NSpace Design Principles

(M. Simpson, 6/30/2005)

Generalities

Abstraction controls overall complexity, by encapsulating implementation details of individual components behind well-defined interfaces. It allows layering within the overall design, which facilitates development by promoting code reuse and minimizing what an individual developer must learn in order to contribute. By cleanly separating specification from implementation, it protects related components from individual implementation changes.

Flexibility accommodates change and growth by allowing individual implementors to customize and enhance their instance through well-defined mechanisms, and contribute those enhancements back to the core design as they reach maturity, for reuse and further development by other implementors. It also promotes adoption by allowing the definition of a simple set of core functionalities with a minimum set of deployment prerequisites.

Modularity allows a complex project to be understood as a hierarchical assemblage of simpler components. It promotes independence between individual components, insulating them from code changes in their neighbors, and minimizing the component's required knowledge of its place in the system as a whole. It facilitates granularity and simplicity of each component, enhancing maintainablity.

Elegance enables simplicity of use and ease of maintenance, which in turn enhance overall reliability. An elegant design allows for simple, clean enhancement and extension.

Specifics

Use a small set of **model** types (Container, Bitstream, Persistent Identifier) to represent data within the system.

Use a small set of **transaction** objects (Container Management, Bitstream Archiving, Persistent Identifier) to represent operations to be performed against the model.

Split the architecture into cleanly-divided **layers** (frontend, backend; service, chain, support), to handle logically separate parts of the transaction cycle, and to allow development within a given layer to become immediately useful to other layers without reimplementation.

Define **components** in terms of required functionality (Java interfaces) rather than specific implementations (Java classes), so that development on a specific component implementation can occur without affecting other components, and to allow implementors to assemble components as needed to support local requirements.

Use a transparent **framework** to allow simple runtime configuration and management of individual components, and determine which components become part of a given instance. Minimize how much a given component must know about the framework to do its job.

An **implementor** wants something that is simple to understand and powerful to use.

A **developer** wants something that is simple to understand and easy to extend.

> The only way to write complex software that won't fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local and you can have some hope of fixing or optimizing a part without breaking the whole.
>
> Eric S. Raymond, *The Art of Unix Programming*

NSpace Framework Technology Primer

(M. Simpson, 6/30/2005)

Representational State Transfer (REST):

Costello, Roger L. "Building Web Services the REST Way." *XFront* website, 3 January 2003.
<http://www.xfront.com/REST-Web-Services.html>.

Fielding, Roy T. "Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation, University of California, Irvine, 2000.
<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

Fielding, Roy T. and Richard N. Taylor. "Principled Design of the Modern Web Architecture". *ACM Transactions on Internet Technology*, Vol. 2, No. 2, May 2002.
<http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf>.

Baker, Mark, Jeff Bone, et al. *RestWiki* website.
<http://rest.blueoxen.net/cgi-bin/wiki.pl>.

Staged Event-Driven Architecture (SEDA):

Welsh, Matt. "SEDA: An Architecture for Highly Concurrent Server Applications". *Harvard Electrical Engineering and Computer Science* website, 22 July 2003.
<http://www.eecs.harvard.edu/~mdw/proj/seda/>.

Welsh, Matt. "An Architecture for Highly Concurrent, Well-Conditioned Internet Services". Ph.D. Thesis, University of California, Berkeley, August 2002.
<http://www.eecs.harvard.edu/~mdw/papers/mdw-phdthesis.pdf>.

Morissette, Jean, et al. *JCyclone Project* website.
<http://jcyclone.org/>.

Inversion of Control Design Pattern (IoC):

Fowler, Martin. "Inversion of Control Containers and the Dependency Injection Pattern". Personal website, 23 January 2004.
<http://www.martinfowler.com/articles/injection.html>.

Spille, Mike. "Inversion of Control Containers". *Pyrasun 2.0 - The Spille Blog*, 6 November 2004.
<http://www.pyrasun.com/mike/mt/archives/2004/11/06/15.46.14/>.

Hammant, Paul, et al. *PicoContainer Project* website.
<http://www.picocontainer.org/>.

Current Project Dependencies:

XML Support for REST: jdom 1.0 <http://www.jdom.org/>
HTTP Support for REST: commons-httpclient 3.0-rc2 <http://jakarta.apache.org/commons/httpclient/>
Required by HttpClient: commons-codec 1.3 <http://jakarta.apache.org/commons/codec/>
Required by HttpClient: commons-logging 1.0.4 <http://jakarta.apache.org/commons/logging/>

Lexer/Parser Generator: antlr 2.7.5 <http://www.antlr.org/>
IoC Framework: picocontainer 1.1 <http://www.picocontainer.org/>
SEDA Framework: seda 3.0 <http://www.eecs.harvard.edu/~mdw/proj/seda/>
RDBMS Services: derby 10.0.2.1 <http://incubator.apache.org/derby/>
Logging Infrastructure: log4j 1.2.8 <http://logging.apache.org/log4j/>

Global Unique Identifier Generation: jug 1.1.2 <http://jug.safehaus.org/>

NSpace Architectural Overview

(M. Simpson, 6/30/2005)

A **service** receives transaction objects from one or more frontends and delivers them to the associated support chain for processing.

Following processing, the service returns the completed transaction object to the frontend that originated the transaction.

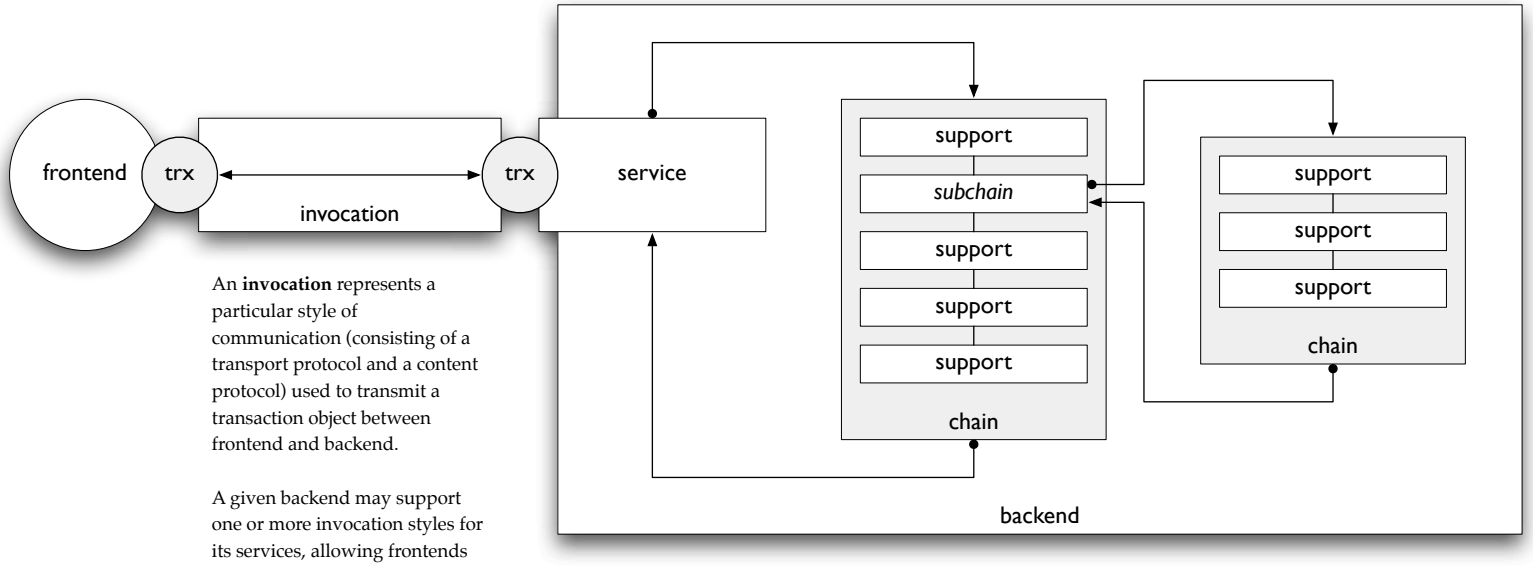
A **chain** consists of one or more supports configured to perform the processing for a particular service. Transaction objects pass through the chain, with each support given an opportunity to perform work on the transaction in turn.

Chains may be nested, allowing the creation of **subchains** representing processing steps common to several different services.

A support module, or more simply, a **support,** receives a transaction object, performs work on that object, and returns it for further processing by other supports in the chain.

A **frontend** acts as a client to one or more backends, creating transaction objects and invoking them against backend services.

Frontends can be written in any language for which transaction objects and invocations exist, and may be local or remote in relation to a given backend.



An **invocation** represents a particular style of communication (consisting of a transport protocol and a content protocol) used to transmit a transaction object between frontend and backend.

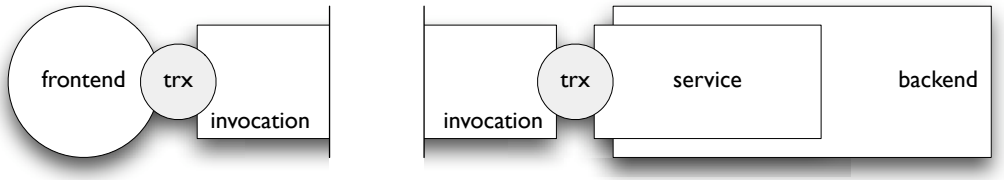
A given backend may support one or more invocation styles for its services, allowing frontends to communicate with it via several different protocols.

A **backend** acts as a server to any number of frontends, maintaining mappings between services and chains of supports and controlling the flow of transaction objects through the system.

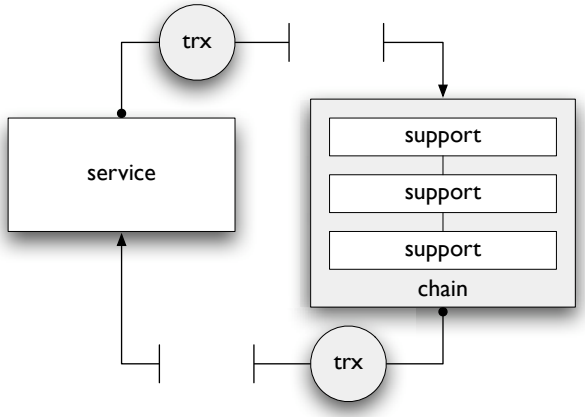
A **transaction** represents a single discrete request addressed to a particular service on a particular backend, and the response to that request. Each transaction type is associated with a set of **commands** that represents a specific functionality within a domain of related functions.

Points of Isolation in the NSpace Architecture

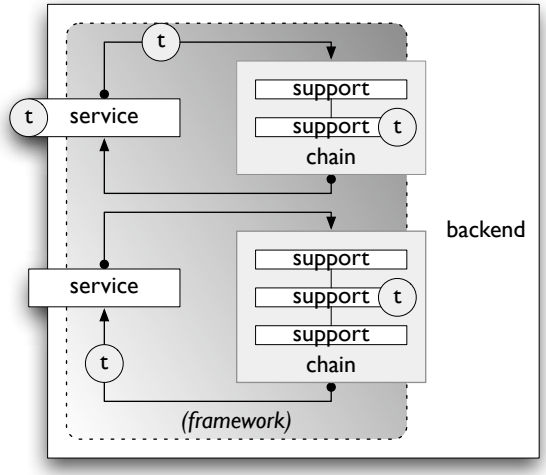
(M. Simpson, 6/30/2005)



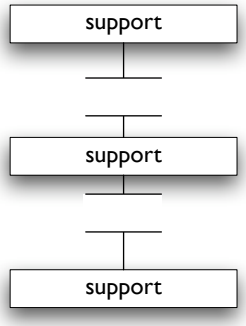
The **invocation isolation** between frontend and backend guarantees that client and server implementations are independent. Given support in the invocation layer, frontends may be written in arbitrary languages, and communicate successfully via various protocols with multiple backends without either client or server knowing implementation details about their partner. Frontend and backend implementations may evolve independently, because transaction objects and the invocation layer decouple client and server, insulating each from side effects of changes made to the opposing code.



The **service isolation** within the backend guarantees that a service remains independent of implementation details within its supports. A service delivers an uncompleted transaction object to its associated chain, and expects a completed transaction object in return, but is ignorant of the processing details that were required to complete the transaction. Services also remain isolated from each other, in that a given service implements a discrete domain of functionality, independent of all other services.



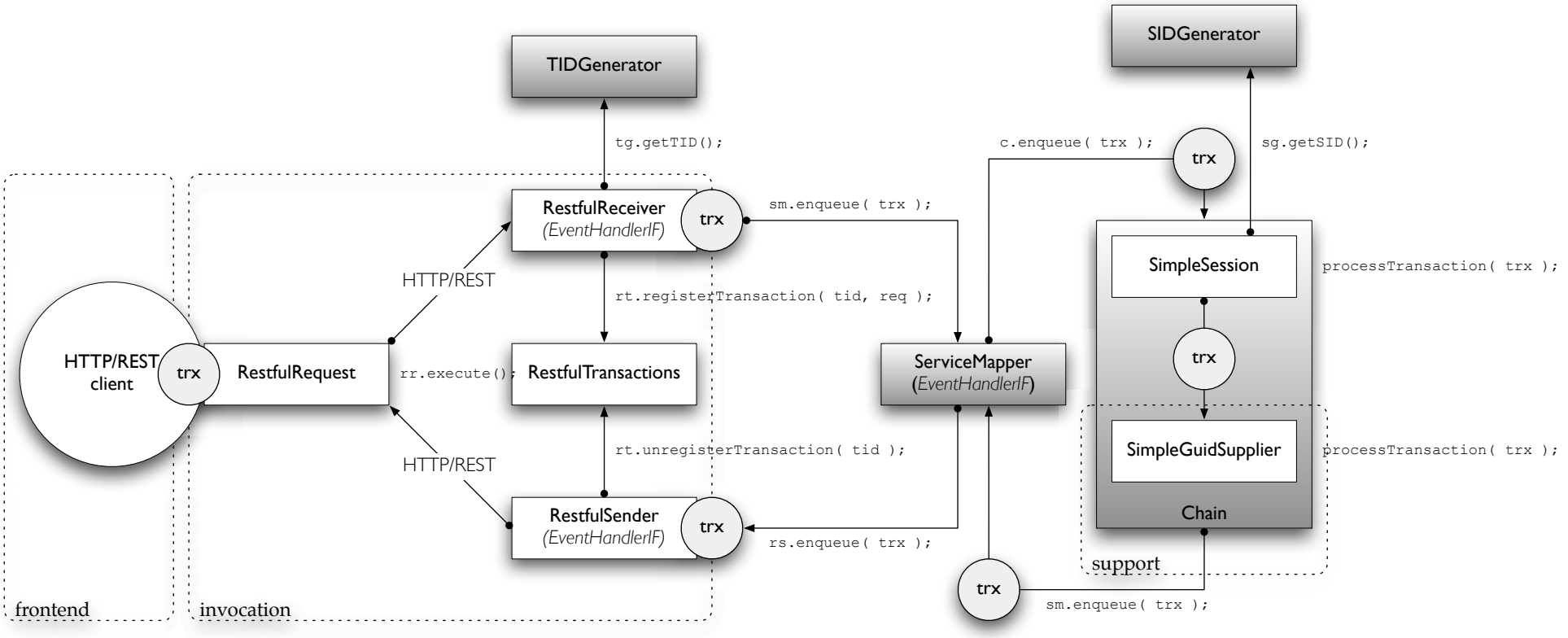
The **framework isolation** within the backend guarantees that runtime configuration, transaction queuing, and adaptive load balancing amongst the various backend components are all handled transparently to both the implementor and the developer. Framework development and tuning can occur independently of service configuration and support development, which are isolated from code changes made to the framework layer.

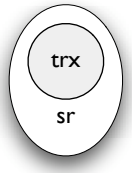


The **support isolation** within a service chain guarantees that a particular support module remains independent from the implementation details of all other supports, and unaware of its specific usage within the context of a particular backend's runtime environment. Supports can thus implement discrete processing tasks at a highly granular level, if desired, and an individual support implementation can evolve independently from all other support implementations.

A Simple But Complete NSpace Transaction

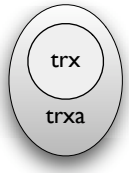
(M. Simpson, 6/30/2005)





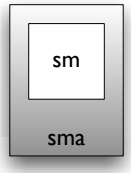

```
public final GUIDTransaction
  implements NSpaceTransaction { ... }

public final RestfulRequest
  implements NSpaceServiceRequest { ... }
```




```
public final GUIDTransaction
  implements NSpaceTransaction { ... }

public final NSpaceTransactionAdapter
  implements QueueElementIF { ... }
```

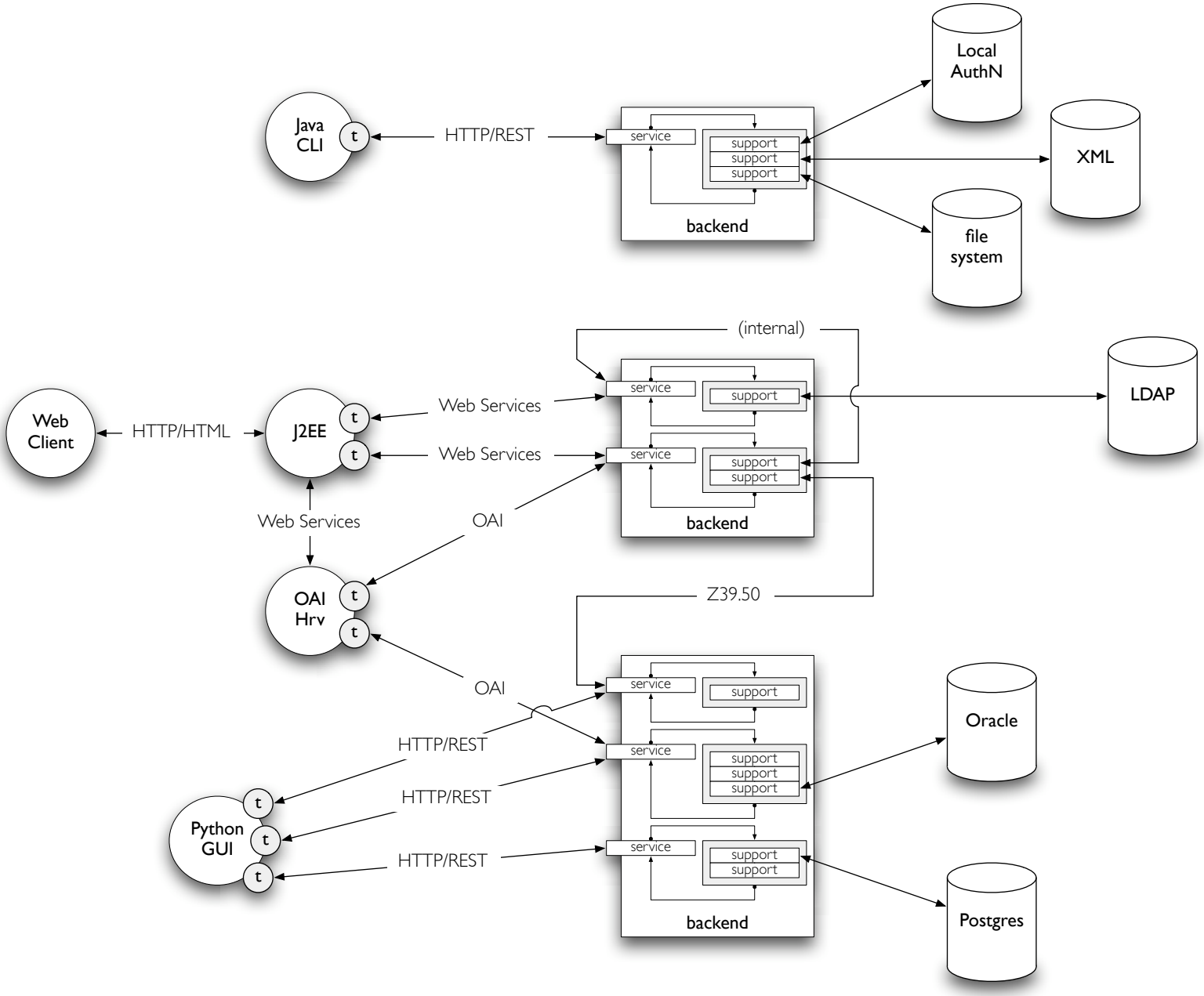



```
public final SimpleGuidSupplier
  implements NSpaceSupportModule { ... }

public final NSpaceSupportModuleAdapter
  implements EventHandlerIF { ... }
```

Distributed Interacting Repository Network

(M. Simpson, 6/30/2005)



NSpace Development Infrastructure

(M. Simpson, 6/30/2005)

Generalities

The project should use a **central repository and version control system** for all source code. The repository should support a proper client/server model, have decent access control, and allow for easy branching, merging, and tagging.

The project should use a consistent, centralized, **project-level build manager** to provide a uniform build and testing environment.

The use of at least one popular, widely available **integrated development environment** should be extensively documented.

Unit and regression testing should be used to formalize project requirements, ensure code quality, and provide automated real-time reporting on current code status.

A **consistent coding style** should be used for all core code. An automated code formatter should be used to enforce this style without placing undue extra workload on individual developers.

There should be **several centralized project communication channels** (mailing list, wiki, blog, issue tracking, code browsing, project documentation, etc.) to allow efficient and appropriate information flow. These communication channels should be able to be integrated at a high level for individualized monitoring.

Infrastructure components should already have or support the development of plugins to facilitate **cross-component integration**.

Specifics

Use **Subversion** as a source code repository and version control system.
<http://subversion.tigris.org/>

Use **Maven** for project management and as an automated build environment.
<http://maven.apache.org/>

Use **Eclipse** as a primary development environment; use the Subversion and Maven plugins for Eclipse to cleanly integrate version control and project management tasks into the development environment. Provide installation and configuration documentation to help new developers get started on the project.
<http://www.eclipse.org/>

Use **JUnit** for the unit and regression testing framework; use the JUnit plugin for Maven to automate nightly testing.
<http://www.junit.org/>

Create a "house style" code formatting guide, and use **Jalopy** to format repository code into the recommended styles; use the Jalopy plugin for Maven to automate code reformatting.
<http://jalopy.sourceforge.net/>

Use **Trac** for integrated collaborative documentation development, issue tracking, code browsing, and project timeline and reporting; use **b2evolution** for developer and project status logging and commentary.
<http://www.edgewall.com/trac/>
<http://b2.evolution.net/>

Use **RSS** feeds for all of the above components to tie together information flow and change notification across the project as a whole.
<http://blogs.law.harvard.edu/tech/rss>

> This suggests that peer production will thrive where projects have three characteristics ... [T]hey must be divisible into components, or modules, each of which can be produced independently of the production of the others ... [T]he modules should be predominately fine-grained, or small in size ... Heterogeneous granularity will allow people with different levels of motivation to collaborate by making smaller- or larger-grained contributions, consistent with their levels of motivation ... [F]inally, a successful peer production enterprise must have low-cost integration, which includes both quality control over the modules and a mechanism for integrating the contributions into the finished product.
>
> Yochai Benkler, *Coase's Penguin, or, Linux and the Nature of the Firm*

Developer Jumpstart Guide

(M. Simpson, 6/30/2005)

The general outline of the steps necessary to begin developing the NSpace prototype is as follows:

(1) Install prerequisite applications.

NSpace development requires that a number of supporting applications be installed and functioning properly on your development workstation. The current recommended development platform is:

JDK 1.4.2, Subversion 1.1.4, Maven 1.0.2, Eclipse 3.0.2

You will also need the following Eclipse plugins:

JavaSVN Library 0.8.8.1 (MacOS X only), JavaSVN Subclipse Extension 0.8.8.1 (MacOS X only)
Maven 1.0.2 (optional), Mevenide 0.3.1
Subclipse 0.9.30, Version Control with Subversion 1.1.1 (optional)

To obtain the above, once Eclipse is installed, use the "Software Updates" functionality to add these update sites:

http://tmate.org/svn/
http://subclipse.tigris.org/update
http://mevenide.codehaus.org/release/eclipse/update/3.0/

You should then be able to easily obtain the latest versions of all of the above Eclipse plugins.

(2) Check out or export the Subversion archive.

Use the command line interface to Subversion to either check out or export a copy of the NSpace codebase. If you are not planning on contributing code back to the project, export the latest tagged version:

```
$ svn export --username <name>  svn://murmur.doit.wisc.edu/ats/lira/npace/tags/v.0.2.0 nspace
```

If you are planning on contributing to the development efforts, checkout the latest HEAD from the trunk of the project:

```
$ svn checkout --username <name> svn://murmur.doit.wisc.edu/ats/lira/nspace/trunk nspace
```

(3) Use Maven to generate Eclipse configuration files.

Change into the top-level directory of the source code, and use Maven's Eclipse plugin to generate Eclipse project configuration files for each of the NSpace subprojects:

```
$ cd nspace
$ maven -Dgoal=eclipse multiproject:goal
```

This will also download and cache all of the JAR dependencies for NSpace in your local Maven repository cache.

(4) Import NSpace subprojects into Eclipse.

Start Eclipse and go to the workbench. In the "Navigator" pane, right-click and choose "Import...", then "Existing Project into Workspace", then browse to the "core" directory underneath the top-level "nspace" directory of your Subversion export or checkout. Click the "Finish" button to import the Maven subproject into an Eclipse project. Do the same for the two remaining subprojects ("contrib", "local").

IMPORTANT NOTE: Make sure you DON'T import the top-level directory: you want to IMPORT EACH MAVEN SUBPROJECT SEPARATELY as Eclipse projects. This is the recommended way to keep Eclipse, Maven, Subversion, and all of the Eclipse plugins playing together happily.

Developer Jumpstart Guide, cont.

(M. Simpson, 6/30/2005)

After you have imported the projects, if you are going to be doing development (i.e. if you used "svn checkout" to get your working copy) you need to have Eclipse recognize the files as versioned checkouts: in the "Navigator" pane, right-click on the "core" project, pick "Team", "Share Project...", "SVN" as the repository type, and click "Next" to let Eclipse automatically discover and interpret the ".svn" control files. Then click "Finish", and enter your Subversion username and password if prompted. Repeat these steps for the "contrib" and "local" Eclipse projects.

(5) Tour the code.

I recommend you start with the documents in these subdirectories:

core/src/site/sdocbook
core/src/site/xdoc

As you read through them, take the time to look through the referenced classes in the appropriate packages.

It is probably also useful to print out the diagrams here:

core/src/site/diagrams

and use them for reference while reading the other documentation.

(6) Begin developing.

I recommend you follow this daily routine while developing:

(a) Before you begin working, update your working copy with the latest changes, by changing to the top-level checkout directory ("nspace", if you followed the steps above), and doing "svn update" in the command-line client.

(b) (optional) Occasionally, if new dependencies have been added to the project by yourself or other developers, you may need to regenerate the Eclipse project files to add them to the compilation classpath, by changing to the top-level checkout directory and doing "maven -Dgoal=eclipse multiproject:goal".

(c) As you work, do frequent commits to the archive from inside Eclipse (from the "Navigator" pane, right-click on a file or tree, and pick "Team", "Commit".

(d) When you're finished for the day, after you've shut down Eclipse, do a final commit from the top-level "nspace" directory, using the command-line client ("svn commit").

Note: this guide was created from the content of the "DEVELOPERS" document at the top level of the NSpace project heirarchy in the main source code repository on 30 June 2005. If you are reading this in hardcopy, it is absolutely certain that it is out of date, and many of the specifics (site names, path names, etc.) may have drifted in the intervening time. Contact Mike Simpson <mike.simpson@doit.wisc.edu> for the most current information on getting started with NSpace development. Thank you for your interest.