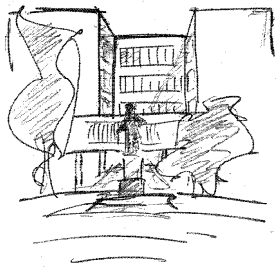
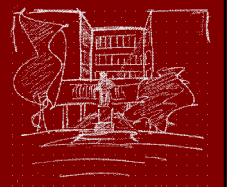
Развој софтвера 6



Саша Малков
Универзитет у Београду
Математички факултет
2023/2024

[P290]
Развој софтвера
Саша Малков



Тема 9.1
Агилни развој софтвера

Шта је агилни развој софтвера?

- Агилни развој софтвера (АРС) је фамилија методологија развоја софтвера која је настала крајем последње деценије XX века
  - назив је обликован 2001. када је формулисан *Манифест агилног развоја софтвера*
  - употребљава се и термин *агилне методологије*

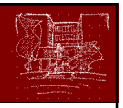
Шта је агилни развој софтвера? (2)

- Агилни развој софтвера
  - има много сличности са ОО методологијама, али са другачијим приступом планирању
    - углавном се претпоставља употреба техника ОО пројектовања и програмирања
  - прописује скуп принципа и скуп техника за њихово остваривање
  - промовише динамичан и дисциплинован тимски рад

Шта није агилни развој софтвера?

- Агилни развој софтвера није
  - одсуство систематичност и структурног приступа
  - анархично или својевољно понашање чланова тима
  - потпуно одсуство документације
  - селективна примена само неких од принципа АРС-а
  - најлакши метод развоја софтвера
  - универзално решење за све проблеме
  - прави начин рада за неискусне или недовољно стручне програмере
  - ...

Шта није агилни развој софтвера? (2)

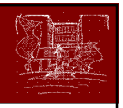


- Назив методологије је атрактиван
  - многи тврде да примењују или познају АРС, иако то није тачно
- Неки принципи изгледају неуређено
  - многи сматрају АРС за површну методологију, иако није то није тачно
- Већина принципа изгледа сасвим лако примењиво
  - многи сматрају АРС за универзалну методологију, иако то није тачно

Околности у којима се развија

- Динамичан пораст броја, величине и сложености софтверских пројеката
- Учестала унапређења технологија
- Учестало подизање (техничког и пословног) нивоа захтева
- Успешност пројеката све више зависи од
  - повећане искоришћености већ написаног кода
  - могућности унапређивања написаног кода
  - могућности реаговања на промене услова и захтева током развоја

Манифест АРС



- Манифест агилног развоја софтвера написали су 2001. године водећи стручњаци у области различитих агилних методологија, оснивачи Агилног савеза (енгл. *Agile Alliance*)
  - Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Ken Schwaber, Jeff Sutherland, Dave Thomas

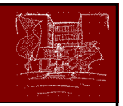
Манифест АРС (2)

"Откривамо боље начине развоја софтвера
развијајући га и помажући другима у томе.
Кроз тај рад научили смо да више вреднујемо:

Људе и односе међу њима – од процеса и алата
Софтвер који ради – од исцрпне документације
Сарадњу са клијентима – од преговарања око уговора
Реаговање на промене – од придржавања плана

Односно, иако ценимо вредности на десној страни,
сматрамо за вреднije оне које су на левој."

Манифест АРС (3)



- Манифест
  - препознаје 4 основне препоставке, на којима почива агилни развој софтвера
  - одређује 12 основних принципа агилних методологија
- Конкретне агилне методологије
  - могу имати додатне претпоставке
  - могу имати додатне принципе
  - одређују методе и технике којима се принципи остварују

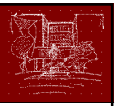
Основне претпоставке АРС

- Четири основне претпоставке (манифест):
  - Појединци и сарадња испред процеса и алата
  - Функционалан софтвер испред исцрпне документације
  - Сарадња са клијентом пре него преговарање
  - Реаговање на промене пре него праћење плана

Основне претпоставке АРС

- Четири основне претпоставке:
  - **Појединци и сарадња испред процеса и алата**
    - људи су најважнији део успешног развоја
    - добар процес не може успети са лошим људима
    - лош процес и најбоље људе чини непродуктивним
    - групе добрих људи ће бити неуспешне ако нема сарадње
    - добри алати помажу развој
    - превише пажње посвећене алатима је једнако лоше као потпуно одсуство алата
    - изградња тима је важнија од изградње окружења
  - Функционалан софтвер испред исцрпне документације
  - Сарадња са клијентом пре него преговарање
  - Реаговање на промене пре него праћење плана

Основне претпоставке АРС

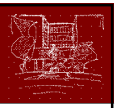


- Четири основне претпоставке:
  - Појединци и сарадња испред процеса и алата
  - **Функционалан софтвер испред исцрпне документације**
    - софтвер без документације је пропаст
      - код није идеално средство за комуникацију
      - неопходна је разумљива документација
    - превише документације може бити горе него премало
      - превише времена иде на одржавање и синхронизацију
    - увек је добро писати пропратну документацију
      - али она мора да буде кратка и на високом концептуалном нивоу
    - увек имати на уму да циљ није документација него софтвер
    - "*не правити документацију осим ако је одмах и значајно потребна*"
      - (Р.Мартин, тзв. "*први закон о документацији*")
  - Сарадња са клијентом пре него преговарање
  - Реаговање на промене пре него праћење плана

Основне претпоставке АРС

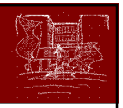
- Четири основне претпоставке:
  - Појединци и сарадња испред процеса и алата
  - Функционалан софтвер испред исцрпне документације
  - **Сарадња са клијентом пре него преговарање**
    - софтвер не може да се наручује као намештај
      - у иоле сложенијем случају практично је неизводиво да се напише опис задатка и препусти некоме да направи софтвер
    - увек је велики проблем изражавање жеља клијента на јасан и довољно експлицитан начин
    - успех пројекта захтева редовну комуникацију развојног тима и клијента
    - прецизно уговарање захтева, рокова, фаза и трошкова пре почетка пројекта је суштински немогуће у већини случајева
  - Реаговање на промене пре него праћење плана

Основне претпоставке АРС



- Четири основне претпоставке:
  - Појединци и сарадња испред процеса и алата
  - Функционалан софтвер испред исцрпне документације
  - Сарадња са клијентом пре него преговарање
  - **Реаговање на промене пре него праћење плана**
    - способност реаговања на промене често одређује успешност пројекта
    - планови морају да буду прилагодљиви променама
      - промене ће сигурно наступити, питање је само када и које
    - планирање не би требало да иде далеко у будућност
      - са током развоја мењају се и циљеви и приоритети
    - детаљно планирање је неопходно, али за кратак период
      - детаљни планови су корисни али се тешко одржавају на дужи период
    - наравно, визија и дефинисани главни циљеви морају да постоје
      - мада се и они могу мењати

Основни принципи АРС (1)



- Манифест агилног развоја прописује принципе агилног развоја софтвера:
  - Највиши приоритет је задовољити клијента кроз брзо и непрекидно испоручивање вредног софтвера.
  - Увек отворено прихватати промене, чак и у касним фазама развоја. АРС уважава промене као средство постизања квалитета за клијента.
  - Испоручивати функционалан софтвер што чешће, са интервалом од пар недеља до пар месеци.
  - Пословни људи и развијаоци морају сарађивати свакодневно на пројекту.
  - ...

Основни принципи APC (2)

- Манифест агилног развоја прописује принципе агилног развоја софтвера:
  - ...
  - Заснивати пројекат на мотивисаним појединцима. Пружити им окружење и потребну подршку и имати поверења да ће обавити посао.
  - Најефикаснији начин за размену информација у тиму је разговор лицем у лице.
  - Функционалан софтвер је основно мерило напретка.
  - APC промовише уздржани развој. Спонзори, развијаоци и корисници би требало да буду у стању да непрекидно одржавају уједначен ритам.
  - ...

Основни принципи APC (3)

- Манифест агилног развоја прописује принципе агилног развоја софтвера:
  - ...
  - Непрекидно посвећивање пажње техничкој дотераности и добром дизајну подиже агилност.
  - Једноставност. Уметност максимизовања количине посла који се не обавља је од суштинског значаја.
  - Најбоље архитектуре, захтеви и пројекти потичу из самоорганизованих тимова.
  - У редовним интервалима тим мора да сагледа свој рад и могућности унапређивања свог понашања и ефикасности.

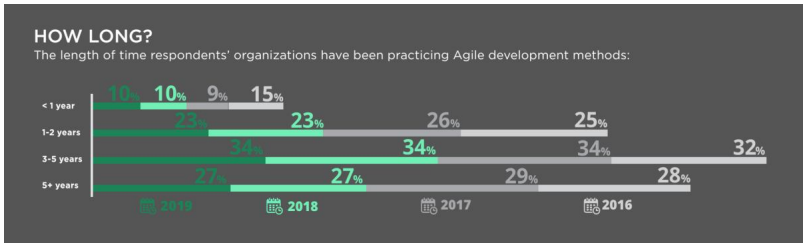
Методологије агилног развоја

- Агилни развој софтвера је окосница више различитих методологија:
  - Агилно моделирање
  - Агилан обједињен процес (*AUP*)
  - Екстремно програмирање (*XP*)
  - Отворен обједињен процес (*OpenUP*)
  - *Scrum*
  - *Kanban*
  - и друге

Агилни развој данас

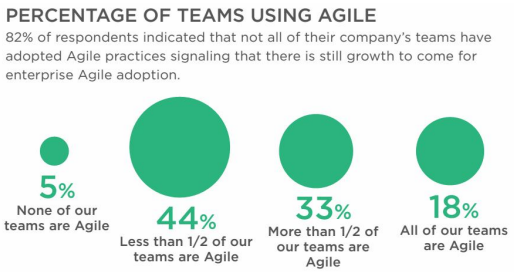
- Представићемо неке статистичке податке, као резултат јавног истраживања које редовно спроводи *CollabNet VersionOne*
  - 14. извештај, односи се на 2019. годину
  - *https://www.stateofagile.com/*
- Уводне напомене:
  - 95% анкетираних лица наводи да се у њиховој организацији користе агилне методологије – то указује да статистике о заступљености агилних методологија можда нису објективне, али унутар APC би требало да су веродостојније
    - у претходном извештају је било 97%

Колико дуго се користи APC?



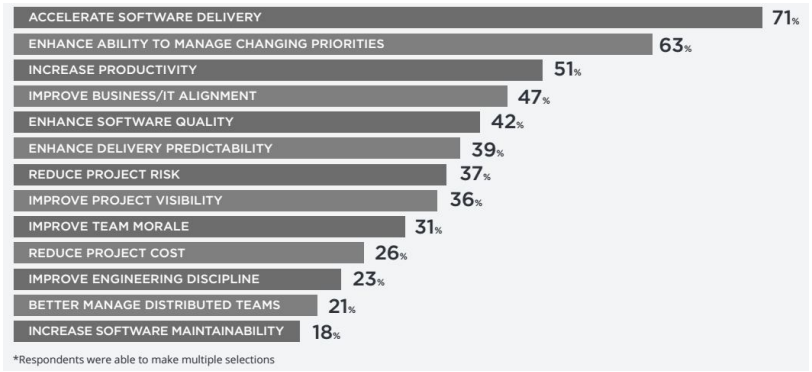
- Релативно много организација је прешло на APC у последњих 5 година
  - значи да се усвајање APC још увек одвија
  - или су компаније релативно младе
  - занимљиво је да се бројеви не мењају ;-)

Колико доследно се користи APC?



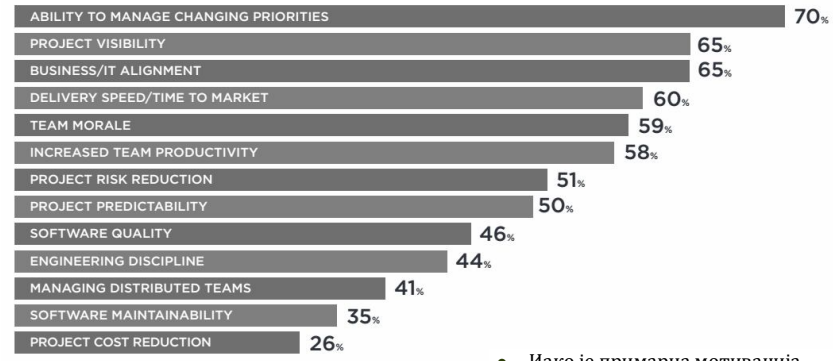
- Тек у 51% организација се APC користи у већини тимова
  - и ово потврђује да је усвајање APC још увек у току

Зашто се користи APC?



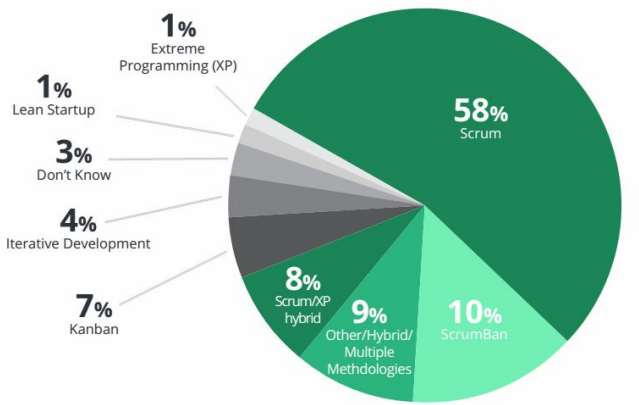
- Најчешћи су разлози везани за продуктивност
- Следе социјални разлози
  - повећање мотивације запослених
  - развојна дисциплина

Које су уочене користи од примене APC?



- Иако је примарна мотивација продуктивност, међу најважнијим последицама је већи тимски морал

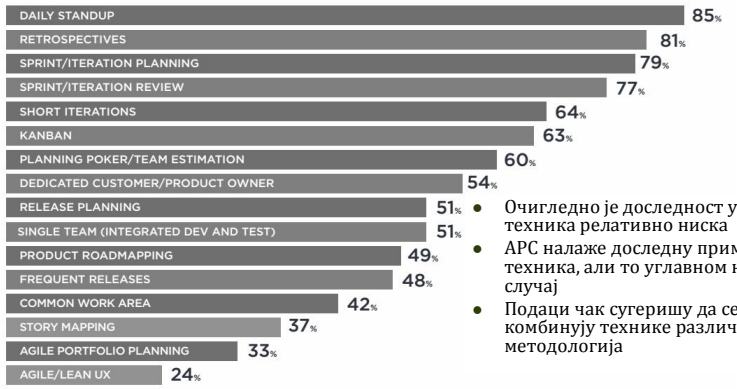
Које методологије АРС су најзаступљеније?



Total exceeds 100% due to rounding.

Које агилне технике се примењују?

Notable changes in Agile techniques and practices that respondents said their organization uses were an increase in product roadmapping (49% this year compared to 45% last year) and a decrease in release planning (51% this year compared to 57% last year).

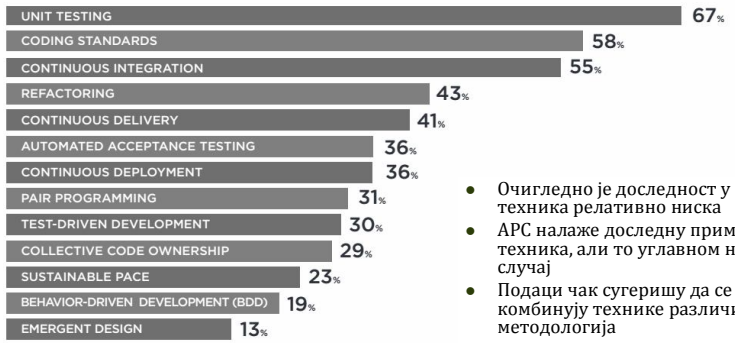


- Очигледно је доследност у примени техника релативно ниска
- АРС налаже доследну примену техника, али то углавном није случај
- Подаци чак сугеришу да се комбинују технике различитих методологија

* истраживање разликује "технике" и "праксе" иако има преклапања

Које агилне "праксе" се примењују?

The overall rank order of engineering practices employed remained almost the same this year over last. Automated acceptance testing increased 3% while pair programming, test-driven development, and behavior-driven development each fell 3%.



*Respondents were able to make multiple selections

- Очигледно је доследност у примени техника релативно ниска
- АРС налаже доследну примену техника, али то углавном није случај
- Подаци чак сугеришу да се комбинују технике различитих методологија

* истраживање разликује "технике" и "праксе" иако има преклапања

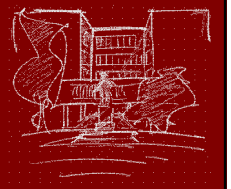
Који алати се примењују?

	CURRENTLY USE		FUTURE PLANS TO USE	
	2019	2018	2019	2018
Kanban board	76%	75%	10%	9%
Taskboard	66%	70%	10%	10%
Bug tracker	63%	67%	15%	12%
Spreadsheet	64%	66%	7%	6%
Agile project managment tool	65%	65%	13%	12%
Wiki	60%	62%	14%	12%
Automated build tool	55%	59%	24%	20%
Unit test tool	55%	54%	20%	17%
Continuous Integration tool	54%	51%	26%	26%
Wireframes	49%	51%	15%	13%
Product roadmapping	51%	50%	28%	27%
Traditional project management tool	44%	46%	8%	6%
Requirements management tool	46%	44%	18%	17%
Release/deployment automation tool	45%	44%	31%	29%
Automated acceptance tool	37%	39%	32%	25%
Static analysis	38%	38%	19%	14%
Project & portfolio management (PPM) tool	39%	36%	26%	24%
Story mapping tool	30%	29%	27%	21%
Timecards	30%	29%	12%	9%
Index cards	26%	28%	14%	9%
Refactoring tool	26%	22%	26%	18%
Customer idea management tool	19%	18%	24%	18%

[P290]

Развој софтвера

Саша Малков



Тема 9.2

Екстремно програмирање

Екстремно програмирање

- Једна од најпознатијих агилних методологија
- Чине га једноставне, међусобно независне методе
  - у складу су са претпоставкама и принципима APC
  - прецизизније описују начин остваривања принципа APC
- Више није међу најзаступљенијим методологијама
  - али остварила је велики утицај на друге методологије
  - и њене методе се веома користе

Методи екстремног програмирања

- Клијент је члан тима
- Корисничке целине (*user stories*)
- Кратки циклуси
- Праћење тока развоја
- Тестови прихватљивости
- Програмирање у пару
- Развој вођен тестовима
- Колективно власништво
- Непрекидна интеграција
- Уздржан ритам
- Отворен радни простор
- *Игра* планирања
- Једноставан дизајн
- Рефакторисање
- Метафора

(размотрићемо их детаљније)

ЕП: Клијент је члан тима

- Клијенти су особе или групе које дефинишу циљеве и приоритете
- Клијенти и развијаоци морају сарађивати
- У екстремном програмирању клијенти имају улогу члана тима
- Најбоље је да буду физички близу због остваривања непосредне комуникације

ЕП: Корисничке целине (*user stories*)

- Ради оквирног планирања потребно је сагледавати захтеве
  - али не и потпуно прецизне елементе захтева
  - неопходно је знати где и каквих детаља има, али не и саме детаље
  - детаљи се мењају током времена
  - прве процене (рокова и трошкова) служе само за управљање приоритетима и никога не обавезују

ЕП: Корисничке целине (*user stories*) (2)

- Уобичајено:
  - захтеви се износе као целине, у свега пар речи
  - оквирне процене се дају одмах или веома брзо
- Корисничка целина служи као референца на нешто што ће се детаљније разматрати када дође на ред за имплементацију

ЕП: Кратки циклуси

- За екстремно програмирање је уобичајена редовна испорука
  - итерације
    - мањи ниво испоруке
    - уобичајено релативно често, на пример на сваке две недеље
  - издања
    - већи ниво испоруке
    - обично обухвата неколико итерација, на пример 6

ЕП: Кратки циклуси – План итерације

- План итерације обухвата:
  - буџет и трајање
    - одређују се на самом почетку планирања итерације
  - корисничке целине које улазе у итерацију
    - по избору клијента, а уз консултације са развојним тимом
    - не одређују се приоритети целина у оквиру итерације
    - скуп корисничких целина не би требало да се мења током рада на итерацији

ЕП: Кратки циклуси – Ток итерације

- Ток итерације:
  - развијаоци праве послове и раде по редоследу који сами одреде
  - на пола трајања итерације снима се стање и процењује се да ли ће итерација бити довршена у року
    - ако неће, обавештава се клијент и доносе се одговарајуће одлуке
  - на крају сваке итерације систем се испоручује
    - може али не мора да иде у продукцију
  - клијенти дају повратне информације за сваку итерацију

ЕП: Кратки циклуси – План издања

- План издања
  - уобичајено се ради о значајној целини која иде у продукцију
  - састоји се од корисничких целина са проценама и приоритетима
  - највише оквирно дефинише садржаје појединачних итерација
  - скуп корисничких целина и њихов распоред по итерацијама се уобичајено мења током рада на издању
  - чак и број итерација се може променити
    - нису пожељне веће промене броја итерација

ЕП: Праћење тока развоја

- Ток развојног процеса се *мери* тако да се лако сагледава напредак
- Уводе се различите нумеричке мере прогреса, на пример:
  - број препознатих корисничких целина;
  - број корисничких целина које су ушле у имплементацију;
  - број имплементираних и прихваћених корисничких целина;
  - број пронађених грешака;
  - број исправљених грешака;
  - број програмских датотека;
  - број класа;
  - број линија кода;
  - број тестова
  - и друге...
- Ову праксу неки аутори запостављају, неки други је истичу

ЕП: Тестови прихватљивости

- Детаљи о корисничким целинама се документују у облику *тестова прихватљивости*
  - Одређује их клијент
  - Пишу се непосредно пре или чак паралелно са имплементацијом исте целине
  - Ако је могуће, пишу се на неком скрипт језику, који омогућава да се изводе аутоматизовано и са понављањем
- Када се тест успешно прође, он се додаје у колекцију положених тестова
  - Они се понављају сваки пут при изградњи система (неколико пута дневно)
  - Тако се обезбеђује да када се захтев једанпут задовољи он више не буде доведен у питање каснијим развојем

ЕП: Програмирање у пару (1)

- Сав *продукциони* код се пише у паровима од по два програмера који раде заједно на истој радној станици
  - један члан тима пише код
  - други у ходу проверава и унапређује код
  - неопходна је интензивна сарадња
  - улоге се често мењају
    - и по више пута у току једног сата
- Чланови парова се мењају бар једанпут дневно
  - током итерације сваки члан тима мора
    - да ради у пару са свим осталим члановима
    - да ради на свим или скоро свим деловима итерације

ЕП: Програмирање у пару (2)

- Последице
  - Веома брзо ширење информација о пројекту међу члановима тима
  - Ширење знања и вештина међу члановима тима
  - Студије показују да се не смањује ефикасност програмера, већ се значајно смањује број грешака
  - Специјалности и даље остају на појединцима, али су и други упознати са резултатима, одлукама и разлозима за њихово доношење

ЕП: Развој вођен тестовима (1)

- Продукциони код се пише са циљем да задовољи тестове јединица (*unit tests*)
  - Почиње се од писања тестова који не пролазе зато што не постоји одговарајућа функционалност
  - Затим се пише код који омогућава да тестови прођу
  - Итерације писања тестова и кода се веома брзо смењују, често на сваки минут
  - Тестови и код еволуирају заједно, тако да тестови буду тек нешто испред кода

ЕП: Развој вођен тестовима (2)

- Последице:
  - Заједно са кодом добија се и веома комплетна колекција тестова
  - Колекција тестова омогућава програмеру да проверава да ли јединица кода ради исправно или не
  - Спречава се настајање грешака у коду приликом накнадних измена
  - Помаже се *рефакторисање*
  - Метод врши притисак да се раздвајају јединице кода, чиме се добија бољи дизајн пројекта

ЕП: Колективно власништво

- Сваки пар има право да провери било који модул и да га унапреди
- Ниједан програмер није појединачно одговоран за било који конкретан модул или технологију
- Свако ради на свим нивоима система, од базе података, преко средњег слоја, па све до кориничког интерфејса
  - то не значи да се не поштују специјалности
  - свако ће највише радити у својој специјалности, али ће моћи да се укључи и у друге целине и да учи о другим специјалностима

ЕП: Непрекидна интеграција (1)

- Интензивна употреба система за управљање верзијама омогућава да свако по више пута дневно узима и поставља нове верзије кода
- Тимови обично користе неограничавајућу контролу кода
  - свако може да преузме и мења било који модул
  - ако је неко мењао у међувремену, програмер је дужан да уклапа своје измене (*merge*)
  - да би се избегла већа уклапања кода, интегрисање се обавља веома често

ЕП: Непрекидна интеграција (2)

- Поступак:
  - пар ради на једном послу сат или два
  - праве тестове и продукциони код
  - у неком погодном тренутку, много пре него што је посао довршен, код се интегрише
    - пре сваке интеграције проверава се да ли пролазе сви тестови
    - ако је потребно, обавља се уклапање кода
  - након интеграције се понавља изградња читавог система
    - дословно читавог
    - у зависности од итерације/издања, ако је потребно, чак се понавља резање *CD*-ова, инсталација софтвера,...
  - затим се понављају сви тестови јединица кода, као и сви тестови прихватљивости из колекције положених

ЕП: Непрекидна интеграција (3)

- Последице
  - изградња система више пута дневно
  - тестирање изграђеног система више пута дневно
  - мала вероватноћа накнадног настајања грешака и њихово лако уочавање
  - значајно једноставнији поступак дебаговања
    - тестови указују да постоји проблем
    - тестови указују на његову просторну и временску локацију
    - историја верзија омогућава сужавање просторне и временске локације настајања проблема

ЕП: Уздржан ритам

- *"Развој софтвера није спринт, него маратон."*
  - Р.Мартин
- Агилни развој промовише устаљен ритам рада
  - Екстремно програмирање *забрањује* прековремени рад
  - Једини изузетак је последња недеља издања
    - "ако је тим довољно близу циља, спринт је допуштен"
- Последице:
  - Бољи односи у тиму
  - Растерећеност чланова тима
  - Мањи број грешака
  - Ефикаснија искоришћеност радног времена

ЕП: Отворен радни простор

- Тим би требало да ради заједно у једној великој отвореној просторији
  - На сваком столу су две или три радне станице
  - За сваком радном станицом су по две столице
  - На зидовима су табле и панои
- Уобичајени тон је тиха комуникација
  - Свако по потреби може да дозове било кога
  - Сви знају ако је неко запао у проблеме
  - Сви могу да интензивно комуницирају
- Последице
  - Иако би се иницијално могло посумњати у ефикасност рада у таквом окружењу, у пракси се показује да такав радни простор значајно подиже ефикасност

ЕП: *Игра* планирања

- Суштина планирања у контексту екстремног програмирања је у подели одговорности између клијента и развојног тима
  - клијенти одлучују шта је значајна карактеристика софтвера
  - развијаоци одлучују колико та карактеристика кошта
- При планирању итерације
  - развијаоци одређују обим (времена и новца)
    - тј. дају горњу процену могућности
  - клијенти у то уклапају корисничке целине
- Са брзим итерацијама клијенти и развијаоци се брзо навикавају и улазе у добар ритам

ЕП: Једноставан дизајн (1)

- Циљ је да дизајн буде што једноставнији и што изражајнији
  - Пажња се посвећује само ономе што је планирано за текућу итерацију
  - Не разматра се оно што ће (можда) доћи касније
- Дизајн система се мења од итерације до итерације
- Развој обично не почиње од инфраструктуре
  - избор базе података обично није прва ствар
  - ни избор средњег слоја обично није прва ствар
  - обично је примарно да прва група корисничких целина проради на *најједноставнији могући начин*
  - инфраструктура се затим додаје према потребама

ЕП: Једноставан дизајн (2)

- Три основна правила:
  - Размотрити прво најједноставније решење које би могло да уради посао
  - Неће бити потребно
  - Једанпут и само једанпут

ЕП: Једноставан дизајн (2)

- Три основна правила:
  - **Размотрити прво најједноставније решење које би могло да уради посао**
    - пример: ако нешто може да се реши помоћу датотека, нема разлога да се одмах уплићу база података или објекти средњег слоја
    - пример: ако нешто може без паралелног израчунавања, онда тако ваља и решити
    - циљ је са што мање рада и у што краћем времену доћи до циља – дизајн ће се унапређивати касније, ако буде потребно
  - Неће бити потребно
  - Једанпут и само једанпут

ЕП: Једноставан дизајн (2)

- Три основна правила:
  - Размотрити прво најједноставније решење које би могло да уради посао
  - **Неће бити потребно**
    - када год се претпоставља да ће нешто бити потребно *можда* или *за неко време*, одговор је “неће бити потребно”
    - нешто се додаје само ако је сасвим извесно да је потребно сада или да ће бити потребно у врло скорој будућности
    - циљ је избегавати компликован дизајн ради нечега што можда никада неће затребати
  - Једанпут и само једанпут

ЕП: Једноставан дизајн (2)

- Три основна правила:
  - Размотрити прво најједноставније решење које би могло да уради посао
  - Неће бити потребно
  - **Једанпут и само једанпут**
    - екстремно програмирање не допушта понављања у коду
    - када год се наиђе на понављање, оно се мора отклонити
      - прављењем метода или класе
    - чак и случајеви када су неки делови кода *веома слични* се морају апстраховати
    - најбољи пут за отклањање редундантности је апстраховање
      - ако су неке две ствари сличне, сигурно постоји апстракција која их обједињује
    - отклањањем редунданци се промовише прављење апстракција и смањивање спрегнутости (*decoupling*) кода

ЕП: Једноставан дизајн (3)

- ВАЖНО:
  - Разматрање најједноставнијих решења и једноставног дизајна никако не значи да код сме бити лоше дизајниран
  - "*Једноставно*" не значи "*на брзину*" ни "*непажљиво*"

ЕП: Рефакторисање

- "*Код је кварљив*"
  - Р.Мартин
- Када се код гограђује, чак и идеалном дизајну временом може да опадне квалитет
- Екстремно програмирање се супротставља опадању квалитета дизајна честим *рефакторисањем*

ЕП: Рефакторисање (2)

- Рефакторисање је предузимање низа малих трансформација кода којима се унапређује структура кода без промене понашања система
  - свака појединачна трансформација је сасвим једноставна, скоро тривијална
  - после сваке трансформације се тестира измењена јединица кода
  - понављају се трансформације све док се не дође до чистог и добро структурираног дизајна

ЕП: Рефакторисање (3)

- Рефакторисање се примењује
  - непрекидно, *у ходу*
  - сваки пут када се уочи да нека измена нарушава постојећи дизајн
  - паралелно са развојем тестова и продукционог кода

(рефакторисање ће бити посебна тема на једном од наредних часова)

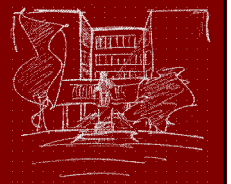
ЕП: Метафора

- Метафора је велика слика читавог система
  - Представља визију система као целине
  - Из ње (не)посредно потичу сви конкретни модули и захтеви
  - Метафора одговара концепту визије у другим методологијама
- Често се формализује у виду речника појмова који идентификују најважније концепте система и проблема који би он требало да реши
  - Тај речник појмова је често симболичног или апстрактног карактера

[P290]

Развој софтвера

Саша Малков



Тема 9.3

Скрам

Скрам

- Методологију ***Скрам*** (*енгл. Scrum*) су развили Кен Швабер и Џеф Сатерленд у првој половини 1990-их
  - представљена је 1995.
  - замишљена је као “лагани развојни оквир”
  - основна идеја потиче из 1980-их, када су Хиротака Такеучи и Икуђиро Нонака дефинисали сличну стратегију за развој производа

Скрам (2)

- Скрам је агилна методологија
- Већину особина дели са другим агилним методологијама
- Не прописује основне технике и праксе, већ их дели са другим методологијама
- Има специфичан приступ организацији рада
  - релативно флексибилан приступ
  - претпоставља способан тим који може да га користи без детаљних инструкција

Вредности

- Углавном засноване на људима:
  - посвећеност
  - фокусираност
  - отвореност
  - поштовање
  - одважност

Тим

- Чине га
  - један ***стручњак за Скрам*** (*енгл. Scrum Master*)
  - један ***власник производа*** (*енгл. Product Owner*)
  - више ***развијалаца***
- У тиму нема даље унутрашње организације
  - нема раздвајања на делове тима или подтимове
  - нема додатне унутрашње хијерархије

Развојни циклус

- ***Власник производа*** наручује рад на неком проблему стављајући га у списак ***необављених послова***
- ***Скрам тим*** реализује изабране послове као ***допринос*** (“увећана вредност”) током ***спринта***
- ***Скрам тим*** и ***улагачи*** процењују резултате и припремају се за наредни ***спринт***
- ***Понављање***

Спринт

- Један ***спринт*** је један заокружен развојни циклус и обухвата остале активности
  - мора да има јасно дефинисане циљеве
  - у току спринта не смеју се уводити промене које доводе у питање циљеве спринта
  - мора да буде временски ограничен, обично на један месец или краће
  - нови спринт почиње одмах по завршетку претходног
  - све остале активности се одвијају у оквиру неког спринта

Планирање спринта

- Спринт започиње ***планирањем спринта***
- Планирање мора да одговори на питања:
  - зашто је тај спринт важан?
  - шта у њему може да се уради?
  - како ће изабрани послови да се ураде?
- Планирање је временски ограничено на највише 8 сати за једномесечни спринт
  - за краћи спринт је уобичајено и краће

Дневни скрам

- Дневни скрам служи за сагледавање тренутног стања и тока спринта
  - траје око 15 минута
  - обично сваког дана на истом месту и у исто време
  - укључује све чланове тима
  - може да обухвати и промене планова

Сагледавање спринта (*Review*)

- Служи за темељније сагледавање резултата спринта
- Обухвата и планирање предстојећих промена
- Учествују тим и улагачи
- То је претпоследња активност у оквиру спринта
- Траје до 4 сата за једномесечни спринт

Ретроспектива

- Служи за сагледавање стања тима током и након спринта
- Циљ је планирање начина за подизање ефикасности тима
- Разматрају се сви елементи рада током спринта:
  - појединачни доприноси
  - односи у тиму
  - алати и послови
- Учествује само тим
- Последња активност у оквиру спринта
- Траје до 3 сата за једномесечни спринт

Артефакти

- Артефакти су јединице посла или вредности
- Сваки артефакт је посвећен нечему
- Основни артефакти су
  - необављени послови – посвећени циљу производа
  - послови спринта – посвећени циљу спринта
  - допринос – посвећен довршавању делова посла

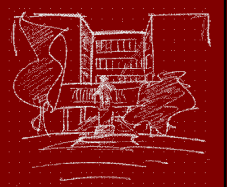
Артефакти (2)

- Необављени послови
  - уређена листа послова које ваља урадити
  - означавају се послови спремни за наредни спринт
  - посвећени су остваривању циља на нивоу производа
- Послови спринта
  - изабрани послови за један спринт
  - посвећени су остваривању циља на нивоу спринта
- Допринос
  - видљив корак напред према испуњењу крајњег циља
  - тежи се да сваки допринос буде и употребљив

[P290]

Развој софтвера

Саша Малков



Тема 10.1

Развој вођен тестовима

Врсте тестова

- Тестови јединица кода (*Unit Test*)
  - Тестирање појединачних делова кода
- Интегрални тестови (*Integration Test*)
  - Тестирање повезаних целина кода
- Тестови система (*System Test*)
  - Тестирање система као целине
- Тестови прихватљивости (*Acceptance Test*)
  - Тестирање система из угла корисника
    - често се изводи помоћу скриптова који симулирају или емулирају кориснички интерфејс

Развој вођен тестовима

- енгл. *Test Driven Development*
- Односи се примарно на тестове јединица кода
- Основни принципи:
  - Тестови претходе коду
  - Систематичност

Тестови претходе коду

- Пре писања било каквог кода, прво се праве одговарајући тестови
- Ниједна функција програма се не развија све док не постоји тест који не успева због њеног одсуства
- Ниједна линија кода се не додаје све док не постоји тест који због ње не успева
- Прво се додају тестови који не пролазе, па тек онда функционалност коју они захтевају и проверавају

Тестови претходе коду - примена

- Свака итерација почиње писањем тестова
  - они често најпре не могу ни да се преведу, зато што не постоје одговарајући методи или класе
- Затим се напише костур кода који омогућава да се тестови преведу али да не пролазе
  - нпр. сви методи враћају неки константан резултат
- Затим се пише код који омогућава да тестови прођу

Тестови претходе коду – примена (2)

- Итерације писања тестова и кода се веома брзо смењују, често на сваки минут
- Пожељно је да тестови и код еволуирају заједно, тако да тестови буду тек нешто испред кода
- Није добро написати велики број тестова без одговарајућег кода

Систематичност

- Свака карактеристика софтвера се мора покрити тестовима
- Сви гранични случајеви морају бити обухваћени тестовима
- Свака линија кода мора бити тестирана

Систематичност (2)

- Накнадно уочавање багова обично је последица недовољно систематичних тестова
  - да су били довољно систематични, тестови би показали проблем, уместо да се касније манифестује као баг

Технике

- Ако се софтвер развија
  - од врха према дну
    - тестира се помоћу *умешака* (*stub*)
      - привремени делови кода који “имплементирају” интерфејс или друге методе тако да не раде ништа
      - једина намена им је да омогуће да се програм преведе, иако неће радити исправно
  - од дна према врху
    - тестира се помоћу *извођача* (*driver*)
      - привремени делови кода који користе мање функционалне целине у одсуству већих функционалних целина чији развој тек следи

Улога тестова (1)

- Тестови су вид верификације
  - Колекција тестова омогућава програмеру да проверава да ли јединица кода ради исправно или не
  - За сваку функцију програма постоје одговарајући тестови који проверавају њену исправност
  - Колекција тестова представља препреку срљању
    - Указује на непотпуности или неисправности
  - Помаже се рано препознавање грешака како на нивоу кода тако и на концептуалном нивоу
  - Метод врши додатни притисак да се раздвајају јединице кода, чиме се добија бољи дизајн пројекта

Улога тестова (2)

- Помаже се рефакторисање
  - Лако се препознаје настајање грешака у коду приликом накнадних измена
  - Онемогућава се промена понашања система при трансформисању кода

Улога тестова (3)

- Више углова посматрања кода
  - Прављење тестова ставља програмера на место корисника кода који се пише
  - У првом плану су интерфејс јединице кода и апстракција њеног понашања
  - Добија се *лако употребљив* код
  - Добија се *лако проверљив* код
    - што је касније веома тешко постићи ако није од почетка добро обликован

Улога тестова (4)

- Тестови су вид документације
  - Представљају облик спецификације захтева
  - Описују услове функционисања кода
  - Описују начин употребе интерфејса јединице кода
    - сваки тест представља пример употребе интерфејса
  - Представљају карактеристичне примере
    - обично тестови почивају на тзв. граничним случајевима
  - Заједно са кодом добија се и веома комплетна колекција тестова
  - Овај вид документације је *увек ажуран*
    - преводи се и проверава са кодом

Улога тестова (5)

- Дебаговање
  - Сваки пут када се уочи нека неисправност која није препозната постојећим тестовима
    - то значи да постојећи тестови нису довољно систематични
    - додаје се нови тест, или више нових тестова
  - Дебаговање се своди на задовољавање тестова
  - За разлику од привремених провера (које се често користе при дебаговању), тестови остају као трајне провере (и потврде) исправности

Шта тестови јединица нису?

- Тестови јединица ***нису довољни*** да докажу исправност софтвера
  - Добро обликовани тестови ће довести до испољавања већине багова
  - Како год да су обликовани, тестови најчешће (скоро никада) не могу покрити све могуће услове у којима се софтвер може наћи
- За доказивање коректности неопходно је употребљавати неке формалне методе (било мануелне или аутоматске)

Јединица кода

- *Јединица кода* може бити
  - операција, функција, метод
  - структура података
  - класа
  - више мањих интегрисаних јединица кода
    - софтверски пакет
    - подсистем
  - спољашњи подсистем
    - тестирање да ли интерфејс одговара спецификацији

Предмет тестирања

- Функционални захтеви
- Зависност постуслова од предуслова
  - Провера да ли јединица кода ради исправно за различите очекиване категорије улаза
    - да ли израчунава исправан излаз?
    - да ли на исправан начин мења стање система?
- Робусност
  - Провера исправности понашања у случају неисправних улазних података
- Интеграција
  - Резултат интеграције мањих јединица кода је већа јединица кода
  - И она може и мора бити предмет тестирања
- Интерфејс спољашњег подсистема
  - Провера да ли интерфејс одговара спецификацији

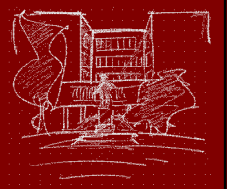
Тестови јединица и *C++*

- Постоји много библиотека:
  - ***CppUnit***
  - *CppUnitLite*
  - *Boost.Test*
  - *Unit++*
  - *CxxTest*
  - *Google Test*
  - *CPUnit*
  - ***Catch2***
- Одличан (мада не најсвежији) преглед више библиотека:
  - ***Exploring the C++ Unit Testing Framework Jungle***, *http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle*
    (ОБАВЕЗНО ПРОЧИТАТИ)

[P290]

Развој софтвера

Саша Малков



Тема 10.2

CppUnit

CppUnit

- Релативно једноставна употреба
- Релативно прилагодљив подсистем
- Омогућава сложене тестове и окружења за тестове
- Подржава изузетке
- Релативно богат скуп провера
- Подржава различите начине извештавања о тестовима
- Подржава библиотеке тестова
- Велики број корисника и солидна јавна подршка

Начела употребе

- У оквиру пројекта
  - направи се нови циљ – програм за извршавање тестова
    - може по један за сваки продукциони циљ, али може и један универзалан
  - дода му се преведена библиотека *CppUnit*
  - напише се главна функција тако да покреће тестове
  - за сваку јединицу продукционог кода пише се по једна или више тест јединица кода
  - сваки пут када се преводи циљни продукциони код преводи се и програм за извршавање тестова

Макрои за тестове

- Скуп макроа за проверавање услова
- У случају грешке избацују изузетке

Макрои за тестове (2)

```
void testVektor()
{
    try {
        Vektor v(1,2,3);
        CPPUNIT_ASSERT( v.getX() == 1 );
        CPPUNIT_ASSERT( v.getY() == 2 );
        CPPUNIT_ASSERT( v.getZ() == 3 );
        CPPUNIT_ASSERT( v.getZ() == 4 );
        cout << "OK" << endl;
    }catch( exception& e ){
        cerr << "ERROR: " << e.what() << endl;
    }
}
```


Макрои за тестове (3)

```
void testVektor()
{
    try {
        Vektor v(1,2,3);
        CPPUNIT_ASSERT( v.getX() == 1 );
        CPPUNIT_ASSERT_MESSAGE(
            "Neispravno X!", v.getX() == 1 );
        CPPUNIT_ASSERT_EQUAL( 2.0, v.getY() );
        CPPUNIT_ASSERT_EQUAL_MESSAGE(
            "Neispravno Z!", 3.0, v.getZ() );
        CPPUNIT_ASSERT_DOUBLES_EQUAL(
            1, v.getX(), 0.0001 );
        cout << "OK" << endl;
    }catch( exception& e ){
        cerr << "ERROR: " << e.what() << endl;
    }
}
```


Тест-случај (*Test Case*)

- Класа са тестовима
- Наслеђује класу
  - *CppUnit::TestCase*
- Конструктору се прослеђује назив
- Главни метод је
  - *void runTest()*

Тест-случај (*Test Case*) (2)

```
class VektorTest : public CppUnit::TestCase
{
public:
    VektorTest() : CppUnit::TestCase( "VektorTest" )
        {}

    void runTest()
    {
        Vektor v(1,2,3);
        CPPUNIT_ASSERT( v.getX() == 1 );
        CPPUNIT_ASSERT_MESSAGE(
            "Neispravno X!", v.getX() == 1 );
        CPPUNIT_ASSERT_EQUAL( 2.0, v.getY() );
        CPPUNIT_ASSERT_EQUAL_MESSAGE(
            "Neispravno Z!", 3.0, v.getZ() );
        CPPUNIT_ASSERT_DOUBLES_EQUAL( 1, v.getX(), 0.0001 );
    }
};
```

Извршавање тестова

- Библиотека обухвата алате за извршавање тестова
- Могу се лако додавати и прилагођавати окружењу
- Основни извршавач тестова за конзолу је:
  - *CppUnit::TextTestRunner*
- Додају му се објекти случајева тестова:
  - *addTest(CppUnit::TestCase*)*
- Покреће се методом
  - *run()*

Извршавање тестова (2)

```
class VektorTest : public CppUnit::TestCase
{
public:
    VektorTest() : CppUnit::TestCase( "VektorTest" )
        {}

    void runTest()
    {...}
};

int main()
{
    CppUnit::TextTestRunner runner;
    runner.addTest(new VektorTest);
    runner.run();
    return 0;
}
```


Свита тестова (*Test Suite*)

- За лакше руковање већим скуповима тестова постоје *свите тестова (Test Suite)*
  - називају се и *комплети тестова*
- Макрои означавају границе свите тестова у оквиру једне класе *CppUnit::TestCase*
  - CPPUNIT_TEST_SUITE(VektorTest);
  - CPPUNIT_TEST(constructorTest);
  - CPPUNIT_TEST_SUITE_END();

Свита тестова (*Test Suite*) (2)

```
class VektorTest : public CppUnit::TestCase
{
    CPPUNIT_TEST_SUITE( VektorTest );
    CPPUNIT_TEST( constructorTest );
    CPPUNIT_TEST_SUITE_END();

    void constructorTest();
};

void VektorTest::constructorTest()
{
    Vektor v(1,2,3);
    CPPUNIT_ASSERT( v.getX() == 1 );
    CPPUNIT_ASSERT_MESSAGE( "Neispravno X!", v.getX() == 1 );
    CPPUNIT_ASSERT_EQUAL( 2.0, v.getY() );
    CPPUNIT_ASSERT_EQUAL_MESSAGE( "Neispravno Z!", 3.0, v.getZ() );
    CPPUNIT_ASSERT_DOUBLES_EQUAL( 1, v.getX(), 0.0001 );
}
```

Извршавање свите тестова

- Макрои за дефинисање свита тестова имплементирају и статички метод за прављење објекта свите тестова:
  - *suite()*
- Извршавачу се не додаје објекат него статички направљена свита:
  - *runner.addTest(VektorTest::suite());*

Извршавање свита тестова (2)

```
class VektorTest : public CppUnit::TestCase
{
    CPPUNIT_TEST_SUITE( VektorTest );
    CPPUNIT_TEST( constructorTest );
    CPPUNIT_TEST_SUITE_END();

    void constructorTest();
};

void VektorTest::constructorTest()
{...}

int main()
{
    CppUnit::TextTestRunner runner;
    runner.addTest( VektorTest::suite() );
    runner.run();
    return 0;
}
```


Уобичајена организација датотека

- Класе програма:
  - className.h
  - className.cpp
- Тестови:
  - className.test.h
  - className.test.cpp
- Програм за тестирање
  - main.cpp или test.cpp или слично

Регистар тестова

- Слабе стране претходне организације:
  - програм за тестирање мора да укључи заглавља свих класа тестова
  - извршавачу се морају додати сви тестови
- Једноставније је са употребом регистра
  - за сваку имплементирану класу тестова се прави по статички објекат чији конструктор врши регистровање теста у јединственом статичком регистру
  - при извршавању се консултује регистар
- Додатна корист – нису неопходна заглавља класа тестова, јер се нигде не користе

Регистар тестова (2)

```
#include <cppunit/extensions/HelperMacros.h>
#include "Vektor.h"

class VektorTest : public CppUnit::TestCase
{
    CPPUNIT_TEST_SUITE( VektorTest );
    CPPUNIT_TEST( constructorTest );
    CPPUNIT_TEST_SUITE_END();

    void constructorTest() {...}
};

CPPUNIT_TEST_SUITE_REGISTRATION( VektorTest );
```


Регистар тестова (3)

```
#include <cppunit/ui/text/TestRunner.h>
#include <cppunit/extensions/TestFactoryRegistry.h>

using namespace std;

int main()
{
    CppUnit::TextTestRunner runner;
    runner.addTest(
        CppUnit::TestFactoryRegistry::getRegistry()
            .makeTest()
        );
    runner.run();
    return 0;
}
```


Припремање тестова

- Некад је потребно да се пре извршавања сваког теста једне свите изврши нека припрема
  - на пример, сложени услови тестирања, велики број објеката и слично
- У таквим случајевима се уместо класе *CppUnit::TestCase* употребљава класа
  - *CppUnit::TestFixture*
- За припремање се користи метод
  - *void setUp()*
  - позива се пре сваког теста
- За деиницијализацију се користи метод
  - *void tearDown()*
  - позива се после сваког теста
- По потреби се дефинишу подаци објекта теста

Припремање тестова (2)

```
class VektorTest : public CppUnit::TestFixture
{
    CPPUNIT_TEST_SUITE( VektorTest );
    ...
    CPPUNIT_TEST_SUITE_END();

public:
    void setUp()
    {
        // ovo se izvrsava pre svakog testa
        ...
    }

    void tearDown() {
        // ovo se izvrsava posle svakog testa
        ...
    }

    void constructorTest() ...
    void eqTest() ...
    void streamTest() ...
};

CPPUNIT_TEST_SUITE_REGISTRATION( VektorTest );
```

Врсте претпоставки (1)

- CPPUNIT_ASSERT(*услов*)
  - Претпоставка да је *услов* задовољен
- CPPUNIT_ASSERT_MESSAGE(*порука*, *услов*)
  - Претпоставка да је *услов* задовољен
  - Кориснички дефинисана порука која се исписује ако услов није испуњен
- CPPUNIT_FAIL(*порука*)
  - Експлицитан неуспех, са датом поруком

Врсте претпоставки (2)

- CPPUNIT_ASSERT_EQUAL(*очекивана*, *стварна*)
  - Претпоставка да су дате две вредности једнаке
  - Морају да буду
    - истог типа
    - да се могу проследити у излазни ток
    - могу се поредити оператором ==
- CPPUNIT_ASSERT_EQUAL_MESSAGE(*порука*, *очекивана*, *стварна*)
  - Као ... али са датом поруком
- CPPUNIT_ASSERT_DOUBLES_EQUAL(*очекивана*, *стварна*, *делта*)
  - Претпоставка да су дате две вредности једнаке, са допуштеном грешком *делта*

Врсте претпоставки (3)

- CPPUNIT_ASSERT_THROW(*израз*, *ТипИзузетка*)
  - Претпоставка да израчунавање датог *израза* избацује изузетак датог типа *ТипИзузетка*
- CPPUNIT_ASSERT_NO_THROW(*израз*)
  - Претпоставка да израчунавање *израза* не избацује изузетак
- CPPUNIT_ASSERT_ASSERTION_FAIL(*претпоставка*)
  - Претпоставка да дата *претпоставка* не успева
  - користи се за тестирање претпоставки
- CPPUNIT_ASSERT_ASSERTION_PASS(*претпоставка*)
  - Претпоставка да дата *претпоставка* успева
  - користи се за тестирање претпоставки

Инсталација

- Пројекат је јаван и отворен:
  - http://sourceforge.net/projects/cppunit/
- Инсталација:
  - Може да се разликује за актуелну верзију!!!
  - преузме се одговарајућа верзија
  - распакује се
  - направи се библиотека одговарајућим преводиоцем
    - cppunit\src*
    - упутства за различите преводиоце постоје на Мрежи

Инсталација – *Eclipse Win32* (1)

- Претпоставка
  - Постоји инсталиран *Eclipse CDT* за *Win32*
- Опис поступка
  - Преузме се актуелна верзија
    - *http://sourceforge.net/projects/cppunit/files/cppunit/1.12.1/cppunit-1.12.1.tar.gz/download*
  - Распакује се у директоријум *CPPUNITROOT*
    - на пример *C:\Programs\cppunit-1.12.1*
      (потенцијално проблематично ако има бланко у путањи)
  - Покрене се програм *MSYS*
  - Пређе се у *CPPUNITROOT*
  - Покрене се скрипт: *./configure*
  - …

Инсталација – *Eclipse Win32* (2)

- Опис поступка
  - …
  - Направи се нови пројекат у оквиру *Eclipse CDT*
    - *File / New / C++ Project*
      - *Project Name = CppUnitLib*
      - *Project Type = Static Library / Empty Project*
    - *Project / Properties*
      - *C/C++ Build / Settings / GCC C++ Compiler / Directories*
        - *Add… (Include Path) / File System… / C:\Programs\cppunit-1.12.1\include*
    - *File / New / Folder*
      - *src*
    - десни клик на нови фолдер / *Import… / General / File System / C:\Programs\cppunit-1.12.1\src\cppunit*
      - означе се све датотеке **.h* и **.cpp*
    - …

Инсталација – *Eclipse Win32* (3)

- Опис поступка
  - …
    - …
    - *Project / Build Project*
    - по потреби се поступак превођења понови за различите конфигурације, да би се добиле и верзија за дебаговање и извршна верзија
- Детаљно упутство
  - *http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=CppUnitWithEclipse*

Употреба (*Eclipse* пројекат)

- Претпоставка
  - преведена је библиотека за одговарајући преводилац
- Направи се нови пројекат
  - *File / New / C++ Project*
    - *Project Name = Test*
    - *Project Type = Executable / Empty Project*
  - *File / New / Source Folder*
    - *src*
  - *File / New / Source Folder*
    - *test*
  - *File / New / Folder*
    - *lib*
  - ископирати из *CppUnitLib/Debug/libCppUnitLib.a* → *lib*
  - *Project / Properties*
    - *C/C++ Build / Settings / GCC C++ Compiler / Directories*
      - *Add… (Include Path) / File System… / C:\Programs\cppunit-1.12.1\include*
    - *C/C++ Build / Settings / MinGW C++ Linker / Libraries*
      - *Libraries / Add… / CppUnitLib*
      - *Libraries Search Path / Add… / Workspace… / lib*

Употреба (2)

- У пројекту се направи датотека *src/main.cpp*

```
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TestRunner.h>

int main( int argc, char **argv)
{
  CppUnit::TextUi::TestRunner runner;
  CppUnit::TestFactoryRegistry& registry =
      CppUnit::TestFactoryRegistry::getRegistry();
  runner.addTest( registry.makeTest() );
  bool wasSuccessful = runner.run( "", false );
  return wasSuccessful;
}
```


Употреба (3)

- У пројекту се направи датотека *test/testVector.cpp*
  - Направимо тестове који описују жељени интерфејс класе *Vector*

```
#include <cppunit/extensions/HelperMacros.h>
#include "../src/Vector.h"

class VectorTest : public CppUnit::TestFixture {
  CPPUNIT_TEST_SUITE( VectorTest );
      CPPUNIT_TEST( testConstruction );
  CPPUNIT_TEST_SUITE_END();
public:
  void testConstruction() {
      CPPUNIT_ASSERT( Vector(10,2).getX() == 10 );
      CPPUNIT_ASSERT( Vector(10,2).getY() == 2 );
  }
};

CPPUNIT_TEST_SUITE_REGISTRATION( VectorTest );
```


Употреба (4)

- У пројекту се направи датотека *src/Vector.h*
  - Имплементирамо класу *Vector* тек толико да прође превођење:

```
class Vector {
public:
   Vector() {};
   Vector(double x, double y) {};
   virtual ~Vector() {};
   bool operator==(const Vector& v)
        { return false; }
   Vector operator+(const Vector& v)
        { return Vector(); }
   int getX() const    { return X; }
   int getY() const    { return Y; }
private:
   int X, Y;
};
```


Употреба (5)

- Ако је потребно извршити више тестова са истим објектима
  - Тестовима додајемо податке и неопходне методе

```
class VectorTest : public CppUnit::TestFixture {
...
private:
   Vector *v_10_1, *v_1_1, *v_11_2;
public:
   void setUp() {
        v_10_1 = new Vector(10, 1);
        v_1_1 = new Vector(1, 1);
        v_11_2 = new Vector(11, 2);
   }
   void tearDown() {
        delete v_10_1;
        delete v_1_1;
        delete v_11_2;
   }
...
```

Употреба (6)

- Према потреби додајемо још тестова

```
class VectorTest : public CppUnit::TestFixture {
   CPPUNIT_TEST_SUITE( VectorTest );
        CPPUNIT_TEST( testConstruction );
        CPPUNIT_TEST( testEquality );
        CPPUNIT_TEST( testAddition );
   CPPUNIT_TEST_SUITE_END();
...
   void testEquality() {
        CPPUNIT_ASSERT( *v_10_1 == *v_10_1 );
        CPPUNIT_ASSERT( !(*v_10_1 == *v_11_2) );
   }
   void testAddition() {
        CPPUNIT_ASSERT( *v_10_1 + *v_1_1 == *v_11_2 );
   }
...
```

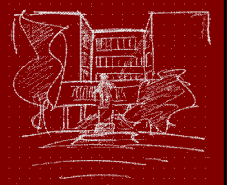

Употреба (7)

- Додајемо уметке у код све док програм не успе да се преведе
  - Ако су методи релативно једноставни, можемо одмах писати исправан код
- Покренемо преведени програм и проверавамо тестове
  - Унапређујемо код док тестови не прођу
- За сваку наредну допуну понављамо поступак
  - Прво додајемо тестове
  - Затим имплементирамо допуне

[P290]

Развој софтвера

Саша Малков



Тема 10.3

Catch2

Catch2

- Бибилиотека *Catch2* је развијена са нешто другачијим циљевима него *CppUnit*
  - Нема тако исцрпну колекцију макроа
  - Једноставнија употреба
  - Олакшано проширивање и прилагођавање
- Не постоји преведена бибилиотека, већ се дистрибуира само заглавље *catch.hpp*
- *Catch2* је у суштини исто што и претходна верзија *Catch*
  - име библиотеке је промењено да би се лакше проналазило, зато што је реч *catch* исувише честа
  - имплементација јесте измењена тако да сада почива више на шаблонима него на макроима

Употреба

- У модулу са тестовима се укључи заглавље:
  `#include "catch.hpp"`
- Напише се тест:

```
TEST_CASE( "...име теста...", "[ознака]" ) {
    REQUIRE( ...услов... );
    REQUIRE( ...услов... );
    REQUIRE( ...услов... );
}
```

- Име теста једнозначно идентификује тест
- Ознаке (тагови) (једна или више) служе за груписање (класификацију) тестова
- Основни облици провере су *REQUIRE* и *CHECK*
  - *REQUIRE* – прекида тест у случају неиспуњења
  - *CHECK* – не прекида тест у случају неиспуњења

Главни програм

- Основни облик главног програма се пише у дословно два реда:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

- Макро у првом реду означава да ће при укључивању заглавља бити направљена функција *main*
  - Наравно, ако желимо можемо да напишемо сопствену функцију, али то обично није потребно

Покретање тестова

```
test [<test name, pattern or tags>...] [options]
```

- Неке опције:
  - `-l, --list-tests` списак тестова
  - `-t, --list-tags` списак ознака
  - `-a, --abort` прекидање после прве грешке
  - `-x, --abortx <no. failures>` прек.после *N* грешака

Напреднија употреба

- Нема потребе да се пишу посебне класе за тестирање
- За све се користе описани тестови

```
TEST_CASE( "...име теста...", "[ознака]" ) {
    ...
}
```

- Посебне конструкције могу да се користе за:
  - писање свита тестова
  - тестирање вођено понашањем

"Свите" тестова и припрема

- Свита тестова се пише у облику:

```
TEST_CASE( "...", "[tag]" ) {
   ... иницијализација (=setUp) ...
   SECTION( "...назив..." ){
      ... исто као и код обичног тестирања ...
   }
   SECTION( "...назив..." ){
      ... исто као и код обичног тестирања ...
   }
   SECTION( "...назив..." ){
      ... исто као и код обичног тестирања ...
   }
   ... деиницијализација (=tearDown) ...
}
```

- Пре сваке секције се прво изврши иницијализација
- После сваке секције се изврши деиницијализација
- Секције могу и да се уграђују једна у другу (са подиницијализацијама...)

Облик "вођен понашањем"

- Идеја писања тестова у облику вођеном према понашању
  - Основа тестирања је "сценарио"
  - Састоји се од једне или више група понашања
  - Свака група понашања почиње иницијализацијом...
  - ...и садржи један или више услова у облику "ако... онда..."
    - део "ако" мења почетну иницијализацију
    - део "онда" проверава услове који након тога морају да важе
- У суштини еквивалентно свитама са иницијализацијом и секцијама, али се јасније раздвајају целине

Облик "вођен понашањем" (2)

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
   GIVEN( "A vector with some items" ) {
      std::vector<int> v( 5 );
      REQUIRE( v.size() == 5 );
      REQUIRE( v.capacity() >= 5 );
      WHEN( "the size is increased" ) {
         v.resize( 10 );
         THEN( "the size and capacity change" ) {
            REQUIRE( v.size() == 10 );
            REQUIRE( v.capacity() >= 10 );
         }
      }
      WHEN( "the size is reduced" ) {
         v.resize( 0 );
         THEN( "the size changes but not capacity" ) {
            REQUIRE( v.size() == 0 );
            REQUIRE( v.capacity() >= 5 );
         }
      }
   }
}
```

- Тестови имају облик реченица
- Имплементација је иста као у случају комплета тестова
- SCENARIO === TEST_CASE
- GIVEN === SECTION
- WHEN === SECTION
- THEN === SECTION

Напредне могућности

- Постоји интерфејс за писање сопствених класа за поређење (*Matcher*), као и неки примери:

```
REQUIRE_THAT( str, EndsWith( "as a service" ) );
REQUIRE_THAT( str, EndsWith( "as a service" )
   || (StartsWith( "Big data" )
       && !Contains( "web scale" )
));
```

- Специјални тагови, нпр: [!hide] [!throws] [!shouldFail]...
- Прилагодљиви извештаји (*XML*,...)
- ...

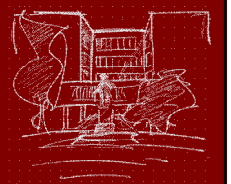
Тестови прихватљивости

- Тестови прихватљивости су концептуално другачији од тестова јединица кода
  - Односе се на тестирање понашања система као целине
  - Не имплементирају се истим алатима
  - Уобичајено се дефинишу неким специфичним скрипт језиком
    - релативно често се употребљава *XML*
  - Обухватају све аспекте употребе система
    - дефинисање података
    - извршавање операција
    - проверавање резултата извршавања операција

Литература за тему

- *Robert C. Martin*, ***Agile Software Development – Principles, Patterns and Practices***, *Prentice Hall, 2003*.
- *Kent Back*, ***Test Driven Development***, *Addison-Wesley Professional, 2002*.
- *Manifesto for Agile Software Development, http://agilemanifesto.org/*
- *Abrahamsson,... –* ***Agile software development methods – Reviews and analysis***, *Espoo, 2002, http://www.pss-europe.com/P478.pdf*
- *Ken Schwaber & Jeff Sutherland,* ***The Scrum Guide***, *https://scrumguides.org/*
- ***Exploring the C++ Unit Testing Framework Jungle***, *http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle*
- ***Catch Tutorial*** *https://github.com/catchorg/Catch2/blob/devel/docs/tutorial.md*

Хвала на пажњи!



МАТФ
Универзитет у Београду
Математички факултет