є стандартом для інтерактивних систем. Зусилля, що вкладаються у визначення, проектування і реалізацію такого інтерфейсу, складають значну частину вартості розробки додатка. Розробники не повинні нав'язувати користувачам свою точку зору на проектований інтерфейс. Користувачі повинні брати активну участь у процесі проектування інтерфейсу. Такий погляд на розробку інтерфейсу привів до підходу, названого проектуванням, орієнтованим на користувача (user-centred design), який заснований на прототипуванні інтерфейсу і участі користувача в процесі його проектування.

У цьому аспекті прототипування – необхідна частина процесу проектування користувацького інтерфейсу. Через динамічну природу користувацьких інтерфейсів текстових описів і діаграм недостатньо для формування вимог до інтерфейсу. Тому еволюційне прототипування за участю кінцевого користувача – єдиний прийнятний спосіб розробки графічного інтерфейсу для програмних систем.

Генератори інтерфейсів - це графічні системи проектування екранних форм, де інтерфейси компонуються з елементів типу меню, полів, піктограм і кнопок, які, у свою чергу, можна просто вибрати з меню і помістити в екранну форму. Як уже згадувалося, системи цього типу – необхідна частина систем програмування баз даних. Генератори інтерфейсів створюють добре структуровану програму, згенеровану по специфікації інтерфейсу.

Мільйони людей сьогодні мають доступ до Web-броузерів. Вони підтримують мову розмітки гіпертексту HTML, який дозволяє створювати користувацькі інтерфейси. Кнопки, поля, форми і таблиці можуть бути включені до Web-сторінки так само, як і засоби мультимедіа. Сценарії обробки подій і даних, пов'язані з об'єктами інтерфейсу, можуть виконуватися або на машині Web-клієнта, або на Web-сервері.

Через широкі можливості Web-броузерів і потужності мови HTML у цей час усе більше користувацьких інтерфейсів будуються як Web-орієнтовані. Такі інтерфейси – приналежність не тільки нових систем; вони заміняють інтерфейси,

побудовані на текстових формах, у широкому колі наслідуваних систем.

Для Web-орієнтованих інтерфейсів прототипи можна створювати за допомогою стандартних редакторів Web-сторінок, які, по суті, будують користувацькі інтерфейси. Об'єкти на Web-сторінці визначаються, як і пов'язані з ними операції, за допомогою вбудованих засобів мови HTML (наприклад, зв'язування з іншою сторінкою) або за допомогою мови Java або сценаріїв.

Лабораторна робота № 4

Тема: Формальні специфікації ПЗ

Ціль – представити формальні специфікації, які можна використовувати для деталізації специфікації системних вимог.

- розуміти, чому методи формальної специфікації допомагають виявляти проблеми в системних вимогах;
- знати, як для формування вимог, що описують інтерфейс, використовуються алгебраїчні методи формальних специфікацій;
- мати представлення про те, як формальні методи можна використовувати для специфікування поведінки системи.

Короткі теоретичні відомості:

Традиційні технічні дисципліни, такі, як електротехніка або цивільне будівництво, зазвичай легко адаптують кращі математичні методи. Наприклад, у машинобудуванні не виникало труднощів із застосуванням методів математичного аналізу. Однак інженерія програмного забезпечення не йде таким шляхом. Так звані формальні методи розробки програмних систем широко не використовуються. Багато компаній, що розробляють ПЗ, не вважають економічно вигідним застосування цих методів у процесі розробки.

Термін “формальні методи” має на увазі ряд операцій, до складу яких входять створення формальної специфікації системи, аналіз і доказ специфікації, реалізація системи на основі перетворення формальної специфікації в програми і верифікація програм. Усі ці дії залежать від формальної специфікації програмного забезпечення. Формальна специфікація – це системна специфікація, записана мовою, словник, синтаксис і семантика якого визначені формально. Необхідність формального визначення мови передбачає, що ця мова ґрунтується на математичних концепціях. Тут використовується область математики, яка називається дискретною математикою і ґрунтується на алгебрі, теорії множин і

алгебрі логіки.

В 1980-х роках багато дослідників уважали, що формальні специфікації і формальні методи є найбільш ефективним шляхом поліпшення якості ПЗ. Вони привели вагомі доводи на користь того, що строгий і детальний аналіз, який є невід'ємною частиною формальних методів, приведе до створення програм з малою кількістю помилок. Вони пророкували, що до XXI сторіччю більша частина програмного забезпечення буде розроблятися з використанням формальних методів.

Тепер ясно, що ці пророкування не збулися, причому із цілого ряду причин.

1. *Успіхи інженерії програмного забезпечення.* Використання в процесі проектування і розробки програмних систем таких технологій інженерії ПЗ, як структурні методи, керування конфігурацією, приховання інформації і т.д., привело до підвищення якості програмних продуктів. Це суперечить представленням, що існували, що підвищення якості програм може бути досягнуте тільки шляхом доказу їх правильності.
2. *Зміни на ринку програмних продуктів.* В 1980-х роках якість була ключовою проблемою в створенні ПЗ. У цей час головним критерієм при розробці багатьох класів ПЗ є не якість, а час поставки його на ринок. Програмне забезпечення повинне бути розроблене швидко, і клієнти готові прийняти його з деякими дефектами, якщо буде забезпечена швидка поставка. Методи швидкої розробки ПЗ не дуже добре узгодяться з формальною специфікацією. Зазвичай, якість є важливим показником, але вона повинне досягатися в контексті зі швидкою поставкою на ринок.
3. *Обмежена область застосування формальних методів.* Формальні методи зазвичай погано підходять для опису взаємодій користувача і визначення користувацьких інтерфейсів. Користувацькі інтерфейси є складовою частиною більшості сучасних систем, тут користь від застосування формальних методів обмежена.
4. *Обмежена масштабованість формальних методів.* В успішних проектах формальні методи, як правило, використовувалися для

розробки тільки щодо невеликого критичного ядра системи. Проблема збільшується недоліком засобів підтримки таких методів.

Ці фактори привели до того, що ризики застосування формальних методів у більшості проектів ПЗ переважують можливі вигоди від їхнього використання. Витрати і проблеми введення формальних методів у процес розробки дуже високі. Однак формальна специфікація є відмінним способом виявлення помилок у вимогах і засобом, що забезпечують однозначність системної специфікації. У всіх успішних проектах, що використовували формальні методи, повідомлялося про зменшення кількості помилок у закінчених програмних системах.

Таким чином, у системах, де можливе застосування формальних методів, воно може бути виправданим і, імовірно, буде рентабельним. Використання формальних методів зростає в спеціальній області розробки критичних систем, де дуже важливі такі властивості, як безпека, безвідмовність і захищеність. Для таких систем атестація вимагає великих витрат, а вартість відмов досить велика. У цьому випадку рентабельно використовувати формальні методи, оскільки вони можуть зменшити ці витрати.

Прикладами критичних систем, при розробці яких успішно застосовувалися формальні методи, є інформаційні системи керування повітряним транспортом, системи сигналізації на залізниці, бортові системи космічних кораблів і медичні системи керування. Вони також були використані для специфікування пакетів програмних засобів, частини системи CICS (абонентська інформаційно-керуюча система) компанії IBM і ядра систем, що працюють у режимі реального часу.

Метод “чистої кімнати” розробки ПЗ, ґрунтується на формальних доказах того, що програма відповідає своїй специфікації.

1. Формальні специфікації в процесі розробки ПЗ

Визначено три рівні специфікації програмного забезпечення. Це користувацькі і системні вимоги і специфікація структури програмної системи. Користувацькі вимоги найбільш узагальнені, специфікація структури найбільш детальна. Формальні математичні специфікації перебувають десь між

системними вимогами і специфікацією структури. Вони не містять деталей реалізації системи, але повинні представляти її повну математичну модель.

У міру розробки специфікації участь замовника зменшується, а участь підрядників і безпосередньо розробників ПЗ зростає. На ранніх стадіях розробки специфікація повинна бути “орієнтована на замовника” і написана так, щоб він міг її зрозуміти. Однак на заключній стадії процесу розробки повинна бути отримана специфікація, в основному призначена для підрядників і розробників ПЗ, оскільки вона буде бути основою для реалізації системи. Ця кінцева специфікація може бути формальною.

На рис. 6.1 показані етапи розробки специфікації ПЗ і їх взаємозв'язку із процесом проектування. Етапи розробки специфікації, показані на рис. 6.1, не є незалежними і не обов'язково розробляються в наведеній послідовності. На рис. 6.2 показано, що розробка специфікації і проектування можуть виконуватися паралельно, коли інформація від етапів розробки специфікації передається до етапів проектування і навпаки.

Рис. 6.1. Розробка специфікації і проектування

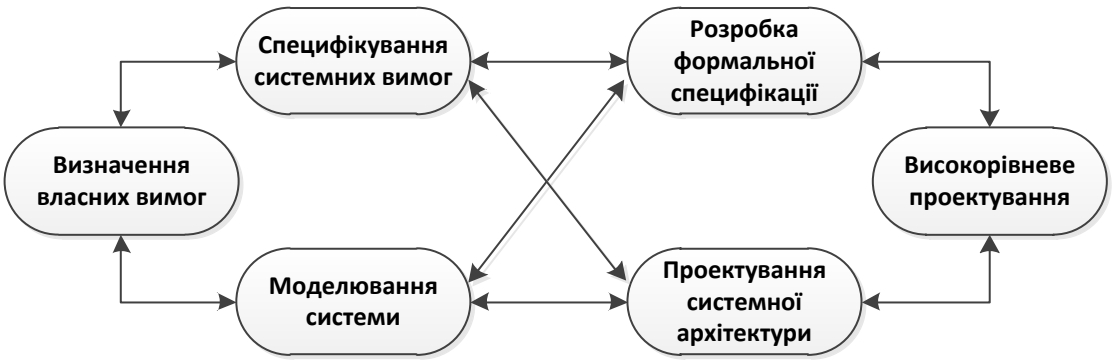

Рис. 6.2. Розробка формальної специфікації

Створення формальної специфікації вимагає детального аналізу системи, яка дозволяє виявити помилки і невідповідності в специфікації неформальних вимог. Ця можливість виявлення помилок – найбільш важливий аргумент для використання формальної специфікації. Проблеми у вимогах, які залишаються невиявленими до останніх стадій процесу розробки ПЗ, зазвичай вимагають більших витрат на виправлення.

Розробка і аналіз формальної специфікації вимагають додаткових витрат. На рис. 6.3 показана вартість створення ПЗ при розробці формальної специфікації і без неї. При звичайному процесі розробки ПЗ вартість атестації системи становить близько 50% усієї вартості розробки, а вартість проектування і реалізації системи у два рази більше вартості розробки специфікації. При використанні формальної специфікації вартості розробки специфікації і реалізації системи порівнянні, а вартість атестації значно знижується, оскільки в процесі розробки формальної специфікації виявляються і усуваються недоробки у вимогах, тим самим виключається переробка системи на останніх стадіях її створення.

Існує два основні підходи до розробки формальної специфікації, які використовуються для написання деталізованих специфікацій нетривіальних програмних систем.

1. Алгебраїчний підхід, при якому система описується в термінах операцій і їх відносин.
2. Підхід, орієнтований на моделювання, при якому модель системи будується з використанням математичних конструкцій, таких, як множини і послідовності, а системні операції визначаються тим, як вони змінюють стани системи.

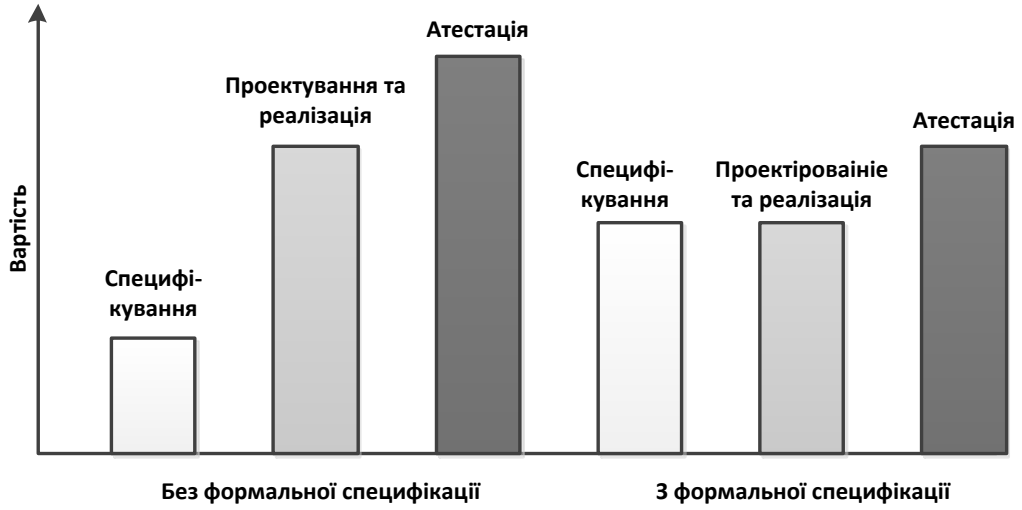

Рис. 6.3. Вартість розробки ПЗ з формальною специфікацією

Для розробки формальних специфікацій послідовних і паралельних систем у цей час створено кілька мов, представлених у табл. 6.1. Наведені далі приклади показують, як побудова формальних специфікацій приводить до точних і деталізованих специфікацій, але тут не обговорюються мови розробки специфікацій і методи специфікування.

Таблиця 6.1. Мови розробки формальних специфікацій

Тип мови	**Послідовні системи**	**Паралельні системи**
Алгебраїчний	Larch, OBJ	Lotos
Заснований на моделях	Z, VDM, B	CSP, мережі Петри

2. Специфицирование інтерфейсів

Великі системи зазвичай розбиваються на підсистеми, які розробляються незалежно друг від друга. Підсистеми можуть використовувати інші підсистеми, тому необхідною частиною процесу специфікування є визначення інтерфейсів підсистем. Якщо інтерфейси визначені і погоджені, підсистеми можна розробляти не залежно друг від друга.

Інтерфейс підсистеми часто визначається як набір абстрактних типів даних

і об'єктів (рис. 6.4), при цьому тільки через інтерфейс доступні опис даних і операції над ними. Тому специфікацію інтерфейсу підсистеми можна розглядати як об'єднання специфікацій компонентів, що в підсумку і складе опис інтерфейсу підсистеми.

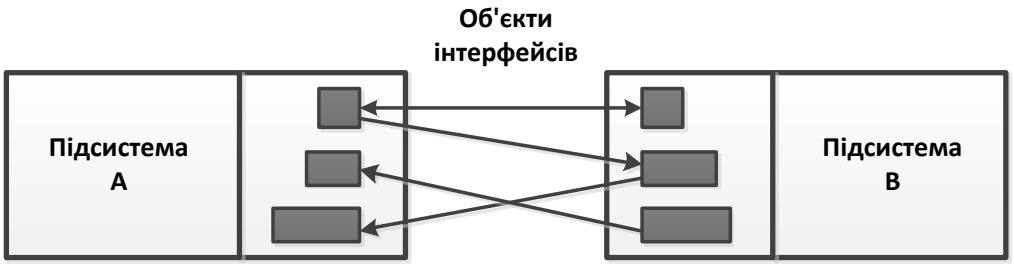

Рис. 6.4. Об'єкти інтерфейсів підсистем

Точні специфікації інтерфейсів підсистем необхідні для розробників, які пишуть програмний код, що звертається до сервісів інших підсистем. Специфікації інтерфейсів містять інформацію про те, які сервіси доступні в інших підсистемах і як одержати до них доступ. Ясний і однозначний інтерфейс підсистем зменшує ймовірність помилок у взаєминах між ними.

Алгебраїчний підхід спочатку був розроблений для опису інтерфейсів абстрактних типів даних, де типи даних визначаються скоріше специфікаціями операцій над даними, чим способом представлення самих даних. Це дуже схоже на визначення класів об'єктів. Алгебраїчний підхід до формальних специфікацій визначає абстрактний тип даних у термінах операцій над даними.

Структура специфікації об'єкта показана на рис. 6.5 і складається із чотирьох компонентів.

- Введення, де оголошується клас **(sort)** об'єктів. Клас – ця загальна назва для безлічі об'єктів. Він зазвичай реалізується як тип даних. Введення може також включати оголошення імпорту **(imports),** де вказуються імена

специфікацій, що визначають інші класи. Імпортування специфікацій робить ці класи доступними для використання.

- Описова частина, у якій неформально описуються операції, асоційовані із класом. Це робить формальну специфікацію більш простою для розуміння. Формальна специфікація доповнює цей опис, забезпечуючи однозначний синтаксис і семантику операцій.
- Частини сигнатур, у якій визначається синтаксис інтерфейсу об'єктного класу або абстрактного типу даних. Тут описуються імена операцій, кількість і типи їх параметрів, а також класи вихідних результатів операцій.
- Частини аксіом, де визначається семантика операцій за допомогою створення ряду аксіом, які характеризують поведінку абстрактного типу даних. Ці аксіоми зв'язують операції створення об'єктів класу з операціями, що перевіряють їх значення.

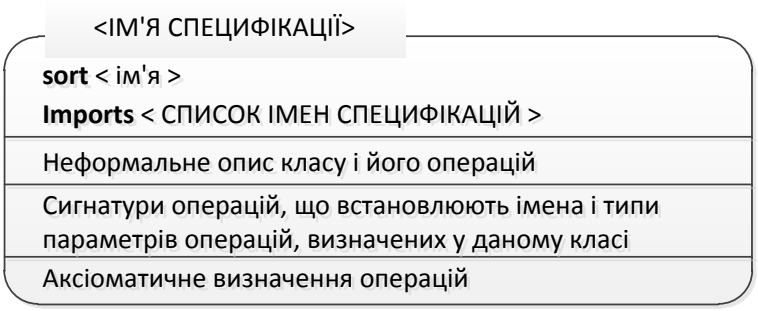

Рис. 6.5. Структура алгебраїчної специфікації

Процеси розробки формальної специфікації інтерфейсу підсистеми включає наступні дії

1. *Структурування специфікації.* Представлення неформальної специфікації інтерфейсу у вигляді безлічі абстрактних типів даних або об'єктних класів. Також неформально визначаються операції, асоційовані з кожним класом.
2. *Іменування специфікацій.* Задаються імена для кожної специфікації абстрактного типу, визначаються параметри специфікацій (якщо вони

необхідні) і імена обумовлених класів.

3. *Визначення операцій.* На підставі списку виконуваних інтерфейсом функцій для кожної специфікації визначається пов'язаний з нею набір операцій. Необхідно передбачити операції по створенню екземплярів класів, по зміні значень екземплярів класів і по перевірці цих значень. Імовірно, прийде додати нові функції до спочатку певного списку функцій інтерфейсу.
4. *Неформальна специфікація операцій.* Написання неформальної специфікації для кожної операції, де повинно бути зазначене, як операції впливають на обумовлений клас.
5. *Визначення синтаксису операцій.* Визначення синтаксису і параметрів для кожної операції. Це частина сигнатури формальної специфікації.
6. *Визначення аксіом.* Визначення семантики операцій шляхом опису умов, які повинні виконуватися для різних комбінацій операцій.

Для пояснення методики алгебраїчної специфікації, розглянемо просту структуру даних зв'язаного списку, специфікація якого показана на рис. 6.6.

Припустимо, що перший етап розробки специфікації списку, а саме структурування специфікації, виконаний. Імена специфікації і ім'я класу може бути тим самим, хоча корисно проводити відмінність між ними, використовуючи яку-небудь угоду. Наприклад, використовуються заголовні букви для імені специфікації (LIST) і прописні букви з першою заголовною буквою для імені класу (List). Оскільки списки можуть містити елементи різних типів, специфікація має загальний параметр Elem (Елемент). Тип Elem може представляти ціле число, рядок, список і т.д.

LIST (Elem)

sort List
imports INTEGER

Визначення списку , в якому елементи додаються в кінець , а вилучаються из вершины (начала) списка. Операция **Create** (Створити) створює порожній список; операция **Cons** (Конструирование) создает новый список, содержащий зазначений елемент; **Length** (Довжина) повертає розмір списку; **Head** (Верхній елемент) повертає елемент з вершини списку ; операція **Tail** змінює список шляхом видалення з нього першого елементу (елементу на вершині списку). Значення типу **Elem** (Елемент) не визначені.

Create → List
Cons (List, Bern) → List
Head (List) → Elem
Length (List) → Integer
Tail (List) → List

Head (Create) = Undefined **exception** (порожній список)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

Рис. 6.6. Специфікація зв'язаного списку

У загальному випадку для кожного абстрактного типу даних набір необхідних операцій повинен містити операцію по створенню нового екземпляра цього типу даних і операцію по конструюванню екземпляра даного типу з наявних елементів (у нашому прикладі це операції **Create** і **Cons).** У випадку застосування списків також повинні бути операції для одержання значення першого елемента списку (у нашому прикладі операція **Head),** операція, яка змінює список шляхом видалення його першого елемента **(Tail),** і операція підрахунку кількості елементів у списку **(Length).**

При визначенні синтаксису цих операцій слід установити, які для них необхідні параметри і які повинні бути результати операцій. У загальному випадку вхідні параметри належать або обумовленому класу (у цьому випадку клас **List)** або більш загальному (родовому) класу. Результати операцій також належать тим же класам або деякому іншому, наприклад **Integer** або **Boolean;** операція **Length** повертає ціле число. Декларація імпорту, що повідомляє використання специфікації цілих чисел, повинна бути включена в специфікацію. Аксіоми, що визначають семантику абстрактного типу даних, написані з використанням раніше визначених операцій.

Операції над абстрактним типом даних зазвичай відносяться до одного із двох класів.

1. *Операції конструювання,* які створюють або змінюють об'єкти класу. Зазвичай їх називають **Create** (Створити), **Update** (Змінити), **Add** (Додати) або, як у нашому випадку, **Cons** (Конструювання).
2. *Операції перевірки*, які повертають атрибути класу. Зазвичай їм дають імена, відповідні до імен атрибута, або імена, подібні **Eval** (Значення), **Get** (Одержати) і т.п.

Гарним емпіричним правилом для написання алгебраїчної специфікації є створення аксіом для кожної операції конструювання із застосуванням усіх операцій перевірки. Це означає, що якщо є **m** операцій конструювання і **n** операцій перевірки, то повинно бути визначено **m** x **n** аксіом.

Операції конструювання, пов'язані з абстрактним типом даних, часто дуже складні і можуть визначатися через інші операції конструювання і перевірки. Якщо операції конструювання визначені за допомогою інших операцій, то необхідно визначити операції перевірки, використовуючи більш примітивні конструкції.

У специфікації списку операціями конструювання є **Create, Cons** і **Tail,** які створюють списки. Операціями перевірки є **Head** і **Length,** які використовуються для одержання значень атрибутів списку. Операція **Tail** не є примітивною конструкцією, тому для неї можна не визначати аксіоми з використанням операцій **Head** і **Length,** але в такому випадку **Tail** необхідно визначити за допомогою примітивних конструкцій.

При написанні алгебраїчних специфікацій часто використовується рекурсія. Результат операції **Tail** – список, сформований із вхідного списку шляхом видалення верхнього елемента. Це визначення підказує, як використовувати рекурсію для побудови даної операції. Операція визначається на порожніх списках, потім рекурсивно переходить на непусті списки і завершується, коли результатом знову буде порожній список.

Іноді простіше зрозуміти рекурсивні перетворення, використовуючи

короткий приклад. Припустимо, що є список [5, **7],** де елемент 5 – початок (вершина) списку, а елемент **7–** кінець списку. Операція **Cons([5, 7], 9)** повинна повернути список [5, **7, 9],** а операція **Tail,** застосована до цього списку, повинна повернути список **[7, 9].** Приведемо послідовність рекурсивних перетворень, що приводить до цього результату.

```
Tail ([5, 7, 9]) =
  = Tail (Cons ([5, 7], 9)) =
  = Cons (Tail ([5, 7]), 9) =
  = Cons (Tail (Cons ([5], 7)), 9) =
  = Cons (Cons (Tail ([5]), 7), 9) =
      = Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
      = Cons (Cons ([Create], 7), 9) =
  = Cons ([7], 9) =
      = [7, 9]
```

Тут систематично використовувалися аксіоми для **Tail,** що привело до очікуваного результату. Аксіому для операції **Head** можна перевірити подібним способом.

Тепер розглянемо, як цю методику можна використовувати при розробці специфікації критичної системи. Припустимо, що є система керування повітряним рухом, метою якої є контроль над певним сектором повітряного простору. У кожному контрольованому секторі може перебувати кілька літаків, що мають різні ідентифікатори. З міркувань безпеки всі літаки повинні бути розведені по висоті принаймні на 300 метрів. У випадку спроби яким-небудь літаком порушити це обмеження, система повинна видати попереджуючий сигнал.

SECTOR

sort Sector
imports INTEGER, BOOLEAN

Enter додає літак в сектор, якщо дозволяють умови безпеки
Leave видаляє літак з сектора
Move переміщує літак з однієї висоти на іншу, якщо дозволяють умови
Lookup визначає висоту літака

Create створює порожній сектор
Put додає літак в сектор без перевірки умов безпеки
In-space перевіряє, чи є літак в секторі
Occupied перевіряє, чи доступна зазначена висота

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

Enter (S. CS, H) =
    if In-space (S, CS) then S exception (Літак вже в секторі)
    elsif Occupied (S, H) then S exception (Висота зайнята)
    else Put (S, CS, H)

Leave (Create, CS) = Create exception (Літак не в секторі)
Leave (Put (S, CS), HI) CS)=
    if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S. CS, H) =
    if S = Create then Create exception (В секторі немає літаків)
    elsif not In-space (S, CS) then S exception (Літак не в секторі)
    elsif Occupied (S, H) then S exception (Висота зайнята)
    else Put (Leave (S, CS), CS, H)

-- NO-HEIGHT - константа, що показує, що значення висоти не повернуто

Lookup (Create, CS) = NO-HEIGHT exception (Літак не в секторі)
Lookup (Put (S, CS1, H1). CS) =
    if CS = CS1 then HI else Lookup (S. CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H)=
    if (H1 > H and H1 - H ≤ 300) or (H > H1 and H-H1 ≤ 300) then true
    else Occupied (S, H)
In-space (Create, CS) = false
In-space (Put (S. CS1. H1). CS) =
  if CS = CSI then true else In-space (S. CS)

Рис. 6.7. Специфікація класу Sector, що представляє сектор контрольованого повітряного простору

Щоб спростити опис, визначається тільки обмежене число операцій над об'єктом-сектором. У реальній системі, зазвичай, буде набагато більше операцій і більш складними будуть умови безпеки польоту літаків. Основні операції наступні.

1. Enter (Введення). Додає літак (який представляється ідентифікатором) у повітряний простір на зазначеній висоті. На цій висоті або на відстані ближче 300 метрів від нього не повинно бути іншого літака.
2. **Leave** (Вихід). Видаляє зазначений літак з контрольованого сектору. Вона застосовується, якщо літак переміщається в сусідній сектор.
3. **Move** (Переміщення). Переміщає літак з однієї висоти на іншу. Знову перевіряються умови безпеки – відстань між літаками повинна бути не менше 300 метрів.
4. **Lookup** (Перегляд). Визначає поточну висоту літака в секторі.

Визначення цих операцій спроститься, якщо визначені також інші операції.

1. **Create** (Створити). Стандартна операція для будь-якого абстрактного типу даних. Вона створює порожній екземпляр даного типу. У нашому випадку ця операція задає сектор, у якому відсутні літаки.
2. **Put** (Помістити). Більш проста версія операції **Enter.** Вона додає новий літак у сектор без перевірки обмежень.
3. **In-space** (Перевірка простору). Повертає значення істини, якщо зазначений літак знаходиться в контрольованому секторі, а якщо ні, то її значення неправильне.
4. **Occupied** (Зайнятий). Повертає значення істини, якщо усередині 300-метрової зони по висоті є літак, а якщо ні, то її значення неправильне.

Прості операції визначаються для того, щоб згодом використовувати їх як блоки для компонування більш складних операцій. Алгебраїчна специфікація класу Sector (Сектор) показана на рис. 6.7.

По суті, основними операціями конструювання є **Create** і **Put,** які використані в специфікаціях інших операцій. **Occupied** і **In-space** є операціями перевірки, які визначають використання операцій **Create** і **Put.** Як виконуються ці і інші операції, легко зрозуміти зі специфікації.

3. Специфікація поведінки систем

Прості алгебраїчні методи підходять для опису інтерфейсів, коли операції, асоційовані з об'єктом, не залежать від стану об'єкта. Тоді результати будь-якої операції не залежать від результатів попередніх операцій. Якщо ця умова не виконується, алгебраїчні методи можуть стати громіздкими. Алгебраїчні описи поведінки систем часто штучні і важкі для розуміння.

Альтернативним підходом до створення формальних специфікацій, який широко використовується в програмних проектах, є специфікація, заснована на моделях системи. Такі специфікації використовують моделі станів системи. Системні операції визначаються за допомогою змін станів системної моделі. У такий спосіб визначається поведінка системи.

Для розробки специфікацій, заснованих на системних моделях, використовуються системи нотацій методів VDM, У и Z. Тут використовується нотація методу Z. У цьому методі система описується на основі теорії множин, яка доповнюється спеціальними логічними структурами, розробленими для специфікування програмних систем. Повний опис методу Z дуже об'ємний**.**

Формальні специфікації можуть бути важкими для читання і громіздкими, особливо якщо використовуються великі математичні формули. Це сповільнює розробку ПЗ. Розробники методу Z звернули увагу на цю проблему. У цьому методі специфікації представляються як неформальний текст, доповнений формальними описами. Формальна частина специфікації складається з невеликих простих описів (називаних схемами), які візуально відділяються від іншого тексту специфікації (рис. 6.8). Схеми використовуються для введення змінних станів і визначення обмежень і операцій над станами. У методі також передбачені операції, виконувані над схемами, зокрема для побудови, перейменування і приховання схем.

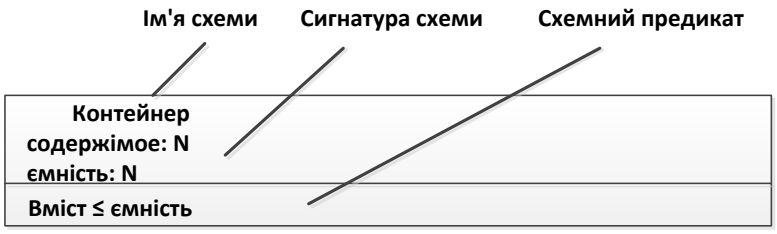

Рис. 6.8. Структура Z-схеми

Сигнатура схеми визначає сутності, які складають стан системи, схемні предикати – це набір умов, які повинні бути завжди істинні для цих сутностей. Якщо схема визначає операції, предикати можуть представляти перед-і постумови для цих операцій.

Для ілюстрації застосування методу Z у розробці специфікації критичної системи розглянемо спрощений приклад системи інсулінового насоса, використовуваної діабетиками. У діабетиків порушений природній метаболізм цукру, їм потрібні ін'єкції інсуліну – гормону, необхідного для метаболізму глюкози. Дана система контролює рівень цукру в крові хворого і, якщо потрібно, робить автоматичну ін'єкцію інсуліну.

Рівень цукру в крові хворого перевіряється через рівні проміжки часу, і, якщо він збільшується, проводиться ін'єкція інсуліну, яка знижує рівень цукру. На рис. 6.9 показана структура інсулінового насоса.

1. *Набір голок.* Приєднані до насоса, використовуються для ін'єкції інсуліну в тіло діабетика.
2. *Датчик.* Вимірює рівень цукру в крові хворого. У формальній специфікації операція одержання даних від датчика названа **reading?.**
3. *Насос.* Передає інсулін з резервуара до набору голок. У формальній специфікації величина дози інсуліну названа dose.
4. *Керуючий пристрій.* Управляє об'єктами системи. Має перемикач “Включене – Виключене”, кнопку скасування і кнопку установки дози інсуліну. Для спрощення формальної специфікації ці елементи керування не описані.

5. *Пристрій аварійної сигналізації.* Подає звуковий сигнал при виникненні проблем. У специфікації сигнал на вході пристрою аварійної сигналізації позначений як **alarm!.**
6. *Індікапюри.* Є два індикатори: один показує останній аналіз рівня цукру в крові, іншої – інформацію, що видається системою користувачеві. Ці індикатори у формальній специфікації позначаються **display 1**! і **dispiay2!.**
7. *Годинник.* Забезпечують керуючий пристрій значенням поточного часу.

Навіть для невеликої системи, подібній системи керування ін'єкціями інсуліну, формальна специфікація досить велика. Хоча основні системні операції прості, існує багато аварійних ситуацій, які необхідно враховувати. Тут наведені тільки деякі з основних Z-схем специфікації і пояснене, що вони означають.

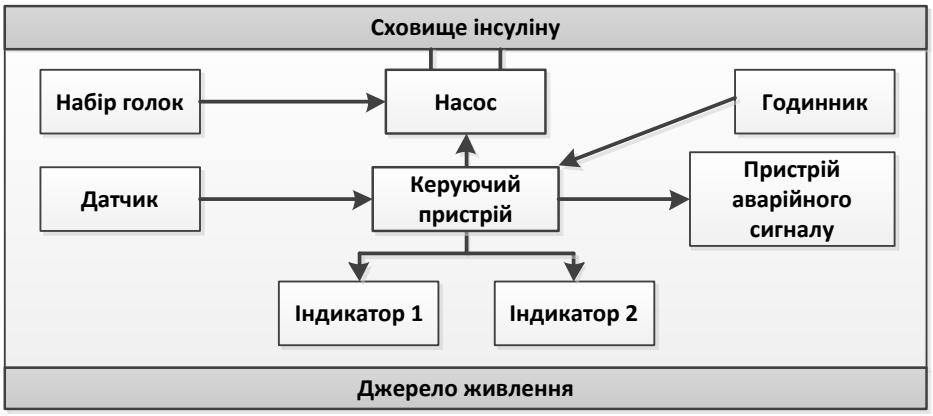

Рис. 6.9. Блок-схема інсулінового насоса

Основна схема Insulin-pump (інсуліновий насос), яка моделює стани інсулінового насоса, показана на рис. 6.10. Схема розбита на дві частини. У верхній частині оголошуються імена і типи, у нижній наведені умови, які повинні завжди виконуватися.

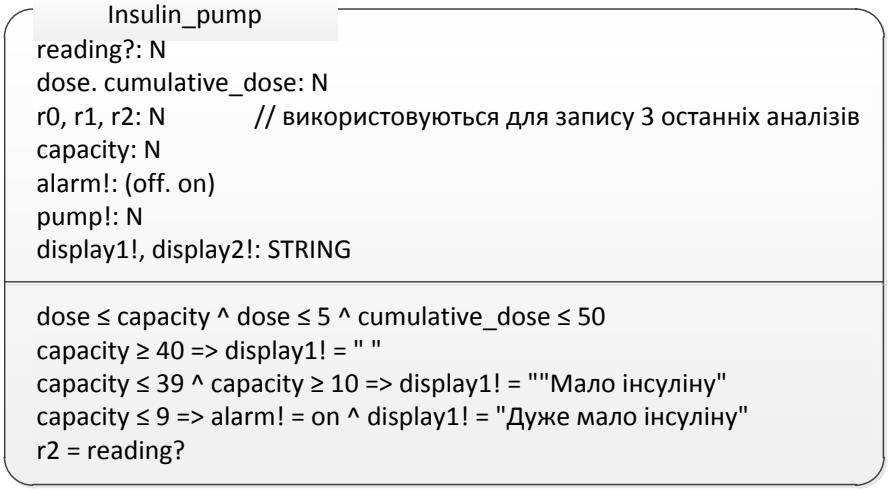

Рис. 6.10. Z-схема інсулінового насоса

Стан системи моделюється за допомогою ряду змінних. Відповідно до угод, прийнятих у методі Z, імена, що закінчуються знаком ?, використовуються для представлення вхідних даних, а імена, що закінчуються знаком !, представляють вихідні дані. У схемі інсулінового насоса оголошені наступні імена.

1. **reading?.** Це ненегативне ціле число, яке представляє дані від датчика, що визначає кількість цукру в крові. Це вхідна величина.
2. **dose, cumulative_dose.** Це також натуральні числа, що представляють відповідно дозу інсуліну і сумарну дозу інсуліну, введену за певний період часу.
3. **r0, r1, r2.** Представляють останні три значення, отримані від датчика, і використовуються для обчислення зміни цукру в крові.
4. **capacity.** Натуральне число, що представляє обсяг інсуліну в сховищі (резервуарі).
5. **alarm!.** Ця вихідна величина повідомляє про аварійні ситуації в системі.
6. **pump!.** Це натуральне число, що представляє керуючі сигнали, що посилають до фізичного насоса.
7. **display1!, display2!.** Ці вихідні величини строкового типу представляють два текстові індикатори. Один індикатор використовується для відображення текстових повідомлень, інший – для показу введеної дози інсуліну.

Схемні предикати визначають ряд умов, які завжди повинні бути дійсними.

1. Доза повинна бути менше або дорівнювати кількості інсуліну в резервуарі.
2. Однократна доза не повинна перевищувати 5 одиниць інсуліну, а загальна доза, введена за певний проміжок часу – 50 одиниць інсуліну. Це умови безпеки.
3. **displayl!.** Показує повідомлення про стан інсулінового резервуара. Якщо резервуар містить 40 або більше одиниць інсуліну, повідомлень немає. Коли в резервуарі інсуліну від 10 до 40 одиниць, на дисплеї відображається попередження, якщо ж у резервуара менше 10 одиниць, звучить звуковий сигнал і відображається відповідне попередження.
4. Робота інсулінового насоса полягає у визначенні кожні 10 хвилин кількості цукру в крові пацієнта і введенні інсуліну, якщо рівень цукру збільшується (цей дуже спрощений опис), кількість інсуліну, що вводиться, обчислюється згідно зі схемою DOSAGE (Дозування), яка наведена на рис. 6.11. Із цієї схеми видно, що на дозу, що вводиться, впливає безліч різних факторів.

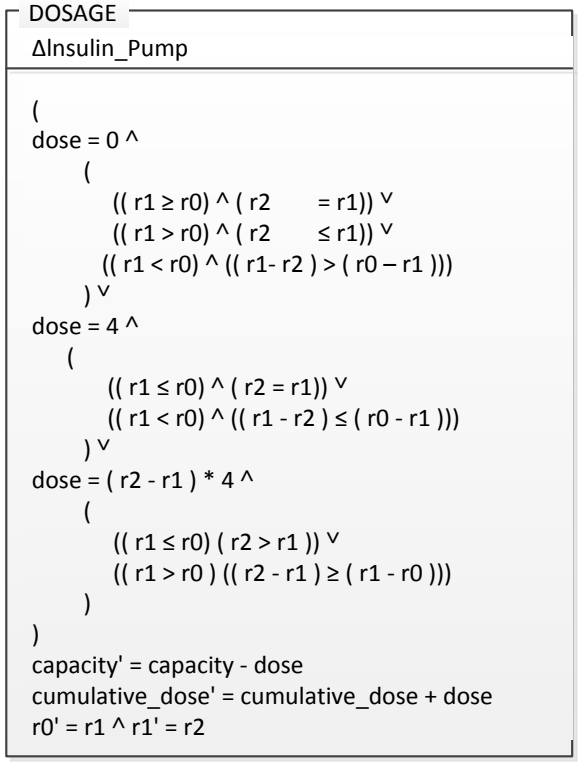

Рис. 6.11. Z-схеми обчислення дози

На рис. 6.11 показаний часто використовуваний засіб методу Z, а саме дельта-схема. Якщо ім'я схеми включене в розділ описів, то це рівносильне включенню всіх імен даної схеми, а її умови включаються в предикатну частину. У цьому випадку в схему DOSAGE включаються імена і умови схеми **lnsulin_Pump.** Якщо імені схеми передує символ Д, то вводиться новий набір величин, імена яких збігаються з іменами даної схеми, але до них додається символ ' (апостроф). Так позначаються значення змінних станів, змінені після виконання операції. Наприклад, якщо операція змінює значення змінної **val,** то після операції ця змінна буде позначатися **val'.** Дельта-схема DOSAGE вводить імена **capacity', cumulative_dose'** і т.д.

Схеми, що моделюють вихідні дані інсулінового насоса, показані на рис. 6.12. Це моделі індикаторів і пристрою аварійної сигналізації. Тут також використовується дельта-схема. Зі схеми DISPLAY видно, що індикатор **display2!** показує обчислену дозу **(Nat_to string** – функція перетворення), індикатор **displayl!** відображає або попереджуюче повідомлення, або ОК. У схемі ALARM показані умови, коли активізується аварійна сигналізація. Вона включається, якщо рівень цукру в крові дуже низький

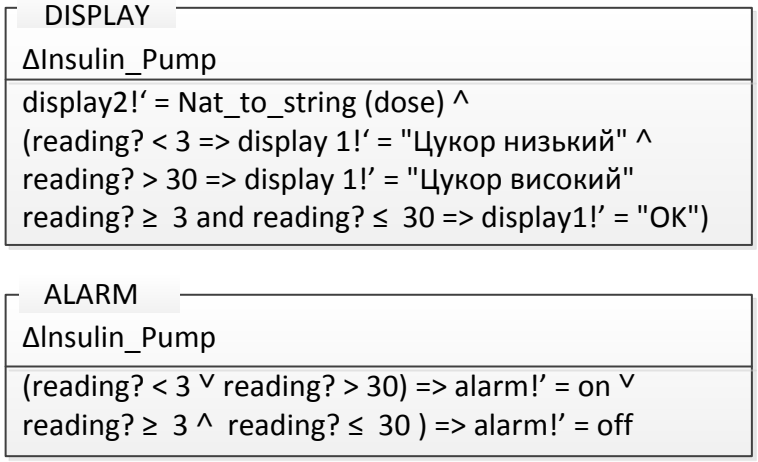

Рис. 6.12. Схеми вихідних даних

Предикати у всіх Z-схемах повинні бути погоджені, тобто не повинно бути

умов в одній схемі, які суперечать предикатам в іншій схемі. Якщо в специфікації замічені протиріччя, можна застосувати різні математичні методи аналізу Z-специфікації. У загальній схемі інсулінового насоса **lnsulin_Pump** установлено, що індикатор **displayl**! повинен показувати стан резервуара інсуліну. Однак у схемі **DISPLAY** цей індикатор повинен показувати рівень цукру в крові. Тут протиріччя, яке слід дозволити під час обговорення системи з медичними експертами і потенційними користувачами.

Потрібно враховувати, що датчик контролює рівень цукру в крові кожні 10 хвилин. Хоча це, зазвичай, можливо, але досить громіздко; неформальний опис дій більш короткий і зрозумілий, чим формальна специфікація.

Основною перевагою застосування формальної специфікації є можливість виявлення проблем у системних вимогах. У неформальній специфікації легко упустити ці проблеми, які однаково будуть вирішені, але на більш пізній стадії процесу розробки ПЗ.

Завдання:

1. Банківські автомати використовують інформацію з картки клієнта, що надає ідентифікатор банку, номер рахунку і персональний ідентифікатор клієнта. Вони також одержують інформацію із центральної бази даних і вносять у неї зміни по завершенню транзакції. Використовуючи ваші знання роботи банкоматів, напишіть Z-схему, що визначає стани системи, перевірку картки клієнта і зняття коштів.

Питання:

1. Поясніть, чому архітектурне проектування системи повинне передувати розробці формальної специфікації.
2. Перед вами поставлене завдання “продажу” методів формальної специфікації організації, що розробляє програмне забезпечення. Як ви будете пояснювати переваги формальної специфікації скептично настроєним розробникам ПЗ?

3. Поясніть, чому необхідно визначати інтерфейси підсистем як можна точніше і чому алгебраїчна специфікація найбільше підходить для специфікування інтерфейсів підсистем.

4. Абстрактний тип даних, що представляє стек, має наступні операції:
   New (Створити) – створює порожній стек;
   Push (Додати) – додає елемент у вершину стека;
   Top (Вершина) – повертає елемент на вершині стека;
   Retract (Вилучити) – видаляє елемент із вершини стека і повертає модифікований стек; Empty (Порожній) – повертає значення істини, якщо стек порожній.
   Визначите цей абстрактний тип даних, використовуючи алгебраїчну специфікацію.

Лабораторна робота № 5

Тема: Проектування систем реального часу

Ціль - познайомитися з технологією проектування систем реального часу і з деякими загальними архітектурами таких систем.

- знати основні концепції систем реального часу і розуміти, чому ці системи зазвичай реалізовані у вигляді паралельних процесів;
- освоїти основні етапи процесу проектування систем реального часу;
- знати призначення керуючої програми системи реального часу;
- познайомитися із загальними архітектурами процесів систем спостереження і керування, а також систем збору даних.

Завдання:

1. Спроектуйте архітектуру процесів для системи спостереження, що збирає дані із групи датчиків, що вимірюють состав повітря і розташованих навколо міста. У системі 5000 датчиків, організованих у групи по 100 штук. Кожний датчик повинен перевірятися 4 рази в секунду. Якщо більше 30% датчиків у групі зафіксують, що якість повітря нижче припустимого рівня, активізується попереджуючий світловий сигнал. Усі датчики передають зібрані дані центральному комп'ютеру, який кожні 15 хв генерує звіт про состав повітря в місті.

Питання:

1. Чому системи реального часу зазвичай реалізовані як безліч паралельних процесів? Проілюструйте свою відповідь прикладами.
2. Поясніть, чому об'єктно-орієнтовані методи розробки ПЗ не завжди підходять до систем реального часу.
3. Сильні і слабкі сторони Java як мови програмування для реалізації систем реального часу.

4. Якщо в бортовій системі безпеки поїзда при зборі даних зі шляхових передавачів використовуються періодичні процеси, яку частоту збору даних слід запланувати, щоб система гарантовано одержувала інформацію від передавачів? Обґрунтуйте свою відповідь.

Короткі теоретичні відомості:

У цей час комп'ютери застосовуються для керування широким спектром різноманітних систем, починаючи від простих домашніх пристроїв і закінчуючи великими промисловими комплексами. Ці комп'ютери безпосередньо взаємодіють із апаратними пристроями. Програмне забезпечення таких систем (керуючий комп'ютер плюс керовані об'єкти) являє собою вбудовану систему реального часу, завдання якої - реагувати на події, які генеруються обладнанням, тобто у відповідь на ці події виробляти керуючі сигнали. Таке ПЗ *вбудовується* в великі апаратні системи і повинне забезпечувати реакцію на події, що відбуваються в оточенні системи, у режимі *реального часу.*

Системи реального часу відрізняються від інших типів програмних систем. Їхнє коректне функціонування залежить від здатності системи реагувати на події через заданий (як правило, короткий) інтервал часу.

Система реального часу - це програмна система, правильне функціонування якої залежить від результатів її роботи і від періоду часу, протягом якого отриманий результат. “М'яка” система реального часу - це система, у якій операції *віддаляються,* якщо протягом певного інтервалу часу не виданий результат. “Тверда" система реального часу - це система, операції якої стають *некоректними*, тобто виробляється сигнал про помилку, якщо протягом певного інтервалу часу результат не виданий.

Систему реального часу можна розглядати як систему “стимул-відгук”. При одержанні певного вхідного стимулу (вхідного сигналу) система генерує пов'язаний з ним відгук (відповідна дія або відповідний сигнал). Отже, поведінка системи реального часу можна визначити за допомогою списку вхідних сигналів, одержуваних системою, пов'язаних з ними відповідних сигналів (відгуків) і інтервалу часу, протягом якого система повинна відреагувати на вхідний сигнал.

Вхідні сигнали діляться на два класи.

1. *Періодичні сигнали* відбуваються через визначені інтервали часу. Наприклад, система перевіряє датчик кожні 50 мілісекунд, і вживає дії (реагує) залежно від значень, отриманих від датчика (стимулу).
2. *Аперіодичні сигнали* відбуваються нерегулярно. Зазвичай вони "повідомляють про себе" за допомогою механізму переривань. Прикладами аперіодичного сигналу може бути переривання, яке виробляється по завершенню передачі вхід/вихід і розміщення даних у буфері обміну.

У системах реального часу періодичні вхідні сигнали зазвичай генеруються сенсорами (датчиками), взаємодіючими із системою. Вони надають інформацію про стан зовнішнього оточення системи. Системні відгуки (відповідні сигнали) направляються групі виконавчих механізмів, керуючих апаратними пристроями, які потім впливають на оточення системи. Аперіодичні вхідні сигнали можуть генеруватися і сенсорами, і виконавчими механізмами. Як правило, аперіодичні сигнали означають виняткові ситуації, наприклад помилки в роботі апаратури. На рис. 8.1 показана модель "сенсор-система-виконавчий механізм" для вбудованої системи реального часу.

Рис. 8.1. Загальна модель системи реального часу

Системи реального часу повинні реагувати на вхідні сигнали, що відбуваються в різні моменти часу. Отже, архітектуру такої системи необхідно організувати так, щоб керування переходило до відповідного до оброблювача якнайшвидше після одержання вхідного сигналу. У послідовних програмах такий механізм передачі керування неможливий. Тому зазвичай системи реального часу проектують як множину паралельних взаємодіючих процесів. Частина системи - керуюча програма, часто називана диспетчером, - управляє всіма процесами.

Більшість моделей “стимул-відгук” систем реального часу зводяться до узагальненої архітектурної моделі, що полягає із трьох типів процесів (рис. 8.2). Для кожного типу сенсора є процес керування сенсором; обчислювальний процес визначає необхідний відповідний сигнал на отриманий системою вхідний сигнал; процеси керування виконавчими механізмами управляють діями цих механізмів. Така модель дозволяє швидко зібрати дані із усіх наявних сенсорів (до того, як відбудеться наступне введення даних), обробити їх і одержати відповідний сигнал від відповідного виконавчого механізму.

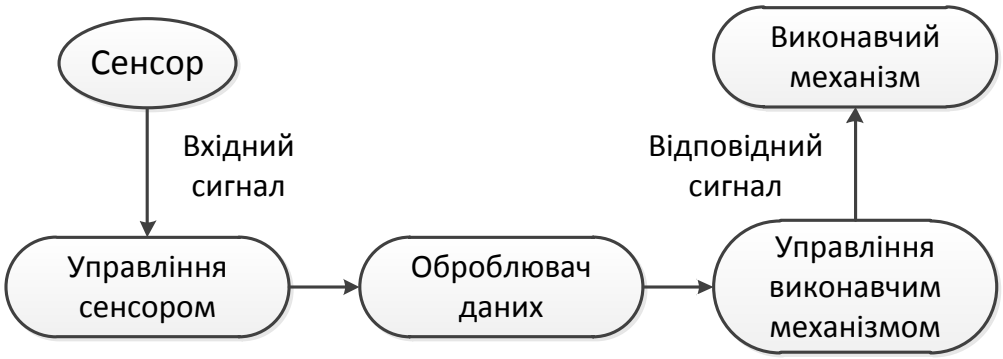

Рис. 8.2. Процеси керування сенсорами і виконавчими механізмами

1. Проектування систем

У процесі проектування системи ухвалюються рішення про те, які властивості будуть реалізовані програмною частиною системи, а які - апаратними засобами. Тимчасові обмеження і інші вимоги передбачають, що деякі функції системи, наприклад обробку сигналів, необхідно реалізувати на спеціально розробленому устаткуванні. Таким чином, процес проектування систем

реального часу містить у собі проектування устаткування (апаратури) спеціального призначення і проектування програмного забезпечення.

Апаратні компоненти забезпечують більш високу продуктивність, чим еквівалентне їм (по виконуваних функціях) програмне забезпечення. Апаратним засобам можна доручити “вузькі” місця системної обробки сигналів і, таким чином, уникнути дорогої оптимізації ПЗ. Якщо за продуктивність системи відповідають апаратні компоненти, при проектуванні ПЗ основну увагу можна приділити його переносимості, а питання, пов'язані із продуктивністю, відходять на другий план.

Рішення про розподіл функцій по апаратним і програмним компонентам слід ухвалювати якнайпізніше, оскільки архітектура системи повинна складатися з автономних компонентів, які можна реалізувати як апаратно, так і програмно. Саме така структура відповідає цілям розробника, що проектує зручну в обслуговуванні систему. Отже, результатом процесу проектування високоякісної системи повинна бути система, яку можна реалізувати і апаратними і програмними засобами.

Відмінності процесу проектування систем реального часу від інших систем полягає в тому, що вже на перших етапах проектування необхідно враховувати час реакції системи. У центрі процесу проектування системи реального часу - події (вхідні сигнали), а не об'єкти або функції. Процеси проектування таких систем складається з декількох етапів.

1. Визначення безлічі вхідних сигналів, які будуть оброблятися системою системних реакцій, що і відповідають їм, тобто відповідних сигналів.
2. Для кожного вхідного сигналу і відповідного йому відповідного сигналу обчислюються тимчасові обмеження. Вони застосовуються до обробки як вхідних, так і відповідних сигналів.
3. Об'єднання процесів обробки вхідних і відповідних сигналів у вигляді сукупності паралельних процесів. У коректній моделі системної архітектури кожний процес пов'язаний з певним класом вхідних і відповідних сигналів (як показано на рис. 8.2).
4. Розробка алгоритмів, що виконують необхідні обчислення для всіх вхідних і відповідних сигналів. Щоб одержати представлення про обсяги

обчислювальних і часових витрат у процесі обробки сигналів, розробка алгоритмів зазвичай проводиться на ранніх етапах процесу проектування.

5. Розробка тимчасового графіка роботи системи.
6. Складання системи, що працює під управлінням диспетчера керуючої програми.

Зазвичай, описаний процес проектування є ітераційним. Як тільки визначена структура обчислювальних процесів і часовий графік роботи, необхідно зробити всебічний аналіз і провести імітацію роботи системи, щоб упевнитися в тому, що вона задовольняє тимчасовим обмеженням. У результаті аналізу може виявитися, що система не відповідає тимчасовим вимогам. У такому випадку для підвищення продуктивності системи необхідно змінити структуру обчислювальних процесів, алгоритм, що управляє керування програми або всі ці компоненти разом.

В системах реального часу складно аналізувати тимчасові залежності. Через непередбачену природу аперіодичних вхідних сигналів розробники змушено робити деякі попередні припущення щодо ймовірності появ аперіодичних сигналів. Зроблені припущення можуть виявитися невірними, і після розробки системи її показники продуктивності не будуть задовольняти тимчасовим вимогам.

Усі процеси в системі реального часу повинні бути скоординовані. Механізм координації процесів забезпечує виключення конфліктів при використанні загальних ресурсів. Коли один процес використовує загальний ресурс (або об'єкт), інші процеси не повинні мати доступ до цього ресурсу. До механізмів, що забезпечують взаємне виключення процесів, відносяться семафори, монітори і метод критичних областей.

Моделювання систем реального часу

Системи реального часу повинні реагувати на події, що відбуваються через нерегулярні інтервали часу. Такі події (або вхідні сигнали) часто приводять до переходу системи з одного стану в інший. Тому одним зі способів опису систем

реального часу може бути модель кінцевого автомата і відповідна діаграма станів.

У моделі кінцевого автомата в кожний момент часу система знаходиться в одному зі своїх станів. Одержавши вхідний сигнал, вона переходить в інший стан. Наприклад, система керування клапаном може перейти зі стану “Клапан відкритий” у стан “Клапан закритий" після одержання певної команди оператора (вхідний сигнал).

На рис. 8.3 показана модель кінцевого автомата для звичайної мікрохвильової печі, обладнаної кнопками включення живлення, таймера і запуску системи. Стани системи позначені округленими прямокутниками, вхідні сигнали, що викликають перехід системи з одного стану в інший, показані стрілками. На діаграмі показані всі стани печі, також названі дії виконавчих механізмів системи або дії по виводу інформації.

Переглядати послідовність роботи системи потрібно зліва направо. У початковому стані **Очікування,** користувач може вибрати режим повної або половинної потужності. Наступний стан наступає при натисканні на кнопку таймера і установці часу роботи печі. Якщо двері печі закриті, система переходить у стан **Дія.**

У цьому стані йде процес готування їжі, після завершення якого стан **пекти** вертається в стан **Очікування.**

Моделі кінцевого автомата - гарний спосіб представлення структури систем реального часу. Тому такі моделі є невід'ємною частиною методів проектування систем реального часу. Метод Харела (Harel), що базується на *діаграмах станів,* спрямований на рішення проблеми внутрішньої складності моделей кінцевого автомата. Діаграма станів структурує моделі таким чином, що групи стану можна було б розглядати як єдині сутності. Крім того, за допомогою діаграм станів паралельні системи можна представити у вигляді моделі станів. Моделі станів підтримуються також UML.

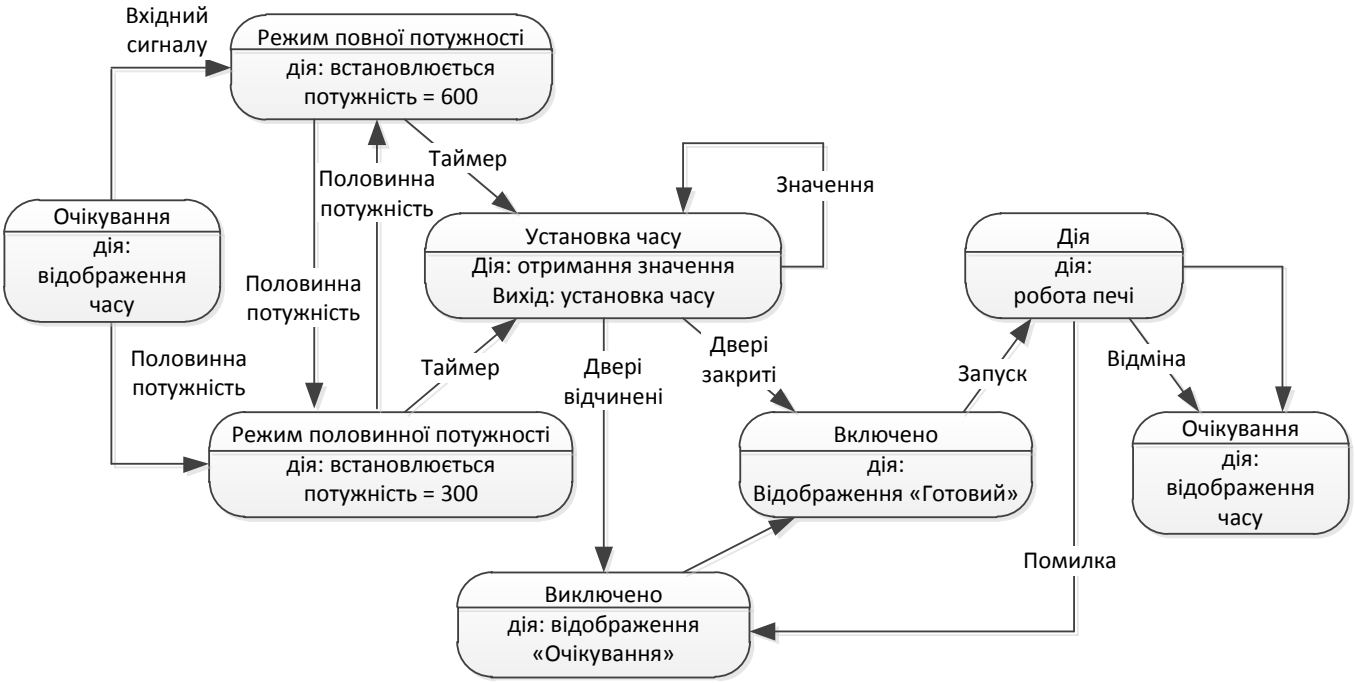

Рис. 8.3. Модель кінцевого автомата для мікрохвильової печі

Програмування систем реального часу

На архітектуру системи реального часу впливає мова програмування, яка використовується для реалізації системи. Дотепер тверді системи реального часу часто програмуються на асемблерних мовах. Мови більш високого рівня також дають можливість згенерувати ефективний програмний код. Наприклад, мова С дозволяє писати досить ефективні програми. Однак у ньому немає конструкцій, що підтримують паралельність процесів і керування спільно використовуваними ресурсами. Крім того, програми на С часто складні для розуміння.

Мова Ada споконвічно розроблявся для реалізації вбудованих систем, а тому має у своєму розпорядженні такі засоби, як керування процесами, виключення і правила представлення. Його засіб *рандеву* (rendezvous) - відмінний механізм для синхронізації завдань (процесів). На жаль, перша версія мови Ada (Ada 83) виявилася непридатною для реалізації твердих систем реального часу. У ній були відсутні засоби, що дозволяють установити граничні строки завершення завдань, не було вбудованих виключень для випадку перевищення граничних строків і пропонувався строгий алгоритм обслуговування черги “першим прийшов - першим вийшов”. При перегляді стандартів мови Ada головна увага

приділялася саме цим моментам. У переглянутій версії мови підтримуються захищені типи, що дозволило більш просто реалізовувати захищені поділювані структури даних і забезпечувати більш повний контроль при виконанні і синхронізації завдань. Однак при програмуванні систем реального часу поліпшена версія мови Ada все-таки не забезпечує достатнього контролю над твердими системами реального часу.

Перші версії мови Java розроблялися для створення невеликих систем, що вбудовуються, таких, наприклад, як контролери пристроїв і приладів. Розробники Java включили кілька засобів для підтримки паралельних процесів у вигляді паралельних об'єктів (потоків) і синхронізованих методів. Але оскільки в подібних системах немає строгих тимчасових обмежень, то і у мові Java не передбачені засоби, що дозволяють управляти плануванням потоків або запускати потоки в конкретні моменти часу.

Тому Java не підходить для програмування твердих систем реального часу або систем, у яких є строгий часовий графік процесів. Перелічимо основні проблеми Java як мови програмування систем реального часу.

1. Не можна вказати час, протягом якого повинен виконуватися потік.
2. Не контролюємо процес очищення пам'яті - він може початися в будь-який час. Тому неможливо передбачити поведінку потоків у часі.
3. Не можна визначити розміри черги, пов'язаної з поділюваними ресурсами.
4. Реалізація віртуальної машини Java відрізняється для різних комп'ютерів.
5. У мові немає засобів для детального аналізу розподілу часу роботи процесорів.

У цей час ведеться робота з розв'язку деяких із цих проблем і формується нова версія мови Java для програмування систем реального часу. Однак не зовсім зрозуміло, яким чином цю версію можна відокремити від лежачої в її основі віртуальної машини Java: властивість переносимості мови завжди конфліктувала з характеристиками режиму реального часу.

2. Керуючі програми

Керуюча програма (диспетчер) системи реального часу є аналогом операційної системи комп'ютера. Вона управляє процесами і розподілом ресурсів у системах реального часу, запускає і зупиняє відповідні процеси для обробки вхідних сигналів і розподіляє ресурси пам'яті і процесора. Однак зазвичай в керуючих програмах відсутні більш складні засоби, властиві операційним системам, наприклад засоби керування файлами.

Незважаючи на те що на ринку програмних продуктів існує кілька керуючих програм систем реального часу, їх часто проектують самостійно як частини систем через спеціальні вимоги, пропоновані до конкретних систем реального часу.

Компоненти керуючої програми (рис. 8.4) залежать від розмірів і складності проектованої системи реального часу. Зазвичай керуючі програми, за винятком найпростіших, складаються з наступних компонентів.

1. *Годинник реального часу* періодично надавав інформацію для планування процесів.
2. *Оброблювач переривань управляє* аперіодичними запитами до сервісів.
3. *Планувальник* переглядає список процесів, які призначені на виконання, і вибирає один з них.
4. *Адміністратор ресурсів*, одержавши процес, запланований на виконання, виділяє необхідні ресурси пам'яті і процесора.
5. *Диспетчер* запускає на виконання який-небудь процес.

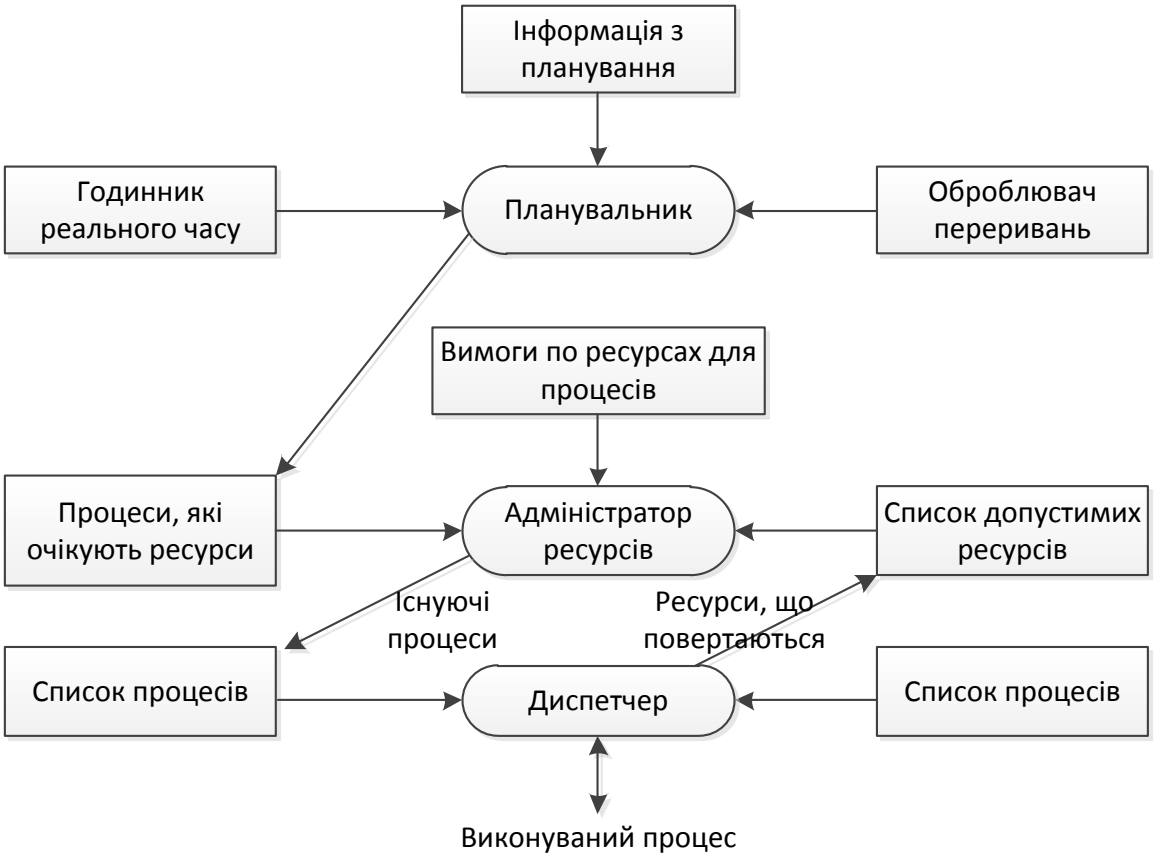

Рис. 8.4. Компоненти керуючої програми реального часу

Керуючі програми систем, що надають сервіси на постійній основі, наприклад телекомунікаційних або моніторингових систем з високими вимогами до надійності, можуть мати ще кілька компонентів.

- *Конфигуратор* відповідає за динамічне переконфігурування апаратних засобів. Не припиняючи роботу системи, з неї можна витягти апаратні модулі і змінити систему за допомогою додавання нових апаратних засобів.
- *Менеджер несправностей* відповідає за виявлення апаратних і програмних несправностей і вживає відповідні дії по їхньому виправленню.

Вхідні сигнали, оброблювані системою реального часу, зазвичай мають кілька рівнів пріоритетів. Для одних сигналів, наприклад пов'язаних з винятковими ситуаціями, важливо, щоб їх обробка завершувалася протягом певного інтервалу часу. Якщо процес із більш високим пріоритетом запитує

сервіс, то виконання інших процесів повинно бути припинене. Внаслідок цього адміністратор системи повинен уміти управляти принаймні двома рівнями пріоритетів системних процесів.

1. *Рівень переривань* є найвищим рівнем пріоритетів. Він привласнюється тим процесам, на які необхідно швидко відреагувати. Прикладом такого процесу може бути процес годин реального часу.
2. *Тактовий рівень* пріоритетів привласнюється періодичним процесам.

Ще один рівень пріоритетів може бути у фонових процесів, на виконання яких не накладаються тверді тимчасові обмеження (наприклад, процес самотестування). Ці процеси виконуються тоді, коли є вільні ресурси процесора.

Усередині кожного рівня пріоритетів різним класам процесів можна призначити інші пріоритети. Наприклад, може бути кілька рівнів переривань. Щоб уникнути втрати даних переривання від більш швидкого пристрою повинен витісняти обробку переривань від більш повільного пристрою.

Керування процесами

Керування процесами - це вибір процесу на виконання, виділення для нього ресурсів пам'яті і процесора і запуск процесу.

Періодичними називаються процеси, які повинні виконуватися через фіксований визначений проміжок часу (наприклад, при зборі даних або керуванні виконавчими механізмами). Керуюча програма системи реального часу для визначення моменту запуску процесу використовує свій годинник реального часу. У більшості систем реального часу є кілька класів періодичних процесів з різними періодами (інтервалами часу між виконанням процесів) і тривалістю виконання. Керуюча програма повинна бути здатна в будь-який момент часу вибрати процес, призначений на виконання.

Годинник реального часу конфігурувався так, щоб періодично подавати тактовий сигнал, період між сигналами становить зазвичай кілька мілісекунд. Сигнал годин ініціює процес на рівні переривань, який запускає планувальник процесів для керування періодичними процесами. Процес на рівні переривань

зазвичай сам не управляє періодичними процесами, оскільки обробка переривань повинна завершуватися якнайшвидше.

Дії, виконувані керуючою програмою при керуванні періодичними процесами, показані на рис. 8.5. Планувальник переглядає список періодичних процесів і вибирає з нього на виконання один процес. Вибір залежить від пріоритету процесу, періоду процесу, передбачуваної тривалості виконання і кінцевих строків завершення процесу. Іноді за один період між тактовими сигналами годин необхідно виконати два процеси з різними тривалостями виконання. У такій ситуації один процес необхідно призупинити на час, що відповідає його тривалості.

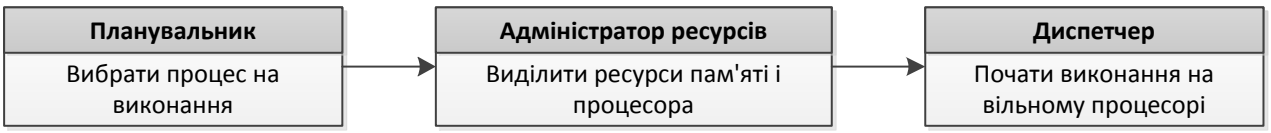

Рис. 8.5. Дії керуючої програми при запуску процесу

Якщо керуючою програмою зареєстроване переривання, це означає, що до одного із сервісів зроблений запит. Механізм переривань передає керування визначеній комірці пам'яті, у якій утримується команда перемикання на програму обслуговування переривань. Ця програма повинна бути простою, короткою і швидко виконуватися. Під час обслуговування переривань усі інші переривання системою ігноруються. Щоб зменшити ймовірність втрати даних, час перебування системи в такому стані повинен бути мінімальним.

Програма, що виконує сервісну функцію, повинна перекрити доступ наступним перериванням, щоб не перервати саму себе. Вона повинна виявити причину переривання і ініціювати процес із високим пріоритетом для обробки сигналу, що викликав переривання. У деяких системах високошвидкісного збору даних оброблювач переривань зберігає для наступної обробки дані, які в момент одержання переривання перебували в буфері. Після обробки переривання керування знову переходить до керуючої програми.

У будь-який момент часу може бути кілька призначених на виконання процесів з різними рівнями пріоритетів. Планувальник встановлює порядок виконання процесів. Ефективне планування відіграє важливу роль, якщо необхідно відповідати вимогам, які пред'являються до системи реального часу. Існує дві основні стратегії планування процесів.

1. *Невитісняюче планування.* Один процес планується на виконання, він запускається і виконується до кінця або блокується за якимись причинами, наприклад при очікуванні введення даних. При такому плануванні можуть виникнути проблеми, пов'язані з тим, що у випадку декількох процесів з різними пріоритетами процес із високим пріоритетом повинен чекати завершення процесу з низьким пріоритетом.
2. *Витісняюче планування.* Виконання процесу може бути припинене, якщо до сервісу зробили запити від процесів з більш високим пріоритетом. Процес із більш високим пріоритетом має перевагу перед процесом з більш низьким рівнем пріоритету, і тому йому виділяється процесор.

У рамках цих стратегій розроблено безліч різних алгоритмів планування. До них відноситься циклічне планування, при якому кожний процес виконується по черзі, і планування по швидкості, коли при першому виконанні одержують більш високий пріоритет процеси з коротким періодом виконання.

Інформація про призначений на виконання процес передається адміністраторові ресурсів. Він виділяє для обраного процесу необхідну пам'ять, а в багатопроцесорній системі - ще і процесор. Потім процес міститься в “список призначень”, тобто в список процесів, призначених на виконання. Коли процесор завершує виконання якого-небудь процесу і стає вільним, викликається диспетчер. Він переглядає наявний список, вибирає процес, який можна виконувати на вільному процесорі, і запускає його на виконання.

3. Системи спостереження і керування

У цей час можна виділити кілька класів стандартних систем реального часу: моніторингові системи (системи спостереження), системи збору даних, системи керування та ін. Кожному типу систем відповідає особлива структура процесів, тому при проектуванні системи, як правило, архітектуру створюють по одному з існуючих стандартних типів. Таким чином, замість обговорення загальних проблем проектування систем реального часу тут краще розглянути проектування за допомогою узагальнених моделей.

Системи спостереження і керування - важливий клас систем реального часу. Їхнім основним призначенням є перевірка сенсорів (датчиків), що надають інформацію про оточення системи, і виконання відповідних дій залежно від інформації, яка надійшла від сенсорів. Системи спостереження виконують дії після реєстрації особливого значення сенсора. Системи керування безупинно управляють апаратними виконавчими механізмами на підставі значень, одержуваних від сенсорів.

Розглянемо наступний приклад.

Нехай у будинку встановлена система охоронної сигналізації. У системі використовується кілька типів сенсорів: датчики руху, встановлені в окремих кімнатах; датчики на вікнах першого поверху, які подають сигнал, якщо розбивається вікно; дверні датчики, що фіксують відкривання дверей. Усього в системі 50 датчиків на вікнах, 30 на дверях і 200 датчиків руху.

Коли який-небудь датчик фіксує присутність стороннього, система автоматично викликає місцеву поліцію і, використовуючи звуковий синтезатор, повідомляє місце розташування датчика (номер кімнати), від якого йде сигнал. У кімнатах, розташованих біля активного датчика, включається світлова сигналізація і звуковий аварійний сигнал. Система сигналізації зазвичай включається через мережу, але може працювати і від батарей. Проблеми з електроживленням реєструються спеціальною програмою, що контролює напруга електромережі. Якщо в мережі реєструється спадання напруги, програма перемикає систему сигналізації на резервне живлення від батарей.

У цьому прикладі описана “м'яка” система реального часу, тому що тут немає твердих тимчасових вимог. У такій системі не потрібно реєструвати події, що відбуваються з високою швидкістю, тому опитування датчиків може проводитися два рази в секунду.

Процес проектування починається з опису аперіодичних вхідних сигналів, одержуваних системою, і пов'язаних з ними реакцій системи. Тут ми обмежимося спрощеним проектом, у якому не враховуються сигнали, породжувані процедурами самоперевірки, і зовнішні сигнали, генеруємі при тестуванні системи або при її вимиканні у випадку неправильної тривоги. В остаточному підсумку система обробляє тільки два типи вхідних сигналів.

1. *Відключення електроживлення* генерується програмою, що контролює електричне коло. У відповідь на цей сигнал система перемикає мережу на резервне живлення за допомогою подачі сигналу електронному приладу перемикання живлення.
2. *Сигнал про вторгнення* є вхідним і генерується одним з датчиків системи. У відповідь на нього система визначає номер кімнати, у якій знаходиться активний датчик, викликає поліцію, ініціюючи звуковий синтезатор, і включає звуковий сигнал тривоги і світлову сигналізацію будинку в місці порушення.

На наступному кроці процесу проектування визначаються тимчасові обмеження для кожного вхідного і відповідного сигналів системи. У табл. 8.1 перераховані ці тимчасові обмеження. До різних типів датчиків, що генерують вхідні сигнали, пред'явлені різні тимчасові вимоги.

Таблиця 8.1. Тимчасові обмеження на вхідні і відповідні сигнали системи

Сигнал	**Тимчасові обмеження**
Відключення електроживлення	Перемикання на живлення від батарей повинне відбутися протягом 50 мс
Сигналізація на двері	Кожний сигнальний датчик на дверях перевіряється двічі в секунду
Сигналізація на вікнах	Кожний датчик на вікні перевіряється двічі в секунду
Датчик руху	Кожний датчик руху опитується двічі в секунду
Звуковий сигнал	Звуковий сигнал повинен пролунати через півсекунди після сигналу датчика
Включення світлової сигналізації	Світлова сигналізація повинна ввімкнутися через півсекунди після сигналу датчика
Зв'язок	Виклик у поліцію повинен початися протягом 2 с після сигналу датчика
Синтезатор мови	Синтезоване повідомлення повинне бути готове через 4 с після сигналу датчика

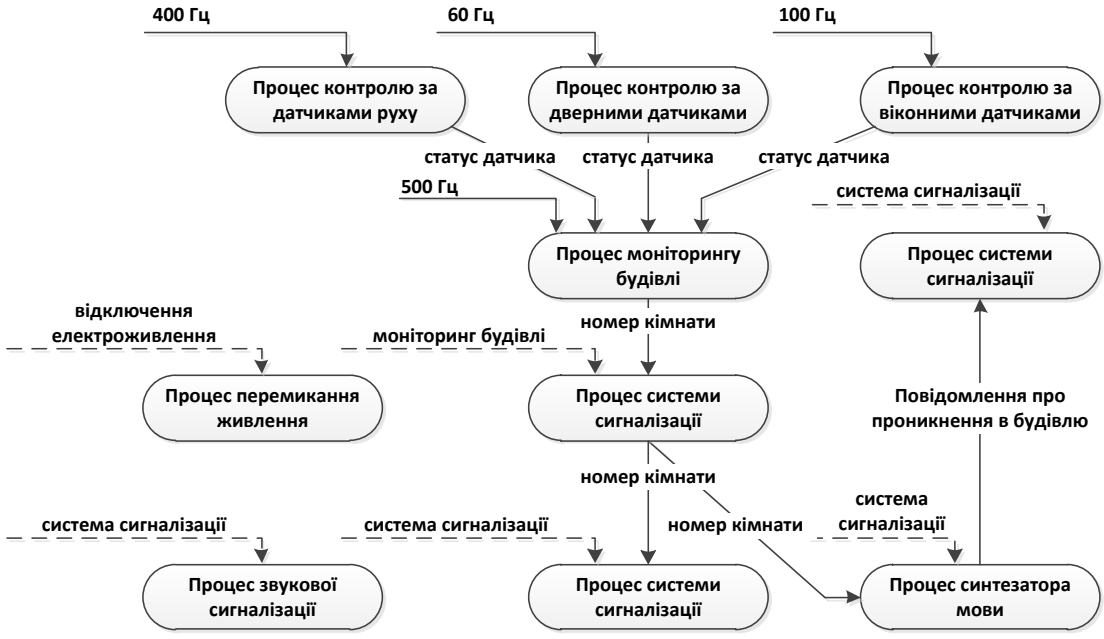

Рис. 8.6. Архітектура процесів системи охоронної сигналізації

На наступному етапі проектування розподіляються системні функції по паралельних процесах. Періодично потрібно опитувати три типи датчиків, тому у кожного типу датчиків є пов'язаний з ними процес. Крім цього, є система керування перериваннями, що контролює електроживлення, система зв'язку з поліцією, синтезатор мови, система включення звукової сигналізації і система, що включає світлову сигналізацію біля датчика. Кожною системою управляє незалежний процес. На рис. 8.6 показана архітектура процесів системи.

На схемі, представленої на рис. 8.6, стрілки (із примітками), що з'єднують окремі процеси, позначають потоки даних між процесами із зазначенням типу даних. Написи над стрілками праворуч над кожним процесом указують систему, що управляє даним процесом. На стрілках вгорі вказано мінімальне значення частоти виконання процесу.

Частота виконання процесів визначається кількістю датчиків і тимчасовими вимогами, що пред'являються системі. Припустимо, що в системі є 30 дверних датчиків, які потрібно перевіряти двічі на секунду. Отже, пов'язаний із дверним датчиком процес повинен виконуватися 60 раз у секунду (частота 60 Гц). Також 400 раз на секунду виконується процес, контролюючий датчик руху.

Аперіодичні процеси позначені стрілками з пунктирними лініями. На цих лініях зазначені події, які викликають даний процес. Усі основні виконавчі процеси (звукової і світлової сигналізації та ін.) починаються командою із процесу **Система безпеки;** їм не потрібні дані з інших процесів. Процесу, що управляє електроживленням, також не потрібні дані з інших частин системи.

Усі представлені процеси можна реалізувати мовою Java як потоки. Лістинг 8.1 містить код Java реалізації процесу **Buildingmonitor** (моніторинг будинку), що опитує датчики системи. У випадку сигналу тривоги програма активізує систему сигналізації. Нехай у прикладі система відповідає тимчасовим вимогам. У мові Java немає засобів для завдання частоти виконання потоків.

Лістинг 8.1. Реалізація процесу моніторингу будинку

```
class Buildingmonitor extends Thread {
      Buildingsensor win, door, move;

      Siren siren = new Siren ();
      Lights lights = new Lights ();
      Synthesizer synthesizer = new Synthesizer ();
      Doorsensors doors = new Doorsensors(30);
      Windowsensors windows = new Windowsensors(50);
      Movementsensors movements = new Movementsensors(200) ;
      Powermonitor pm = new Powermonitor();

      Buildingmonitor()
      {
         //ініціалізація датчиків і запуск процесів
         siren.start ();
         lights.start();
         synthesizer.start();
         windows.start();
         doors.start()movements.start();
         pm.start();
      }

      public void run ()
      {
         int room = 0;
         while (true)
         {

//перевірка датчиків руху два рази в секунду (400 Гц) move =
//movements.getval();
//перевірка віконних датчиків два рази в секунду (100 Гц)
win = windows.getval();
//перевірка дверних датчиків два рази в секунду (60 Гц)
door = doors.getval();
      if(move.sensorval==li door.sensorval==l|win.sensorval==l)
      {
         //датчик зареєстрував порушення
         if(move.sensorval == 1) room = move.room;
         if (door.sensorval == 1) room = door.room;
         if(win.sensorval == 1) room = win.room;
         lights.on(room);
         siren.on();
         synthesizer.on(room);
         break;
      }
      }
      lights.shutdown();siren.shutdown();
      synthesizer.shutdown();
      windows.shutdown();doors.shutdown();
      movements.shutdown();
      )//run }
//Buildingmonitor
```

Оскільки дана система не містить строгих тимчасових вимог, її можна реалізувати мовою Java. Зазвичай, немає ніякої гарантії відповідності тимчасовим

специфікаціям. У нашій системі всі датчики опитуються однакова кількість раз, чого не буває в реальних системах.

Після визначення архітектури процесів системи починається розробка алгоритмів обробки вхідних сигналів і генерації відповідних сигналів. Цей етап проектування необхідно виконувати якомога раніше, щоб упевнитися, що система буде відповідати тимчасовим вимогам. Якщо відповідні алгоритми виявляються складними, може виникнути необхідність у зміні тимчасових обмежень. Зазвичай алгоритми систем реального часу порівняно прості. Вони перевіряють комірки пам'яті, виконують деякі прості розрахунки і управляють передачею сигналу. У прикладі системи сигналізації проектування алгоритмів не розглядається.

Останнім етапом процесу проектування є складання тимчасового графіка виконання процесів. У прикладі немає процесів зі строгими строками виконання. Розглянемо пріоритети процесів. Усі процеси, що опитують датчики, мають один і гот же пріоритет. Процесу, що управляє електроживленням, необхідно призначити більш високий пріоритет. Пріоритети процесів, керуючих системою сигналізації, і процесів, що опитують датчики, повинні бути однакові.

Систему охоронної сигналізації можна віднести скоріше до систем спостереження, чим до систем керування, тому що в ній немає виконавчих механізмів, що прямо залежать від значень датчиків. Прикладом системи керування може служити система керування опаленням будинку. Система спостерігає за температурними датчиками, установленими в різних кімнатах будинку і перемикає нагрівальні прилади залежно від реальної температури і температури, установленої в термореле. Термореле, у свою чергу, контролює опалювальний котел.

Архітектура процесів такої системи показана на рис. 8.7; у загальному вигляді вона виглядає подібно системі сигналізації.

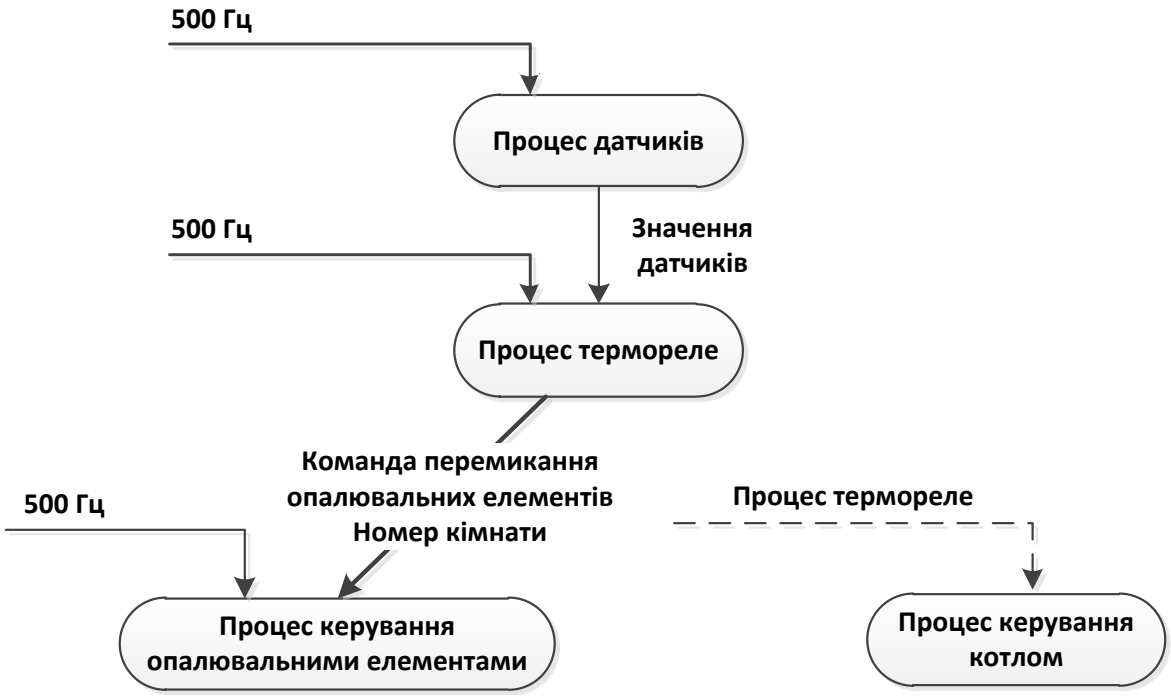

Рис. 8.7. Архітектура процесів системи в правління опаленням

4. Системи збору даних

Ці системи представляють інший клас систем реального часу, які зазвичай базуються на узагальненій архітектурній моделі. Такі системи збирають дані із сенсорів з метою їх наступної обробки і аналізу.

Для ілюстрації цього класу систем розглянемо модель, представлену на рис. 8.8. Тут зображена система, що збирає дані з датчиків, які вимірюють потік нейтронів у ядерному реакторі. Дані, зібрані з різних датчиків, містяться в буфер, з якого потім вилучаються і обробляються. На моніторі оператора відображається середнє значення інтенсивності потоку нейтронів.

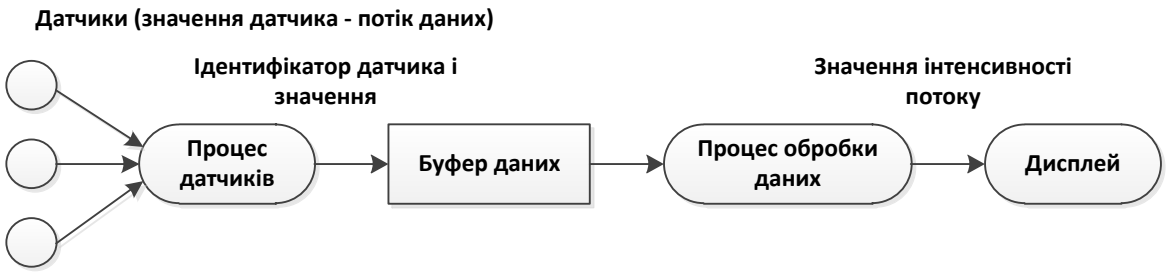

Рис. 8.8. Архітектура системи, спостереження за інтенсивністю потоку нейтронів

Кожний датчик пов'язаний із процесом, який перетворить аналоговий сигнал, що показує інтенсивність вхідного потоку, у цифровий. Сигнал разом з ідентифікатором датчика записується в буфер, де зберігаються дані. Процес, відповідальний за обробку даних, бере їх з буфера, обробляє і передає процесу відображення для виводу на операторну консоль.

У системах реального часу, що ведуть збір і обробку даних, швидкості виконання та періоди процесу збору і процесу обробки можуть не збігатися. Якщо обробляються великі обсяги даних, збір даних може виконуватися швидше, чим їх обробка. Якщо ж виконуються тільки прості обчислення, швидше відбувається обробка даних, а не їх збір.

Щоб згладити різницю у швидкостях збору і обробки даних, у більшості подібних систем для зберігання вхідних даних використовується кільцевий буфер. Процеси, що створюють дані (процеси-виробники), поставляють інформацію в буфер. Процеси, що обробляють дані (процеси-споживачі), беруть дані з буфера (рис. 8.9).

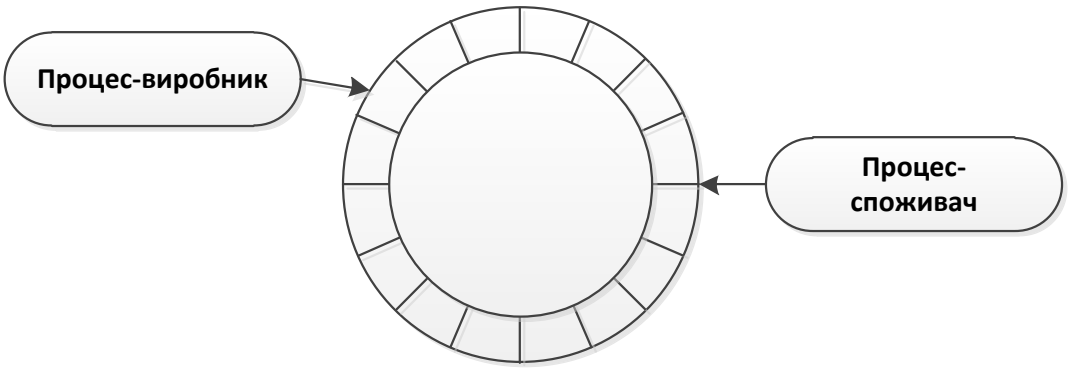

Рис. 8.9. Кільцевий буфер у системі збору даних

Очевидно, необхідно запобігти одночасному доступу процесу-виробника і процесу-споживача до тих же елементів буфера. Крім того, система повинна відслідковувати, щоб процес-виробник не додавав дані в повний буфер, а процес-споживач не забирав дані з порожнього буфера.

У лістингу 8.2 показана можлива реалізація буфера даних як об'єкта Java. Значення в буфері мають тип **Sensorrecord** (запис даних датчика). Визначено два методи - **get** і **put:** метод **get** бере елементи з буфера, метод **put** додає елемент у буфер. При оголошенні типу **Circularbuffer** (кільцевий буфер) програмний конструктор задає розмір буфера.

Лістинг 8.2. Реалізація кільцевого буфера

```
class Circularbuffer
{
   int bufsize;
   Sensorrecord [] store;
   int numberofentries = 0;
   int front = 0, back = 0;

   Circularbuffer (int n) {
        bufsize = n;
        store = new Sensorrecord [bufsize];
   } //Circularbuffer

   synchronized void put (Sensorrecord rec) throws
Interruptedexception
   {
        if(numberofentries == bufsize)
             wait ();
        store [back] = new Sensorrecord (rec.sensorld,rec.sensorval);
        back = back + 1;
        if(back == bufsize)
             back = 0;
        numberofentries = numberofentries + 1;
        notify();
   }//put
synchronized Sensorrecord get () throws Interruptedexception
{
   Sensorrecord result = new Sensorrecord(-1,-1);
   if(numberofentries == 0)
        wait ();
   result = store [front];
   front = front + 1;
   if(front == bufsize)
        front = 0;
   numberofentries = numberofentries - 1;
   notify();
   return result;
}// get
   }//Circularbuffer
```

Модифікатор **synchronized,** пов'язаний з методами **get** і **put,** указує на те, що дані методи не повинні виконуватися паралельно. При виклику одного із цих методів система реального часу блокує екземпляр об'єкта, щоб у це ж час не

відбувся виклик іншого методу і відповідно не проводилися маніпуляції на тій же ділянці буфера. Виклики методів **wait** і **notify** з методів **get** і **put** гарантують, що вхідні дані не можна покласти в повний буфер або взяти з порожнього буфера. Метод **wait** викликає потік і припиняє, поки інший потік за допомогою методу **notify** не відправить йому повідомлення про зняття очікування. При виклику методу **wait** блокування на захищені дані об'єкта знімається. Метод **notify** відновляє виконання одного з потоків, що очікують.

Список літератури

1. Лешек А. Мацяшек. Аналіз і проектування інформаційних систем за допомогою UML 2.0 / Лешек А. Мацяшек. - М.: Вільямс, 2008. - 816 с.

3. Шалыто А.А. SWITCH - технологія. Алгоритмізація і програмування завдань логічного керування / А.А. Шалыто. - Спб.: Наука, 1998. - 628 с.

4. Карло Гецци. Основи інженерії програмного забезпечення / Карло Гецци, Мехди Джазайери, Дино Мандриоли. - Спб.: Бхв-Петербург, 2005. - 832 с.

5. Соммервилл, Иан. Інженерія програмного забезпечення, 6-е видання, пров. з англ. А.А. Минько. - М.: Видавничий будинок "Вільямс", 2002. - 624 с.

6. Эдвард Йордон. Орієнтований^-орієнтований-об'єктно-орієнтований аналіз і проектування систем / Эдвард Йордон, Карл Аргила. - М.: Лорі, 2010. - 264 с.

7. Эрик Эванс. Предметно – орієнтоване проектування (DDD). Структуризація складних програмних систем / Эрик Эванс, пров. з англ. В. Бродів. - К.: Вільямс, 2010. - 448 с.

8. Гама Є. Приймання об'єктно-орієнтованого проектування. Паттерны проектування / Є. Гама, Р. Хелм, Р. Джонсон, Дж. Влиссидес. пер. с англ. А. Слинкин. - К.: Пітер, 2007. - 366 с.

9. Joey F. George. Object-Oriented Systems Analysis and Design. [Joey F. George, Dinesh Batra, Joseph S. Valacich, Jeffrey A. Hoffer]; (2nd Edition). - Prentice Hall; 2 edition (October 27, 2006). - 550 p.

10. Noushin Ashrafi. Object Oriented Systems Analysis and Design / Noushin Ashrafi, Hessam Ashrafi. - Prentice Hall; 1 edition (September 20, 2008). - 648 p.

11. Michele Lanza. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems / Michele Lanza, Radu Marinescu. - Springer; Softcover reprint of hardcover 1st ed. 2006 edition (December 2, 2010). - 220 p.

12. Grady Booch. Object-Oriented Analysis and Design with Applications (3rd Edition) / [Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, Kelli A. Houston. - Addison-Wesley Professional; 3 edition (April 30,

2007). - 720 p.

13. Jeffrey Whitten. Systems Analysis and Design Methods / Jeffrey Whitten, Lonnie Bentley. - Mcgraw-hill/Irwin; 7th edition (November 22, 2005). - 768 p.

14. Alan Dennis. Systems Analysis and Design / Alan Dennis, Barbara Haley Wixom, Roberta M. Roth. - Wiley; 4 edition (December 10, 2008). - 576 p.

15. Соммервилл, Иан. Інженерія програмного забезпечення, 6-е видання.: Пер. с англ. / Соммервилл, Иан. - М.: Видавничий будинок "Вільяму", 2002. - 624 с.

16. Смірнов В.В. Технологія проектування програмних систем. Лекції / В.В. Смірнов, Н.В. Смірнова. - Кіровоград: КНТУ.