Continuous integration from the ground up

GILLES QUERRET – RIVERSIDE SOFTWARE

About the speaker

- Pronounced \ʒil.ke.ʁe\
- Started Riverside Software in 2007
- Continuous integration and source code analysis in OpenEdge

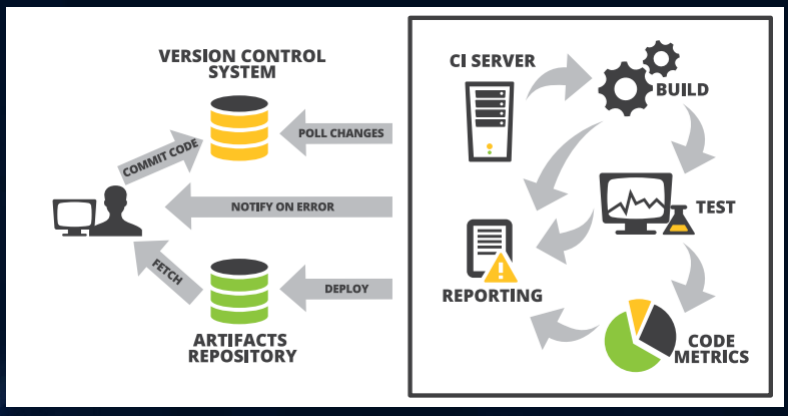
Continuous integration

Continuous Integration is a software development practice of performing software integration frequently…several times a day, in fact. Ideally, your software application or system should be built automatically after each commit into a shared version control repository.

Upon each successful build, the system integrity should be verified using automated tests that cover if not all, then at least most of the functionality. If some tests fail, the developer responsible is notified instantly and the problem can be identified and solved quickly.

Using this approach, you can deliver working and reliable code to the customer much faster, while also mitigating the risk of releasing unstable, buggy software to your users.

Continuous integration

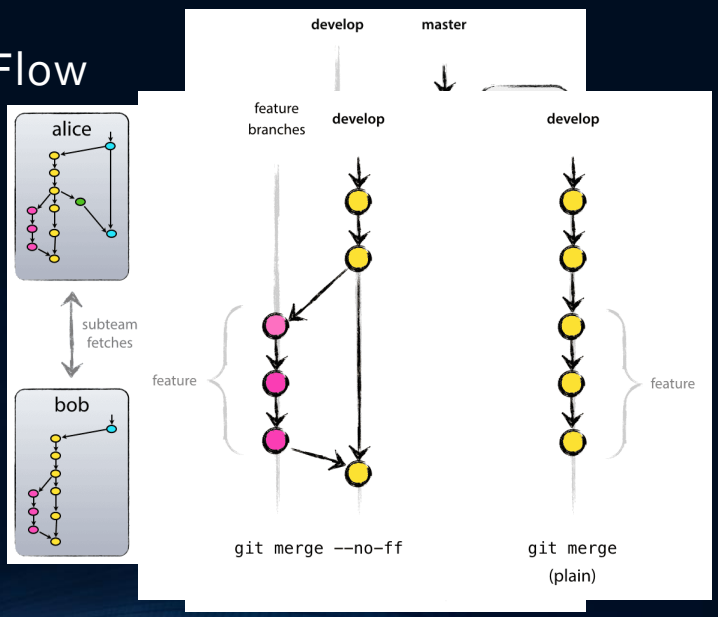


Source code repositories

Source code repositories

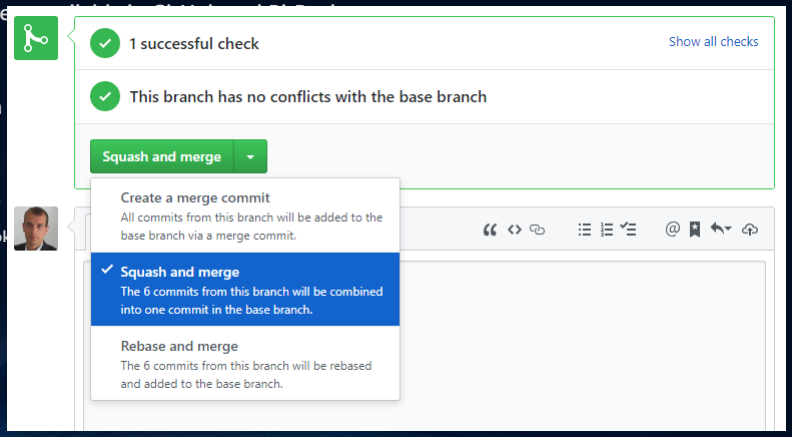
- If you can generate something, don't store it in the repository
  - Such as rcode, DLL,
- Separate requirements and dependencies
  - OpenEdge is a requirement, your custom assemblies are dependencies
- Don't forget database versioning
- Associate a bug tracker to your code repository

Git / GitFlow



Pull requests

- Conce
- Ask a
- Code
  - Look



Build automation

- Progress Developer Studio or the AppBuilder are not build tools
- Check out source code, then execute a single command line to generate a standalone binary

Build automation

- Looking for something stable, flexible and portable across platforms? Use Ant !
- Ant is an open-source product to deal with software builds
- XML based syntax, and provides lots of standard tasks
- PCT is an open-source extension to deal with the OpenEdge environment
  - Included in OpenEdge 11.7.3 !

Build automation

```
<PCTCreateBase dbName="ged" destDir="${db}"
   codepage="utf" schemaFile="db/schema1.df,db/schema2.df"
   structFile="db/struct.st" blockSize="4"
   dlcHome="${DLC}" />



<PCTRun procedure="src/initDb.p" paramFile="conf/param.pf"
     dlcHome="${DLC}" cpstream="utf-8">
  <DBConnection dbName="ged" dbDir="${db}" singleUser="yes" />
  <DBConnection dbName="cust" dbDir="${db}" singleUser="yes" />
</PCTRun>
```

Build automation

```
<PCTCompile destDir="${build}" graphicalMode="true" dlcHome="${DLC}"
    md5="false" minSize="false" cpstream="iso8859-15"
    inputChars="16384" debugListing="true">

  <fileset dir="src/core" includes="**/*.p,**/*.w" />
  <fileset dir="src/module1" includes="**/*.p,**/*.w" />
  <fileset dir="src/oo" includes="**/*.cls" />

  <PCTConnection dbName="ged" dbDir="${db}" />
  <PCTConnection dbName="cust" dbDir="${db}" />

  <propath>
    <pathelement location="src/core" />
    <pathelement location="src/oo" />
  </propath>
</PCTCompile>
```

Continuous integration servers

- Many products, but same functionalities
  - Define and trigger jobs
  - Store deliverables (and keep history)
  - Make them easily accessible
  - Keep users informed of build result
- Choosing a CI server :
  - Free or not
  - Integration with your tools
  - Plugins

Continuous integration servers

- Always use a clean server
- Use distributed jobs
- Ideally, you should have one job configuration for all branches
- Keep deliverables only for production jobs. Keep only a dozen for integration
- Only send alerts for failures !

Jenkins pipelines

- History of job kept in Jenkins
- Jobs executed on multiple nodes
- Large number of branches

- Jobs described in Groovy-like DSL
- Stored in code repository
- Jenkins monitors branches on repository

Jenkins pipelines

```
#!groovy
stage ('Build') {
  node ('windows') {
    checkout([
      $class: 'GitSCM',
      branches: scm.branches,
      extensions: scm.extensions + [[$class: 'CleanCheckout']],
      userRemoteConfigs: scm.userRemoteConfigs
    ])
    withEnv([
      "PATH+ANT=${tool name: 'Ant 1.9', type: 'hudson.tasks.Ant$AntInstallation'}/bin",
      "DLC=${tool name: 'OpenEdge-11.7', type: 'jenkinsci.plugin.openedge.OpenEdgeInstallation'}" ]) {
      bat "ant -DDLC=%DLC% -lib Z:\\Tools\\PCT\\PCT-Latest.jar build"
    }
  }
}
```

Jenkins pipelines

```
[pct:compile] PCTCompile - Progress Code Compiler
[pct:compile] Error compiling file 'Z:/jenkins-public/Windows-Node1/workspace/OSS-PDO-App_TestGilles-
47S633NPQMWFB7CHZWUZKO6U4VXMZNTCMF5W3IRADOJJG737C7SQ/src/mainwin.w' ...
[pct:compile]  ... in main file at line 382 column 5
[pct:compile]      RN mainwin2.r.
[pct:compile] -----^
[pct:compile] ** Unable to understand after -- "RN". (247)
[pct:compile]
[pct:compile] 168 file(s) compiled
[pct:compile] Failed to compile  1  file(s)
```

Automated tests

- From cheap to expensive:
  - Unit testing
  - API or service tests
  - UI tests
- Make sure that unit tests are executed as part of the standard build pipeline
- And failures trigger an alert

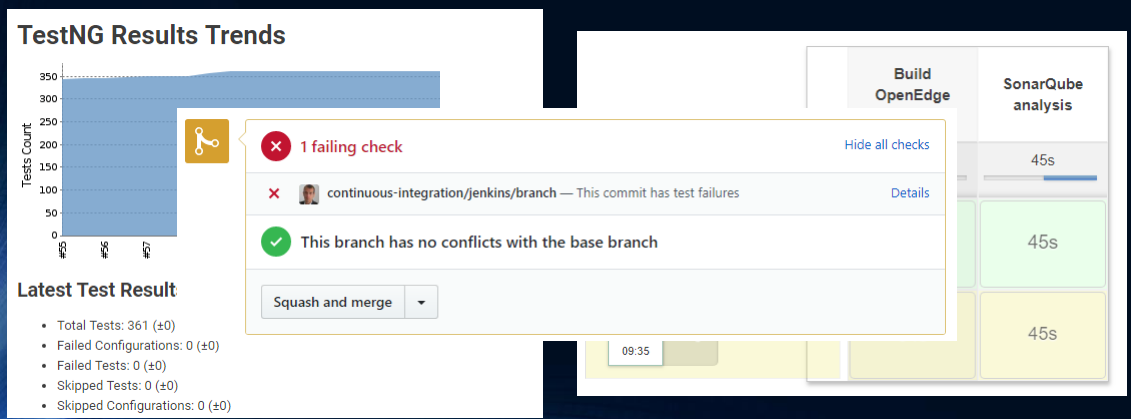
Automated tests

- Yesterday – ABL Unit Testing Part 1 by Mike Fechner
- Yesterday – ABL Unit Testing Part 2 by James Palmer

- Explanation of the various frameworks
- How to write tests
- How to mock components

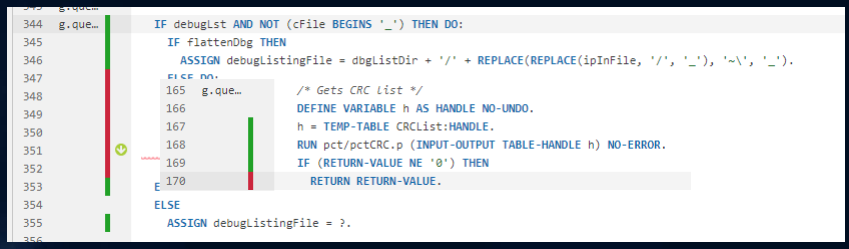
Automated tests

- Unit tests failure has to be trapped by the CI server

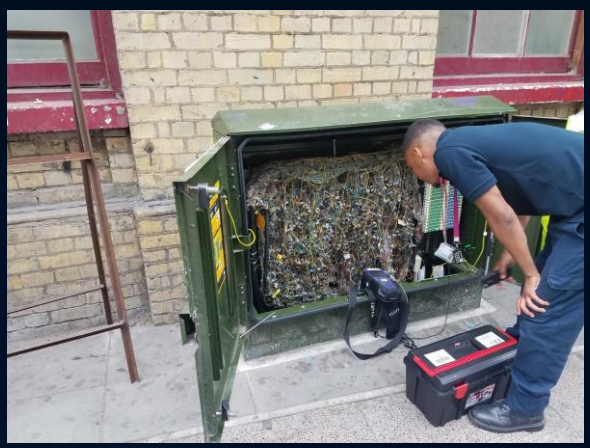


Automated tests – Code coverage

- Lines of code executed during tests compared to total number of lines of code

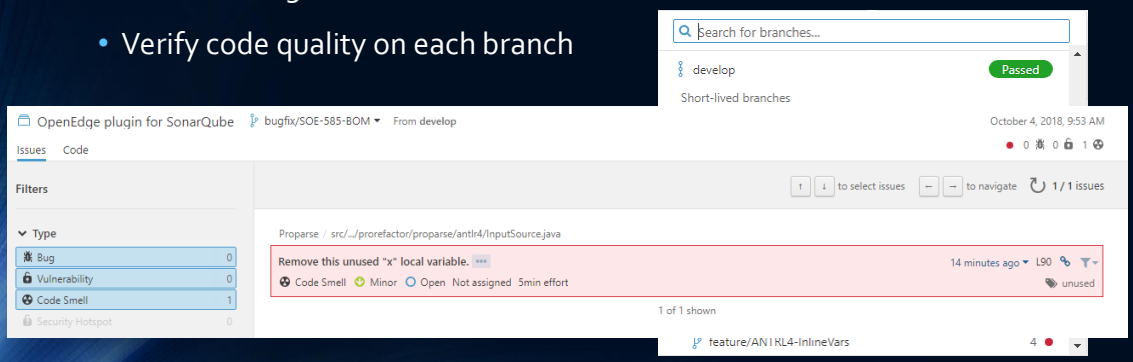


Source code analysis



Source code analysis

- Code coverage on tests
- Verify code quality on each branch



Automated deployment

- Virtual machines (vmware for example)
- Use clean VM with prerequisites being installed
- Use snapshots

- Docker for server-side deployment