Next Generation Testing

Cédric Beust, Google

Alexandru Popescu, InfoQ

QCon

San Francisco

November 2007

Menu

1) TestNG features that we like

2) Designing for testability

TestNG Overview

- Annotations based
- Groups
- Dependent test methods
- Parallel/Multithreaded testing
  - Thread pools, timeouts
- Customizable runtime configuration
- Flexible plug-in API

TestNG features we like

- Groups
- Data Providers
- Dependent tests

→ Groups

- Data providers
- Dependent tests

TestNG Groups

- Problem: configure what tests should be run
- Most of the time, you do this:
  - either by artificial grouping (directory based, name based, etc.)
  - creating configuration like classes that describe the inclusion rules (same as `suite()` in xUnit)

Groups with TestNG

- Test method:
  `@Test(groups={"one", "two"})`

- Configuration method:
  @Before and @After methods can also belong to groups

- Define special group lifecycle methods:
  @BeforeGroup/@AfterGroup

Running groups

- Supported by all TestNG launchers
  - Command line
  - Ant
  - Eclipse (launch configurations can be shared between the team members)
  - IntelliJ IDEA plug-ins
- Exclude running specific groups

Example

```
public class GroupTest {
  @Test(groups={"one", "two"}
  public void commonTest() {}

  @Test(groups={"one"})
  public void groupOneOnly() {}

  @Test(groups={"two"})
  public void groupTwoOnly() {}

  @BeforeGroups(groups={“one”})
  public void beforeGroupOne() {
      // run only before group “one”
  }
}
```

Group categories

Examples of group names:

- test **type**: unit, functional, integration, system, acceptance.
- test **size**: small, medium, large
- **functional description**: web, gui, html, jsp, servlet, database, back-end.
- **speed of the test**: slow, fast.
- **procedural**: check-in, smoke-test, milestone, release.
- **platform**: os.win32, os.linux, os.mac-os
- **hardware**: single-core, multi-core, dual-cpu, memory.1gig, memory.10gig
- **runtime schedule**: week-days, weekends, nightly, monthly

Hints on using groups

- Groups are not mutually exclusive

```
@Test(groups = { "fast", "database"})
@Test(groups = { "slow", "database" })
```

- Use regular naming pattern for groups

```
@Test(groups = { "os.linux.debian" })
@Test(groups = { "database.table.ACCOUNTS" })
@Test(groups = { "database.ejb3.connection" })
```

- TestNG has the ability to parse regular expressions to locate the groups you want to run
  - running the groups "database.*" will run all the database tests
  - or narrow down the set of tests to "database.ejb3.*"

- Groups

→ Data Providers

- Dependent tests

Data Providers

- Data Providers allow you to separate data from the logic of your tests
- Data can come from Java, flat file, database, network, etc…
- You can have as many Data Providers as you want (e.g. “string-provider”, “url-provider”, etc…)

What makes a Data Provider?

- Use the @DataProvider annotated a method with @DataProvider
- Method must return Object[][]
- Name the data provider to be used by your test method:

  ```
  @Test(dataProvider=“clusters”)
  ```
- TestNG will handle the type conversions

@DataProvider example

Directory made of .properties file:

cluster1.properties, cluster2.properties, etc…

Property file example: (host=port)

```
169.1.3.2=6552
169.5.12.3=2002
```

@DataProvider Example

```
@Test(dataProvider=”hosts")
public void verifyHost(Properties settings){
  Enumeration keys = settings.keys();
  while (keys.hasMoreElements()) {
    String host =
      settings.keys.nextElement();
    String port =
      settings.getProperty(host);
    // perform test on host/port
  }
}
```

@DataProvider Example

```
@DataProvider(name = “hosts”)
public Object[][] loadHosts() {
   File rootDir = new File("root");
   String[] names= rootDir.list(new FilenameFilter() {
     public boolean accept(File dir, String name) {
       return name.endsWith(".properties");
     }
   });

   Object[][] result = new Object[names.length][];
   for(int i= 0; i < names.length; i++) {
     Properties prop = new Properties();
     prop.load(new FileInputStream(new File(rootDir, names[i])));
     result[i] = new Object[] {prop};
   }
   return result;
 }
```

- Groups

- Data providers

→ Dependent tests

Method dependency

- Problem:
  - certain test methods depend on the success of previous methods
  - you don't want to duplicate your efforts while writing tests
- Example: DAO testing:
  - One method to launch the server: embedded DB/connect to DB
  - One test method to test if the table to work on is available
  - Methods to verify functionality insert(), findById(), update(), delete()

Example

```
public class DaoTest {
  @BeforeMethod initConnections() {}
  @Test public void insert() {}
  @Test public void findById() {}
  @Test public void deleteById() {}
}
```

Problems:

- `initConnections()` fails
- 4 FAILURES
- What we want: 1 FAILURE, 3 SKIPS

Example

```
public class DaoTest {
   MyDao dao;

   @BeforeClass public void initConnections() {}

  @Test public void isSetupOk() {
    assert dao.getConnection() != null;
  }

  @Test(dependsOnMethods={"isSetupOk"})
  public void insert() {}

  @Test(dependsOnMethods={"insert"})
  public void findById() {}
}
```

Problems:

- Doesn't scale very well
- Breaks if you refactor

Example

```
public class DaoTest {
  @BeforeClass public void initConnections() {}

  @Test(groups= "prepare")
  public void isSetupOk() {
    assert dao.getConnection() != null;
    // ...
  }

  @Test(groups="create", dependsOnGroups="prepare")
  public void insert() {}

  @Test(groups= "retrieve", dependsOnGroups = "create")
  public void findById() {}
}
```

Benefits:

- Method names can change
- Easy to add future test methods

What groups give you

- a way to order methods;
- order not just individual methods, but collections of methods grouped logically
- a mechanism to accurately report failures due to failed dependency
- a way to exactly reproduce the failure scenario

Conclusion

- Groups, Data Providers and Dependent Tests are very popular features
- TestNG has many more features, see for yourself!

http://testng.org

Designing for Testability

Do we need to design for testability?

Unfortunately, yes!

Requires forethought and giving up on certain ideas

What’s so hard about testing?

Identifying the enemy

Statics! In all shapes: singletons, global variables, static fields

Extreme encapsulation

Enemy #1 : Statics

Hard to test:

```
void f() {
  Database db =
    Database.getInstance();
  db.query("DELETE FROM ACCOUNTS");
}
```

Statics

Better

```
void f(Database db) {
  db.query(”…”);
}
```

Statics

Product:

```
 db = Database.getInstance();
 f(db)
```

Test:

```
 db = new Mock(Database.class);
 f(db);
```

Even better

Use a dependency injection framework!

Highly recommended: Guice ("juice"), by Bob Lee.

```
void f(@Inject Database db) {
  db.query("...");
}
```

Spring also an option

Enemy # 2:
Extreme encapsulation

Everything private and final

Reasonable from an OO perspective

Adversely impacts testing

Questioning existing practices

Beware of certain design patterns such as Singleton or Abstract Factory

It's okay to open up a class to make it more testable (package protected is your friend!)

And now…

The big elephant in the room…

Test-Driven Development!!!

Test-driven development

Show of hands:

Who…

1) Writes tests **first** most of the time?
2) Writes tests **last** most of the time?
3) Does a mix of both?

TestNG and TDD

Perfect project for a TDD approach

Yet, only ~10% of the tests I wrote were developed using TDD

Is it just me?

Problems with TDD

Promotes micro-design over macro-design

Hard to apply in practice

No clear evidence that it produces better designs than "tests last"

TDD promotes micro-design

Focuses on the immediate problem at hand

"Simplest thing that could possibly work" can lead to short-sighted designs

Risk of churn (throw-away code)

TDD is hard to apply in practice

Forces you to a design that might be good for testing but not optimal for your users (or even yourself)

Makes you spend a lot of time with compilation and IDE errors (negates IDE benefits)

Counter-intuitive

TDD: good or evil?

Great to train junior programmers or non-test savvy developers

Not so great for more experienced developers

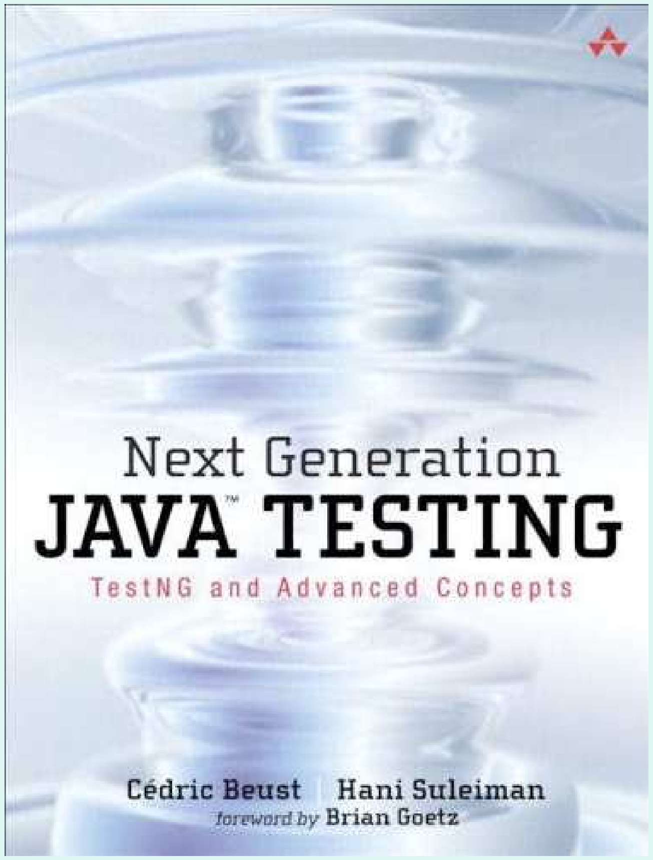
Conclusion

When in doubt, remember that tests are for users, not for developers

Be open to giving up on some established software engineering practices

Don't feel bad if you're not using TDD

One last thing:



Available from Amazon.

Thank you for your attention!

Questions?