# Reassessment of Java Code Management

*Random Thoughts*

Andrey Petrov

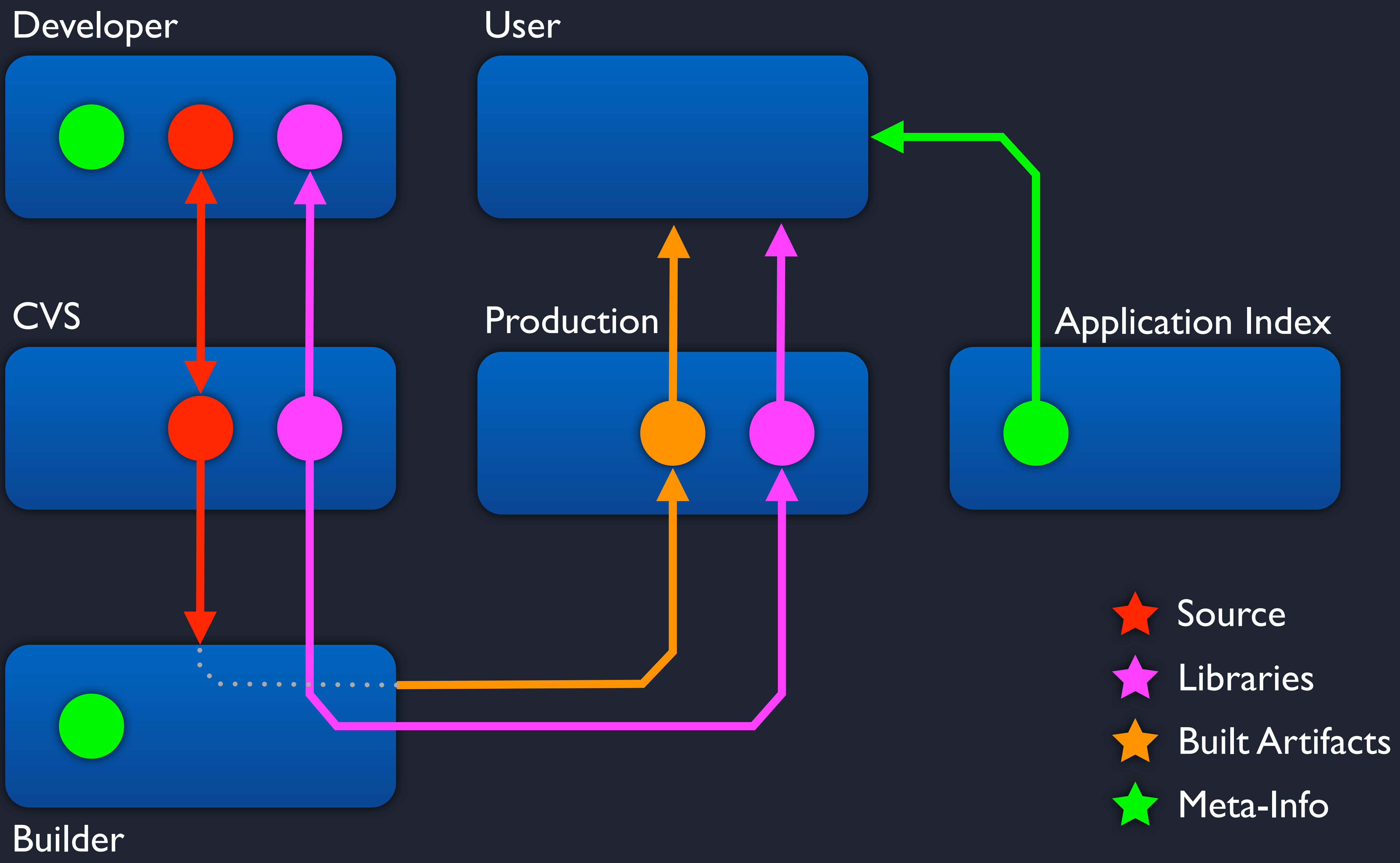
January 8, 2010

## Principal Issues

- Customization of favorite IDEs.
- Lack of common project templates.
- Guessing the purpose, scope, dependencies, and build nuances of existing units of code.
- Deployment over a network share.
- Volume of Webstart downloads.
- All-at-once builds.
- Lack of a staging (test) environment.
- Stupid CVS.

## Code Infrastructure

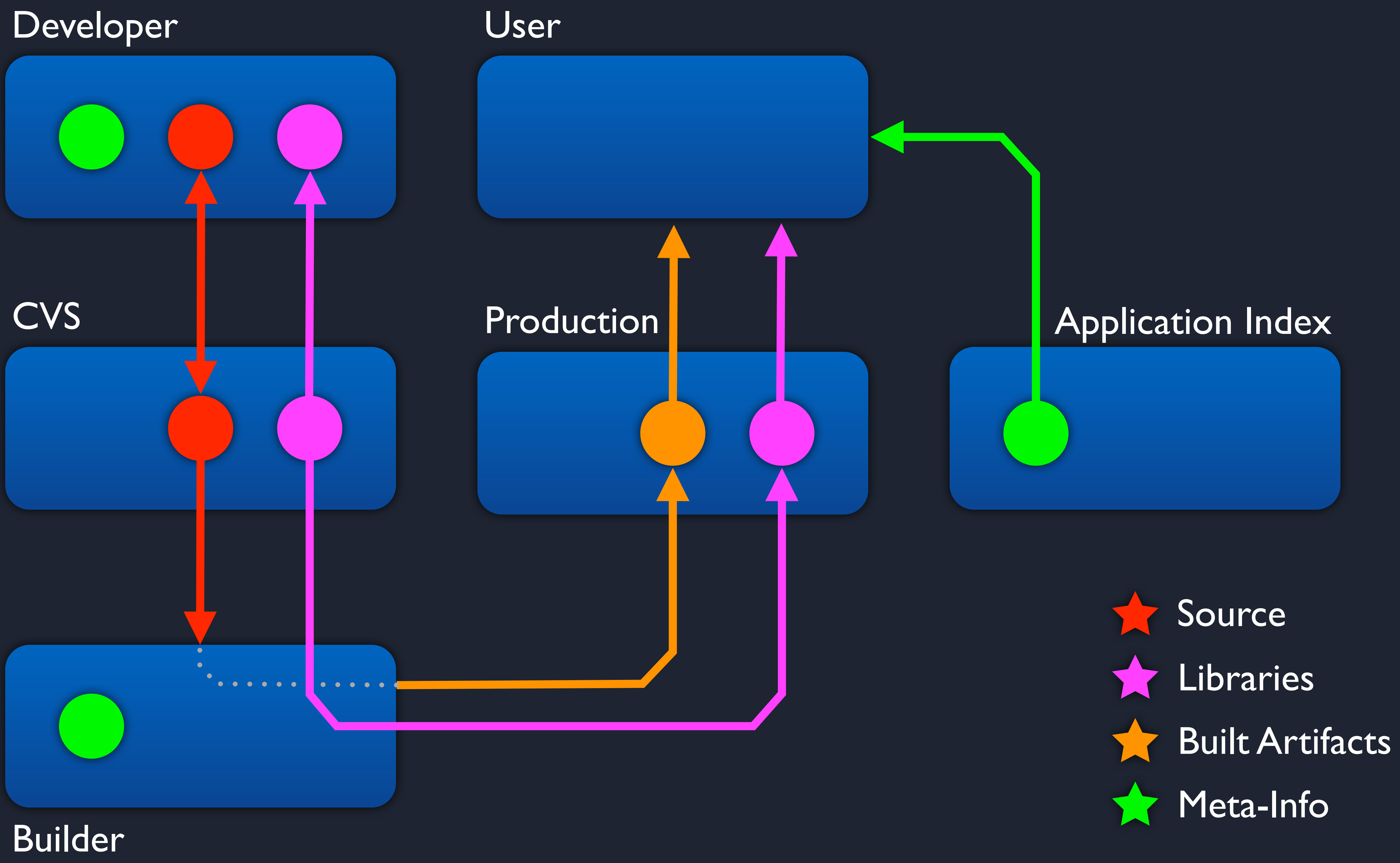
# How It Works Now



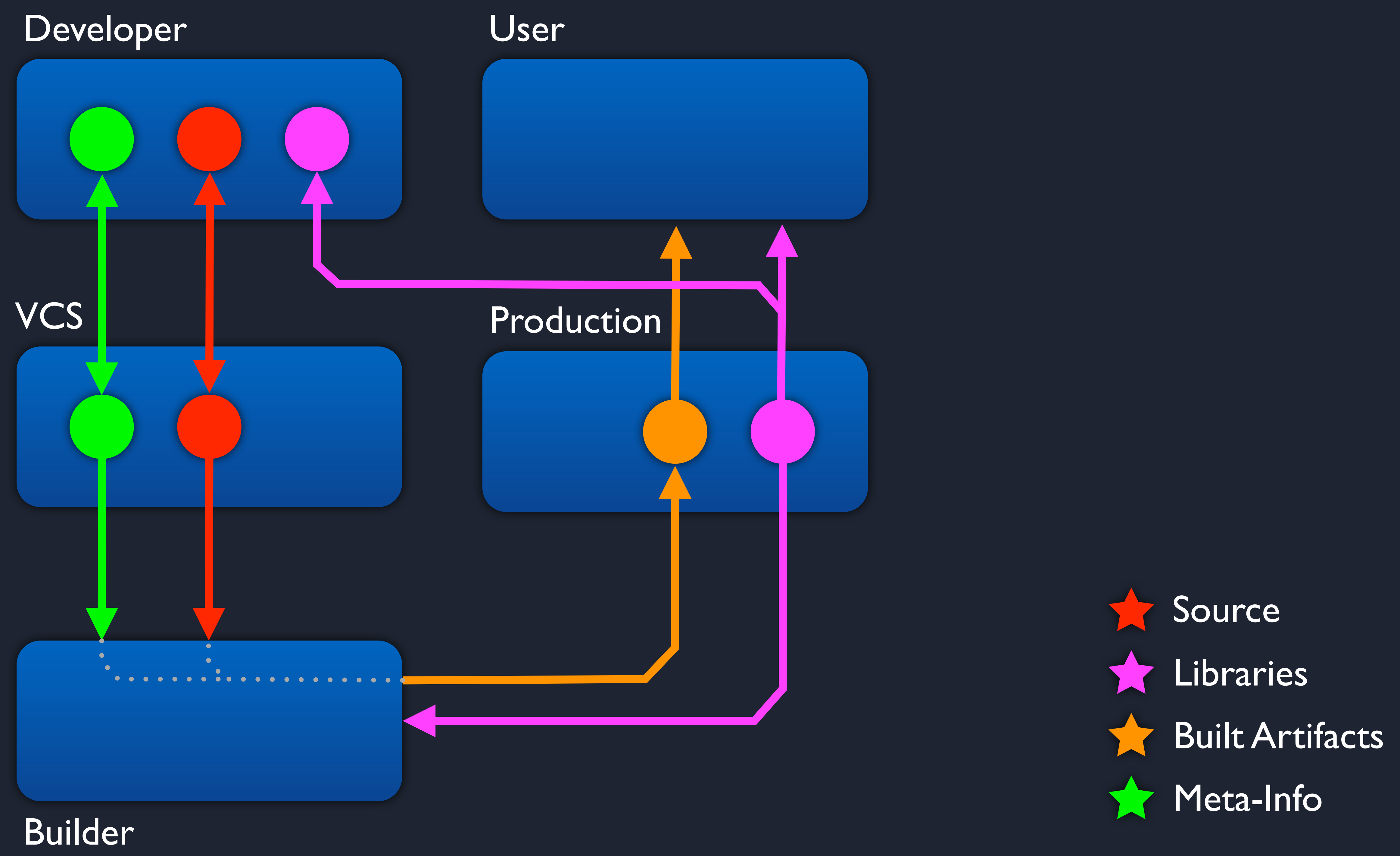
## What’s Wrong

- Multiple sources of meta-information.
  - Build procedures are not portable.
- 3-party libraries are duplicated in CVS and in the production repository.
- Can we combine the source code repository and the production repository?

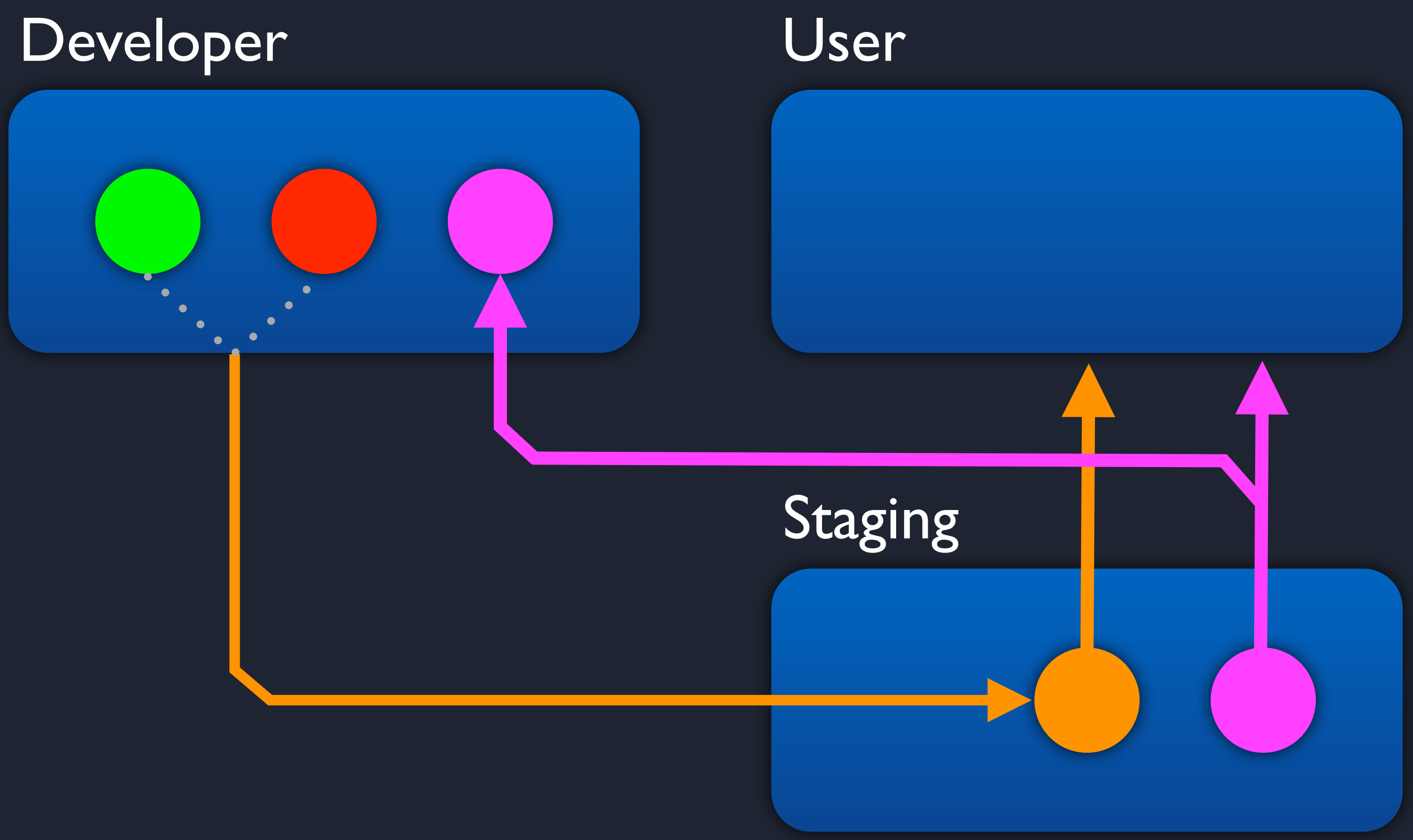
# How It Works Now



# What's Desired



# Run-Time Testing



★ Source
★ Libraries
★ Built Artifacts
★ Meta-Info

## Webstart

## How It Works Now

- JNLP file is generated dynamically according to information from the Application Index.
  - Among other things, it lists the names of required libraries.
- For every library, the client checks whether it is already cached, and whether a newer version is available on the server.
- The missing and newer libraries are downloaded and cached.
- The application is launched.

## What's Wrong

- Slow.
- Application libraries are coarse-grained.
- Lack of version attributes on individual libraries.
  - Many jars don't change between releases.
  - Have to check time stamps to distinguish different versions of a jar.
- JNLP file is not bound to a particular application version.

## What’s Desired

- Automatically created jars for every application.
- Elaborated versioning scheme for libraries.
- Background download of libraries.
  - The application starts immediately if a previous version is available in the cache.
- Pre-load of the system cache on public consoles.
- Generation of JNLP files by the building system.

## Why We Can't Use Incremental Download

- Download and verification of 25M's *gov.jar* takes...
  - whole file: **9.5** s.
  - minuscule patch: **20** s.

## Development Environment

## What's Wrong

- Customization of favorite IDEs.
  - Eclipse excluded.
- Lack of common project templates.
- Guessing the purpose, scope, dependencies, and build nuances of existing units of code.
- Deployment over a network share.
- Volume of Webstart downloads.
- All-at-once builds.
- Lack of a staging (test) environment.

## What’s Desired

- Splitting the source code into smaller parts.
- Describing what every part is, and how it can be used: checked out, compiled, packaged, and run.
  - In a machine-readable format!
- Keeping the descriptions along with the code.
- Using a common code management client,
  - Multi-platform.
  - Command-line and integration with IDEs.
- Always using local copies of external libraries.

# Development Lifecycle

**I**

- Create a new empty application from a template.
- Check out an existing application form VCS.

Version X

**II**

- Edit
- Clean
- Build
- Test

**III**

- Submit built artifacts to the staging environment.
- Commit source code and meta-info to VCS.

Version Y

**IV**

- Building
- Testing
- Deployment to production

🟦 Server

🟩 Client

## Building System

## What’s Wrong

- Not portable.
- Have to rebuild the full code base all the time.
  - Release is huge, difference is tiny.
- Lack of staging environment.
- *Beta* builds make no sense.
- Difficult to handle custom building procedures, native libraries.
- Can we compile less and unit-test more?
- Wizardry of dependency management.

## CVS Issues

- Slow.
- Awkward interface and unexpected results.
- Lack of reliable historical data and reports.
- Lack of a secure connection.

## Practical Steps

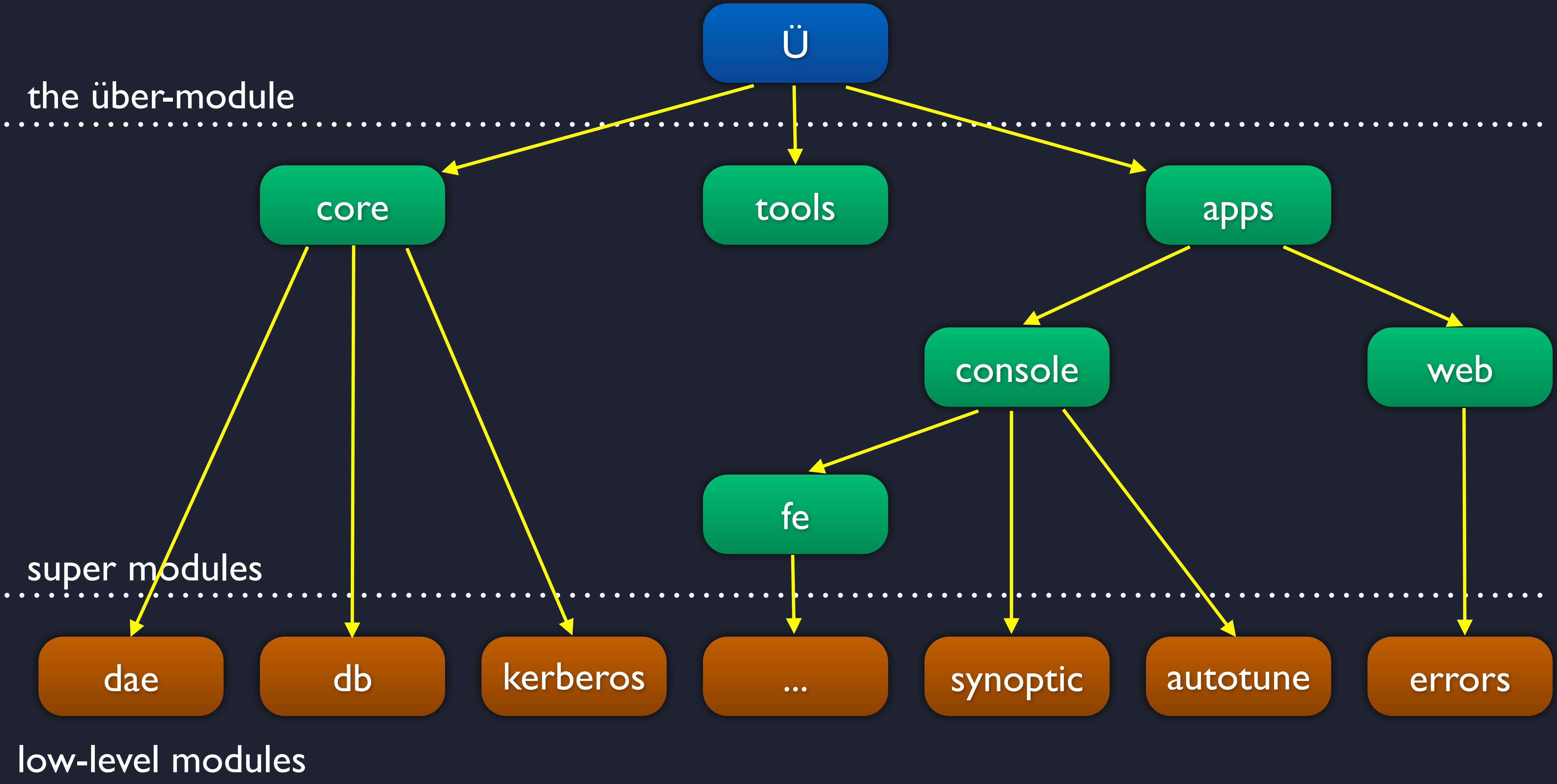
## The Plan

1. Physically split the gov tree into a fresh repository and annotate the new modules. No refactoring unless absolutely necessary.
2. Develop the development building environment.
3. Develop and set up the central building system.
4. Update Application Index.
5. Set up a [new?] version control system.
6. Set up new production and staging repositories.

## Code Modules

- Can be checked out and built independently.
- An atomic module is an application or a library.
- Don’t have to follow package boundaries.
  - However, each file from the gov tree goes to exactly one module.
- Use a standard directory layout:
  - main source, test source, resources, meta-info.
- Get built into one or several deployment units:
  - *jar* and *war* archives, *jnlp* files, ...

# Module Hierarchy



## Module Meta-Information

- Describes what the module is:
  - unique identifier,
  - version,
  - external dependencies,
  - location in a source control repository,
  - variations to the building procedure,
  - author, keeper, etc.
- Can be inherited from a top-level descriptor.
- Supersedes Application Index.

## Building System

- Apache Maven.
  - Strong similarities with Common Build.
- Unrelated to Ant.
  - "What to do" (Maven) vs. "How to do" (Ant).
- Rich set of defaults based on best practices.
  - Customizable.
- Support command-line interface and integration with common IDEs. Platform-independent.
- Comprehensive open-source repositories.

## Maven Command Line

- Creating an empty project:

```
mvn archetype:create
        -DgroupId=gov.fnal.controls
        -DartefactId=example
        -DpackageName=gov.fnal.controls.example
```

- Compile a project:

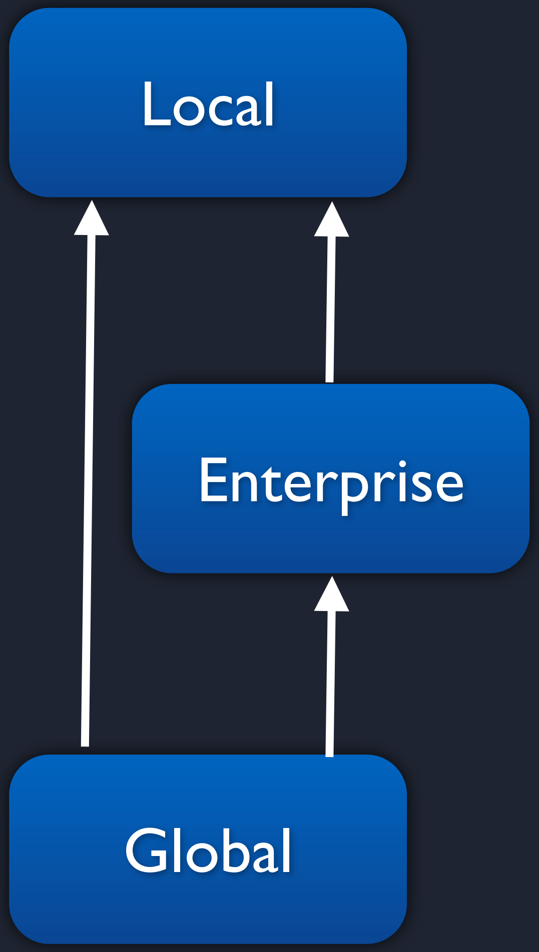
```
mvn compile:compile
```

- Build a project and deploy in the local repository:

```
mvn install
```

# Maven Repositories

Working copies



- Proprietary code
- Signed libraries
- Maven customization

Well-categorized open source

## Standard Directory Layout

```
src/
    main/
        java/
        resources/
        webapp/
    test/
        java/
        resources/
target/
pom.xml
```

## Project Object Model (POM)

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>gov.fnal.controls</groupId>
    <artifactId>parameter-page</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>Parameter Page</name>
    <url>http://www-bd.fnal.gov</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

## Version Control System

- Subversion? Git?
- A nice opportunity to switch.
- Interoperability with CVS is desired.
- The system have to remain usable for a long time.

## Considerations on Migration

- The build product should be binary compatible.
  - Actual jars will be different.
- Don’t have to switch all at once.
  - One module at a time?
  - Maintain two repository, synchronize regularly?
  - Keep the CVS repository, start new project in the new repository?