МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Вінницький національний технічний університет

“ЗАТВЕРДЖУЮ”

Директор ІнАЕКСУ

_________ А.С.Васюра
(підпис) (ініціали, прізвище)

“____”___________ 20__ р.

Методичні вказівки практичних занять з дисципліни:

Програмування

Укладач:
доцент каф. АІВТ
Довгалець С.М.

Ухвалено методичною
комісією ІнАЕКСУ

Протокол №_____від__________

Голова методичної комісії

________ А.С.Васюра
(підпис) (ініціали, прізвище)

Методичні вказівки
затверджено на засіданні
кафедри АІВТ.

Протокол №6 від 08.12.2009р.

Завідувач кафедри АІВТ

________ Р.Н. Квєтний
(підпис) (ініціали, прізвище)

У даному методичному виданні висвітлюються основи конструювання програмного забезпечення. Зокрема розкриваються поняття про методи ефективного та оптимального кодування в першу чергу на мовах високого рівня. Наводяться дані щодо якісного стилю програмування, механізмам відладки та тестування програм.

ЗМІСТ

Вступ 4
1. Основи конструювання програмного забезпечення (КПЗ). 5
1.1. Місце КПЗ в життєвому циклі програмної системи 5
1.2. Фундаментальні складові конструювання програмного забезпечення 8
1.3. Мінімізація складності 8
1.4. Очікування змін. 9
1.5. Конструювання з можливістю перевірки. 9
2. Стандарти у конструюванні. 10
3. Високоякісне кодування. 14
3.1. Правила написання якісного коду. Рівень класів. 14
3.2. Принципи використання змінних. 16
3.3. Структурне програмування. 21
4. Удосконалення програмного забезпечення 23
4.1. Рефакторинг 23
4.1.1. Еволюція програми. 23
4.1.2. Поняття рефакторингу 24
4.1.3. Ознаки того, що потрібен ре факторинг 24
4.1.4. Рівні рефакторингу. 26
4.1.5. Безпечний рефакторинг. 28
4.1.5. Стратегії рефакторингу. 29
4.2. Якість конструювання. 30
4.2.1. Тестування коду розробником. 30
4.2.2. TDD (Test-Driven Development) 31
4.2.2. Переваги, які надає TDD. 33
4.2.3. Фреймворк JUnit 33
5. Практикум. 38
5.1. Рефакторинг в середовищі Eclipse. 38
5.2. Коректний та некоректний підхід - практичні приклади та зразки 38
5.3. Створення програм у відповідності з принципами написання якісного коду. 45
5.4. Unit-тестування. 46
5.5. Рефакторинг. 46
5.6. Система керування версіями Subversion (SVN).. 46
Література 48

ВСТУП

Базову складову професійної діяльності фахівців в галузі програмної інженерії формують вміння та навички конструювання програмного забезпечення. До складу обов'язкового обсягу практичних навичок фахівця напряму «Системна інженерія» повинні входити поняття про методи ефективного та оптимального в певному сенсі кодування алгоритмів в першу чергу на мовах високого рівня.

Програміст повинен генерувати не просто будь-який код, який працює, а і обов'язково володіти якісним стилем програмування, методами документування, застосовувати методи мінімізації коду, проводити ефективний пошук помилок, зокрема не явних, на етапі відладки та вміти якісно тестувати власний програмний продукт.

Розгляданню саме цих питань і присвячені навчально-методичні матеріали, в яких представлений матеріал, що направлений на набуття насамперед практичних навичок ефективного програмування.

1. ОСНОВИ КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

1.1. Місце КПЗ в життєвому циклі програмної системи.

Розробка програмного забезпечення (ПЗ) – це складний процес, в який входить багато складових. В загальному випадку це:

- визначення проблеми;
- вироблення вимог;
- створення плану конструювання;
- розробка архітектури ПЗ, або високорівневе проектування;
- детальне проектування;
- кодування і відлагодження;
- блочне тестування;
- інтеграційне тестування;
- інтеграція;
- тестування системи;
- корегувальне супроводження.

Термін конструювання програмного забезпечення (software construction) описує детальне створення робочої програмної системи за допомогою комбінації кодування, верифікації (перевірки), модульного тестування (unit testing), інтеграційного тестування та відлагодження.

На рис.1 показано місце конструювання як частину кроків серед процесів, що проходять при побудові ПЗ.

Рис.1. Конструювання серед процесів побудови ПЗ

Процеси конструювання зображені всередині сірого еліпсу. Головними компонентами конструювання є кодування та відлагодження, однак воно включає і детальне проектування, блочне тестування, інтеграційне тестування та інші процеси.

Іноді конструювання називають "кодуванням" або "програмуванням". Кодування в даному випадку видається не найкращим терміном, так як воно має на увазі механічну трансляцію розробленого плану в команди мови програмування, тоді як конструювання є зовсім не механічним процесом і часто пов’язане з творчістю та аналізом. Сенс слів "конструювання" та "програмування" досить близький.

Дана область знань пов'язана з іншими областями. Найбільш сильний зв'язок існує з проектуванням (Software Design) і тестуванням (Software Testing). Причиною цього є те, що сам по собі процес конструювання програмного забезпечення зачіпає важливі аспекти діяльності з проектування й тестування. Крім того, конструювання відштовхується від результатів проектування, а тестування (у будь-якій своїй формі) передбачає роботу з результатами конструювання. Досить складно визначити межі між проектуванням, конструюванням і тестуванням, тому що всі вони пов'язані в єдиний комплекс процесів життєвого циклу і, в залежності від обраної моделі життєвого циклу і застосовуваних методів (методології), таке розділення може мати різний вигляд.

Хоча ряд операцій з проектування детального дизайну може відбуватися до стадії конструювання, великий обсяг такого роду проектних робіт відбувається паралельно з конструюванням або як його частина. Це є сутність зв'язку з областю знань "Проектування програмного забезпечення".

У свою чергу, протягом всієї діяльності з конструювання, інженери використовують модульне і інтеграційне тестування. Таким чином пов'язана дана галузь знань з "Тестуванням програмного забезпечення".

У процесі конструювання звичайно створюється більша частина активів програмного проекту - конфігураційних елементів (configuration items). Тому в реальних проектах просто неможливо розглядати діяльність по конструюванню у відриві від галузі знань "конфігураційного управління" (Software Configuration Management).

Так як конструювання неможливе без використання відповідного інструментарію і, ймовірно, дана діяльність є найбільш інструментально-насиченою, важливу роль у конструюванні грає область знань "Інструменти і методи програмної інженерії" (Software Engineering Tools and Methods).

Безумовно, питання забезпечення якості значимі для всіх галузей знань і етапів життєвого циклу. У той же час, код є основним результуючим елементом програмного проекту. Таким чином, явно напрошується і присутній зв'язок обговорюваних питань з областю знань "Якість програмного забезпечення" (Software Quality).

З пов'язаних дисциплін програмної інженерії (Related Disciplines of Software Engineering) найбільш тісний і природний зв'язок даної галузі знань

існує з комп'ютерними науками (computer science). Саме в них, звичайно, розглядаються питання побудови та використання алгоритмів і практик кодування. Нарешті, конструювання стосується і управління проектами (project management), причому, в тій мірі, наскільки діяльність з управління конструюванням важлива для досягнення результатів конструювання.

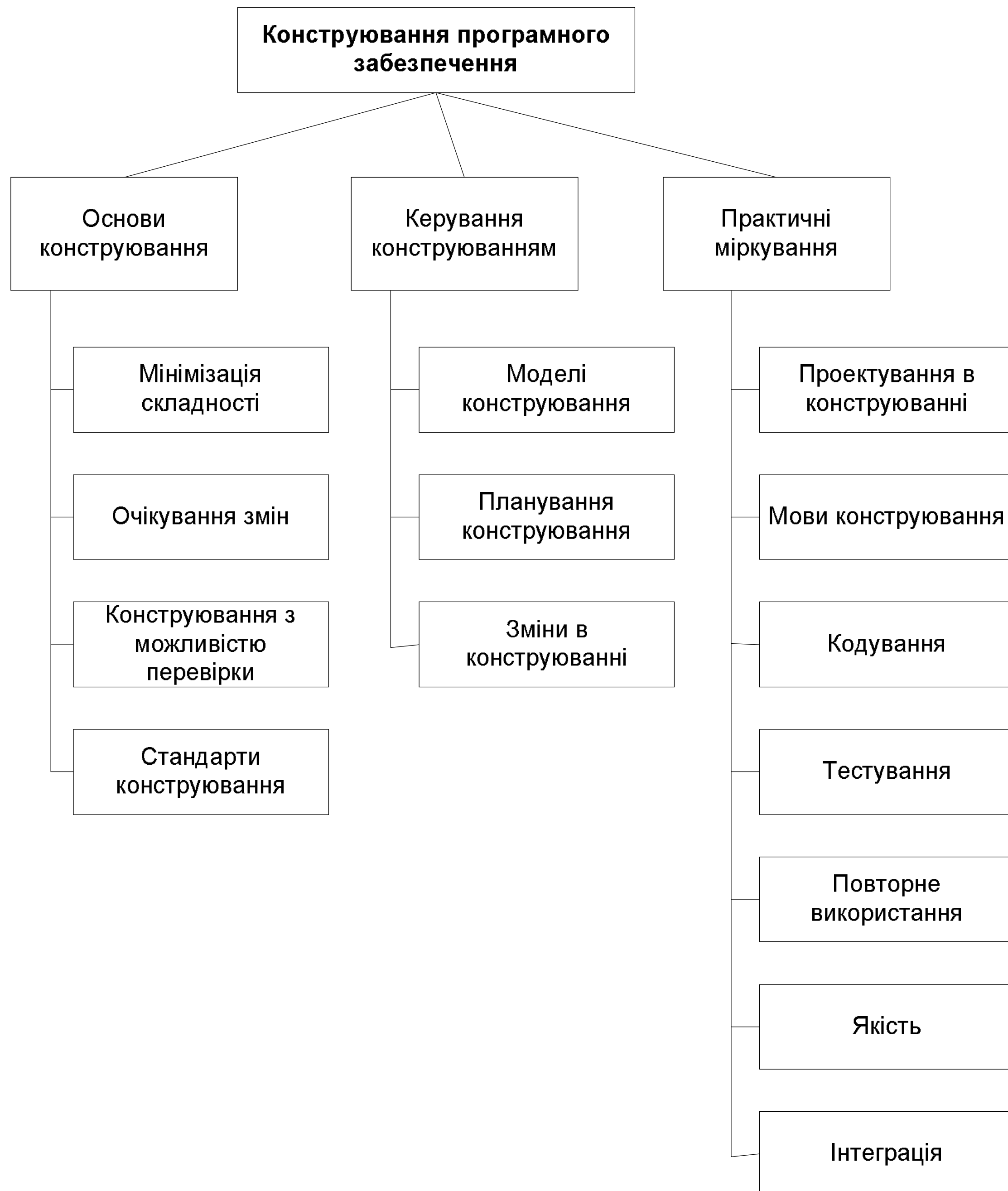


Рис.2. Область знань "Конструювання програмного забезпечення"

Далі наведено деякі конкретні задачі, що виникають в процесі розробки ПЗ, пов’язані з конструюванням:

- перевірка виконання умов, необхідних для успішного конструювання;
- визначення способів подальшого тестування коду;
- проектування та написання класів та методів;

- створення та присвоєння імен змінних та іменованих констант;
- вибір управляючих структур та організації блоків команд;
- блочне тестування, інтеграційне тестування і відлагодження власного коду;
- взаємний огляд коду та низькорівневих програмних структур членами групи;
- "шліфування" коду шляхом його ретельного форматування та коментування;
- інтеграція програмних компонентів, створених окремо;
- оптимізація коду, яка направлена на підвищення його швидкодії і зниження міри використання ресурсів.

З іншого боку, з видів діяльності, що проходять в процесі розробки ПЗ, до конструювання не відносяться: керування процесом розробки, виробки вимог, розробка високорівневої архітектури програми, проектування інтерфейсу користувача, тестування системи і її супроводження – для кожного з цих пунктів є своя наука.

1.2. Фундаментальні складові конструювання програмного забезпечення.

Фундаментальні основи конструювання програмного забезпечення включають:

- Мінімізація складності
- Очікування змін
- Конструювання з можливістю перевірки
- Стандарти у конструюванні

Перші три концепції застосовуються не тільки до конструювання, але й проектування, і лежать в основі сучасних методологій управління життєвим циклом програмних систем.

1.3. Мінімізація складності (Minimizing Complexity)

Основною причиною того, чому люди використовують комп'ютери в бізнес-цілях, є обмежені можливості людей в обробці і зберіганні складних структур і великих обсягів інформації, зокрема, протягом тривалого періоду часу. Це міркування є однією з основних рушійних сил у конструюванні програмного забезпечення: мінімізація складності. Потреба у зменшенні складності впливає на всі аспекти конструювання і особливо критична для процесів верифікації (перевірки) і тестування результатів конструювання, тобто самих програмних систем.

Зменшення складності у конструюванні програмного забезпечення досягається за допомогою звертання особливої уваги на створення простого коду, який легко читається, іноді на шкоду прагненню зробити його ідеальним (наприклад, з точки зору гнучкості або слідування тим чи іншим уявленням про

красу, витонченість коду, вправність тих чи інших прийомів тощо). Це не означає, що повинно обмежуватися застосування тих чи інших розвинених мовних можливостей використовуваних засобів програмування. Мається на увазі "лише" надання більшої значимості читаності коду, простоті тестування, прийнятного рівня продуктивності та задоволенню заданих критеріїв, замість постійного вдосконалення коду, не оглядаючись на терміни, функціональність і інші характеристики та обмеження проекту.

Мінімізація складності досягається, зокрема, за рахунок слідування стандартам, використанням низки специфічних технік кодування і підтримкою практик, спрямованих на забезпечення якості в конструюванні.

1.4. Очікування змін (Anticipating Changes)

Більшість програмних систем змінюються з плином часу. Причин цьому - безліч. Очікування змін є однією з рушійних сил конструювання програмного забезпечення. Програмне забезпечення не є ізольованим від зовнішнього оточення (як системного, так і з точки зору галузі діяльності, для автоматизації задач і проблем якого воно застосовується). Більш того, програмні системи є частиною середовища, що змінюється, і повинні змінюватися разом з нею, а, іноді, і бути джерелом змін самого середовища.

Очікування змін підтримується рядом технік кодування.

1.5. Конструювання з можливістю перевірки (Constructing for Verification)

"Конструювання для перевірки" (а саме такий сенс закладений в оригінальній назві даної підтеми) припускає, що побудова програмних систем повинна вестися таким чином, щоб сама програмна система допомагала вести пошук причин збоїв, будучи прозорою для застосування різних методів перевірки (і, до речі, внесення необхідних змін), як на стадії незалежного тестування (наприклад, інженерами-тестувальниками), так і в процесі операційної діяльності - експлуатації, коли особливо важлива можливість швидкого виявлення та виправлення помилок що виникають.

Серед технік, спрямованих на досягнення такого результату конструювання:

- Огляд, оцінка коду (code review)
- Модульне тестування (unit-testing)
- Структурування коду для і спільно з застосуванням автоматизованих засобів тестування (automated testing)
- Обмежене застосування складних або важких для розуміння мовних структур

2. СТАНДАРТИ У КОНСТРУЮВАННІ (STANDARDS IN CONSTRUCTING)

Стандарти, які безпосередньо застосовуються при конструюванні, включають:

- Комунікаційні методи (наприклад, стандарти форматів документів і оформлення вмісту)
- Мови програмування і відповідні стилі кодування (наприклад, Java Language Specification, що є частиною стандартної документації JDK - Java Development Kit і Java Style Guide, що пропонує загальний стиль кодування для мови програмування Java)
- Платформи (наприклад, стандарти програмних інтерфейсів для викликів функцій операційного середовища, такі як прикладні програмні інтерфейси платформи Windows - Win32 API, Application Programming Interface або .NET Framework SDK, Software Development Kit)
- Інструменти (не в термінах середовищ розробки, але можливих засобів конструювання - наприклад, UML як один зі стандартів для визначення нотацій для діаграм, що представляють структура коду і його елементів або деяких аспектів поведінки коду)

Використання зовнішніх стандартів. Конструювання залежить від зовнішніх стандартів, пов'язаних з мовами програмування, використовуваним інструментальним забезпеченням, технічними інтерфейсами і взаємним впливом конструювання програмного забезпечення та інших галузей знань програмної інженерії (в тому числі, пов'язаних дисциплін, наприклад, управління проектами). Стандарти створюються різними органами, наприклад, консорціумом OMG - Object Management Group (зокрема. Стандарти CORBA, UML, MDA, ...), міжнародними організаціями з стандартизації такими, як ISO/IEC, IEEE, TMF, ..., виробниками платформ, операційних середовищ і т.д. (Наприклад, Microsoft, Sun Microsystems, CISCO, NOKIA, ...), виробниками інструментів, систем управління базами даних і т.п. (Borland, IBM, Microsoft, Sun, Oracle, ...). Розуміння цього факту дозволяє визначити достатній і повний набір стандартів, які застосовуються у проектній команді або організації в цілому.

Кожна програмна система протягом свого існування проходить з певною послідовністю фази або стадії від задуму до його втілення в програми, експлуатацію та вилучення. Така послідовність фаз називається життєвим циклом розробки (Software life cycle processes). На кожній фазі відбувається певна сукупність процесів, кожен з яких породжує певний продукт, використовуючи певні ресурси.

Усі продукти всіх процесів програмної інженерії являють собою певні описи — тексти вимог до розробки, погодження домовленостей, документацію, тексти програм, інструкції з експлуатації тощо.

Головними ресурсами програмної інженерії, які визначають ефективність її розробок, є час і вартість цих розробок.

Різновиди діяльності, котрі становлять процеси життєвого циклу програмної системи, зафіксовано в міжнародному стандарті ISO/IEC 12207 : 1995—0801 : Informational Technology - Software life cycle processes.

Згідно з наведеним стандартом, усі процеси поділено на три групи:

- головні процеси;
- допоміжні процеси;
- організаційні процеси.

До головних процесів віднесено такі:

- процес придбання, який ініціює життєвий цикл системи та визначає організацію-покупця автоматизованої системи, програмної системи або сервісу;
- процес розроблення, який означає дії організації — розробника програмного продукту;
- процес постачання, який означає дії під час передавання розробленого продукту покупцеві;
- процес експлуатації, який означає дії з обслуговування системи під час її використання — консультації користувачів, вивчення їхніх побажань тощо;
- процес супроводження, який означає дії з керування модифікаціями, підтримки актуального стану та функціональної придатності, інсталяцію та вилучення версій програмних систем у користувача.

У свою чергу, до процесу розроблення входять такі процеси:

- інженерія вимог до системи;
- проектування;
- кодування й тестування.

До допоміжних процесів віднесено ті, що так чи інакше забезпечують якість продукту. Терміном якість продукту позначено сукупність властивостей, які зумовлюють можливість задоволення потреб замовника, котрий сформулював їх у формі вимог на розробку.

До організаційних процесів віднесено менеджмент розробки, створення структури організації, навчання персоналу, визначення відповідальності кожного з учасників процесів життєвого циклу розробки.

Стандарт ISO/IEC 12207:1995 - 0801: Informational Technology - Software life cycle processes є головним чинником визначення змісту діяльності у сфері програмної інженерії, і всі знання, яких потребують професіонали з програмної інженерії, формулюються стосовно процесів, визначених цим стандартом.

Зупинимося докладніше на процесах розробки програмного забезпечення, які в сукупності мають забезпечити шлях від усвідомлення потреб замовника до

передавання йому готового продукту. На цьому шляху виділяють низку характерних робіт:

- Визначення вимог. Збір та аналіз вимог замовника виконавцем і подання їх у нотації, яка є зрозумілою як для замовника, так і для виконавця.
- Проектування. Перетворення вимог до розробки в послідовність проектних рішень щодо способів реалізації вимог: формування загальної архітектури програмної системи та принципів її прив’язки до конкретного середовища функціонування; визначення детального складу модулів кожної з архітектурних компонент.
- Реалізація. Перетворення проектних рішень на програмну систему, яка реалізує такі рішення.
- Тестування. Перевірка кожного з модулів та способів їхньої інтеграції; тестування програмного продукту в цілому (так звана верифікація); тестування відповідності функцій працюючої програмної системи вимогам (requirements), поставленим до неї замовником (так звана валідація).
- Експлуатація та супроводження готової програмної системи.

Базовими вітчизняними нормативними документами в галузі програмної інженерії є:

- ДСТУ 2873-94. Системи обробки інформації. Програмування. Терміни та визначення.
- ДСТУ 2941-94. Системи оброблення інформації. Розроблення систем. Терміни та визначення.
- ДСТУ 4302:2004. Інформаційні технології. Настанови щодо документування комп’ютерних програм.
- ДСТУ ISO/IEC 12119:2003. Інформаційні технології. Пакети програм тестування і вимоги до якості.
- ДСТУ ISO/IEC 14764:2002. Інформаційні технології. Супроводження програмного забезпечення.
- ДСТУ ISO/IEC 90003:2006. Програмна інженерія. Настанови щодо застосування ISO 9001:2000 до програмного забезпечення (ISO/IEC 90003:2004, IDT)
- ДСТУ ISO/IEC TR 12182:2004. Інформаційні технології. Класифікація програмних засобів (ISO/IEC TR 12182:1998, IDT)
- ДСТУ ISO/IEC 14598-1:2004. Інформаційні технології. Оцінювання програмного продукту. Частина 1. Загальний огляд (ISO/IEC 14598-1:1999, IDT)
- ДСТУ ISO/IEC 15288:2005. Інформаційні технології. Процеси життєвого циклу системи (ISO/IEC 15288:2002, IDT)
- ДСТУ ISO/IEC 15939:2008. Інженерія систем і програмних засобів. Процес вимірювання.
- ДСТУ 3327-96. Методика випробування процесорів мов програмування. Загальні вимоги.

- ДСТУ ISO/IEC TR 14369:2003. Інформаційні технології. Мови програмування, їхнє середовище та системний інтерфейс. Настанова щодо підготовки незалежних від мов специфікацій послуг.
- ДСТУ 4072:2001. Інформаційні технології. Мови програмування, їхнє середовище та системний інтерфейс. Настанова щодо підготовки незалежних від мов виклик процедур.
- ДСТУ ISO/IEC 2382-15:2005. Інформаційні технології. Словник термінів. Частина 15. Мови програмування (ISO/IEC 2382-15:1999, IDT)
- ДСТУ 3008-95. "Документація. Звіти у сфері науки і техніки Структура і правила оформлення". К.: Держстандарт України,1995. – 75 с.
- ГОСТ 2.106-96. Единая система конструкторской документации. Текстовые документы. Изд. Офиц – К.: Госстандарт Украины, 1998. – 47 с.
- ГОСТ 2.109-73 ЕСКД. Основные требования к чертежам – М., 1978.
- ГОСТ 2.105-95. Единая система конструкторской документации. Общие требования к текстовым документам. Изд. Офиц – К.: Госстандарт Украины, 1996.
- ГОСТ 7.32-91. Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления

Використання внутрішніх стандартів. Певні стандарти, угоди та процедури можуть бути також створені усередині організації або навіть команди, що працює над проектом. Ці стандарти підтримують координацію між певними видами діяльності, групами операцій, мінімізують складність (у тому числі при взаємодії членів команди та за її межами), можуть бути пов'язані з питаннями очікування та обробки змін, ризиків і питаннями конструювання для перевірки та подальшого тестування. У поєднанні із зовнішніми стандартами, внутрішні стандарти покликані визначити загальні правила гри для всіх членів команди, домовившись про терміни, процедури та інші значущі угоди, незалежно від ступеня формалізації процесів конструювання зокрема і процесів життєвого циклу в загальному випадку.

3. ВИСОКОЯКІСНЕ КОДУВАННЯ.

3.1. Правила написання якісного коду. Рівень класів.

Клас – набір даних і методів, що мають спільну, цілісну, добре визначену сферу відповідальності. Однією з основних умов ефективного програмування є максимізація частини програми, яку можна ігнорувати при роботі над конкретним фрагментом коду. Клас – основний засіб досягнення даної цілі.

Не маючи явного уявлення про поняття абстракції, що використовується при побудові класів, програмісти часто створюють класи, які тільки називаються "класами", будучи насправді лише контейнерами, які вміщують дані та підпрограми, що погано узгоджуються між собою. Класи примітні тим, що дозволяють працювати з сутностями реально світу, а не з низькорівневими сутностями реалізації.

Переваги використання абстракції на рівні класів.

1) Можливість приховування реалізації.

2) Більш висока інформативність інтерфейсу.

3) Легкість оптимізації коду.

4) Легкість читання і зрозумілість коду.

5) Обмеження області використання даних рамками одного класу.

6) Можливість роботи з сутностями реального світу, а не низькорівневими деталями реалізації.

Якісний інтерфейс класів.

Перший і, мабуть, найважливіший етап розробки високоякісного класу – це створення адекватного інтерфейсу. Це має на увазі, зокрема, що клас повинен підтримувати хороший рівень абстракції, приховуючи деталі реалізації коду.

Інтерфейс класу повинен являти собою групу методів, чітко узгоджених один з одним. Розглянемо приклад невдалої побудови інтерфейсу класу (на C++). Інтерфейс, який виражав би погану абстракцію, складався б з набору різнорідних методів.

```
class Program {
public:
// відкриті методи
  void InitializeCommandStack();
  void PushCommand( Command command );
  Command PopCommandO;
  void ShutdownCommandStackO;
  void InitializeReportFormattingO;
  void FormatReport( Report report );
  void PrintReport( Report report );
  void InitializeGlobalDataO;
  void ShutdownGlobalData();
private:
...
};
```

Схоже, даний клас вміщує операції зі стеком команд, форматування звітів, друку звітів та ініціалізації глобальних даних. Важко побачити зв'язок між стеком команд, звітами та глобальними даними. Інтерфейс такого класу не формує узгоджену абстракцію. В даному випадку методи потрібно реорганізувати в більш чіткі класи, інтерфейси яких будуть являти собою більш вдалі абстракції.

Для того, щоб класи мали високоякісні інтерфейси, варто дотримуватись наступних принципів при їх проектуванні:

Виражати в інтерфейсі класу узгоджений рівень абстракції. В ідеалі, кожен клас повинен бути реалізацією тільки одного абстрактного типу даних (АТД). Приклад класу, який має неузгоджений інтерфейс:

```
class EmployeeCensus: public ListContainer {
public:
//абстракція рівня Employee
  void AddEmployee(Employee employee);
  void RemoveEmployee(Employee employee);
//абстракція рівня List
 Employee NextItemInList();
 Employee FirstItem();
 Employee LastItem();
private:
  ...
}
```

Даний клас виражає 2 АТД – Employee та ListContainer. Але чи потрібно, щоб інформація про використання контейнера була частиною абстракції? Зазвичай, це є деталлю реалізації. яку потрібно приховувати.

```
class EmployeeCensus {
public:
// Абстракція, що формується всіма цими методами, тепер відноситься до рівня Employee
  void AddEmployee(Employee employee);
  void RemoveEmployee(Employee employee);
  Employee NextEmployee();
  Employee FirstEmployee();
  Employee LastEmployee();
private:
//Факт використання List прихований
  ListContainer m_EmployeeList;
...
}
```

Деякі програмісти можуть стверджувати, що спадкування від ListContainer зручно, тому що воно підтримує поліморфізм в потрібних випадках. Але цей аргумент не проходить головний тест на доцільність спадкування: чи використовується спадкування для моделювання відношення "є".

Всі відкриті методи повинні бути узгодженими між собою частинами інтерфейсу. Якщо представити відкриті методи класу як люк, що перешкоджає попаданню води в човен, то неузгоджені відкриті методи – це щілини. Вода не буде протікати через них так швидко, як через відчинений люк, але з часом човен все ж потоне. На практиці при змішуванні рівнів абстракції саме так і відбувається. По мірі зміни програми змішані абстракції роблять її все менш і менш зрозумілою, поки код не стає взагалі загадковим.

Надавати методи разом з протилежними до них методами, якщо потрібно. При проектуванні класу потрібно перевірити кожен відкритий метод на предмет того, чи потрібний протилежний йому.

Прибирати сторонню інформацію в інші класи. Іноді стає помітно, що одні методи класу працюють з однією половиною даних, а інші – з іншою. Це значить, що в класі приховуються два різних класи, тому його потрібно розділити.

Побоюватись порушення цілісності інтерфейсу при зміні класу.

Не включати в клас відкриті члени, які погано узгоджуються з абстракцією інтерфейсу. Додаючи новий відкритий метод в клас, завжди потрібно спитати себе, чи узгоджується він з абстракцією, що формує клас. Якщо ні, потрібно знайти інший спосіб внесення змін, який дозволяє зберегти узгодженість абстракції.

Розглядати абстракцію і зв'язність разом. Поняття абстракції і зв'язності (cohesion) тісно пов'язані: інтерфейс класу, який представляє хорошу абстракцію, зазвичай має хорошу зв'язність. І навпаки, класи, що мають високу зв'язність, зазвичай представляють хорошу абстракцію, хоча даний зв'язок виражений слабше.

3.2. Принципи використання змінних.

Грамотне оголошення змінних

Деякі мови програмування підтримують неявне оголошення змінних (наприклад, Visual Basic, JavaScript тощо), тобто якщо в програмі використовується ще не оголошена змінна, компілятор автоматично оголосить її.

Неявне оголошення змінних – дуже небезпечна можливість в мові програмування. Вона може привести до плутанини в змінних, імена яких схожі або до плутанини з типами даних. Наприклад:

```
Dim acctNum as Long
...
acctNo=85647
```

Мови програмування, які вимагають оголошення змінних, заставляють програміста бути більш уважним з даними своєї програми, що є однією з найбільших переваг такого роду мов. В мовах без обов'язкового оголошення змінних рекомендується включати в компіляторі опцію вимоги оголошення змінних, якщо така є (як наприклад у Visual Basic). Рекомендується

оголошувати всі змінні, які планується використовувати, незалежно від того, чи вимагає це компілятор. Крім того, потрібно використовувати можливості середовища програмування по аналізу коду (наприклад, eclipse підкреслює жовтим змінні, які були оголошені, але не використані; по різному відображає поля і локальні змінні; виділяє всі використання змінної, на імені якої стоїть курсор).

Принципи ініціалізації змінних

Неправильна ініціалізація змінних є одним з найбільших джерел помилок в програмах. При неправильній ініціалізації проблеми пояснюються тим, що змінна може мати не те початкове значення, яке очікує програміст. Це може статись з однієї з причин:

- Змінній не було присвоєно значення. Відповідно, вона має те випадкове значення, яке знаходилось у відповідних комірках пам'яті при запуску програми.
- Значення змінної застаріло. Колись змінній було присвоєно значення, але воно втратило актуальність.
- Одним частинам змінної були присвоєні значення, а іншим ні.

Для правильної ініціалізації рекомендовано дотримувати наступних основних принципів.

1) Ініціалізуйте кожну змінну при її оголошенні. Ініціалізація змінних при їх оголошенні – проста методика захисного програмування і хороша страховка від помилок ініціалізації.

2) Ініціалізуйте кожну змінну там, де вона використовується вперше. Visual Basic та деякі інші мови не дозволяють ініціалізувати змінну при її оголошенні. В результаті, код може набувати вигляду, коли змінні спочатку оголошуються, а потім ініціалізуються, і це відбувається далеко від місця першого використання даних змінних.

Приклад поганої ініціалізації змінних (Visual basic)

```
' оголошення всіх змінних
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean
' ініціалізація всіх змінних
accountIndex = 0
total =0.0
done = False
...
' використання змінної accountIndex
...
' використання змінної total
...
' використання змінної done
While Not done
...
```

Краще ініціалізувати кожну змінну ближче до місця її першого використання:

```
Dim accountIndex As Integer
accountIndex = 0
' використання змінної accountIndex
...
Dim total As Double
total = 0.0

'використання змінної total
...
Dim done As Boolean
done = False
' використання змінної done ...
While Not done
```

Другий варіант кращий за перший з декількох причин. Поки виконання першого прикладу дійде до фрагменту, де використовується змінна done, її значення може бути змінено. Навіть якщо при написанні програми це не так, не можна гарантувати, що це не відбудеться в майбутньому при зміні програми. Крім того, в першому прикладі всі змінні ініціалізуються в одному місці, через що складається враження, що всі вони використовуються впродовж всього методу, тоді як насправді змінна done використовується тільки в його кінці. Нарешті, в результаті внесення змін в програму, що дуже навіть може статись, код, що використовує змінну done, може опинитись в циклі, що вимагатиме ініціалізації змінної заново кожного разу. Бачимо, що перший приклад слабше захищений від помилок, ніж другий.

Ці два фрагменти ілюструють принцип близькості: групуйте пов'язані дії разом. Цей принцип стосується також близькості коментарів до коду, що вони описують, близькість коду налаштування циклу до самого циклу тощо.

3) В ідеальному випадку потрібно оголошувати та ініціалізовувати змінну безпосередньо перед першим зверненням до неї. Не всі мови це дозволяють.

4) По мірі можливості потрібно оголошувати змінні як final чи const.

5) Потрібно приділяти особливу увагу лічильникам та акумуляторам.

Змінні i, j, k, sum, total часто грають роль лічильників чи акумуляторів. Нерідко програмісти забувають задати нульове значення лічильнику чи акумулятору перед його черговим використанням.

6) Потрібно ініціалізувати поля класу в його конструкторі.

7) Потрібно уважно слідкувати за повідомленнями компілятора – він попереджає про не ініціалізовані змінні.

Одиничність мети кожної змінної

Потрібно використовувати кожну змінну з єдиною метою. Іноді виникає спокуса використати одну змінну в двох різних місцях для розв’язування двох різних задач. Зазвичай, таким змінним доводиться давати невдале ім’я у відповідності з однією з її цілей, або використовувати для обох задач тимчасову змінну (наприклад, temp). Наприклад:

```
temp = Sqrt( b*b - 4*a*c);
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

Тут складається враження, що змінні temp в першій частині програми і в другій якось пов’язані між собою, хоча це не так. Краще було б написати так:

```
discriminant = Sqrt( b*b - 4*a*c);
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...
oldRoot = root[0];
root[0] = root[1];
root[1] = temp;
```

Потрібно уникати змінних, що мають прихований сенс. Навіть якщо змінна з подвійним сенсом зрозуміла автору програми, вона може бути незрозумілою іншим. Наприклад, якщо додатне число в змінній bytesWritten означає число байтів, записаних у вихідний файл, а від’ємне – номер диску, на який здійснюється запис, то отримуємо так звану "гібридну зв’язність" ("hybrid coupling"), що є недоліком програми. Тому, в даному випадку доцільно використовувати для кожної величини окрему змінну.

Потрібно переконатись в тому, що використовуються всі оголошені змінні. Доведено, що невикористання оголошених змінних веде до збільшення кількості помилок. Компілятор java, наприклад, видає попередження про невикористані змінні.

Принципи вибору імен змінних

Найважливішим принципом іменування змінних полягає в тому, що ім’я повинно повністю і точно описувати сутність, яка представляється змінною, і, відповідно, легко читатись і легко запам’ятовуватись. Приклад:

Невдале іменування:

```
x= x-xx;
xxx= fido + SalesTax( fido );
x = x + LateFee(x1, x ) + xxx;
x = x + Interest(x1, x );
```

Вдале іменування:

```
balance = balance - lastPayment;
```

monthlyTotal = newPurchases + SalesTax(newPurchases);
balance = balance + LateFee(customerID, balance) + monthlyTotal;
balance = balance + Interest(customerID, balance);

Очевидно, що найкращим ім'ям є те, яке в словах повністю описує сенс змінної. Наприклад, змінну, що представляє число місць на стадіоні, можна назвати numberOfSeatsInTheStadium. Особливостями такого іменування є: позитивна – ім'я не потрібно розшифровувати, воно саме за себе говорить; негативна – ім'я занадто довге, щоб бути ефективним при практичному використанні.

Таблиця 1. Ефективні та неефективні імена

Сутність змінної	Вдалі імена	Невдалі імена
Сума, на яку на даний момент виписано чеки	runningTotal, checkTotal	written, ct, checks, CHKTTL, x, xl, x2
Швидкість потягу	velocity, trainVelocity, velocitylnMph	velt, v, tv, x, xl,x2, train
Поточна дата	currentDate, todaysDate	cd, current, c, x, xl, x2, date
Число рядків на сторінці	linesPerPage	lpp, lines, l, x, xl, x2

Хороше мнемонічне ім'я описує зазвичай проблему, а не її вирішення. Наприклад, запис даних про співробітників можна назвати inputRecord або employeeData. Перше – комп'ютерний термін, що виражає ідею введення даних. Друге відноситься до предметної області, тому має перевагу.

Оптимальна довжина змінної лежить, мабуть, десь між довжинами змінних x та maximumNumberOfPointsInModernOlympics. Занадто короткі імена страждають від недостатнього вираження сенсу, в той час, як занадто довгі роблять складнішим набір програми і можуть зіпсувати візуальну структуру. Вважається, що оптимальною є довжина ім'я 10-16 символів. Але це не значить, що всі імена повинні бути такими – просто, якщо є більш короткі імена, потрібно приділити увагу тому, чи досить ясно вони описують сутність змінної.

Таблиця 2. Вдалі та невдалі імена з точки зору довжини

Занадто довгі імена	numberOfPeopleOnTheUsOlympicTeam numberOfSeatsInTheStadium maximumNumberOfPointsInModernOlympics
Занадто короткі імена	n, np, ntm n, ns, nsisd m,mp, max, points
Те, що потрібно	numTeamMembers, teamMemberCount numSeatsInStadium, seatCount teamPointsMax, pointsRecord

Чи завжди короткі імена є невдалими? Ні, не завжди. Якщо змінній присвоюється коротке ім’я, наприклад, і, то сама довжина ім’я говорить про те, що змінна є другорядною і має обмежену область дії. Загалом, довгі імена краще присвоювати глобальним змінним, або таким, які рідко використовуються, а коротші – локальним змінним або змінним, що викликаються в циклах.

В багатьох програмах використовуються змінні, в яких зберігаються обчислені значення – суми, середні величини тощо. Доповнюючи ім’я специфікатором типу Total, Sum, Average, Max, Min, Record, String або Pointer, бажано ставити його в кінці. Такий підхід має декілька переваг. По-перше, найзначущіша частина ім’я стоїть на початку, перша читається і перша сприймається. По-друге, прийнявши дану конвенцію, можна уникнути плутанини, коли записуються семантично однакові імена синтаксично по-різному, наприклад, totalRevenue та revenueTotal. По-третє, імена типу revenueTotal, expenseTotal, revenueAverage, expenseAverage мають приємну для ока симетрію. Винятком з даного правила є позиція специфікатора Num. При розташуванні на початку ім’я специфікатор Num позначає загальне число, наприклад, numCustomers – загальне число замовників. Якщо ж він вказується в кінці ім’я, то позначає індекс: customerNum – номер поточного замовника. Іншою ознакою даної відмінності є буква s в кінці ідентифікатора numCustomers. Але навіть в цьому випадку специфікатор Num часто приводить до плутанини, тому краще замість нього застосовувати Count чи Total для позначення кількості та Index для позначення номера: customerCount, customerIndex.

3.3. Структурне програмування

Термін «структурне програмування» був введений в історичній статті «Structured Programming», представленої Едсжером Дейкстрою на конференції НАТО з розробки ПЗ в 1969 році [26]. З тих самих пір термін «структурний» застосовувався до будь-якої діяльності в області розробки ПЗ, включаючи структурний аналіз, структурний дизайн і структурний валяння дурня. Різні структурні методики не мали між собою нічого спільного, крім того, що всі вони створювалися в той час, коли слово «структурний» надавало їм більшої значущості.

Суть структурного програмування полягає в простій ідеї: програма повинна використовувати керуючі конструкції з одним входом і одним виходом. Така конструкція являє собою блок коду, в якому є тільки одне місце, де він може починатися, і одне - де може закінчуватися. У нього немає інших входів і виходів. Структурне програмування - це не те ж саме, що і структурне проектування зверху вниз. Воно відноситься тільки до рівня кодування. Структурна програма пишеться в упорядкованій, дисциплінованою манері і не містить непередбачуваних переходів з місця на місце. Ви можете читати її зверху вниз, і практично так само вона виконується. Менш дисципліновані підходи призводять до такого вихідного коду, який містить менш зрозумілу і

зручну для читання картину того, як програма виконується. Менша читабельність означає гірше розуміння і врешті гіршу якість програми. Головні концепції структурного програмування, що стосуються питань використання break, continue, throw, catch, return та інших тем, застосовуються до теперішнього часу.

Три базових компонента структурного програмування:

- Послідовність - це набір операторів, що виконуються по порядку. Типові послідовні оператори містять присвоювання і виклики методів.
- Вибір - це така керуюча конструкція, яка змушує оператори виконуватися вибірково. Найбільш частий приклад - вираз if-then-else. Виконується або блок if-then, або else, але не обидва одразу. Один з блоків «вибирається» для виконання. Оператор case - інший приклад керуючого елемента вибору. Оператор switch в C + + і Java, оператор select - все це приклади case. У кожному разі для виконання вибирається один з варіантів. Концептуально оператори if та case схожі. Якщо ваша мова не підтримує оператори case, ви можете емулювати їх за допомогою набору інших операторів
- Ітерація - це керуюча структура, яка примушує групу операторів виконуватися кілька разів. Ітерацію зазвичай називають «циклом». До ітерацій відносяться структури For-Next у Visual Basic і while та for в C + + і Java.

Основна теза структурного програмування свідчить, що будь-яка керуюча логіка програми може бути реалізована за допомогою трьох конструкцій: послідовності, вибору та ітерації [24]. Програмісти іноді надають перевагу мовним конструкціям, що збільшують зручність, але програмування, схоже, розвивається багато в чому завдяки обмеженню того, що ми можемо робити на наших мовах програмування. До введення структурного програмування використання goto вважалось дуже зручним, але код, написаний таким чином, виявився малозрозумілим і не піддавався супроводу. Вважається, що використання будь-яких керуючих структур, відмінних від цих трьох стандартних конструкцій, тобто break, continue, return, throw-catch і т. д., повинні розглядатися під критичним кутом зору.

Одна з причин, з якої стільки уваги приділялося керуючим структурам, полягає в тому, що вони вносять великий внесок у загальну складність програми. Неправильне застосування керуючих структур збільшує складність, правильне - зменшує її.

Дослідники в області обчислювальної техніки вже протягом двох десятиліть усвідомлюють важливість проблеми складності. Багато років тому Дейкстра попереджав про небезпеку складності: «Компетентний програміст повністю усвідомлює суворо обмежені розміри свого черепа, тому підходить до завдань програмування з усією можливою скромністю» [25]. Складність керуючої логіки має велике значення, тому що вона корелює з низькою надійністю і частими помилками [29].

4. УДОСКОНАЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

4.1. Рефакторинг

Існує широко поширена помилкова думка відносно того, що в процесі добре організованого процесу розробки програмної системи ретельно розробляються методичні вимоги і визначається незмінний список задач програми, проект системи відповідає заданим вимогам і, в результаті кодування може здійснюватись лінійно, з початку до кінця, коли більшість коду пишеться один раз, тестується і про нього можна забути. Відповідно до цієї точки зору, суттєва зміна коду відбувається лише під час супроводження ПЗ, тобто вже після випуску початкової версії програми. Насправді ж код суттєво еволюціонує вже впродовж етапу початкової розробки. Багато змін, потреба в яких видимою під час початкової розробки є, як мінімум, такими ж кардинальними, як зміни під час підтримки.

Навіть в проектах з добре побудованим процесом управління вимоги до системи змінюються в процесі її розробки. Дані зміни позначаються на розроблюваному коді, іноді дуже суттєво. Інший факт – сучасні методи розробки збільшують потенціал для зміни коду в процесі конструювання. В старих життєвих циклах фокус був на уникненні змін коду. Сучасні підходи відходять від повної передбачуваності в розробці коду – вони більш кодо-центровані, і в процесі життєвого циклу проекту код еволюціонує більше, ніж колись.

Обов’язково потрібно як писати початковий код, так і вносити зміни, пам’ятаючи про майбутні зміни в коді. "Основне правило еволюції програмного забезпечення" говорить, що впродовж еволюції внутрішня якість програми повинна покращуватись.

4.1.1. Еволюція програми

В еволюції програми, як і в еволюції живих організмів, деякі зміни (мутації) є корисними, в той час як багато інших можуть бути шкідливими. Ключовим у питанні еволюції програмного забезпечення є те, чи покращується якість коду, чи погіршується. Програміст повинен розглядати процес внесення змін в програму як можливість покращити якість її дизайну та коду. Якщо ж помічається деградація якості коду, то це однозначний сигнал про те, що еволюція програми йде неправильно.

Другим фактором, який впливає на еволюцію ПЗ є те, чи зміни вносяться під час розробки, чи під час підтримки. Ці зміни розрізняються за декількома параметрами. Зміни під час розробки вносяться в більшості початковим розробником до того, як код є повністю забутим. Система ще не знаходиться в роботі, тому немає такого часового тиску, як під час підтримки. З тієї ж причини, зміни під час розробки дешевші, тому що система знаходиться в більш гнучкому динамічному стані. Дані обставини диктують стиль внесення змін, який відрізняється від того, що використовується під час підтримки.

4.1.2. Поняття рефакторингу

Ключовим методом виконання "основного правила еволюції програмного забезпечення" є рефакторинг. За визначенням Мартіна Фаулера це "зміна, внесена у внутрішню структуру програмного забезпечення для більшої його зрозумілості і здешевлення внесення змін без зміни зовнішньої поведінки програми". Слово “refactoring” в сучасному програмуванні походить від поняття “factoring” в структурному програмуванні, яке означало як можна більшу декомпозицію програми на складові частини.

4.1.3. Ознаки того, що потрібен рефакторинг

Перерахуємо основні ознаки ("попереджувальні знаки" – Мартін Фаулер називав їх "smells"), що вказують на потребу в рефакторингу:

- Є дублювання коду
- Метод занадто довгий
- Цикл занадто довгий або занадто велика вкладеність циклів
- Клас має погану внутрішню зв'язність
- Інтерфейс класу не забезпечує достатній рівень абстракції
- Список параметрів занадто довгий
- Зміни, що вносяться в клас, мають сегментний характер
- Зміни вимагають паралельну модифікацію багатьох класів
- Різні ієрархії класів повинні модифікуватися паралельно
- Пов’язані між собою дані, що використовуються разом, не організовані в один клас
- Метод використовує більше членів іншого класу, ніж свого
- Базовий тип перевизначено
- Клас не має достатньої функціональної навантаженості
- В ланцюгу викликів методів є ланцюг параметрів, що передаються
- Проміжний об’єкт не робить нічого власного
- Клас занадто зв’язаний з іншим класом
- Метод має погане ім’я
- Поля є public
- Клас-нащадок використовує тільки невелику кількість з методів класу-предка
- Коментарі використовуються для пояснення складного коду
- Використовуються глобальні змінні
- Метод потребує код ініціалізації перед викликом або пост-код після виклику
- Програма вміщує код, який здається таким, що "може знадобитись колись"

Розглянемо деякі з них.

Дублювання коду

Воно майже завжди означає, що при початковому проектуванні коду не було правильно проведено декомпозицію. Дублювання коду заставляє

програміста вносити паралельні зміни. Це суперечить широко відомому принципу DRY - Don’t Repeat Yourself.

Метод занадто довгий

В об’єктно-орієнтованому програмуванні методи величиною більші за екран рідко бувають потрібні і зазвичай означають спробу впхнути принципи структурного програмування в об’єктно-орієнтовану програму.

Одним зі шляхів покращення системи є збільшення її модульності – збільшення кількості добре визначених, добре іменованих методів, які добре роблять одну визначену річ. Якщо метод стає зрозумілішим при перенесенні частини його в окремий метод, потрібно створювати цей метод.

Інтерфейс класу не забезпечує достатній рівень абстракції

Навіть класи, які на початку мали раціонально побудований інтерфейс, можуть втратити свою цілісність з плином часу. Інтерфейс класу має тенденцію до розпливання в процесі розробки, особливо коли в клас вносяться термінові зміни. Врешті інтерфейс класу починає негативно впливати на зрозумілість програми.

Зміни, що вносяться в клас, мають сегментний характер

Іноді клас вирішує 2 або більше чітко визначених задач. Якщо так трапляється, програміст модифікує або одну частину класу, або іншу, але дуже рідко зміни зачіпають обидві одночасно. Це явна ознака того, що клас повинен бути розбитим на декілька в чіткій відповідності з задачами.

Пов’язані між собою дані, що використовуються разом, не організовані в один клас.

Якщо є набір змінних, які багаторазово використовуються разом, варто подумати над об’єднанням їх в клас.

В ланцюгу викликів методів є ланцюг параметрів, що передаються

Передача параметрів методу для того, щоб той передав їх як параметри в інший метод, може бути і добре, і ні. Потрібно подивитись, чи така передача узгоджується з абстракцією, вираженою в інтерфейсі кожного з методів. Якщо абстракція для кожного методу є відповідною, то і передача параметрів є відповідною, якщо ні, потрібно подумати над відповідністю інтерфейсів.

Метод має погане ім’я

Якщо виявлено, що метод має невідповідне ім’я, потрібно виправити його в місці опису та всіх місцях використання. Потрібно зробити це зразу при виявленні, тому що з часом це буде зробити ще важче.

Поля є public

public полів потрібно уникати, тому що вони роблять нечіткою лінію між інтерфейсом та реалізацією, порушуючи принцип інкапсуляції і обмежуючи гнучкість в майбутньому. Тому потрібно приховувати доступ до полів за public методами.

Коментарі використовуються для пояснення складного коду

Коментарі є важливими, але вони не повинні слугувати для пояснення поганого коду. Поганий код не потрібно пояснювати – його потрібно переписувати.

Метод потребує код ініціалізації перед викликом або пост-код після виклику

Код наступного виду є "попереджувальним знаком":

```
WithdrawalTransaction withdrawal;
withdrawal.SetCustomerId( customerId );
withdrawal.SetBalance( balance );
withdrawal.SetWithdrawalAmount( withdrawalAmount );
withdrawal.SetWithdrawalDate( withdrawalDate );
ProcessWithdrawal( withdrawal );
customerId = withdrawal.GetCustomerId();
balance = withdrawal.GetBalance();
withdrawalAmount = withdrawal.GetWithdrawalAmount();
withdrawalDate = withdrawal.GetWithdrawalDate();
```

Подібним знаком є випадок, коли створюється спеціальний конструктор для класу WithdrawalTransaction, який має параметрами значення для підмножини його властивостей:

```
withdrawal = new WithdrawalTransaction( customerId, balance,
withdrawalAmount, withdrawalDate );
ProcessWithdrawal( withdrawal );
delete withdrawal;
```

Кожен раз, коли зустрічається таке, потрібно задуматись, чи інтерфейс методу правильно представляє відповідну абстракцію. В даному випадку, напевне, метод *ProcessWithdrawal()* потрібно додати до класу WithdrawalTransaction, або список параметрів ProcessWithdrawal повинен бути змінений, щоб код мав вигляд:

```
ProcessWithdrawal( balance, withdrawalAmount, withdrawalDate );
```

Потрібно відмітити, що якщо подивитись навпаки, то можна побачити схожу проблему. Якщо виявляється, що існує об'єкт WithdrawalTransaction, а потрібно передавати значення декількох його властивостей в інший метод:

```
ProcessWithdrawal( withdrawal.GetCustomerId(), withdrawal.GetBalance(),
    withdrawal.GetWithdrawalAmount(), withdrawal.GetWithdrawalDate() );
```

то потрібно розглянути можливість зміни інтерфейсу методу ProcessWithdrawal, щоб передавати об'єкт, а не його окремі властивості. Будь-який з даних підходів може бути правильним чи ні – це залежить від того, чи абстракція інтерфейсу ProcessWithdrawal() є тим, що очікує отримати чотири окремі порції даних, чи цілий WithdrawalTransaction об'єкт.

4.1.4. Рівні рефакторингу

Термін рефакторинг іноді використовується в широкому значенні як будь-яке внесення змін в програму – з метою виправлення помилок, додавання функціональності, зміни дизайну тощо. Таке використання даного терміну не є точним і його потрібно намагатись уникати. Рефакторинг наголошує не просто на внесенні змін як таких, а на їх цілеспрямованості на досягнення покращення коду, що тягне за собою стійке покращення якості програми і запобігає широко відомій руйнівній спіралі програмної ентропії.

Приклади можливих кроків в процесі рефакторингу

Рефакторинги рівня даних

- Замінити магічне число іменованою константою.
- Замінити ім'я змінної на більш зрозуміле та інформативне.
- Відмовитись від проміжної змінної, записавши вираз в рядку використання його значення.
- Замінити вираз викликом підпрограми.
- Ввести проміжну змінну.
- Замінити змінну, яка використовується багаторазово для різних цілей, групою змінних, кожна з яких використовується з однією метою.
- Використовувати локальні змінні для локальних цілей, а не параметри.
- Перетворити змінну базового типу в об'єкт.
- Перетворити набір значень в клас.
- Перетворити набір значень в клас з підкласами.
- Перетворити масив на об'єкт.
- Інкапсулювати колекцію.
- Замінити запис (структуру) на клас.

Рефакторинги рівня операторів

- Провести декомпозицію логічного виразу.
- Перенести складний логічний вираз в добре іменовану булеву функцію .
- Об'єднати оператори, що знаходяться в різних частинах умовного оператора.
- Використовувати break чи return замість змінних, що служать для виходу з циклу.
- Виконувати return як тільки обчислено результат функції замість присвоєння значення, що повертається, спеціальній змінній.
- Заміняти умови поліморфізмом (особливо case-оператори).
- Створювати і використовувати null об'єкти замість перевірки на значення null.

Рефакторинги рівня методів

- Виокремити метод.
- Перенести код методу у місце виклику.
- Перетворити довгий метод у клас.
- Замінити простий алгоритм більш ефективним складнішим алгоритмом.
- Додати параметр.
- Вилучити параметр.
- Відділити операції читання від операцій запису.
- Об'єднати схожі методи, використавши параметри.
- Розділити методи, чия поведінка залежить від переданого параметра.
- Передавати як параметр цілий об'єкт, а не окремі поля.
- Передавати як параметр окремі поля, а не цілий об'єкт.
- Інкапсулювати приведення типу вниз.

Рефакторинги рівня реалізації класу

- Уникати багаторазового створення об’єктів шляхом використання вказівників на об’єкт, що розділяється декількома частинами програми (для великих, громіздких об’єктів).
- Уникати використання багатьох вказівників шляхом створення багатьох об’єктів (для малих об’єктів).
- Змінювати положення методів або даних в ієрархії.
- Виокремити спеціалізований клас в підклас.
- Скомбінувати схожий код в клас вищого рівня ієрархії.

Рефакторинги рівня інтерфейсу класу

- Перенести метод в інший клас.
- Розбити клас на два.
- Прибрати клас.
- Вилучити проміжний метод.
- Вилучити set()-методи для полів, які не можуть змінюватись.
- Приховувати методи, які не використовуються поза межами класу.
- Об’єднати клас-нащадок та клас-предок, якщо їх реалізація дуже подібна.

Рефакторинги рівня системи

- Робити копії даних, якими програміст не керує.
- Замінити односторонній зв’язок класів двостороннім.
- Замінити двосторонній зв’язок класів одностороннім.
- Створити фабрику об’єктів замість простого конструктора.
- Замінити коди помилок на виключні ситуації або навпаки.

4.1.5. Безпечний рефакторинг

Рефакторинг – ефективний і потужний інструмент програмування, але як і всі потужні інструменти при неправильному використанні він може нанести шкоду. Тому при рефакторингу потрібно користуватись наступними прийомами:

- Збереження початкового коду.
- Обмеження об’єму окремих видів рефакторингу.
- Виконання окремих видів рефакторингу по одному за раз.
- Складання списку дій, які програміст збирається виконати.
- Складання і підтримка списку видів рефакторингу, які потрібно виконати пізніше.
- Часте створення контрольних точок.
- Використання попереджень компілятора.
- Виконання регресивного тестування.
- Створення додаткових тестів.
- Виконання оглядів змін.
- Зміна підходу в залежності від ризикованості рефакторингу.

4.1.6. Стратегії рефакторингу

Число видів рефакторингу, вигідних для конкретної програми, майже нескінчене. Рефакторинг підлягає тому ж закону зниження вигоди, що й інші процеси програмування і до нього теж можна застосувати правило 80/20. Тому доцільно витрачати час на 20% видів рефакторингу, які забезпечать 80% вигоди. При визначенні найважливіших видів рефакторингу варто:

- Виконувати рефакторинг при створенні нових методів.
- Виконувати рефакторинг при створенні нових класів.
- Виконувати рефакторинг при виправленні дефектів.
- Виконувати рефакторинг модулів, в яких велика ймовірність виникнення помилок.
- Виконувати рефакторинг складних модулів.
- При супроводженні програми покращувати фрагменти, які доводиться виправляти.
- Визначити інтерфейс між акуратним і поганим кодом та перенести поганий код на інший бік цього інтерфейсу.

4.2. Якість конструювання

4.2.1. Тестування коду розробником

Існує багато видів тестування програмного забезпечення. Деякі з них зазвичай здійснюються розробником, а деякі спеціальним персоналом – тестувальниками.

Unit-тестування – це запуск підпрограми, класу чи малої програми, створеної одним програмістом або групою програмістів, окремо від системи вищого рівня.

Компонентне тестування – це запуск класу, пакету, малої програми чи іншої програмної одиниці, яка включає роботу групи програмістів або програмістських команд, окремо від системи вищого рівня.

Інтеграційне тестування – це комбінований запуск двох або більше класів, пакетів, компонентів, підсистем, які створені групою програмістів або програмістських команд. Даний тип тестування зазвичай починається тоді, коли завершено створення двох класів і продовжується до закінчення роботи над системою.

Регресійне тестування – це повторення раніше виконаних тестів з метою знаходження дефектів в ПЗ, яке раніше пройшло той же набір тестів.

Системне тестування – це запуск ПЗ в його кінцевій конфігурації, включаючи інтеграцію з іншими програмними та апаратними системами. Тут іде перевірка на безпеку, швидкодію, втрату ресурсів, проблеми таймінгу та інші проблеми, які не можуть бути виявлені на нижчих рівнях інтеграції

Зазвичай, розробник здійснює unit-тестування, компонентне тестування та інтеграційне тестування, яке може включати регресійне та системне тестування. Численні інші види тестування здійснюються спеціальним персоналом (наприклад, тестувальниками) і до нього рідко залучаються розробники (це включає бета-тестування, тестування на прийняття замовником (customer-acceptance tests), тестування продуктивності, конфігураційне тестування, тестування платформи, стресове тестування, тестування зручності використання (usability tests) тощо).

Тестування можна розділити на 2 категорії – тестування "чорного ящика" та тестування "білого ящика". Тестування "білого ящика" має на увазі, що той, хто тестує, знає про внутрішню структуру об’єкту тестування – це якраз підходить для тестування розробником. Не слід плутати поняття "тестування" та "відлагодження" – тестування є засобом виявлення помилок, відлагодження – засіб виявлення і виправлення причин помилок, які вже виявленні.

Рекомендується виділяти на тестування від 8 до 25% часу, що йде на розробку системи.

Коли створювати тести

Попереднє написання тестів, до написання коду, який вони тестують, дозволяє звести до мінімуму інтервал часу між моментами внесення дефекту та його виявлення/виправлення. Є й інші мотиви для попереднього написання тестів:

- створення тестів до написання коду вимагає тих же зусиль – просто змінюється порядок виконання цих двох етапів;
- попереднє написання тестів заставляє хоч трохи задуматись про вимоги і проект до написання коду, що сприяє покращенню коду;
- попереднє написання тестів дозволяє знайти проблеми в вимогах до написання коду, тому що складно створити тест для поганої вимоги.

Програмування з попередніми тестами є дуже ефективною методикою розробки програм, хоча воно теж має загальні обмеження і недоліки тестування розробником.

4.2.2. TDD (Test-Driven Development)

TDD є однією з основоположних частин XP (eXtreme Programming). Ідеї, покладені в основу XP, направлені на те, щоб зробити процес розробки ПЗ простішим і мати короткі, повторювані цикли розробки з постійним зворотнім зв'язком щодо стану програмної системи. TDD дає можливість розробляти складні програмні системи шляхом виконання простих ітерацій, в яких тести задають напрямок розвитку дизайну та реалізації системи.

Найскладнішим у використанні TDD є зміна уявлень програміста про способи побудови коду програми. Замість створення проекту, який описує, які програмні одиниці потрібно створити, створюються тести, кожен з яких описує, як повинна функціонувати маленька програмна одиниця системи. Ці тести визначають дизайн коду, який буде реалізовувати дані програмні одиниці системи.

TDD базується на двох методологіях – unit-тестуванні та рефакторингу. Загальна схема розробки при використанні TDD така:

1. Написати тест, який визначає, як, на думку автора, маленька програмна одиниця повинна поводитись.
2. Створити програмну одиницю якомога простішими засобами так, щоб вона пройшла тест.
3. Здійснити рефакторинг коду, зробивши його більш якісним і перевіряючи за допомогою тестів його правильність після внесення кожної зміни.

Оскільки TDD є ітеративним процесом, то потрібно повторювати дані кроки, поки не буде досягнуто бажаної якості коду.

При використанні TDD системні вимоги (наприклад, сформовані у вигляді use cases) декомпонуються в набір дій, виконання яких приведе до виконання вимог. Для кожної з даних дій спочатку пишеться unit-тест, який повинен перевірити правильність виконання даної дії. Разом зі створенням тесту визначаються чіткі критерії, за якими можна визначити, коли написано достатньо коду для виконання заданої дії. Однією з переваг попереднього написання тестів є те, що це допомагає чіткіше визначити бажану поведінку системи та дати відповіді на деякі основні питання її проектування.

Приклад. Однією з вимог до програми є визначити процент влучності баскетболіста за гру та за сезон. З вимоги можна визначити ряд дій, які повинна робити програмна одиниця:

- ідентифікувати гравця;
- ввести дані, необхідні для визначення проценту влучності в грі;
- ідентифікувати гру;
- обрахувати середній процент влучності за гру;
- обрахувати процент влучності за сезон.

Зробивши спрощене припущення, що гравці ідентифікуватимуться за ім'ям, а ігри за датою, можна написати код:

```
Player wally = PlayerList.getPlayer("Wally Wiffer");
// Get player
wally.addGameA("8/9/2003",15,30);
wally.addGameA("8/16/2003",25,75);
wally.addGameA("8/23/2003",4,16);
float gameA = wally.getGameA("8/9/2003"); // Get
average for game on 8/9
float seasonA = wally.getA(); // Get average season
assertEquals(0.5, gameA); // Check game average
calculation
assertEquals(0,36, seasonA); // Check season average
calculation
```

Написавши даний тест, маємо, що потрібно створити класи PlayerList та Player. В PlayerList повинен бути метод getPlayer(String), а в Player – addGameA(String, int, int), getGameA(String), getA().

Коли тест написано, він повинен бути запущений. Спочатку він, звичайно, повинен виконатись з помилками (оскільки код самої програми ще не створено) – це покаже, що сам тест запускається і функціонує. Потім пишеться програма до того часу, поки в тестах не зникнуть помилки. Потім проводиться рефакторинг; після кожного виду рефакторингу знову запускаються тести для перевірки, чи не зруйновано правильність коду.

Потім іде перехід до наступної дії програми з повторенням тих же кроків. Даний інкрементальний підхід заставляє програміста здійснювати постійний рефакторинг коду і приводить до того, що зменшується небезпека "занадто-проектування" (overdesign) системи і створення роздутого коду, оскільки код додається невеликими, прорефакториними порціями.

Коли розробляється більше і більше коду, число тестів швидко збільшується і створюється ціла система тестів, яку можна виконати в будь-який момент. Дана система тестів є важливою в TDD, тому що її можна використовувати для постійного моніторингу програмної системи і миттєвого виявлення проблем. Оскільки тести виконуються регулярно впродовж процесу розробки, вони допомагають оперативно виявляти і виправляти помилки. Це робить процес розробки ПЗ більш безпечним в плані помилок.

4.2.3. Переваги, які надає TDD

*1.Спрощена, інкрементна розробка***.** Програміст концентрується на розробці невеликих блоків коду, таким чином просуваючись в розробці. Це краще, ніж заздалегідь виконати велику роботу з проектування і потім виявити в проекті масу неузгодженостей. Також однією з основних переваг TDD є те, що вже на початку розробки одержується робоча система, нехай і з обмеженою функціональністю.

2.Можливість постійного регресійного тестування. В програмуванні часто зустрічається ефект доміно – коли невелика зміна в якійсь частині програми може потягти за собою непередбачувані зміни у функціонуванні всієї системи. Частиною TDD є проведення регресійного тестування після внесення кожної зміни в код, що дає можливість виявляти помилки зразу після їх внесення і таким чином захищає програміста від необхідності високовартісного виправлення помилок в майбутньому.

3.Покращена комунікація і централізація знань. Тести є хорошим методом документування коду – більш точним, ніж словесна або графічна форма і дає можливість розробнику описати ідеї дизайну своєї програми в формі, яка буде зрозумілою іншим.

4.Покращене розуміння вимог до програми, і, відповідно, покращений дизайн програми. Написання тестів до коду дає можливість поглянути на неї спочатку з точки зору користувача і уяснити вимоги до програми краще.

5.Покращена інкапсуляція і модульність. В процесі розробки програміст може внести в код небажані зв'язки між класами. Один з принципів TDD стверджує, що unit-тести повинно бути легко виконати. Це означає, що потрібно мінімізувати вимоги необхідні для запуску тестів. Фокусування на простоті тестів веде до покращення модульності класів, зменшуючи залежності.

6.Зменшення складності за рахунок не внесення надлишкового коду. Розробники часто вносять в код надлишкові методи в сподіванні, що ті потім знадобляться, намагаючись зробити програму більш гнучкою, більш придатною до модифікації в майбутньому. Це веде до збільшення складності програми. Наявність набору тестів дає програмісту більшу впевненість в позитивному результаті внесення змін (навіть досить суттєвих) в код і це приводить до відсутності необхідності в надлишковому коді.

4.2.4. Фреймворк JUnit

JUnit 4.x є тестовим фреймворком для Java, який, на відміну від попередніх версій, використовує анотації для позначення тестових методів.

Таблиця 3. Анотації JUnit

Анотація	Опис
@Test public void method()	Анотація @Test позначає, що даний метод є тестовим методом
@Before public void method()	Виконує method() перед кожним тестом. Цей метод може підготовлювати середовище тестування, наприклад, вводити тестові дані, ініціалізувати клас тощо)
@After public void method()	Тестовий метод, який запускається з тестом
@BeforeClass public void method()	Виконує метод перед початком всіх тестів. Це використовується для виконання дій, що вимагають великих витрат, наприклад, під'єднання до бази даних.
@AfterClass public void method()	Виконує метод після закінчення всіх тестів. Це може бути використано для виконання завершальних дій, наприклад для від'єднання від бази даних.
@Ignore	Тестовий метод буде ігноруватись; є корисним тоді, коли тестований код змінився, а тест ще не адаптували до нього чи встановлення середовища виконання тесту займає занадто багато часу.
@Test(expected=IllegalArgumentException.class)	Тестує, якщо метод генерує вказану виключну ситуацію.
@Test(timeout=100)	Виконується з помилкою, якщо метод виконується довше за 100 мілісекунд

Таблиця 4. Методи JUnit

Метод	Опис
fail(String)	Дає можливість методу виконатись не успішно; може бути корисним для визначення того, що певна частина коду не досягнута.
assertsEquals([String message], expected, actual)	Перевіряє, чи значення є однаковими.. Примітка: для масивів перевіряється рівність вказівників, а не значень комірок масивів
assertsEquals([String message], expected, actual, delta)	Використовується для типів float та double; delta вказує значення припустимої розбіжності між значеннями

Метод	Опис
assertNull([message], object)	Перевіряє, чи об’єкт null
assertNotNull([message], object)	Перевіряє, чи об’єкт не null
assertSame([String], expected, actual)	Перевіряє, чи обидва вказівники посилаються на один і той же об’єкт
assertNotSame([String], expected, actual)	Перевіряє, чи обидва вказівники не посилаються на один і той же об’єкт.
assertTrue([message], boolean condition)	Перевіряє, чи булевий вираз дорівнює is true.

Приклад створення JUnit тесту (в середовищі Eclipse).

Припустимо, в нашій програмі існує клас

```
package de.vogella.junit.first;
public class MyClass {
  public int multiply(int x, int y) {
        return x / y;
  }
}
```

Якщо натиснути правою кнопкою мишки на класі і вибрати New ->JUnit Test case, то з’явиться вікно рис.3.

Тести прийнято розміщувати в окремій папці поза межами src, наприклад, в папці test. Потрібно вказати Source folder, вибрати New JUnit 4 test та натиснути Next. Потім потрібно вибрати методи для тестування (рис.4).

Якщо це робиться в перший раз, то потрібно вказати підключення бібліотеки(рис. 5).

Створимо тест:

```
package de.vogella.junit.first;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class MyClassTest {
  @Test
  public void testMultiply() {
        MyClass tester = new MyClass();
        assertEquals("Result", 50, tester.multiply(10,
5));
  }
}
```

Натиснувши правою кнопкою мишки на класі тесту і вибравши Run-As-> Junit Test, одержуємо результат тестування у вікні JUnit (рис.6). Результат вказує на помилку. Виправивши в методі multiply рядок `return x/y;` на рядок `return x*y;`, одержимо правильний результат.

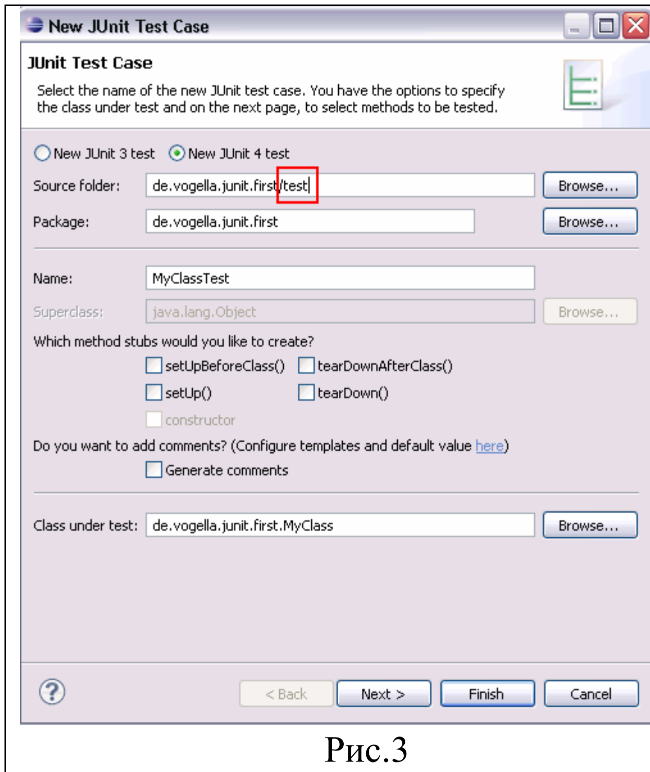


Рис.3

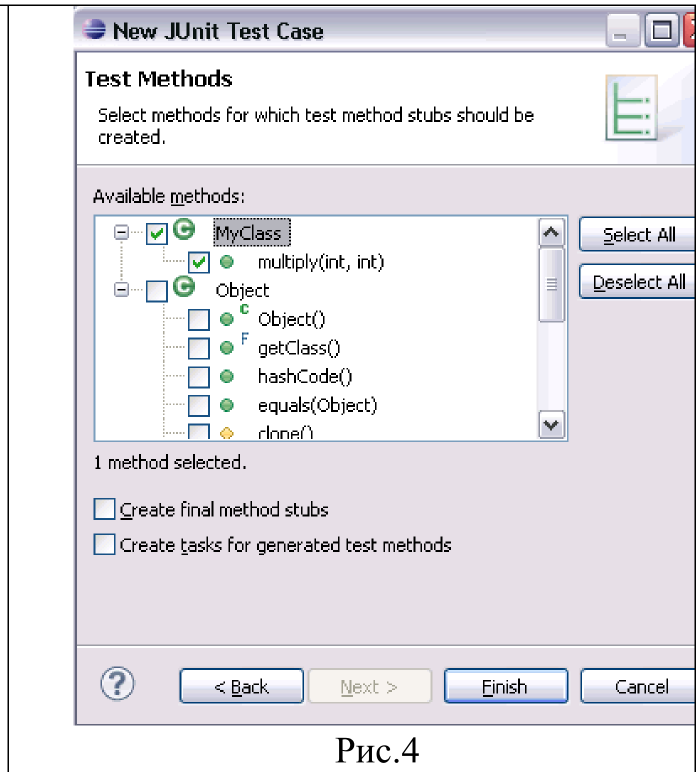


Рис.4

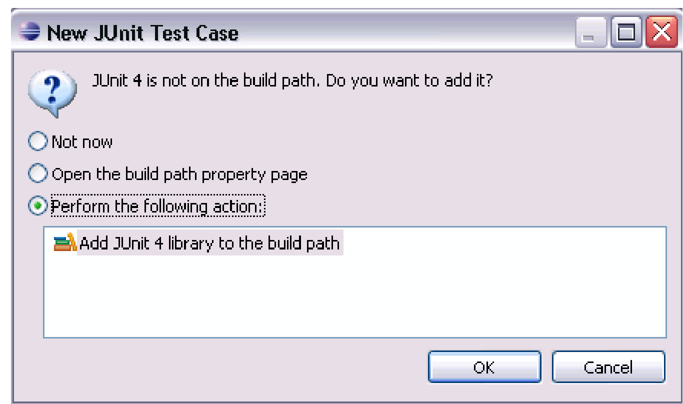
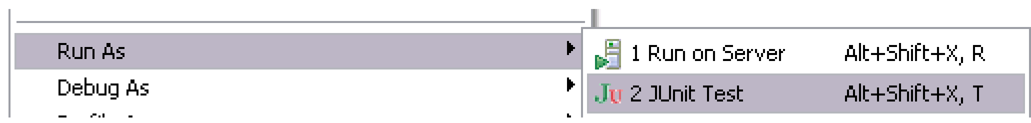


Рис. 5



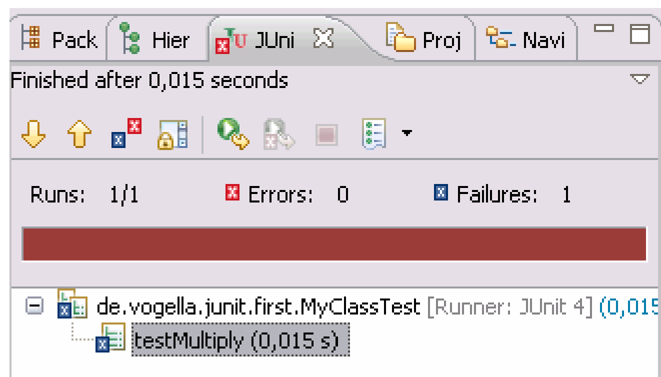


Рис.6

5. ПРАКТИКУМ.

5.1. Рефакторинг в середовищі Eclipse.

1. Створити два класи так, щоб в методі другого викликався метод першого класу. Перейменувати даний метод за допомогою команди рефакторингу Rename (Alt + Shift + R) в першому класі. Звернути увагу на те, як змінився другий клас.

2. Перенести створені два класи в інший пакет за допомогою команди рефакторингу Move.

3. За допомогою команди рефакторингу Change Method Signature (Alt + Shift + C) додати параметр до методу. Звернути увагу на те, як змінився виклик методу.

4. За допомогою команди рефакторингу Extract Method (Alt + Shift + M) розбити метод на два методи.

5. За допомогою команди рефакторингу Extract Local Variable (Alt + Shift + L) винести частину виразу окремо, присвоївши окремій змінній.

6. За допомогою команди рефакторингу Extract Constant зробити з виразу іменовану константу.

7. За допомогою команди рефакторингу Convert Local Variable to Field зробити з локальної змінної поле.

8. За допомогою команди рефакторингу Extract Superclass винести частину класу в новостворений клас-предок.

9. За допомогою команд рефакторингу Push Down та Pull Up перемістити поля з класу-предка в клас-нащадок та навпаки.

10. За допомогою команди рефакторингу Introduce Parameter Object замінити набір параметрів методу базових типів на один параметр-об’єкт.

5.2. Коректний та некоректний підхід - практичні приклади та зразки.

5.2.1. Використання іменованих констант.

Уявімо, в методі потрібно розташувати текстові поля рівно одне під одним, а справа від кожного текстового поля кнопку. Недоцільно для задання координат вказувати безпосередні числа, тут доцільно використати іменовані константи. Це дасть змогу швидко і без зусиль змінити розташування

елементів, а також робить програму більш зрозумілою. В даному випадку не потрібно боятись зайвих рядків коду, тому що зрозумілість програми і простота її модифікації мають величезне значення.

<table>
<tr><th>Некоректний підхід</th><th>Коректний підхід</th></tr>
<tr><td>

```
public void locateVisualElements (){
  for (int i=0;i<4;i++){
textFields[i].setBounds(40,60+i*60,120,50);
buttons[i].setBounds(190,60+i*60,110,50);
  }
}
```

</td><td>

```
final int TEXT_FIELDS_X=40;
final int ELEMENTS_Y=60;
final int BUTTONS_X=40;
final int ELEMENT_SHIFT=60;
final int BUTTONS_WIDTH=110;
final int TEXT_FIELDS_WIDTH=120;
final int ELEMENT_HEIGHT=50;
public void locateVisualElements (){
  for (int i=0;i<4;i++){
textFields[i].setBounds(TEXT_FIELDS_X,
      ELEMENTS_Y +i*
ELEMENT_SHIFT,
      TEXT_FIELDS_WIDTH,
      ELEMENT_HEIGHT);
    buttons[i].setBounds(BUTTONS_X,
      ELEMENTS_Y +i*
ELEMENT_SHIFT,
      BUTTONS_WIDTH,
      ELEMENT_HEIGHT);
  }
}
```

</td></tr>
</table>

5.2.2. Змінні

Імена змінних повинні виражати їхню сутність. Вважається, що оптимальною є довжина ім’я 10-16 символів. Крім того, існують різні конвенції іменування; деякі з правил іменування є обов’язковим для вживання в певних мовах. Наприклад, в мові Java імена змінних починаються з маленької букви, а кожне смислове слово всередині імені починається з великої.

Некоректний підхід	*Коректний підхід*

k=a+b;
c=met(k);
k+=c;
	totalPrice=wholesalePrice+profit;
	vat=calculateVat(totalPrice);
	totalPrice+=vat;

5.2.3. Методи

Параметри методів

Даний приклад ілюструє один з аспектів використання параметрів методів. Параметри методів, як правило, нечисленні. Дуже рідко можна зустріти метод, в якому більше 4 параметрів. Якщо в методі є багато параметрів одного й того ж типу, це неприпустимо – потрібно об'єднати їх в рамках однієї змінної, наприклад, масиву або колекції.

Некоректний підхід	*Коректний підхід*

public void inputData (JTextField tf1, JTextField tf2, JTextField tf3, JTextField tf4){
name=tf1.getText();
height=
Integer.parseInt(tf2.getText());
weight=
Integer.parseInt(tf3.getText());
sort= tf4.getText();
}
	public void inputData **(JTextField tf[])**{
	name=tf[0].getText();
	height=
	Integer.parseInt(tf[1].getText());
	weight=
	Integer.parseInt(tf[2].getText());
	sort= tf[3].getText();
	}

Довжина методів

Загальноприйнято, що довжина методу не повинна перевищувати одного екрану комп'ютера. Якщо написаний вами метод виявився занадто довгим, то його потрібно розбити на декілька методів, кожен з яких повинен мати ім'я, яке виражає сутність того, що він робить. Розбивати метод на декілька зручно за допомогою пункту Extract method засобів рефакторингу візуальних середовищ розробки програм.

Некоректний підхід

```
class StringProcessing{
...
private static String halfStringToUpperCase(
                                        String
initialString) {

 }
 public static void main(String[] args){
  JFrame frame=new JFrame("Laba 1 ");
  frame.setDefaultCloseOperation(

JFrame.EXIT_ON_CLOSE);
  JButton btn=new JButton("Start");
  Label infoLabel=new Label("Lababoratorna1",
                             Label.CENTER);
  final JTextField textIn=new JTextField();
  final JTextField textOut=new JTextField();
  frame.getContentPane().add(infoLabel);
  frame.getContentPane().add(textIn,

BorderLayout.NORTH);
```

Коректний підхід

```
class StringProcessing{
...
 private static String halfStringToUpperCase(
                                        String
initialString) {
   String firstHalf=new String();
   String secondHalf=new String();
   for(int i=0; i<initialString.length()/2;i++)
     firstHalf=firstHalf+initialString.charAt(i);
   for(int i=initialString.length()/2;
                        i<initialString.length();
i++)

secondHalf=secondHalf+initialString.charAt(i);
   firstHalf=firstHalf.toUpperCase();
   return  firstHalf+secondHalf;
 }
 public static void main(String[] args){
  JFrame frame=new JFrame("Laba 1 ");
  frame.setDefaultCloseOperation(
```

```
        frame.getContentPane().add(textOut,

BorderLayout.SOUTH);

frame.getContentPane().add(btn,BorderLayout.EAST);
        frame.setSize(450,100);
        frame.setLocation(300,250);
        frame.setVisible(true);
        frame.setResizable(false);
        btn.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e){
            String initialString = textIn.getText();
            String firstHalf=new String();
            String secondHalf=new String();
            for(int i=0; i<initialString.length()/2;i++)
           firstHalf=firstHalf+initialString.charAt(i);
            for(int                    i=initialString.length()/2;
i<initialString.length();

i++)
              secondHalf=secondHalf+initialString.charAt(i);
           firstHalf=firstHalf.toUpperCase();
           textOut.setText(halfStringToUpperCase(

firstHalf+secondHalf));
          }
        });
        ...
      }
    }
```

```
JFrame.EXIT_ON_CLOSE);
        JButton btn=new JButton("Start");
        Label infoLabel=new Label("Lababoratorna1",

Label.CENTER);
        JTextField textIn=new JTextField();
        JTextField textOut=new JTextField();
        frame.getContentPane().add(infoLabel);
        frame.getContentPane().add(textIn,

BorderLayout.NORTH);
        frame.getContentPane().add(textOut,

BorderLayout.SOUTH);

frame.getContentPane().add(btn,BorderLayout.EAST);
        frame.setSize(450,100);
        frame.setLocation(300,250);
        frame.setVisible(true);
        frame.setResizable(false);
        btn.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e){
            String symbols = textIn.getText();

textOut.setText(halfStringToUpperCase(symbols));
          }
        });
        ...
      }
    }
```

Принцип DRY (Don't Repeat Yourself)

Принцип DRY наголошує на тому, що в програмі не повинно бути повторюваних блоків коду, і, відповідно, не повинно бути випадків, коли в декілька частин програми зміни завжди повинні вноситись паралельно.

Некоректний підхід

```
public void makeTransfer(){
  try{
    userIsLoggedIn();
    userHasEnoughRights();
    userAccountIsActive();
    makeTransfer ();
  }
  catch(Exception e){
    System.out.println("Операція               не
дозволена
  для даного користувача");
  }
}
public void makeDeposit(){
  try{
    userIsLoggedIn();
    userHasEnoughRights();
```

Коректний підхід

```
public void checkUser() throws userException{
    userIsLoggedIn();
    userHasEnoughRights();
    userAccountIsActive();
}
public void makeTransfer(){
  try{
    checkUser()
    makeTransfer ();
  }
  catch(Exception e){
    System.out.println("Операція  не  дозволена
для
  даного користувача");
  }
}
public void makeDeposit(){
```

```
        userAccountIsActive();
        makeDepositOperation();
      }
      catch(Exception e){
        System.out.println("Операція не дозволена
    для даного користувача");
      }
    }
```

```
    try{
      checkUser()
      makeDepositOperation();
    }
    catch(Exception e){
      System.out.println("Операція не дозволена для
   даного користувача");
    }
  }
```

Чітка визначеність та одиничність мети методу

Потрібно намагатись уникати написання в одному методі блоків коду, що мають різне по суті призначення. Зокрема, комбінування в одному методі введення даних через засоби інтерфейсу користувача, обробку даних та виведення.

Некоректний підхід

```
class StringProcessing{
  public static void countLetters(char letter) {
    String initialString = textIn.getText();
    int counter=0;
    for(int i=0; i<initialString.length();i++)
      if (initialString.charAt(i)==letter)
        counter++;
    textOut.setText(""+counter);
  }
  public static void main(String[] args){
    JFrame frame=new JFrame("Laba 1 ");
    frame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE);
    JButton btn=new JButton("Start");
    Label infoLabel=new Label("Lababoratorna1",
Label.CENTER);
    JTextField textIn=new JTextField();
    JTextField textOut=new JTextField();
    frame.getContentPane().add(infoLabel);
    frame.getContentPane().add(textIn,
BorderLayout.NORTH);
    frame.getContentPane().add(textOut,
BorderLayout.SOUTH);
frame.getContentPane().add(btn,BorderLayout.EAST);
    frame.setSize(450,100);
    frame.setLocation(300,250);
    frame.setVisible(true);
    frame.setResizable(false);
    btn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        countLetters('a');
      }
```

Коректний підхід

```
class StringProcessing{
  public static int countLetters(String initialString,
                                 char
letter) {
    int counter=0;
    for(int i=0; i<initialString.length();i++)
      if (initialString.charAt(i)==letter)
        counter++;
    return  counter;
  }
  public static void main(String[] args){
    JFrame frame=new JFrame("Laba 1 ");
    frame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE);
    JButton btn=new JButton("Start");
    Label infoLabel=new Label("Lababoratorna1",
Label.CENTER);
    JTextField textIn=new JTextField();
    JTextField textOut=new JTextField();
    frame.getContentPane().add(infoLabel);
    frame.getContentPane().add(textIn,
BorderLayout.NORTH);
    frame.getContentPane().add(textOut,
BorderLayout.SOUTH);
    frame.getContentPane().add(btn,BorderLayout.EAST);
    frame.setSize(450,100);
    frame.setLocation(300,250);
    frame.setVisible(true);
    frame.setResizable(false);
    btn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        String initialString = textIn.getText();
        textOut.setText(""+countLetters(initialString,'a'));
```

```
    });
  }
}
```

```
      }
    });
  }
}
```

5.2.4. Інкапсуляція

Поля прийнято описувати як private, а для доступу до них створювати спеціальні get та set методи.

Некоректний підхід	*Коректний підхід*

public class Man {
public String name;
public int age;
...
}
public class Run{
public static void main(String[] args){
Man man=new Man();
man.name="John";
...
System.out.println("Name="+man.name);
}
}
	public class Man {
	private String name;
	private int age;
	public String getName(){
	return name;
	}
	public void setName(String name){
	this.name=name;
	}
	public int getAge(){
	return age;
	}
	public void setAge(int age){
	this.age=age;
	}
	...
	}
	public class Run{
	public static void main(String[] args){
	Man man=new Man();
	man.setName("John");
	...
	System.out.println("Name="+man.**getName**());
	}
	}

Поля прийнято описувати як private, особливо, якщо для них існують спеціальні public get та set методи. Якщо клас описаний як public, то його

конструктор теж варто зробити public, якщо немає іншого спеціального методу для створення об’єктів даного класу, наприклад, фабрики об’єктів.

Некоректний підхід	*Коректний підхід*
public class Dog {	public class Dog {
int age;	**private** int age;
String breed;	**private** String breed;
Dog {	**public** Dog {
...	...
}	}
public int getAge(){	public int getAge(){
return age;	return age;
}	}
public int getBreed(){	public int getBreed(){
return breed;	return breed;
}	}
public void setAge(int age){	public void setAge(int age){
this.age=age;	this.age=age;
}	}
public void setBreed(int breed){	public void setBreed(int breed){
this.breed=breed;	this.breed=breed;
}	}
public void barks(){	public void barks(){
...	...
}	}
}	}

5.3. Створення програм у відповідності з принципами написання якісного коду.

Метою даного практикуму є набуття навичок побудови програм у відповідності з принципами написання якісного коду, а саме за рівнями:
Рівень класів:

- розумна абстракція;
- якісний інтерфейс класу;
- розумне об'єднання класів в пакети.

Рівень методів:

- розумна причина для створення методу;
- вдале ім'я методу;
- розумний розмір методу;
- розумний список параметрів методу;
- обгрунтований тип значення, що повертається;
- обгрунтоване використання ключового слова final.

Рівень змінних:

- грамотне оголошення змінних;
- грамотна ініціалізація змінних;
- поміщення змінної у відповідну їй область видимості;
- одиничність мети кожної змінної;
- грамотне іменування змінної.

Рівень операторів:

- грамотний вибір умовного оператора (if або case);
- грамотний вибір оператора циклу;
- відповідне використання нестандартних структур керування (наприклад, рекурсії);
- подолання занадто великої вкладеності.

Рівень констант:

- завжди підставляти константи на місце "магічних чисел";
- вдале ім'я константи.

Загальний рівень:

- форматування коду.

Завдання: Створити програми, які реалізують наступну функціональність. При створенні програм користуватись правилами написання якісного коду.

1) Квадрат, який рухається всередині вікна по діагоналі і "відбивається" від стінок під кутом, "дзеркальним" до кута руху до стінки.
2) На екрані з'являються різнокольорові прямокутники з однаковою шириною і різною висотою, яка задається генератором випадкових чисел. При натисненні кнопки вишикувати дані прямокутники за зростанням висоти.
3) Дано рядок символів. Виявнити, чи є він коректною web-адресою.

5.4. Unit-тестування

Метою даного практикуму є набути навичок в проведенні тестування розробником на прикладі JUnit-тестування в середовищі розробки програм Eclipse.

Завдання: написати JUnit-тести для методів для створених раніше програм.

Хід роботи.

1. Взяти на вибір до 3 написаних вами раніше програми (наприклад, в курсі ООП) для подальшої роботи. Узгодити їх з викладачем.
2. Запустити середовище розробки Eclipse. Відкрити в ньому першу з вибраних програм.
3. Написати JUnit-тести для даної програми у відповідності з логікою програми.
4. Навмисне внести помилку в один з методів програми. Запустити JUnit-тести на виконання. Проглянути результат запуску.
5. Виправити внесену помилку. Виправити помилку, добившись того, щоб JUnit-тести виконувались правильно.
6. Повторити кроки 2-6 для двох інших програм.

5.5. Рефакторинг.

Метою даного практикуму є набуття навичок в проведенні рефакторингу коду. Оволодіти засобами проведення рефакторинг в середовищі розробки програм Eclipse.

Завдання: провести рефакторинг раніше створених вами програм.

Хід роботи.

1. Проглянути список ознак того, що потрібен рефакторинг, з лекційного матеріалу та список кроків рефакторингу.
2. Вибрати 4 з написаних вами програм (наприклад, в рамках курсу ООП). Здійснити рефакторинг даних програм з використанням засобів Eclipse у відповідності з заданими в лекційному матеріалі кроками рефакторингу. Вказати, які конкретні кроки рефакторингу були використані.

5.6. Система керування версіями Subversion (SVN).

Метою даного практикуму є набуття навичок у використанні системи керування версіями Subversion.

Завдання: виконати послідовність операцій SVN з метою їх опрацювання та засвоєння.

Хід роботи.

1. Створити репозиторій для власної програми.
2. В середовищі Eclipse створити java-проект та записати його в репозиторій.

3. Прочитати проект з репозиторію в іншу папку (за допомогою команди svn checkout), створивши таким чином другу локальну копію проекту. Після цього з'явиться можливість змоделювати ситуацію внесення паралельних змін в проект.
4. Відкрити другий проект в середовищі Eclipse.
5. Внести зміни в перший проект. Не забувайте, що для додавання, вилучення, перейменування, копіювання та переміщення файлів та папок в проекті, який працює з subversion, потрібно використовувати команди `svn add`, `svn mkdir`, `svn delete`, `svn copy` та `svn move`. Записати зміни на сервер за допомогою svn commit.
6. Синхронізувати другий проект з першим за допомогою команди svn update.
7. Внести різні зміни в один і той же файл в двох проектах. Записати зміни в першому проекті на сервер за допомогою svn commit. Спробувати записати зміни в другому проекті на сервер за допомогою svn commit. Вирішити конфлікт, використовуючи команду svn diff.
8. Створити дві гілки проекту в папці branches за допомогою команди svn copy.
9. Відкрити кожну з гілок як проект в Eclipse. Внести в кожну з гілок зміни, які не повторюються. Злити гілки назад в єдиний стовбуровий проект за допомогою svn merge.

ЛІТЕРАТУРА

1. ДСТУ 2873-94. Системи обробки інформації. Програмування. Терміни та визначення. - К.: Держстандарт України, 1994.
2. ДСТУ 2941-94. Системи оброблення інформації. Розроблення систем. Терміни та визначення. - К.: Держстандарт України, 1994.
3. ДСТУ 4302:2004. Інформаційні технології. Настанови щодо документування комп’ютерних програм. - К.: Держстандарт України, 2004.
4. ДСТУ ISO/IEC 12119:2003. Інформаційні технології. Пакети програм тестування і вимоги до якості. - К.: Держстандарт України, 2003.
5. ДСТУ ISO/IEC 14764:2002. Інформаційні технології. Супроводження програмного забезпечення. - К.: Держстандарт України, 2002.
6. ДСТУ ISO/IEC 90003:2006. Програмна інженерія. Настанови щодо застосування ISO 9001:2000 до програмного забезпечення (ISO/IEC 90003:2004, IDT) - К.: Держстандарт України, 2006.
7. ДСТУ ISO/IEC TR 12182:2004. Інформаційні технології. Класифікація програмних засобів (ISO/IEC TR 12182:1998, IDT) - К.: Держстандарт України, 2004.
8. ДСТУ ISO/IEC 14598-1:2004. Інформаційні технології. Оцінювання програмного продукту. Частина 1. Загальний огляд (ISO/IEC 14598-1:1999, IDT) - К.: Держстандарт України, 2004.
9. ДСТУ ISO/IEC 15288:2005. Інформаційні технології. Процеси життєвого циклу системи (ISO/IEC 15288:2002, IDT) - К.: Держстандарт України, 2005.
10. ДСТУ ISO/IEC 15939:2008. Інженерія систем і програмних засобів. Процес вимірювання. - К.: Держстандарт України, 2008.
11. ДСТУ 3327-96. Методика випробування процесорів мов програмування. Загальні вимоги. - К.: Держстандарт України, 1996.
12. ДСТУ ISO/IEC TR 14369:2003. Інформаційні технології. Мови програмування, їхнє середовище та системний інтерфейс. Настанова щодо підготовки незалежних від мов специфікацій послуг. - К.: Держстандарт України, 2003.
13. ДСТУ 4072:2001. Інформаційні технології. Мови програмування, їхнє середовище та системний інтерфейс. Настанова щодо підготовки незалежних від мов виклик процедур. - К.: Держстандарт України, 2001.
14. ДСТУ ISO/IEC 2382-15:2005. Інформаційні технології. Словник термінів. Частина 15. Мови програмування (ISO/IEC 2382-15:1999, IDT) - К.: Держстандарт України, 2005.
15. ДСТУ 3008-95. "Документація. Звіти у сфері науки і техніки Структура і правила оформлення". К.: Держстандарт України,1995. – 75 с.
16. ГОСТ 2.106-96. Единая система конструкторской документации. Текстовые документы. Изд. Офиц – К.: Госстандарт Украины, 1998. – 47 с.
17. ГОСТ 2.109-73 ЕСКД. Основные требования к чертежам – М., 1978.
18. ГОСТ 2.105-95. Единая система конструкторской документации. Общие требования к текстовым документам. Изд. Офиц – К.: Госстандарт Украины, 1996.

19. ДСТУ ГОСТ 7.1:2006. Система стандартів з інформації, бібліотечної та видавничої справі. Загальні вимоги та правіла складання. - К.: Держстандарт України, 2007. – 47 с.
20. ДСТУ ГОСТ 2.104:2006. ЕСКД. Основні написи. - К.: Держстандарт України, 2006.
21. Інформатика: Комп’ютерна техніка. Комп’ютерні технології. Посіб. / За ред. О.І.Пушкаря – К: Видавничий центр “Академія”, 2001. – 696с. (Альма-матер)
22. Макконнелл С. Совершенный код. Мастер-класс / Пер. с англ.- М.: Издательско-торговый дом "Русская Редакция"; СПб.: Питер, 2005.- 896 стр.: ил.
23. Основы Программной Инженерии (по SWEBOK). 3. Конструирование программного обеспечения. http://swebok.sorlik.ru/3_software_construction.html.
24. Bohm, Corrado; and Giuseppe Jacopini (May 1966). "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". Communications of the ACM 9 (5): 366–371. doi:10.1145/355592.365646
25. Dijkstra, E. W. (Aug 1972). "The Humble Programmer". Communications of the ACM 15 (10): 859–866. doi:10.1145/355604.361591. http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html. (EWD340) PDF, 1972 ACM Turing Award lecture
26. Dijkstra, E.W., "Structured Programming," Software Engineering Techniques, Buxton, J.N., and Randell, B., eds. Brussels, Belgium, NATO Science Committee, 1969.
27. B. Meyer, Object-Oriented Software Construction, second ed., Prentice Hall, 1997, Chap. 6, 10, 11.
28. Guide to the Software Engineering Body of Knowledge (SWEBOK). CHAPTER 4. SOFTWARE CONSTRUCTION. http://www.computer.org/portal/web/swebok/html/ch4K. Beck, Test-Driven Development: By Example, Addison-Wesley, 2002.
29. McCabe : Complexity Measure, IEEE Transacions on Software Engineering, Volume 2, No 4, pp 308-320, December 1976
30. M. Fowler and al., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2002.
31. Russell Gold, Thomas Hammell, Tom Snyder. Test Driven Development: A J2EE Example.- Apress, 2005.- 296 pages.