

DNA: Enabling Agility

Enabling agility through artefacts which can identify themselves and their provenance.



1. Table of Contents

1. TABLE OF CONTENTS	2
2. OVERVIEW.....	3
3. EXAMPLE SCENARIOS	4
3.1. CASE 1: DETERMINING ENVIRONMENT STATUS.....	4
3.1.1. Scenario	4
3.1.2. Solution.....	4
3.2. CASE 2: PRODUCTION DEFECT – EMERGENCY FIX.....	4
3.2.1. Scenario	4
3.2.2. Solution.....	4
3.3. CASE 3: INCORRECT VERSIONS DEPLOYED?	5
3.3.1. Scenario	5
3.3.2. Solution.....	5
3.4. CASE 4: WAS THIS EVER TESTED?	5
3.4.1. Scenario	5
3.4.2. Solution.....	5
4. THE MECHANISMS	7
5. ADDING DNA TO THE CODEBASE	9
5.1. .JAVA FILES	9
5.2. WEB.XML	9
5.3. APPLICATION.XML.....	9
5.4. OTHER XML FILES	9
5.5. PROPERTIES FILES	10
5.6. NEW AND EXISTING FILES	10
5.7. EXTRACTING DNA FROM BUILT ARTIFACTS.....	10
5.7.1. Java Reflection	11
5.7.2. Textual Extraction.....	11
6. AUTOMATING DNA ADDITION	12

2. Overview

Agile Methodologies rely heavily on automation, and favour working software over comprehensive documentation. On the face of it, this goes against normal SCM practices which rely heavily on comprehensive documentation so that baselines can be recreated.

Not knowing exactly what versions of software are deployed in an environment is wasteful, time-consuming and costly yet it is a common occurrence in Agile projects. Test Managers have no idea the baseline they are testing against, and production defects cannot be quickly traced to their original source components.

This document describes a zero-cost mechanism for being able to provide this information, and to do so in a way that supports Agile methods.

In order to unambiguously identify exactly which artefacts have been deployed into an environment, it is necessary to be able to track every artefact back to the version control repository.

Normally, this is managed by way of some kind of extrinsic database – such as a spreadsheet. On large and complex projects, this can quickly become cumbersome.

This document describes DNA – a zero-cost mechanism which delivers a number of immediate benefits:

- 1. Enabling Agility**
Fully-automated mechanisms integrate with Agile project methods to provide all of the benefits of good SCM practice without the encumbrance of manual record-keeping;
- 2. Known Environment Status**
The status of all environments is known at all times through the automated creation of accurate Bills of Materials;
- 3. Time and Cost Savings**
Time is not wasted trying to identify exactly which source components are producing a particular defect. This is particularly important in the case of defects requiring emergency patching;
- 4. Complete Audit Capability**
The complete history of every artifact, in every environment, is known and can be tracked all the way back to the individuals who have worked on it;
- 5. Effective Quality Control**
Deployed code can easily be associated with the test cycles it passed through en route

3. Example Scenarios

In the following examples, a number of things have been omitted for simplicity:

1. Release Management procedures
2. Branching and merging structure
3. Automated build and deployment
4. Preparation of release notes

since these are not directly related to DNA, although the existence of DNA makes these processes much more efficient.

3.1. Case 1: Determining Environment Status

3.1.1. Scenario

A System Test or UAT run is about to start. The test managers wish to have absolute confidence that fixes to approved defects have been deployed and are ready for testing.

3.1.2. Solution

A search of the SCM logs provides details of which artefacts have been modified as part of fixing each particular defect, along with the unique revision number of each artefact.

The DNA of the deployable archives is queried, and the unique revision numbers are matched up.

Confirmation that the defect fixes have been deployed is very quickly provided to the Test Manager.

3.2. Case 2: Production Defect – Emergency Fix

3.2.1. Scenario

A defect has been found in production, incorrect links being displayed on a page for instance. This defect needs to be fixed and re-deployed immediately.

3.2.2. Solution

Development indicate the individual file(s) that are causing the problem. The DNA of the application deployed in production is queried to identify the unique

revision numbers of these files, and it is very quickly confirmed that these are the versions on the release branch in the SCM repository.

Developers check out the release branch and fix the defect in the correct revisions of the files. Nobody has to perform mental gymnastics about which actual revisions these may be.

The files are checked in, automatically built and deployed. The fixed defect is immediately available for re-testing.

3.3. Case 3: Incorrect Versions Deployed?

3.3.1. Scenario

A defect is raised (in test or production) which developers believe has already been fixed and should have been deployed. There is confusion about which versions of the source artefacts are actually deployed.

3.3.2. Solution

The DNA of the deployed components is interrogated, and the following information is immediately available:

1. Exactly which version of the artefact is in the archive;
2. Exactly which version of the artefact is deployed;
3. Who worked on the artefact, when and what changes were made

As an example, it could quickly be established that the artefact was checked in (and built) after the deployed archive was loaded into the environment (perhaps due to a post-freeze change, or a mis-communication in when the deployment was to happen).

3.4. Case 4: Was This Ever Tested?

This case relies on external project events being recorded in the SCM repository – this Best Practice is described in another document.

3.4.1. Scenario

A defect is found in production and the question is asked – “has this ever worked anywhere before? Has it ever been tested?”

3.4.2. Solution

The offending artefacts are identified by developers, and their DNA is obtained. The revisions in the SCM repository are then interrogated to establish:

1. When the artefacts were tested, and in which environment(s);
2. Which test cases, and test data, were used in the test run;
3. The results of the test runs

and the answer to the “*was this ever tested?*” becomes immediately obvious.

4. The Mechanisms

The mechanisms used to implement the above capabilities will be referred to as “**DNA**” throughout this document.

Like its real-world counterpart, DNA allows artefacts in the system to be unambiguously identified with 100% accuracy. If the DNA is injected at the build stage, then this information comes for free – there is no need to perform any additional work.

It relies on a built-in feature of the SCM repository called keyword substitution. This means that cardinal tokens within the source files are replaced.

The table below gives an example of the tokens that are automatically replaced by a Subversion¹ SCM repository.

\$LastChangedDate\$	This keyword describes the last time the file was known to have been changed in the repository, and looks something like \$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) \$. It may be abbreviated as \$Date\$.
\$LastChangedRevision\$	This keyword describes the last known revision in which this file changed in the repository, and looks something like \$LastChangedRevision: 144 \$. It may be abbreviated as \$Revision\$ or \$Rev\$.
\$LastChangedBy\$	This keyword describes the last known user to change this file in the repository, and looks something like \$LastChangedBy: jbirtley \$. It may be abbreviated as \$Author\$.
\$HeadURL\$	This keyword describes the full URL to the latest version of the file in the repository, and looks something like \$HeadURL: http://zfs-dev3/svn/ep/trunk/README \$. It may be abbreviated as \$URL\$.
\$Id\$	This keyword is a compressed combination of the other keywords. Its substitution looks something like \$Id: README 148 2007-08-01 21:30:43Z

¹ Similar keywords are substituted in other SCM systems, such as CVS, Perforce etc.

	jbirtley \$, and is interpreted to mean that <i>the file README was last changed in revision 148 on the evening of August 1, 2007 by the user jbirtley.</i>
--	---

Table 1: Recognised Keywords

5. Adding DNA To The Codebase

This section describes how to add DNA to the files in the codebase, and offers some suggestions of where the DNA should be located for maximum effect.

5.1. .java Files

Java source files are the most straightforward, and the place where the biggest gains are to be found.

Java files should have code added similar to the following:

```
/**
 * DNA identification
 *
 * The names are prefixed with double underscores to
 * avoid potential conflict with anything else in the class.
 */
public static String __className = new
    Throwable().getStackTrace()[0].getClassName();
public static String __id = __className + ": $Id$";
public static String __rev = __className + ": $Revision$";
public static String __url = __className + ": $HeadURL$";
```

5.2. Web.xml

The DNA of the web.xml could be as follows:

```
<display-name>BuildMonkeyWAR ($Id$) [@TAG@]</display-name>
```

This is to ensure that it can be easily identified in an Application Server console.

5.3. Application.xml

The DNA of the application.xml could be as follows:

```
<display-name>AutomationService ($Id$) [@TAG@]</display-name>
```

This is to ensure that it can be easily identified in an Application Server console.

5.4. Other Xml Files

Other .xml files, unless they have some kind of display-name type property which can contain arbitrary text not affecting behaviour, need to have the DNA embedded into comments:

```
<!-- $Id$ -->
<!-- $Revision$ -->
<!-- $headURL$ -->
```

5.5. Properties Files

The DNA of `.properties` files could be as follows:

```
#####
# ($Id$)
# ($HeadURL$)
# ($Revision$)
#####
```

This can be interrogated from the filesystem, or by recursively interrogating an archive file using UCPLs².

5.6. New and Existing Files

New files need nothing extra to be done in order for Subversion to automatically replace the keywords with values.

Existing files need to have keyword substitution enabled explicitly with the following command:

```
svn propset
  svn:keywords
  "Id Revision HeadURL LastChangedDate LastChangedRevision
  LastChangedBy" <FILENAME...>
```

5.7. Extracting DNA From Built Artifacts

Deployed Artifacts

A very simple web application, with a very small footprint, can be created to be embedded inside all built `.ear` files.

This web application will load java classes (or properties or resources) using the built-in class loader and will then use Java reflection to interrogate and display the DNA.

This web application will not affect normal operation of the system in any way, and can be password-protected.

² Universal Configuration Point Locators [RFC-UCPL, BuildMonkey 2007]

5.7.1. Java Reflection

Java reflection allows Java classes to be interrogated from within the Java VM in order to find out which methods and fields they possess.

By using reflection to check for the existence of named fields, and retrieving their values, we can easily identify the DNA of any individual Java class.

Archive files (such as `.ear`, `.war` and `.jar`) can easily be recursively interrogated to provide this information.

5.7.2. Textual Extraction

Command-line tools, such as the ubiquitous `grep(1)`, can be used to extract string content from files in the file system.

Universal Config Point Locators provide the ability to extract information quickly, record it and compare it in real-time.

This can be used for any type of file on any system.

6. Automating DNA Addition

In most situations, it is preferable to automate the addition of DNA injection as part of the build process. Not only is this more efficient, it also eliminates human error.

A number of ANT tasks, and command-line utilities, are available from BuildMonkey to fully automate DNA injection, manipulation and reporting. Please contact dna@buildmonkey.com for details.