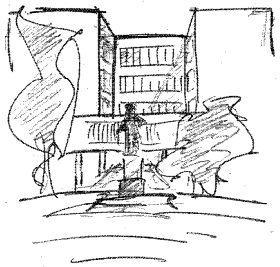
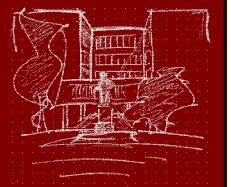
Развој софтвера 8



Саша Малков
Универзитет у Београду
Математички факултет
2023/2024

[P290]
Развој софтвера
Саша Малков



Тема 12

Рефакторисање

Када је програмски код добар?

- Критеријуми су прилично разноврсни:
  - када ради
  - када ради добро
  - када ради брзо
  - када се пише брзо
  - ...
  - када се лако одржава
    - што је програм већи и важнији, то овај критеријум више добија на значају
    - агилни развој скраћује период писања делова програма
      али продужава период одржавања делова програма

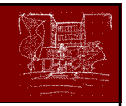
Кварљивост програмског кода

- Агилни развој промовише учестало мењање постојећег кода
  - кратки развојни циклуси
  - не планира се далеко унапред
  - зато се често се праве измене и допуне
- Мењање постојећег програмског кода нарушавају његов добар дизајн
  - долази до понављања делова програма
  - нарушавају се неки установљени односи између делова кода
- Да ли и како дизајно програма може да се поправља?

Рефакторисање

- *Рефакторисање* је предузимање низа малих трансформација кода којима се унапређује структура кода без спољa видљиве промене понашања
  - Свака појединачна трансформација је једноставна, скоро тривијална
  - После сваке трансформације се тестирају јединице кода
  - Понављају се трансформације све док се не дође до чистог и добро структурираног дизајна
- Основни циљ рефакторисања је поправљање дизајна програма

Пример рефакторисања (1)



- **ВАЖНО**!!!
  - При демонстрацији примера, због недостатка времена, прескочићемо писање тестова
    - уместо тога ћемо често преводити и извршавати главни програм
  - у пракси писање тестова **НЕ СМЕ** да се прескаче
- Рефакторисање би **увек** требало да почиње прављењем тестова (ако већ не постоје)
  - тестови проверавају да ли је очувана функционалност кода
  - циљ је да промене ограничимо на структуру/дизајн кода, тј. да при поправљању дизајна не променимо понашање

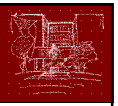
Пример рефакторисања (2)

- Погледаћемо пример кода који ради...
- ...али је тако писан да не може да се непосредно и једноставно проширује
- Да бисмо га проширили, покушаћемо прво да га поправимо
- ... пример ...

Рефакторисање и писање кода

- **Рефакторисање** не мења видљиво понашање, већ само структуру
- **Писање новог кода** не мења постојећи код већ само додаје ново понашање
  - (идеал агилног развоја...)
- **Програмирање** се састоји од наизменичног писања новог кода и рефакторисања

Основна питања

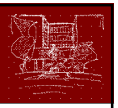


- Зашто рефакторисати?
  - Шта је основна мотивација?
- Када рефакторисати?
  - Како да препознамо прави тренутак?
- Како рефакторисати?
  - Да ли постоје неки описани поступци?

Зашто рефакторисати?

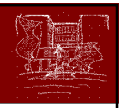
- Рефакторисање подиже ниво квалитета дизајна софтвера
  - то је основни мотив за његово предузимање
  - све остало су последице
- Рефакторисање чини софтвер лакше разумљивим
- Рефакторисање помаже у тражењу грешака
- Рефакторисање подиже брзину писања кода

Када рефакторисати?



- Редовно
- Потенцијално сваки пут пре додавања новог кода
- “Ако се нешто понови по трећи пут”...
- Када се додају нове функције
- Када је потребно пронаћи баг
- Када се проверава исправност и квалитет кода

Како рефакторисати?



- Слично као у случају образаца за пројектовање:
  - уочене су неке препознатљиве слабости у програмском коду
  - које је могуће решити применити рефакторисања
  - као и одговарајуће технике рефакторисања
  - и праве се каталози рефакторисања
- Најпре ћемо размотрити неке од уобилајених проблема...
- ...а затим и неке од одгоаврајућих техника

Кандидати за рефакторисање

- Веома је незахвално дефинисати услове које би неки код требало да задовољава да би га требало рефакторисати
- Уместо тога, издвајају се “кандидати” – случајеви у којима је потребно да се *размотри* да ли рефакторисањем може да се оствари позитиван помак у квалитету дизајна кода

Кандидати за рефакторисање (2)

- “Ако код заудара, потребно га је изменити”
- “Лош дизајн” је субјективна категорија
- Осетљивост програмера на одређене проблеме није иста
  - Програмер са већим искуством ће умети да боље препозна могућа унапређења
  - Програмер може неке трансформације кода да сматра за тривијалне, па онда и да одлаже одговарајућа трансформисања кода као мање значајна

Кандидати за рефакторисање (3)

- Да ли ће се заиста прибећи рефакторисању зависи
  - од могуће добити
  - од способности програмера да препозна добити и погодне технике рефакторисања
- Неку технику је можда боље да не применимо
  - ако није јасно да је њеном применом могуће остварити добит
  - или није јасно да ли та техника уопште може да се исправно примени у датом случају

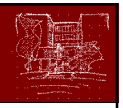
Врсте *заударања*

- Понављање кода
- Дугачак метод
- Велика класа
- Дугачка листа аргумената
- Дивергентне промене
- Дистрибуирана апстракција
- Велика зависност од других класа
- Груписање података
- Поплава примитивних података
- Наредба *switch*
- Паралелне хијерархије наслеђивања
- Лење класе
- Спекулативно уопштавање
- Привремени подаци
- Ланци порука
- Посредник
- Непожељна блискост
- Алтернативне класе са различитим интерфејсима
- Непотпуна библиотека
- Класа податак
- Непожељно наслеђе
- Коментари

Понављање кода

- Један од најчешћих и основних проблема
- Ако се исти или сличан код понавља више пута, потребно је размотрити апстраховање
  - Издвајање метода
  - Повлачење метода уз хијерархију
  - Прављење шаблонских метода
  - Замена алгоритма
  - Издвајање класе

Дугачак метод



- Ако је метод сувише дугачак, сва је прилика да је потребно да се разложи на више мањих
- Пре свега ради лакшег одржавања
  - разумевање
  - дебаговање
  - мењање
  - Издвајање метода
  - Замена привремених вредности упитима
  - Увођење параметарског објекта
  - Замена метода објектом
  - Декомпозиција услова

Велика класа

- Велика класа може да постоји са разлогом
- Ако се прави много објеката неке класе и то за различите намене, то може бити сигнал да је класу потребно поделити на више класа
  - Издвајање класе
  - Издвајање подкласе
  - Издвајање интерфејса
  - Понављање праћених података

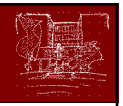
Дугачка листа аргумената

- Ако метод има много аргумената, добро је да се смањи њихов број, ако је могуће
  - Замењивање аргумента методом
  - Задржавање целог објекта
  - Увођење параметарског објекта

Дивергентне промене

- Термин *дивергентне промене* означава случајеве у којима се нека класа мења из више различитих разлога
- То обично значи да разлози промена нису довољно прецизно апстраховани
  - Издвајање класа

Дистрибуирана апстракција



- У неким случајевима, због неке промене морају да се праве бројне мале измене у много различитих класа
- То обично значи да је одговарајућа апстракција дистрибуирана у више класа, што није добро
  - Премештање метода
  - Премештање података
  - Уметање класе

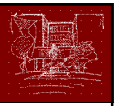
Велика зависност од других класа

- За класу или метод кажемо да сувише зависи од других класа ако користи велики број метода друге класе за читање вредности
- То обично значи да је нека одговорност подељена између више класа на неодговарајући начин
  - Премештање метода
  - Издвајање метода

Груписање података

- Ако се исти подаци понављају у више класа то може да значи да је потребно да се они апстрахују
  - Издвајање класе
  - Увођење параметарског објекта
  - Задржавање целог објекта

Поплава примитивних података



- Често је много боље да се направи једноставна класу него да се користе примитивне вредности
  - Замена податка објектом
  - Замена кодираног податка класом
  - Замена кодираног податка потклассом
  - Замена кодираног податка стањем/стратегијом
  - Издвајање класе
  - Увођење параметарског објекта
  - Замена низа објектом

Наредба *switch*

- Проблем са употребом наредбе *switch* је у тенденцији да се понављају критеријуми и гранања
- То најчешће може да се превазиђе хијерархијама класа
  - Издвајање метода
  - Премештање метода
  - Замена кодираног податка потклассом
  - Замена услова полиморфизмом
  - Замена параметра експлицитним методом
  - Увођење празних објеката

Паралелне хијерархије наслеђивања

- Случај када се две хијерархије паралелно одржавају
  - За сваку (или скоро сваку) класу једне хијерархије постоји одговарајућа класа друге
- Паралелне хијерархије су посебан случај дистрибуираних одговорности
  - Премештање метода
  - Премештање података

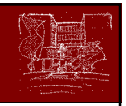
Лење класе

- Класа је *лења* ако нема значајну функцију
- Често је такве класе боље обрисати
  - (супротно од замене примитивних вредности објектима)
  - Сажимање хијерархије
  - Уметање класе

Спекулативно уопштавање

- Ако је код апстрахован зато што се *претпоставља* да би разноликост *могла да наступи* али заправо не постоји
- Сувишно уопштавање (апстраховање) може да значајно отежа одржавање па, па је онда потребно да се уклони
  - Сажимање хијерархије
  - Уметање класе
  - Уклањање аргумента
  - Преименовање метода

Привремени подаци



- Ако се нека привремена променљива користи само у неким посебним случајевима, онда то може да отежава разумевање кода
  - Издвајање класе
  - Увођење празног објекта

Ланци порука

- Ланци порука указују на велики број посредника у обављању неког посла
- Потребно је да се размотри да ли је могуће уклањање сувишних посредника
  - Скривање делегата
  - Издвајање метода
  - Премештање метода

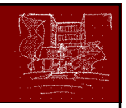
Посредник

- Скривање интерних детаља је једна од основних карактеристика ОО програмирања
- Међутим, може се отићи предалеко
- Ако читава класа представља само посредника, без значајне допуне у понашању, потребно је размотрити њено уклањање
  - Уклањање посредника
  - Уметање метода
  - Замена делегирања наслеђивањем

Непожељна блискост

- Непожељна блискост наступа када нека класа сувише често приступа приватним деловима неке друге класе
  - Премештање метода
  - Премештање податка
  - Промена двосмерног односа у једносмеран
  - Издвајање класе
  - Скривање делегата
  - Замена наслеђивања делегирањем

Алтернативне класе са различитим интерфејсима

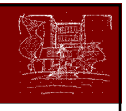


- Ако методи који раде исте или одговарајуће ствари имају различите интерфејсе, то није добро
  - Преименовање метода
  - Премештање метода
  - Издвајање наткласе

Непотпуна библиотека

- Нека библиотечка класа може да буде непотпуна у контексту у коме је потребна, али њу не можемо да мењамо
- Због тога се примењују другачије технике
  - Премештање метода
  - Увођење страног метода
  - Увођење локалног проширења

Класа податак

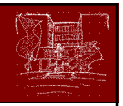


- Класа податак је класа која садржи само податке и приступне методе, а не и додатно понашање
- Такве класе могу да значајно поправљају дизајн
- Али, ако нису добро написане могу правити проблеме
  - Енкапсулирање података
  - Енкапсулирање колекције
  - Уклањање метода за постављање вредности
  - Премештање метода
  - Издвајање метода
  - Скривање метода

Непожељно наслеђе

- Ако класа *не жели* наслеђен део, онда вероватно неки методи нису на правом месту у хијерархији или тој класи није место у хијерархији
  - Спуштање метода низ хијерархију
  - Спуштање податка низ хијерархију
  - Замена наслеђивања делегирањем

Коментари



- Коментари у имплементацији метода сугеришу да код без њих није довољно јасан, тј. да није довољно добро дизајниран
- “Коментари имају добар мирис, али се често користе као дезодоранс за код који заудара”
  - Издвајање метода
  - Преименовање метода
  - Увођење претпоставке

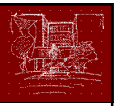
Каталог рефакторисања (1)

- Систематизација рефакторисања подразумева уједначено описивање техника
- Свака техника има
  - име
  - сажет опис случаја у коме се примењује
  - мотивацију
  - опис технике извођења
  - пример

Каталог рефакторисања (2)

- Сва рефакторисања су груписана по намени у:
  - Методе компоновања кода
  - Премештање кода између објеката
  - Организовање података
  - Поједностављивање условних израза
  - Поједностављивање позивања метода
  - Разрешавање уопштавања
  - Велика рефакторисања

Проблеми при рефакторисању (1)



- Рефакторисање понекад може да буде
  - веома тешко
  - практично немогуће

Проблеми при рефакторисању (2)

- Промене структуре базе података
  - мењање структуре података има широког утицаја
  - миграција података је веома неугодна
- “Не правити базу података пре времена”

Проблеми при рефакторисању (3)

- Промене интерфејса
  - све док се промене односе на имплементацију неког система, то је релативно једноставно
  - када почне да се мења интерфејс, потребна је права мера
  - ако се мења јавни интерфејс, често је пожељно да се привремено сачува и стари интерфејс, ради компатибилности
    - али то привремено чини код веома прљавим
- “Не објављивати јавни интерфејс док не сазри”

Проблеми при рефакторисању (4)

- Сложени дизајн
  - увек размотрити рефакторисање пре примене
  - често постоје различите алтернативе, а често не може свака од њих да се (релативно једноставно) примени у пракси

Проблеми при рефакторисању (5)

- Некада није добро рефакторисати
  - ако су проблеми вишеструки и међусобно зависни
  - ако се не могу примењивати мањи кораци без ремећења постојећег понашања кода
  - ако постојећи код не ради
  - ако су сувише близу крајњи рокови
  - ...
  - и посебно, ако је на делу више наведених услова
- У неким случајевима је исплативије да се пише нови код него да се мења постојећи

Шта рефакторисање није

- Рефакторисање **НИЈЕ** универзалан лек за све проблеме дизајна софтвера
- Неки проблеми у дизајну не могу да се реше рефакторисањем већ само великим и сложеним захватима
- Међутим, ако је почетни дизајн добар и ако се редовно примењује рефакторисање, велики проблеми углавном неће настајати

Рефакторисање и перформансе

- Подизање нивоа структуре кода често има за последицу повећавање обима кода или спуштање нивоа перформанси
  - Перформансе се не смеју занемарити, ипак...
  - Степен негативног утицаја на перформансе је обично далеко нижи од оствареног позитивног утицаја на дизајн
  - Писање структурираног кода је далеко ефикасније него писање оптимизованог кода
  - Трошкови развоја су често важнији од трошкова експлоатације

Рефакторисање и перформансе

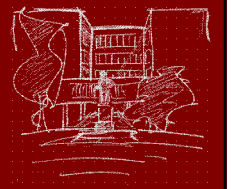
- Више метода:
  - Добро структурирање са дефинисањем циљних перформанси сваке од компоненти
    - Након достизања функционалности приступа се оптимизацији у мери у којој је потребно за достизање циљних услова – тзв. оптимизација уназад
  - Континуално старање о перформансама
    - Током читавог развоја сви се старају да увек понуде оптимална решења – тзв. оптимизација унапред
- У пракси је обично (али не увек) бољи први метод
  - Функционалност пре перформанси
  - Перформансе се плански подижу у складу са дизајном
  - Неоптимизован програм се лакше одржава и мења

Литература за тему

- *Martin Fowler*, ***Refactoring – Improving the Design of Existing Code***, *Addison Wesley, 2000.*
- *William Opdyke*, ***Refactoring C++ Programs***
  - из претходне књиге
  - *http://st-www.cs.illinois.edu/users/opdyke/wfo.990201.c++.refac.html*
- *Martin Fowler*, ***Refactoring Home Page***
  - *www.refactoring.com*

Универзитет у Београду – Математички факултет

Хвала на пажњи!



МАТФ
Универзитет у Београду
Математички факултет