Testautomatisering

Unit Test, forts + repetition

Idag

- FM:
  - Repetition Unit Test + TDD
  - setup
  - Assertions
  - Unit Test, forts
  - Video?
- EM:
  - Handledning

Kort om labben

- Något om labben:
  - Skriv med allt som jag kräver i info.txt
  - ex: inget namn -> jag måste kontrollera att det är rätt fil -> 10 min kontroll / inlämning => 2h längre tid att rätta labben totalt
  - Skicka inte med onödiga filer
  - ex: två filer med nästan samma namn -> 10 min kontroller / inlämning (och jag kanske väljer fel fil) => 2h längre tid att rätta labben totalt

Kort om labben

- Något om labben:
  - => Summa summarum - om ni följer anvisningarna så går det snabbare för mig att rätta labben och ni får tillbaka resultatet snabbare.
  - => Jag har väldigt knappt med tid den närmsta ~1.5 veckan - så 2h extra för att rätta labben kan innebära lång fördröjning

- Snabbrepetition
- Setup
- Assertions

Repetition – Unit Test

Repetition – Unit Test

- Snabbrepetition
  - Hur?

Repetition – Unit Test

- Snabbrepetition

```
Require ”test/unit”
Requre ”account.rb”

class TestAccount < Test::Unit::TestCase

    def test_new_account_should_have_a_user
            # …
    end

end
```

Repetition – Unit Test

- Snabbrepetition - Testmetod

```
def test_new_account_should_have_a_user
    # Arrange
    test_user = User.new

    # Act
    test_account = Account.new(test_user)

    # Assert
    assert_equal(test_account.user, test_user, "...")
end
```

Repetition – Unit Test

- Setup
  - En metod som körs innan alla test
  - Måste heta ”setup”
  - När vi börjar köra copy/paste eller ser att samma variabel dyker upp igen och igen
    - => använd setup
    - => använd instansvariabler
      - (vi vill att alla metoder i klassen skall ha tillgång)

Unit Test

- TDD
  - Test Driven Development
  - I korthet: vi utvecklar testerna innan implementationen de skall testa

Unit Test

- TDD
  - Red -> Green -> Refactor -> repetera…
  - **Red:** Skriv ett test som ger fail
  - **Green** Skriv tillräckligt utav implementationen för att få pass
  - **Refactor:** Är vi nöjd med implementation och test? Om inte – gör förändringar med de test vi skrivit som stöd.
  - … skriv nästa test

Unit Test

- TDD
  - Vad får vi?
    - För det mesta: En robustare implementation
      - (Mer modulär, etc.)
    - (Vissa) Regressionstester på köpet!
      - (Vi kan behöva ytterliggare regressionstester - men vi bör ha en hyfsad plattform att utgå ifrån här).
    - Mindre skräp
      - (Vi skriver endast tillräckligt med implementation för att testet skall passera)
  - M.m.

- TDD – Exempel
  - Det skall finnas möjlighet att lägga till vänner för en User i vårt system.

Unit Test

- TDD – Varning
  - Vi bör ha någon uppfattning om de övergripande strukturen för vårt projekt innan vi börjar med TDD.

Unit Test

Repetition – Unit Test

- Setup
  - Detta kan innebära att hela ”Arrange”-delen blir tom för vissa test
  - Målbilden:
    - Arrange: 1-3 rader
    - Act: 1 rad
    - Assert: 1 rad
  - Men avvikelser från detta är vanligt
    - Passa er för ”stora” test => bryt isär

Repetition – Unit Test

- Snabbrepetition - setup

```
Require ”test/unit”
Requre ”account.rb”

class TestAccount < Test::Unit::TestCase

  def setup
     @test_user = User.new(”name”,”password”, ”id”)
  end

end
```

- Testmetod innan setup

```
def test_new_account_should_have_a_user
    # Arrange
    test_user = User.new

    # Act
    test_account = Account.new(test_user)

    # Assert
    assert_equal(test_account.user, test_user, "...")
end
```

Repetition – Unit Test

Repetition – Unit Test

- Testmetod med setup

```
def test_new_account_should_have_a_user
    # Arrange

    # Act
    test_account = Account.new(@test_user)

    # Assert
    assert_equal(test_account.user, test_user, "...")
end
```

- Exempel setup

Repetition – Unit Test

Repetition – Unit Test

- Assertion - Repetition
  - http://en.wikibooks.org/wiki/Ruby_Programming/Unit_testing

Repetition – Unit Test

- Assertion
  - assert(Boolean, [Message])
  - Hur gör vi om vi vill kontrollera att en variabel är false?

- Assertion
  - is_logged_in = …
  - assert(!is_logged_in, ””)

Repetition – Unit Test

- Assertion
  - assert_equal(expected, actual, [Message])

Repetition – Unit Test

- Assertion
  - assert_not_equal(expected, actual, [Message])

Repetition – Unit Test

- Assertion
  - assert_nil(object, [Message])

Repetition – Unit Test

- Assertion
  - assert_not_nil(object, [Message])

Repetition – Unit Test

Repetition – Unit Test

- Assertion
  - assert_raise(Exception…){…kod…}
    - Min förväntning är att koden inom blocket skall kasta ett exception

Repetition – Unit Test

- Assertion
  - assert_raise(Exception…){…}
    - Hur tar vi reda på vilket exception som är intressant?
    - Hur ser det ut?

- Mer om Unit Test

Unit Test

Unit Test

- A TRIP:
  - **Automatic**: Kör och kontrollera resultat automatiskt
  - **Thorough**: Testa alla kritiska vägar och scenarion
  - **Repeatable**: Var ej beroende av parametrar vi inte har kontroll över
  - **Independent**: Testa en sak, var inte beroende av andra tester
  - **Professional**: Lätta att förstå och underhålla, Korta, Snabba

- Vad gör vi om vi inte har tid att testa allt eller då vi skall införa Unit Tests i ett Legacy-projekt som saknar Unit Tester?

Unit Test

Unit Test

- Fokusera på två områden:
  - 1. Kritisk kod
    - Beroende på affärsnytta och vanliga användarscenarion
  - 2. Kod som tidigare haft mycket buggar
    - Normalt:
      - Kod som är komplex
      - Kod som ändras ofta
      - Dålig implementation “Fulhack”/”Spaghettikod”

Unit Test

- Code Coverage
- Hur stor del av kodbasen för ett projekt som täcks av enhetstester
  - (Vi kan prata om coverage för andra typer av automatiserade test - men för det mesta så gäller det enhetstester).

Unit Test

- Code Coverage
- Flera problem här
  - Är våra tester vettiga?
    - (Sett till de tekniska aspekterna)
  - Är det vi testar viktigt?
    - (Sett till affärsnytta)

Unit Test

- Code Coverage
- Att avgöra vad som är ett vettigt/viktigt test kan vara komplext
- Enhetstester är normalt billiga att utveckla (de går snabbt att implementera)
- Men - vi bör ägna detta någon tanke

Unit Test

- Code Coverage
- Bör normalt sett inte testas:
  - 3:e partsramverk av olika slag
  - språk-definitionen
    - (Vi kan mycket väl testa ett 3:e partsramverk i ett integrationstest dock)

- Fallgropar – Unit Test
  - UNDVIK: För mycket i ett och samma test
  - Ex:
    assert(…,…)
    assert_equal(…,…,…)
    assert_not_nil(…,…,…)
    assert_equal(…,…,…)
  - : Svårförståeligt
    : Svårt att uppdatera/ta bort

    => dela upp testerna

Unit Test

Unit Test

- Fallgropar – Unit Test
  - UNDVIK: För många objekt i testerna
  - Ex:

```
permissions = Permissions.new
role = Role.new(permissions)
user_status = UserStatus.new
user = User.new(role, user_status, ”test”);
```

  - : långsamt
    : döljer intentionen med testet
    : risk för buggar i test
  - => färre objekt / dela upp testerna / bryt isär implementationen
  - => hjälpklasser. Ex: user = testHelper.user1

Unit Test

- Fallgropar – Unit Test
  - UNDVIK: Mycket brus
  - Ex: user = User.new(1, ”test”, ”test”, ”asdf@asdf.se”, ”normal”, ”first”, ”last”)
  - : döljer intentionen med testet
    : långsammare att utveckla
  - => undvik information som inte krävs för testet
  - => Om du måste ha med informationen: user = UserFactory.build(:name => ”user”)

- Video
- What testers and developers can learn from each other
- http://oredev.org/2011/sessions/what-testers-and-developers-can-learn-from-each-other

Video

Nästa vecka

- BDD + Rspec
- Cucumber
- TDD
- Lab 3

- ...

Fin