Антипатерни об’єктно-орієнтованого програмування

Запах коду (Code smell)

- Так називають симптоми програмного коду, які вказують на певну глибоку проблему

Запах коду (Code smell)

- Дублювання коду: ідентичний, або майже ідентичний код
- Довгий метод: метод, функція або процедура, яка стала дуже великою
- Великий клас: клас, який став занадто великим (God object)
- Заздрість (Envy Feature): клас, який надмірно використовує методи іншого класу
- Невідповідна інтимність (Inappropriate intimacy): клас, який має залежність від деталей реалізації іншого класу
- Відмова запиту (*Refused request)*: клас, який перевизначає методи базового класу таким чином, що контракт базового класу фактично ігнорується
- Лінивий клас (Lazy class, Freeloader) : клас, який робить дуже мало. Наприклад, перенаправляє запит до подібного класу (не Адаптер)
- Надмірно довгі ідентифікатори(Excessively long identifiers): використання ідентифікаторів, які неявні в архітектурі програмного забезпечення (>60 символів)
- Надмірне використання літералів: вони повинні бути закодовані як повноцінні стрічки, для поліпшення читабельності і уникнення помилок програмування

Надто багато коду в одну місці

- Проблема сприйняття та обробки інформації
  - Проблема попереднього слайду – саме така проблема :-)
  - Аналогія з GUI – можна ставити 3-10 компонент *(оптимально 5-7) в одному місці. Більше 20 компонент у меню, без жодних розділень та групувань – це помилка
  - Коли багато коду (Spaghetti code), його стає тяжко читати і розуміти
  - Погане розділення по предметній області
- Симптоми (Code smell)
  - Довгий метод: метод, функція або процедура, яка стала дуже великою
  - Великий клас: клас, який став занадто великим (God object)

Рішення довгого методу

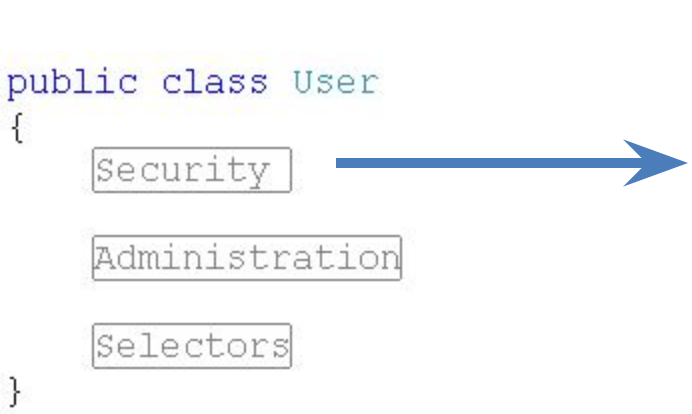
- Розбивають код по класах та функціях
  - Навіть якщо функція викликається один раз і з одного місця, але покращує читабельність, то краще її створити

```
public void CheckStatus() {
    PreCalculateStatuses();
    RunPriorityManager();
    AssignStatusesToRoles();
    AssignRolesToUser();
}
```

Код виглядає краще, коли метод розбитий на кілька функцій, кожна з яких робить щось одне, ніж коли у методі є повна суміш різної функціональності

Велика кількість функцій

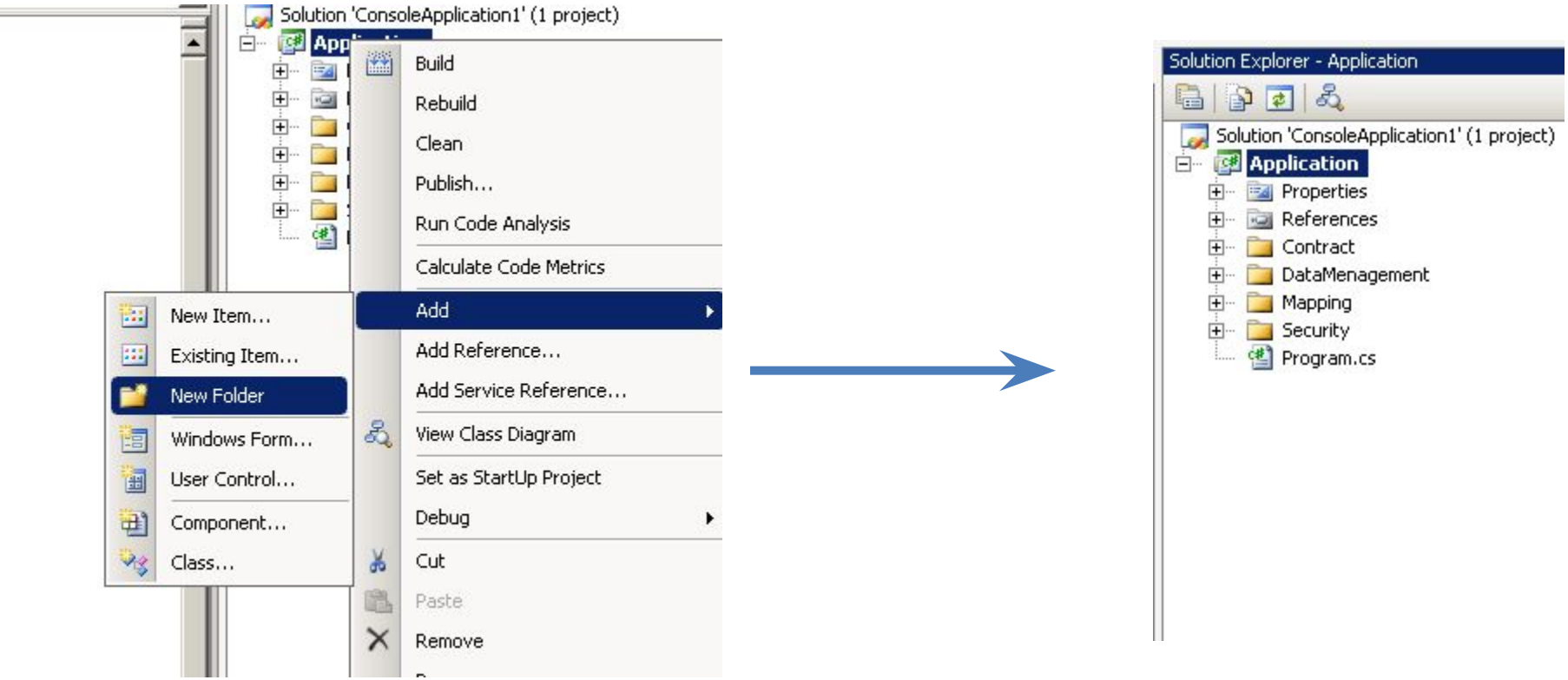
- Розкинути функції по кількох класах
  - Тільки тоді, коли вони логічно діляться на підгрупи відповідно до предметної області
  - Тільки тоді, коли вони не є сильнозв’язаними (багато викликів одна від одної)
- Використати #region / #endregion для групування функцій




```
#region Security

public int SecturePriority { get; set; }

public void CheckStatus() {
...
}

#endregion
```

Велика кількість класів

- Розбивають код по модулях та папках відповідно до предметної області



Запах надмірного старання. Lazy class

- Лінивий клас (Lazy class, Freeloader) : клас, який робить дуже мало. Наприклад, перенаправляє запит до подібного класу (не Адаптер)
  - Зворотня проблема – код ріжеться на надзвичайно малі куски
  - Створення додаткових, непотрібних рівнів між компонентами
  - Lazy function – те саме, але на рівні функції. Проблема може виникнути на рівні модуля, і т.д

Запах неправильних назв

- Code Convention – домовленості про назви змінних, класів, методів та ін.
- Повні назви, які нормально описують призначення класу, функції, тощо
- Code smell
  - Надмірно довгіі дентифікатори(Excessively long identifiers): використання ідентифікаторів, які наявні в архітектурі програмного забезпечення (>60 символів)
  - Надмірне використання літералів: вони повинні бути закодовані як повноцінні стрічки, для поліпшення читабельності і уникнення помилок програмування. Літерали можна використати лише як змінні всередині функцій
  - **Таємничий код** – використання абревіатур або незрозумілих назв
- Інші проблеми і підходи
  - Назви 20 класів починаються з одного довгого слова. Наприклад: SecurityRole, Security.User, Security.Access . Краще зробити додатковий простір (namespace Security) і звертатись до них коротшими іменами в межах цього простору. У зовнішніх класах, якщо імена співпадають можна повністю писати ім’я Security.User
  - Класи, які є подібними за функціональністю , мають схоже називатись

Запах неправильного проектування

- Основна проблема проектування – це розбити функціональність і дані по класах найкращим чином
- Заздрість (Envy Feature): клас, який надмірно використовує методи іншого класу
  - Клас сильно зв’язаний. У такому випадку, швидше за все розділення зовсім неправильне і потрібно зробити з 2 класів 1
- Невідповідна інтимність (Inappropriate intimacy): клас, який має залежність від деталей реалізації іншого класу
  - Порушується ідея слабкого зв’язування. Потрібна краща абстракція класу, або зміна структури класів
- Відмова запиту (*Refused request)*: клас, який перевизначає методи базового класу таким чином, що контракт базового класу фактично ігнорується
  - Порушується ідея абстракції

Антипатерни

- Антипатерни – це типові помилки у створенні програмних продуктів
- Більш точний опис проблем, ніж просто запах коду

Antipatterns leads to dark side

Антипатерни

- Об’єктно-орієнованого проектування
- Загальні у програмуванні
- Методологічні
- В керуванні розробленням ПЗ
- Організаційні
- Соціальні

Антипатерни ООП

- Базовий клас-утиліта (BaseBean): Спадкування функціональності з класу-утиліти замість делегування до нього
- Циклічні залежності (Cycle Dependency)
- Виклик предка (CallSuper): Для реалізації прикладної функціональності методу класу-нащадка потрібно в обов'язковому порядку викликати ті ж методи класу-предка
- Помилка порожнього підкласу (Empty subclass failure): Створення класу (в Perl), який не проходить «перевірку порожнечі підкласу» («Empty Subclass Test») через різну поведінку в порівнянні з класом, який успадковується від нього без змін
- Божественний об'єкт (God object): Концентрація занадто великої кількості функцій в одній частині системи (класі)
- Об'єктна клоака (Object cesspool): перевикористання об'єктів, що знаходяться в непридатному для перевикористання стані
- Полтергейст (комп'ютер) (Poltergeist): Об'єкти, чиє єдине призначення - передавати інформацію іншим об'єктам
- Проблема йо-йо (Yo-yo problem): Надмірна розмитість сильно пов'язаного коду (наприклад, виконуваного по порядку) по ієрархії класів
- Сінглетонізм (Singletonitis): Надмірне використання патерну синглетонами
- Паблік Морозов
-

Cycle Dependency
Циклічні залежності

- Циклічні залежності не є проблемою, якщо класи можуть бути в одному модулі
- Якщо вони мають бути у різних модулях (assembly), то циклічні залежності не дадуть змоги цього зробити

Cycle Dependency
Циклічні залежності

- Може привести до ефекту доміно, коли невеликі локальні зміни в одному модулі поширюється на інші модулі і досягають небажаного глобального ефекту
- Може також призвести до нескінченної рекурсії або інших непередбачених збоїв
- Ще може призвести до витоку пам'яті (memory leaks) шляхом запобігання автоматичному збиранню сміття

Циклічні залежності Вирішення

- Перебудова класів
- Спостерігач Observer
- Приклад: Перенесення циклічної залежності у абстракцію
  - Створення інтерфейсів класів у циклічній залежності
  - Винесення інтерфейсів в вищий по ієрархії модуль
  - Реалізація буде у класах – нащадках, які знаходитимуться у класах, нижчих по ієрархії

Об’єкт “Бог” (“God”object)

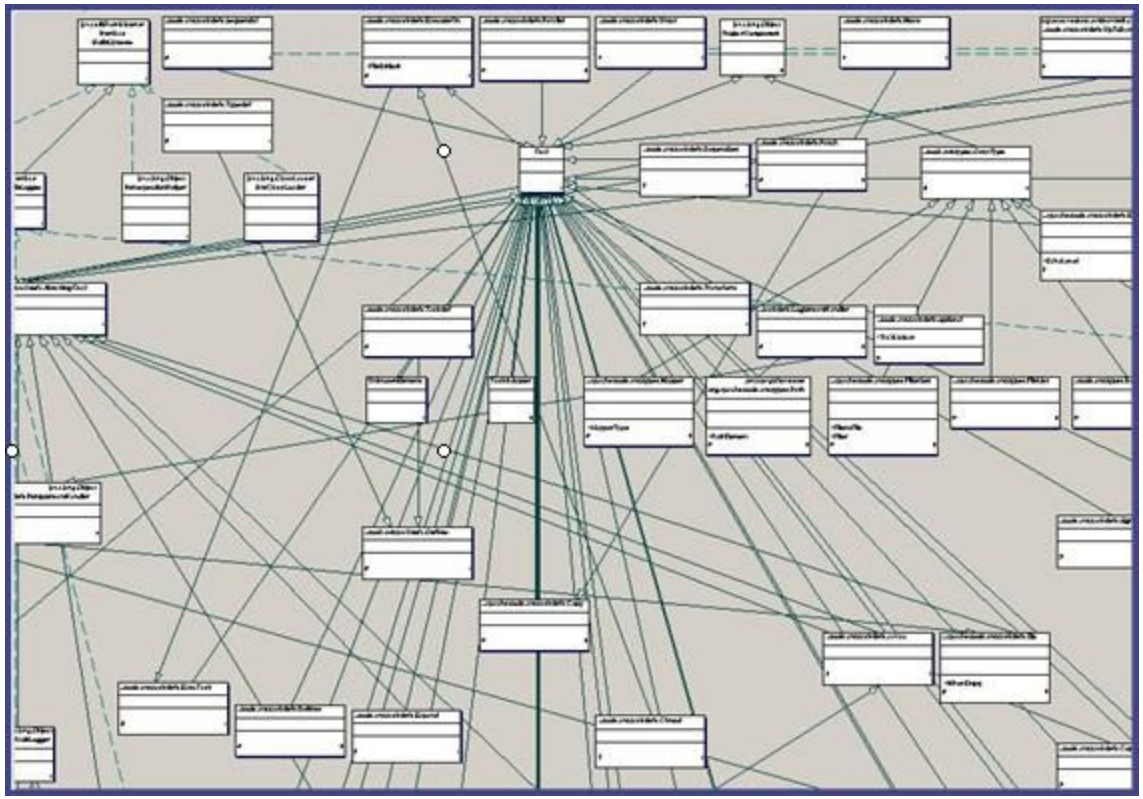
- В об'єктно-орієнтованому програмуванні божественний об'єкт (англ. God object) - це об'єкт, який зберігає в собі «занадто багато» або робить «занадто багато»

Об’єкт Бог – ознаки

- Спочатку він легкий і “хороший”
- З часом об’єкт розростається, стає громіздкий і “тяжкий”
  - Тяжко у ньому щось знайти та змінити
- Більшість об’єктів мають посилання на нього
- Він повинен “знати”всі типи об’єктів

Недоліки

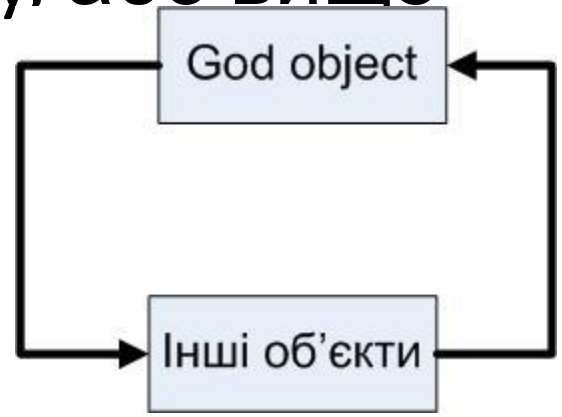
- Циклічні та “волохаті” залежності між God object і рештою об’єктів



Циклічні залежності

- God object знає про всі об’єкти, отже він має бути в доступній Assembly (яку використовують всі об’єкти)
- Але він робить операції зі всіма об’єктами – отже, ці об’єкти повинні міститись у його Assembly, або вище

Отже, всі об’єкти будуть міститись у одній Assembly



Введення нових змін

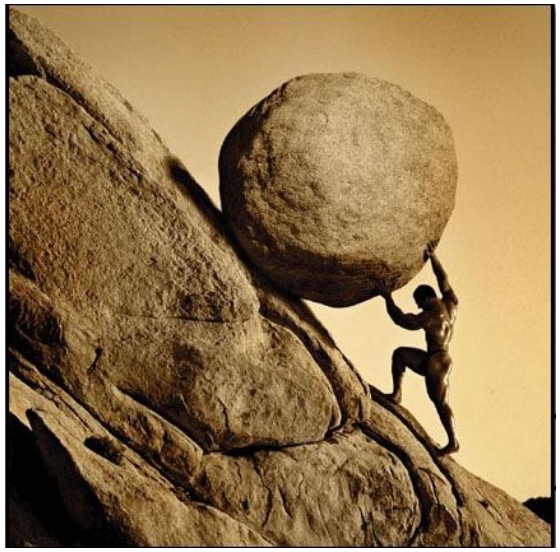
- При змінах в “God object” є потреба міняти код в безлічі місць, і просто не знаєш з якого боку взятись :(

Рефакторинг

- А при спробі порефакторити усе падає, через сильні залежності

Стає все важче його підтримувати

- Те, що почалося як добре розроблений клас, перетворюється на тисячі і тисячі строк коду, оскільки програмісти не знайшли час, щоб реорганізувати й очистити його. Багато інших класів в кінцевому підсумку залежать від God-object і залежності стають “волохатими”



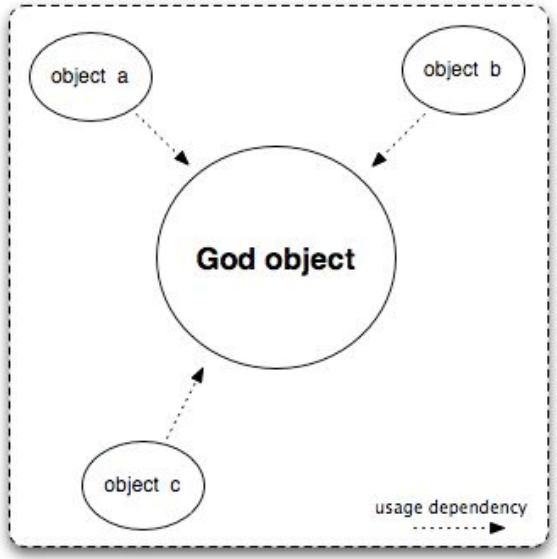
Мінуси – підтримує різні функціональності

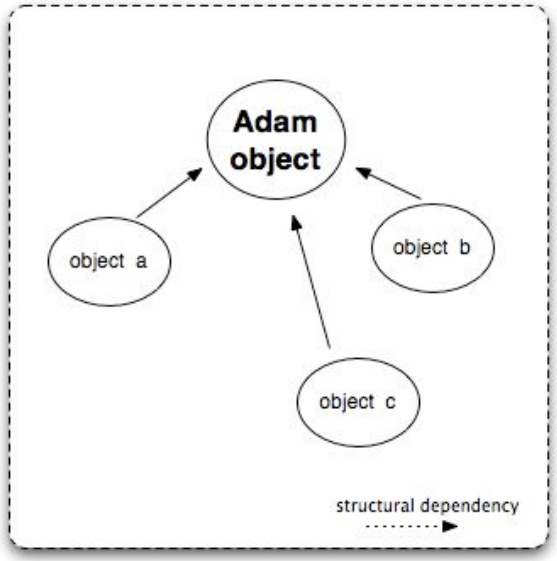
- Універсальна спеціалізація – кожен клас має відповідати за свою ділянку роботи

- Усувають цей антипатерн, відповідно переміщуючи подібну функціональність до різних класів

Об’єкт Адам

- Прагнення до універсальності інколи веде до темної сторони
- Зробити один об’єкт, який би наслідувала велика група об’єктів
- God-object – залежності використання, Адам – структурні залежності





Проблеми Адама (Fragile Base Class)

- Зовсім не у відсутності (або присутності) Єви
- **Зміна у базовому методі Адама приводить до тотальної зміни поведінки усіх нащадків**
- Жорстка прив'язка до структури – неможливо її змінити у нащадках, негнучка система класів
- Абстрактні методи Адама повинні бути переписані у кожному нащадку (що трохи втомлює)
- Якщо Адам – об'єкт, то неможливо наслідувати від ще одного класу його нащадків
- Шаблонний метод – приклад Адама

Реакція програміста на поведінку програми після змін в Адамі

Рішення

- Інколи Адам – це неминуче зло
- “Крихкі” методи зробити віртуальними, щоби можна було змінити їх у нащадках
- Змінні робити приватними і “змушувати” нащадків перевизначати їх
- Інколи можна частину винести у інші класи/інтерфейси. Краще мати кілька “Адамів” для кожної області, ніж одного

Паблік Морозов

- Клас-нащадок, створений відповідно до цього антипатерну, видає за вимогою всі дані класу-предка, незалежно від потреби їх приховування

Назва даного анти-патерну - це каламбур, заснований на співзвуччі ключового слова public (паблік), часто означає відкритий доступ до методів і полів класу в об'єктно-орієнтованих мовах програмування, та імені піонера-героя Павлика Морозова, відомого тим, що він видав свого батька-куркуля

Видає всі приховані поля

- Клас предок має приховані поля та методи
- Клас нащадок тим чи іншим чином робить ці поля і методи доступними для всіх (public полями та методами)

Мінуси

- Надає доступ до всієї інформації (розказує все що треба і не треба)
  - Надлишковість методів і даних – порушення інкапсуляції
  - Можливі порушення безпеки

Оргія об’єктів (Object Orgy)

- Об'єкти недостатньо інкапсульовані, що дозволяє необмежений доступ до своєї внутрішньої структури, як правило, призводить до складності, яку неможливо підтриму
- Прийшла з мови Perl

Оргія об’єктів (Object Orgy)

- Масове використання public спричиняє своєрідні ефекти
- Код стає нечитабельним
- Абстакція стає неефективною
- Код-спагетті

BaseBean

- BaseBean - це утиліта - об'єкт, який має кілька нащадків
- "Бін" частина назви походить від стандартних імен JavaBean
- Правильне проектування дозволяє припустити, що замість успадкування відповідна функціональність повинна бути забезпечена за допомогою делегування

BaseBean

- Наслідування – більш сильна структурна залежність, ніж делегування
  - Послідовність дій, визначену у базовому класі, змінити не можна
  - Нащадки мають підтримувати всі абстрактні дії, визначені в базовому класі
  - Якщо раптом виникає потреба зміни структури, то їх **дуже складно** впровадити

Чітка послідовність дій задана базовим класом

- З’являються дивні послідовності, виду
  - Спочатку прибираємо сніг, потім копаємо траншею, потім обкладаємо її свіжою травою
- if (сніг != null) ПрибратиСніг(сніг)
- КопатиТраншею()
- if (трава != nu) равою(трава)

— Мне нужны три добровольца

Підтримкати всіх дій BaseBean

Class **BaseBean**{

- protected abstract void Validate(Obj)
- protected abstract void OnUpdate(Obj)
- protected bstract void DoUpdate(Obj)
- protected abstract void OnUpdated(Obj)

```
Public void Update(Obj){
    Validate();
    OnUpdate(Obj);
    DoUpdate(Obj);
    OnUpdated(Obj);
     }
    }
```

Підтримка всіх дій BaseBean

```
Class ABean: BaseBean{
protected override void Validate(Obj){//Empty}
protected override void
   OnUpdate(Obj){//Empty}
protected override void DoUpdate(Obj) {
    //The code is just here }
protected override void
   OnUpdated(Obj){//Empty}
}
```

Зміна функції BaseBean

- Зміна – до Update додався виклик SaveToHistory()
  - У всіх класах тепер треба дописувати цей метод
- Зміна – треба поміняти послідовність викликів у функції Validate() викликаємо після OnUpdate(Obj)
  - Неможливо прослідкувати, що відбуватиметься у всіх базових класах

Як покращувати

- Замість прямого наслідування використовувати делегування
- Код легше змінити і стає більш зрозумілим

```
DoSmth(){
base. DoSmth()
//Some additional code
…
}
```

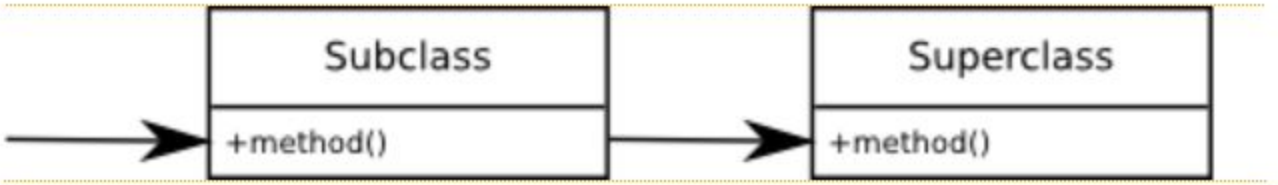
CallSuper

- Базовий клас передбачає, що в похідному підкласі користувачеві необхідно у перевизначеному методі викликати відповідний базовий метод у певній точці виконання
- Методи базового класу можуть бути навмисно частково нереалізованими, вимагаючи їх перевизначення. Але це не вирішує проблему
- **Оскільки програміст повинен пам’ятати послідовність викликів функцій, він точно це забуде**

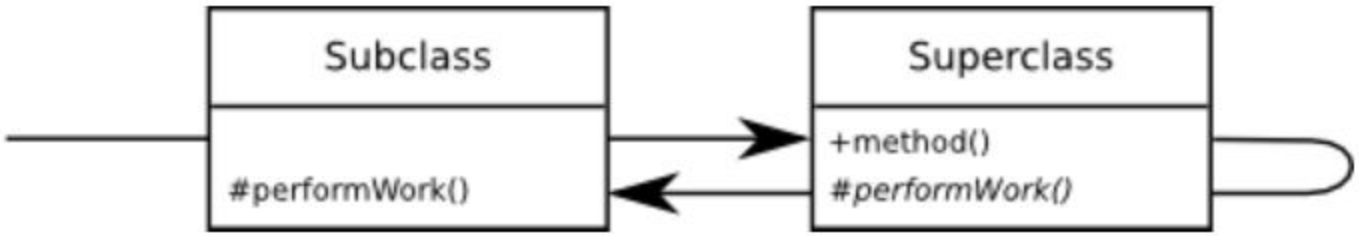
Вирішення

- Template method (Шаблонний метод)

До



Після



Circle-ellipse problem

Сержант – солдатам:

Еліпс – це коло, вписане в квадрат зі сторонами 3x4

Коло-еліпс проблема (квадрату - прямокутника)

- Може виникнути при використанні поліморфізму підтипів при моделюванні об’єктів
  - Який клас має бути базовий – квадрат чи прямокутник?
  - критика ООП, складність класифікації
- Найчастіше виникає при використанні об'єктно-орієнтованого програмування

Проблема

- Коло має метод і змінну Радіус
- клас Еліпс має дві змінні, і метод Розтягнути_у_напрямі()
- Клас нащадок має підтримувати метод базового класу, який для нього є абсурдним
- Liskov substitution principle …

Можливі розв’язки

- Перебудова класів
- Повертати true/false у разі успіху/невдачі (генерувати виняткову ситуацію)
- Immutable – у методах проводити операції не над поточним об’єктом, а над новим і повертати його як результат операції (String)

Об'єктна клоака (Object cesspool)

- Пастка шаблону Object pool
- Перевикористання об'єктів, що знаходяться в непридатному для перевикористання стані

Об’єктний пул Object pool

- Ініціалізує список подібних об’єктів. Клієнти беруть з цього списку
- По закінченню роботи повертаються в пул
- Після повернення повинні бути підготовленими до повторного використання
  - Інакше будуть містити значення змінних, які неможливо передбачити
  - У мовах де є GarbageCollector пул не

Полтергейст (poltergeist або gypsy wagon)

- Michael Akroyd 1996
  - "As a gypsy wagon or a poltergeist appears and disappears mysteriously, so does this short lived object. As a consequence the code is more difficult to maintain and there is unnecessary resource waste. The typical cause for this antipattern is poor object design "

Ознаки

- Короткоживучий об’єкт, який засмічує пам’ять
  - Не путати з довгоживучими об’єктами
- Призначення полтергейста мінімальне
  - передати виклик до інших класів
  - ініціалізувати якісь класи і т.д.

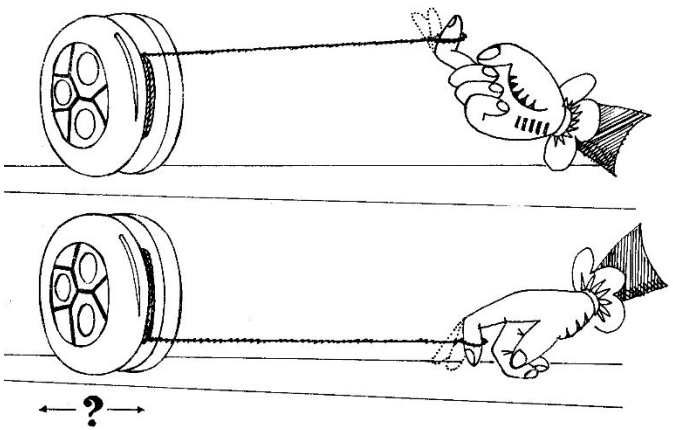
Послідовне зв’язування
Sequential coupling

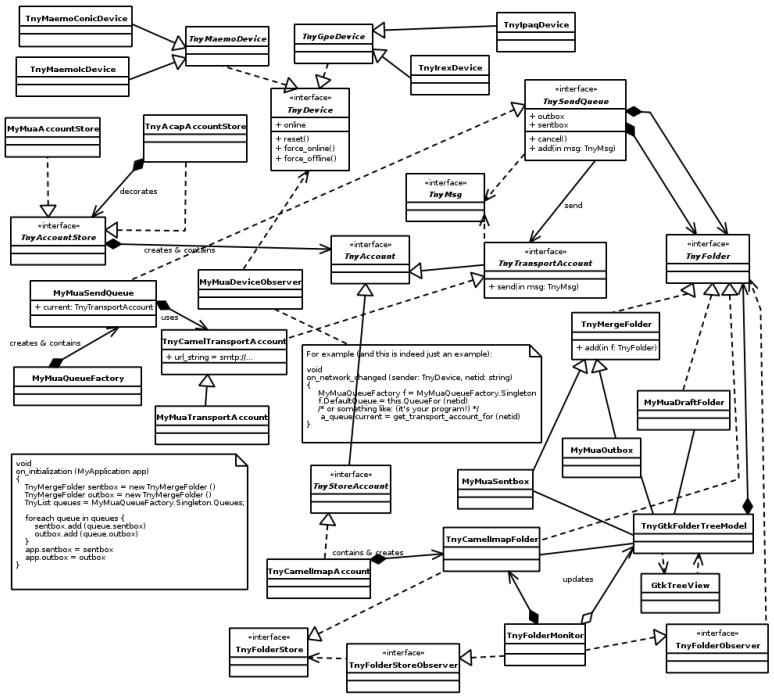
- Методи класу можуть бути викликані лише в певному порядку
  - Базового або делегованого класу
- Порядок не може бути змінений, інакше система неправильно функціонуватиме

Проблема Йо-йо (Yo-yo problem)

- Ієрархія наслідуваних класів роздута
- Стає складно відслідковувати функціональність

Викликаємо метод 1,
він викликає метод 2,
він викликає метод 3,
…..





Сінгелетонізм (Singletonitis)

- Багато авторів критикують шаблон сингелтон
  - Використання статичних ініціалізацій – певне порушення ООП
  - Наприклад у Unit Testing складно користуватись сингелтонами
- Проблема у зловживанні сингелтоном
  - Збирач сміття не видаляє ці об’єкти аж до завершення програми
  - При розширенні класу приходиться відмовлятись від цього шаблону (наприклад, вам потрібно різний такий об’єкт для різних вікон)

Непотрібна складність (Accidental complexity)

- Надлишкова складність, які виникають в комп'ютерних програмах або в процесі їх розробки (програмування)
- Наслідок неефективного планування
- Складність має бути зведено до мінімуму в будь-якій архітектурі, проектуванні і реалізації

Дія на відстані (Action at a distance)

- Термін заснований на концепції дій на відстані у фізиці, маючи на увазі процес, який дозволяє об'єктам взаємодіяти миттєво без посередника, як фотон. Зокрема, Альберт Ейнштейн охарактеризував ці ефекти у квантовій механіці, як "жахливі дії на відстані "
- Поширена помилка, в якій поведінка в одній з частин програми змінюється в залежності від операції в іншій частині програми
- Важко або неможливо визначити, що відбувається (ознака - купа перевірок правильності)
  - Зміни відбуваються у різний час
  - Непередбачувані побічні ефекти від малої зміни
- Усунення проблеми
  - Уникнення глобальних змінних
  - Змінювати дані лише у локальних межах (своїх класах чи своїй частині програми)
  - Шаблон “Стан”

Сліпа віра (Blind faith)

- Недостатня перевірка коректності результату роботи програми
- У моїх програмах помилок немає
  - Жодних Unit-test
  - Жодних try-catch

Сліпа віра (Blind faith)

- У моєму коді ніколи немає помилок

Човновий якір (Boat anchor)

- Збереження більше не використовуваної частини системи
- “не видаляй, можливо нам це знадобиться в майбутньому ”

З’явилось, коли невеликі комп’ютери почали витісняти старих монстрів

На фото міні-комп 1977 року (770 кг)

Активне очікування (Busy spin):

- Споживання ресурсів ЦПУ (процесорного часу) під час очікування події, зазвичай за допомогою постійно повторюваної перевірки (Sleep + if (handled)), замість того, щоб використовувати систему повідомлень

Кешування помилки (Caching failure)

- Забувати скинути кеш помилки після її обробки
- Наприклад у браузері є кеш, якщо помилка закешувалась, то на повторне введення адреси треба не використовувати кеш, а обновити його

Смердючий підгузник (The Diaper Pattern Stinks)

- Скидання помилки без її обробки або передачі її на обробку вище
- try{…} catch{ //do nothing }
- Результат
  - Ніби все працює, але якось дивно
  - Неможливо відслідкувати, де помилка виникає (бо вони постійно перехоплюється)
  - Нагромадження помилок у коді (оскільки вони не виловлюються і не виправляються)

Ховання помилок

```
try {ImportFile(filename);
}
catch
{
 // an exception with almost no information
throw new Exception (“import failed”);
}
```

Хования помилок (Error hiding)

- Аргументом є – навіщо користувачу знати внутрішні нюанси
- Такі помилки не дають інформації користувачу узагалі (де мають бути файли, що можна зробити)

Coding by exception

- Коли є нагромадження перевірок і виняткових ситуацій
- Наприклад, для кожного випадку визначається своя виняткова ситуація
  - If (a>10){throw new ToLargeException()},
  - if (a<10) {throw new AIsNegtiveException()},
  - Правильніше їх об’єднати і не створювати custom-Exception
- При правильному дизайні не буде багато виняткових ситуацій

Cargo cult programming

- Нецільове використання шаблонів проектування чи архітектури

Cargo cult programming

- Копіюємо шаблони проектування або архітектуру без її розуміння – і як результат

Hard code/Soft code

- Жорстка прив’язка до даних у коді (коли потрібно натомість вичитувати з БД чи конфігурації). Часто потрібно для швидкого кодування презентацій, демо, тощо
- Soft code – винесення непотрібних речей у конфігурацію (назв полів, тощо)

Магічні стрічки (Magic strings)

- Використання стрічок для передачі інформації замість створення власних типів даних
- З часом конвертування у стрічки та з них розростається і стає незграбним у підтримці та джерелом

Магічні числа

- Характерні для прикладних обчислень
- У формулах є якісь числа, але незрозуміло, що це таке
- З часом забувається, стає незрозумілим для чого це і що треба змінювати

Потік лави (Lava flow)

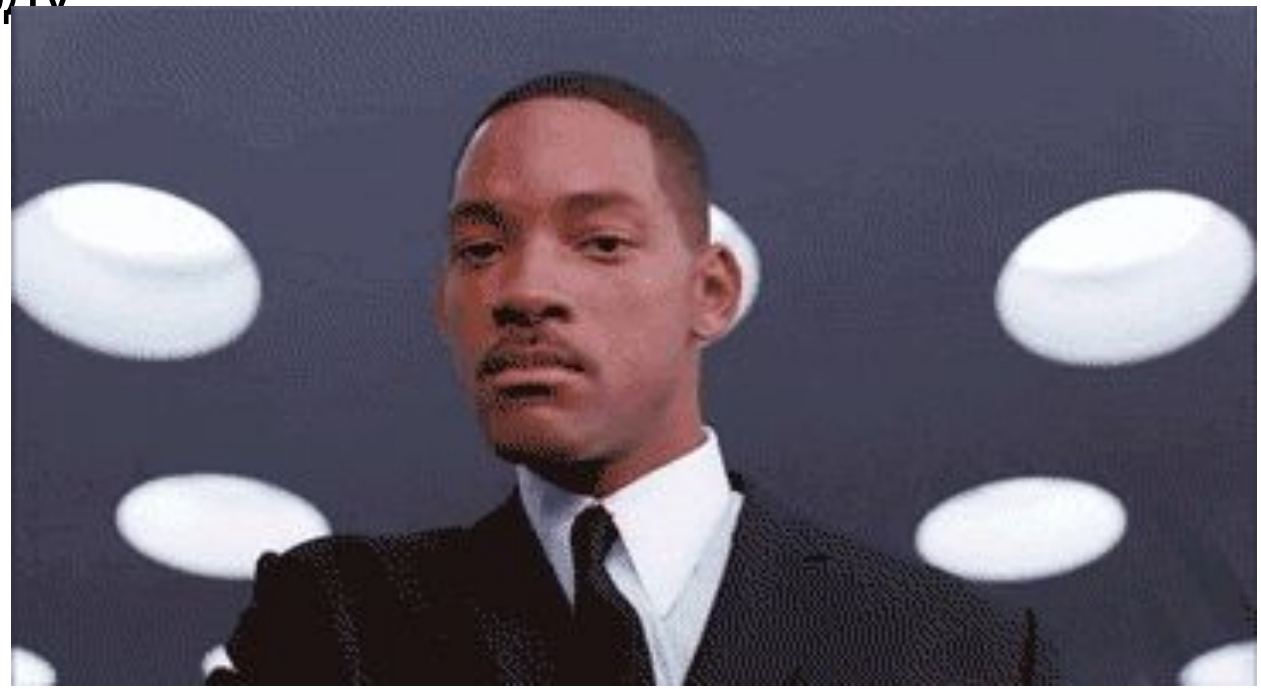
- Збереження небажаного (зайвого або низькоякісного) коду через те, що його вилучення занадто дорого або буде мати непередбачувані наслідки
- Постійний рефакторінг обходится дешевше
- Деградація будинку починається з одного розбитого вікна, яке ніхто не хоче вста

Методологічні антипатерни

Павло Сердюк, НУ "Львівська політехніка", 2013

Copy – paste

- Copy – paste (+ легка модифікація)
- При збільшенні такого коду та їх частій мадифікації легко забути модифікувати одну з частин копійованого коду



Методологічні антипатерни

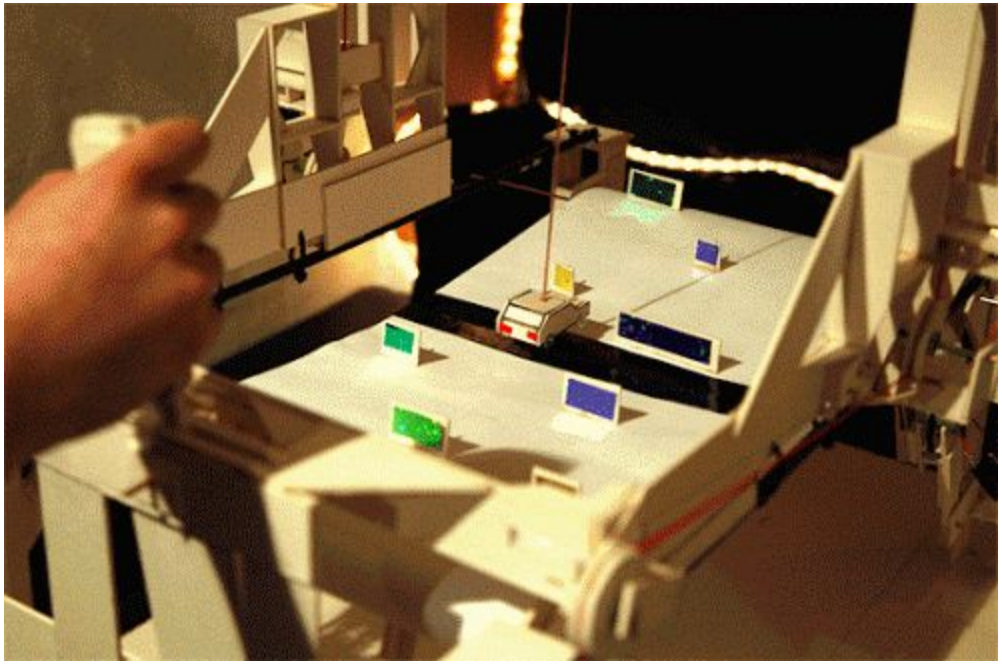
- Дефакторінг. Заміна функціональості документацією (а тут у нас буде басейн)
- Золотий молоток (або Срібна куля) (коли в руках молоток – всі проблеми є цвяхами). Застосування улюбленого інструменту (шаблону) до всіх без виключення проблем
- Фактор неймовірності (Improbability factor): Припущення про неможливість того, що спрацює відома помилка

Програмування випадком

- Програмування випадком (“programming by accident” або ”Programming by permutation”) – процес програмування полягає у **випадковій** зміні малих кусків коду і перевірки, чи програма запрацювала

Методологічні антипатерни

- Винаходження колеса (Reinventing the wheel)
- Непотрібно писати свою CMS :)



Винаходження велосипеда

Винаходження квадратного колеса
(Reinventing the square wheel)

Expectation:

Винаходження квадратного колеса (Reinventing the square wheel)

Reality:

Конфігураційні антипатерни Версіонування

- Пекло залежностей (Dependency hell, DLL hell): Проблеми з версіями потрібних продуктів. Ця проблема виникла в ранніх версіях Microsoft Windows
- За задумом, DLL повинні бути сумісними від версії до версії і взаємозамінними в обидві сторони, але це є радше винятком, ніж правилом
  - Відсутність стандартів на імена, версії та положення DLL у файловій структурі призводить до того, що несумісні DLL легко заміняють один одного або відключають один одного
  - Відсутність стандарту на процедуру встановлення призводить до того, що встановлення нових програм призводить до заміщення працюючих DLL на несумісні версії
  - Відсутність підтримки DLL з боку лінкера і механізмів захисту призводить до того, що несумісні DLL можуть мати одне і те ж ім'я і одну і ту ж версію
  - Відсутні стандартні інструменти ідентифікації та управління системою DLL користувачами та адміністраторами
  - Використання окремих DLL для забезпечення зв'язку між завданнями

Правило 90-90
Аналог принципу Парето 80-20

- "The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time." —Tom Cargill
- "The first 90% of the code takes 90% of development time. The other 90% of code takes the other 90% of time "

Додатова література

- Принципи програмування http://en.wikipedia.org/wiki/Category:Programming_principles
- Проблема Кола-Еліпса. http://en.wikipedia.org/wiki/Circle-ellipse_problem
- Додаткові запахи http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm
- Додаткові запахи. Coding horror. Code smell http://www.codinghorror.com/blog/2006/05/code-smells.html
- Додаткові запахи. http://c2.com/cgi/wiki?CodeSmell
- Martin Fowler. **Refactoring.** http://www.refactoring.com/
- Додаткові шаблони проектування http://design-pattern.ru/patterns
- **Catalog. Patterns in Enterprise Software** http://martinfowler.com/articles/enterprisePatterns.html

Дякую за увагу