BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Introduction to DevOps

Sonika Rathi

Assistant Professor
BITS Pilani

Review CS#4

Version Control System

- Evolution of Version Control
- Version Control System Types
    - Centralized Version Control Systems
    - Distributed Version Control Systems
- Introduction to GIT
- GIT Basics commands
- Creating Repositories, Clone, Push, Commit, Review
- Git Branching
- Git Managing Conflicts
- Git Tagging
- Git workflow
    - Centralized Workflow
    - Feature Branch Workflow
- Best Practices- clean code

Agenda

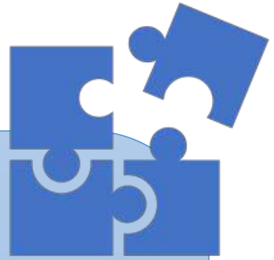
Version Control System

- Manage Dependencies
- Automate the process of assembling software components with build tools
- Use of Build Tools
  - Maven
  - Gradle
- Unit testing
- Automates Test Suite - Selenium
- Continuous Code Inspection
- Code Inspection Tools
  - Sonarqube

Component

Why Component based design

- large-scale code structure within an application
- also refer as “modules”
- In Windows, a component is normally packaged as a DLL
- In UNIX, it may be packaged as an SO (Shared object) file
- In the Java world, it is probably a JAR file



Benefits

- encouraging reuse and good architectural (loose coupling)
- efficient ways for large teams of developers to collaborate

Component

Challenges of component based design

- Components form a series of dependencies, which in turn depend on external libraries
- Each component may have several release branches

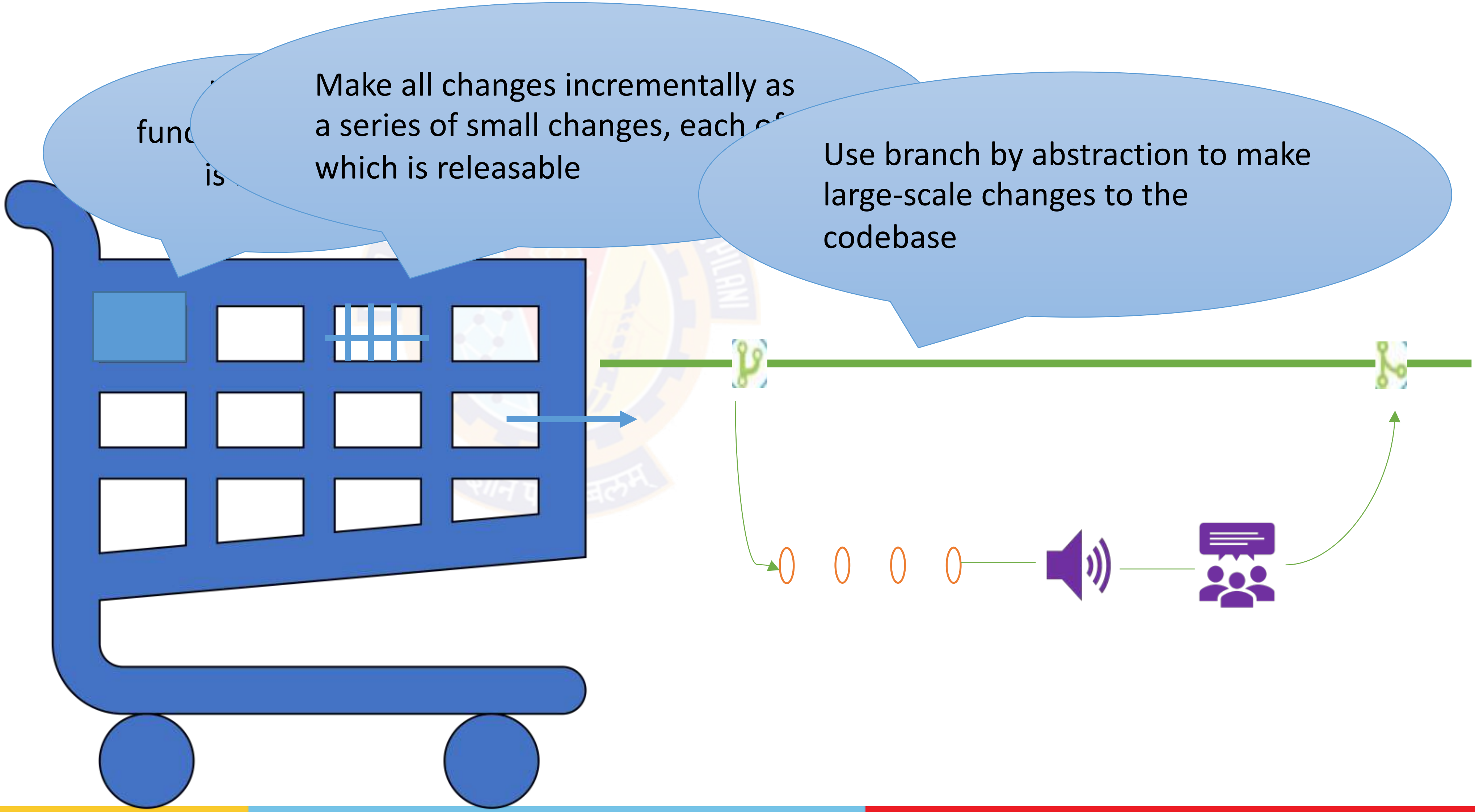
To overcome this we need to follow the best practices

Delay in release

- finding good versions of each of these components
- Then assembled them into a system which even compiles is an extremely difficult process

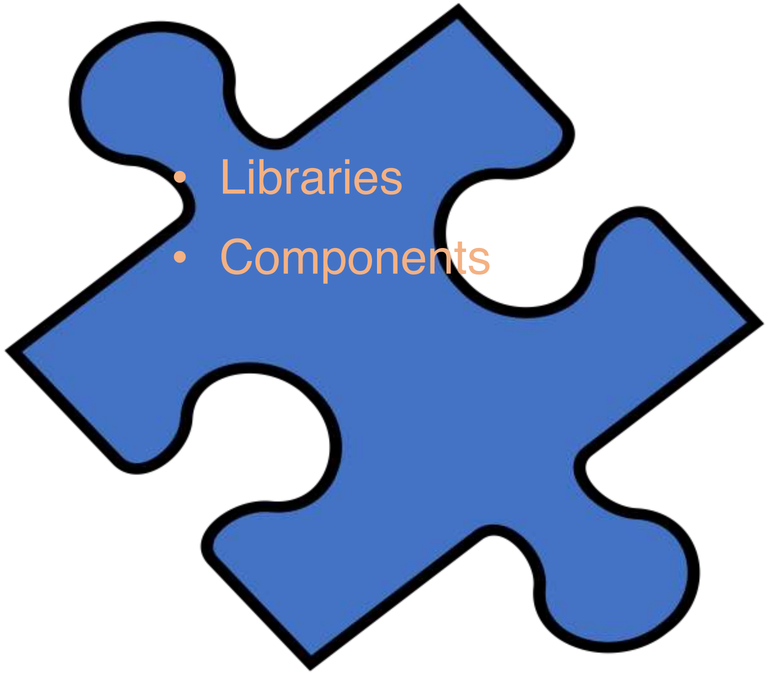
Best Practices for component based design

Keeping Your Application Releasable



Best Practices for component based design

Mange your applications dependencies



- Libraries
- Components

Dependencies

What is dependencies

- Dependency occurs when one piece of software depends upon another in order to build or run
- Software applications → host operating environment
- Like:
- Java applications. → JVM
- .NET applications → CLR,
- Rails applications → Ruby and the Rails framework,
- C applications → C standard library etc.

Dependencies

Dependencies can be

- Build time dependencies and Run time dependencies
- Libraries and components

Libraries	Components
• Software packages that your team does not control, other than choosing which to use • Libraries are Usually updated rarely	• Pieces of software that your application depends upon, but which are also developed by your team, or other teams in your organization • Components are usually updated frequently

This distinction is important because when designing a build process,
there are more things to consider when dealing with components than libraries

Dependencies

Build time dependencies and Run time dependencies

- Ex. In C and C++, your build-time dependencies are simply header files, while at run time you require a binary to be present in the form of a dynamic-link library (DLL) or shared library (SO)
- Managing dependency can be difficult

Build-time dependencies must be present when your application is compiled and linked (if necessary)

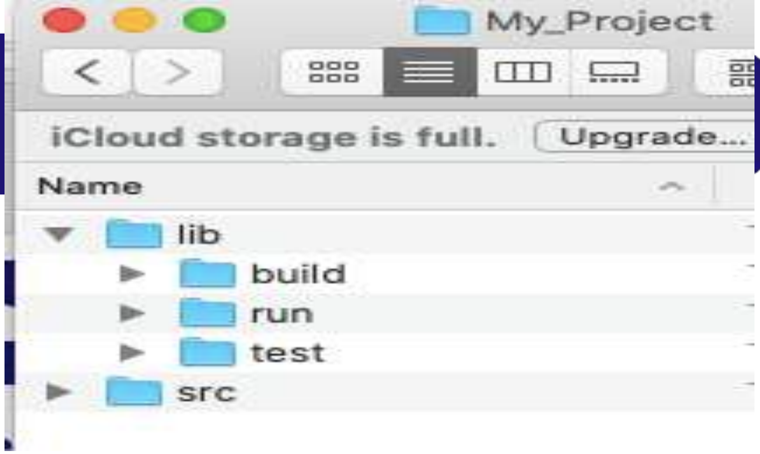
Runtime dependencies must be present when the application runs, performing its usual function

Most common dependency problem

With libraries at run time

- Dependency Hell also refer as DLL Hell
- Occurs when an application depends upon one particular version of something, but is deployed with a different version, or with nothing at all

Managing Libraries



Implementing Version Control

- Simplest solution, and will work fine for small projects
- Create lib directory
- Use a naming convention for libraries that includes their version number; you know exactly which versions you're using

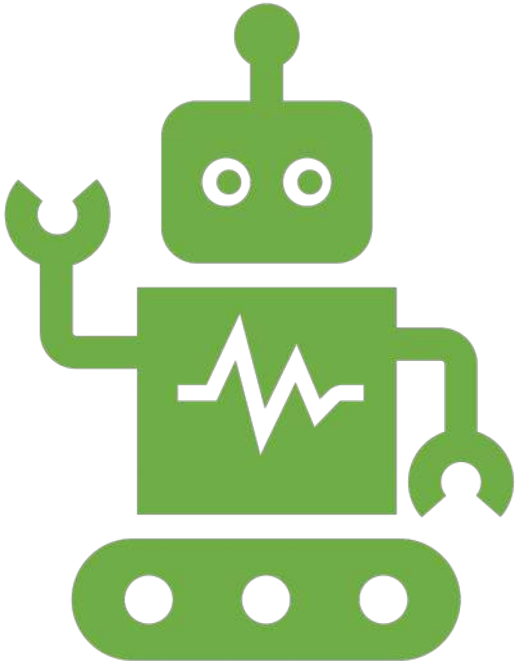
Benefit:
- Everything you need to build your application is in version control
- Once you have a local check-out of the project repository, you know you can repeatably build the same packages that everybody else has

Problems:
- your checked-in library repository may become large and it may become hard to know which of these libraries are still being used by your application
- Another problem crops up if your project must run with other projects on the same platform
- Manually managing transitive dependencies across projects rapidly becomes painful

Managing Libraries

Automated

- Declare libraries and use a tool like Maven or Ivy or gradle
- To download libraries from Internet repositories or (preferably) your organization's own artifact repository



Managing Component

Components to be separated from Codebase

- Part of your codebase needs to be deployed independently (for example, a server or a rich client)
- It takes too long to compile and link the code

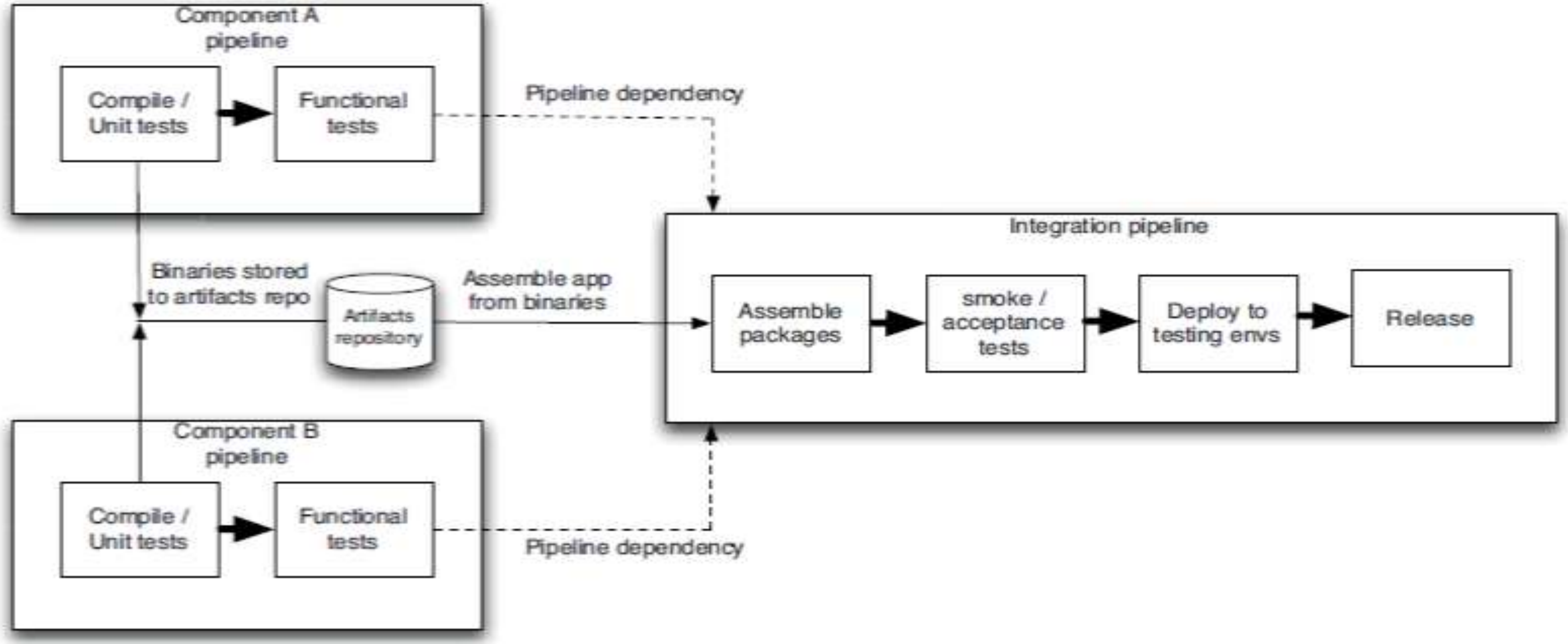
Managing Component

Pipelining Components

- Split your system into several different pipelines
- The build for each component or set of components should have its own pipeline to prove that it is fit for release
- This pipeline will perform the following steps
  - Compile the code, if necessary
  - Assemble one or more binaries that are capable of deployment to any environment
  - Run unit tests
  - Run acceptance tests
  - Support manual testing, where appropriate

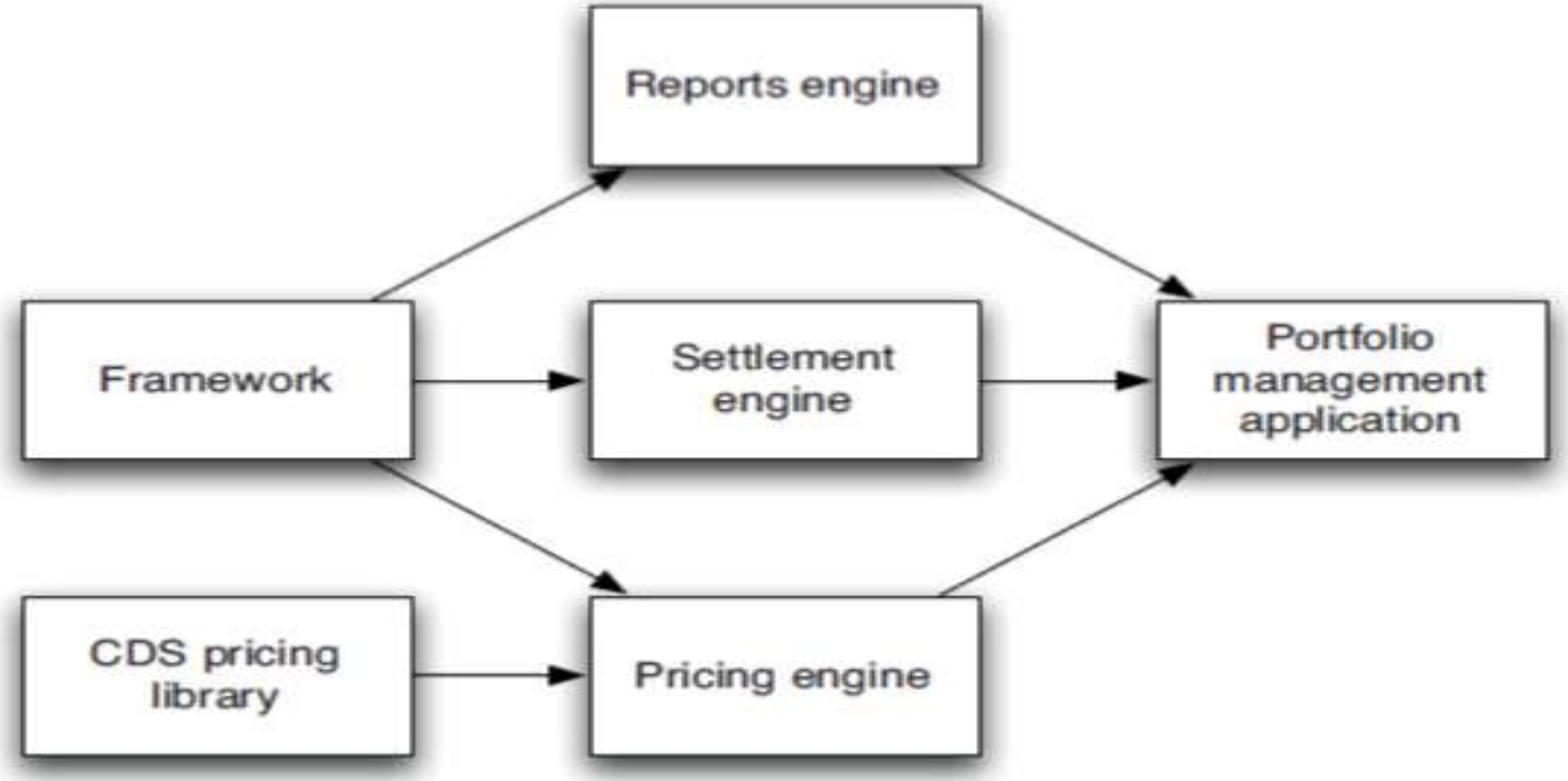
Managing Component

Integration Pipeline



Managing Component

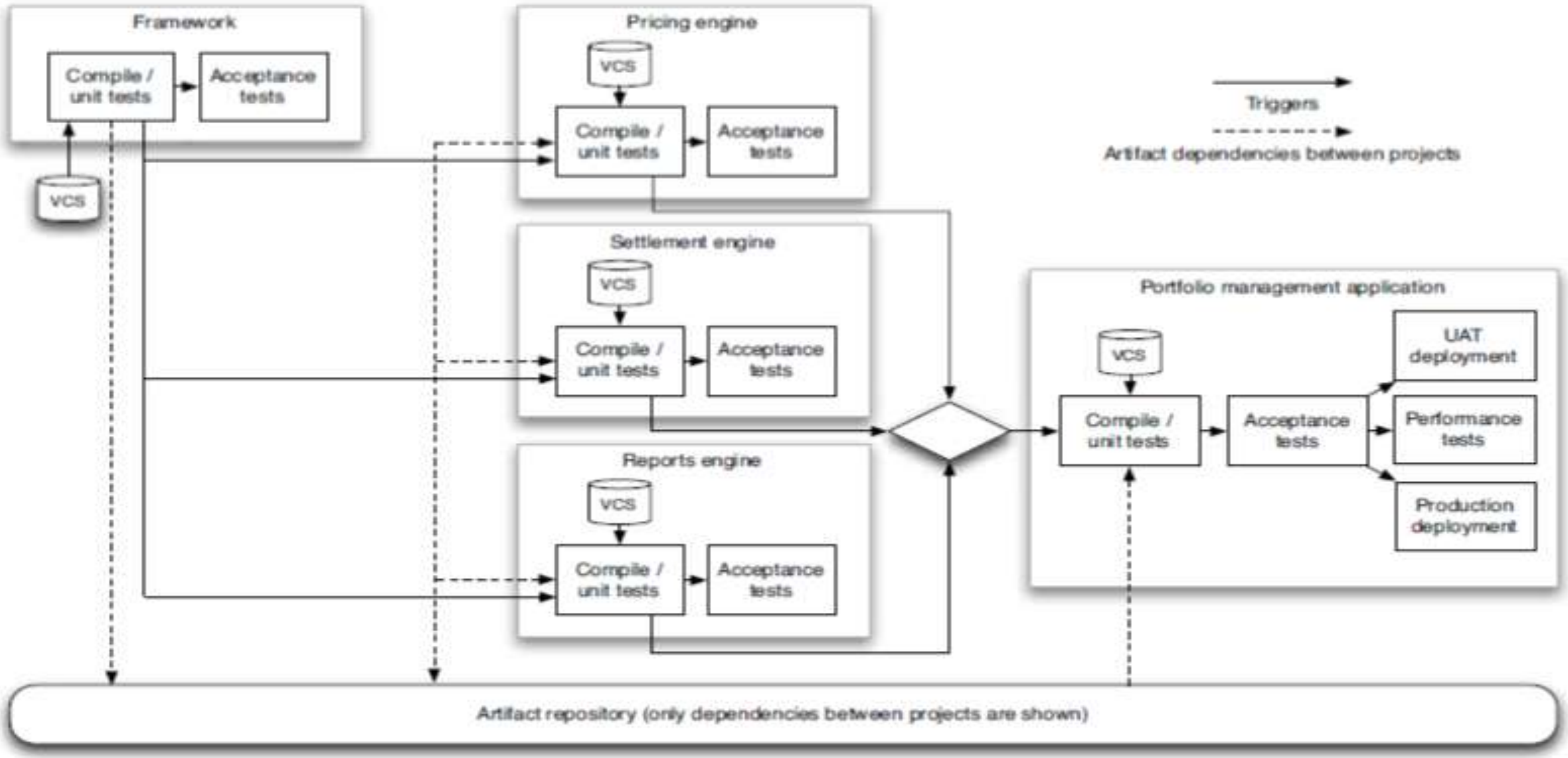
Managing Dependency Graphs



credit default swap (CDS) library that is provided by third party

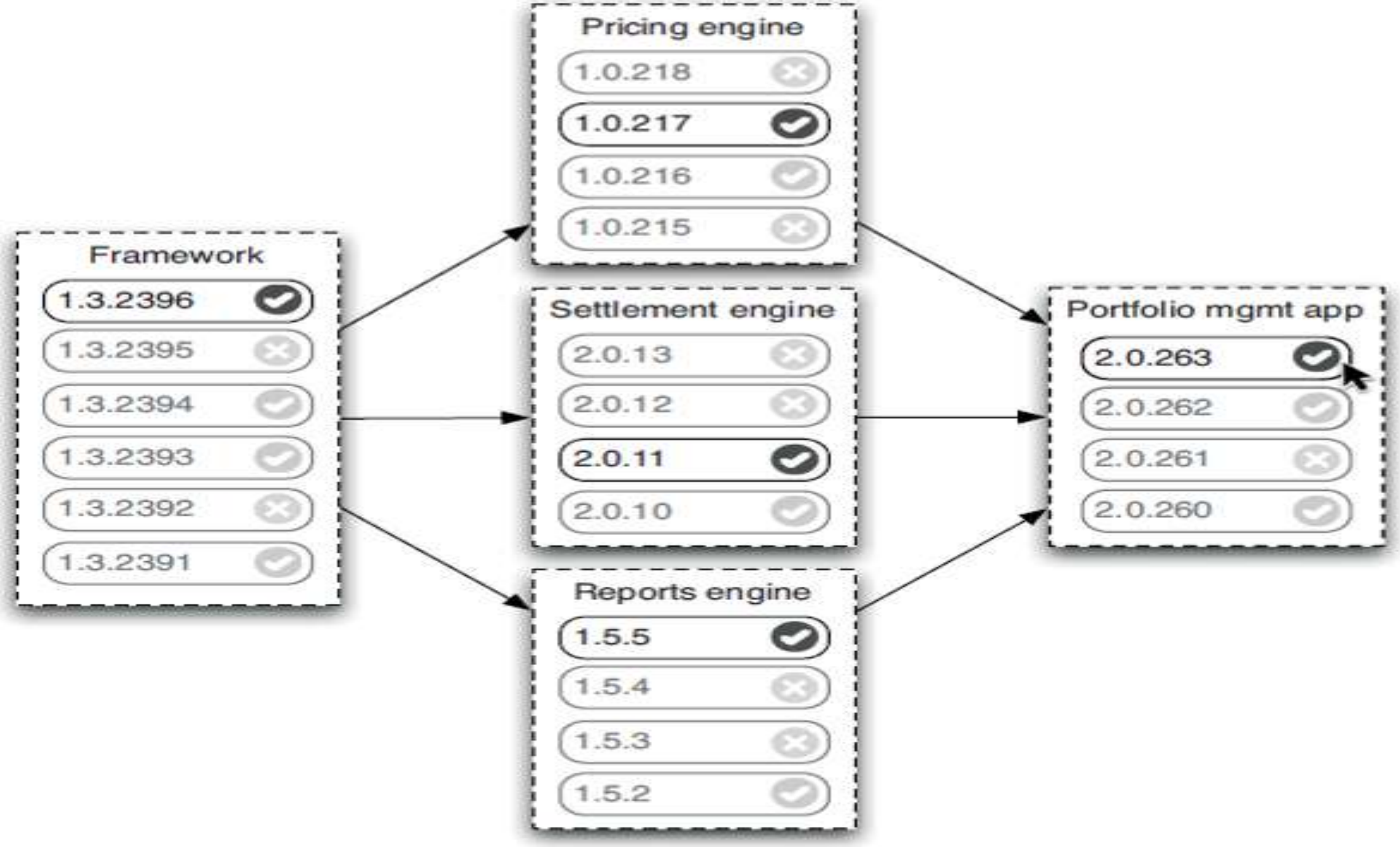
Managing Component

Pipelining Dependency Graphs



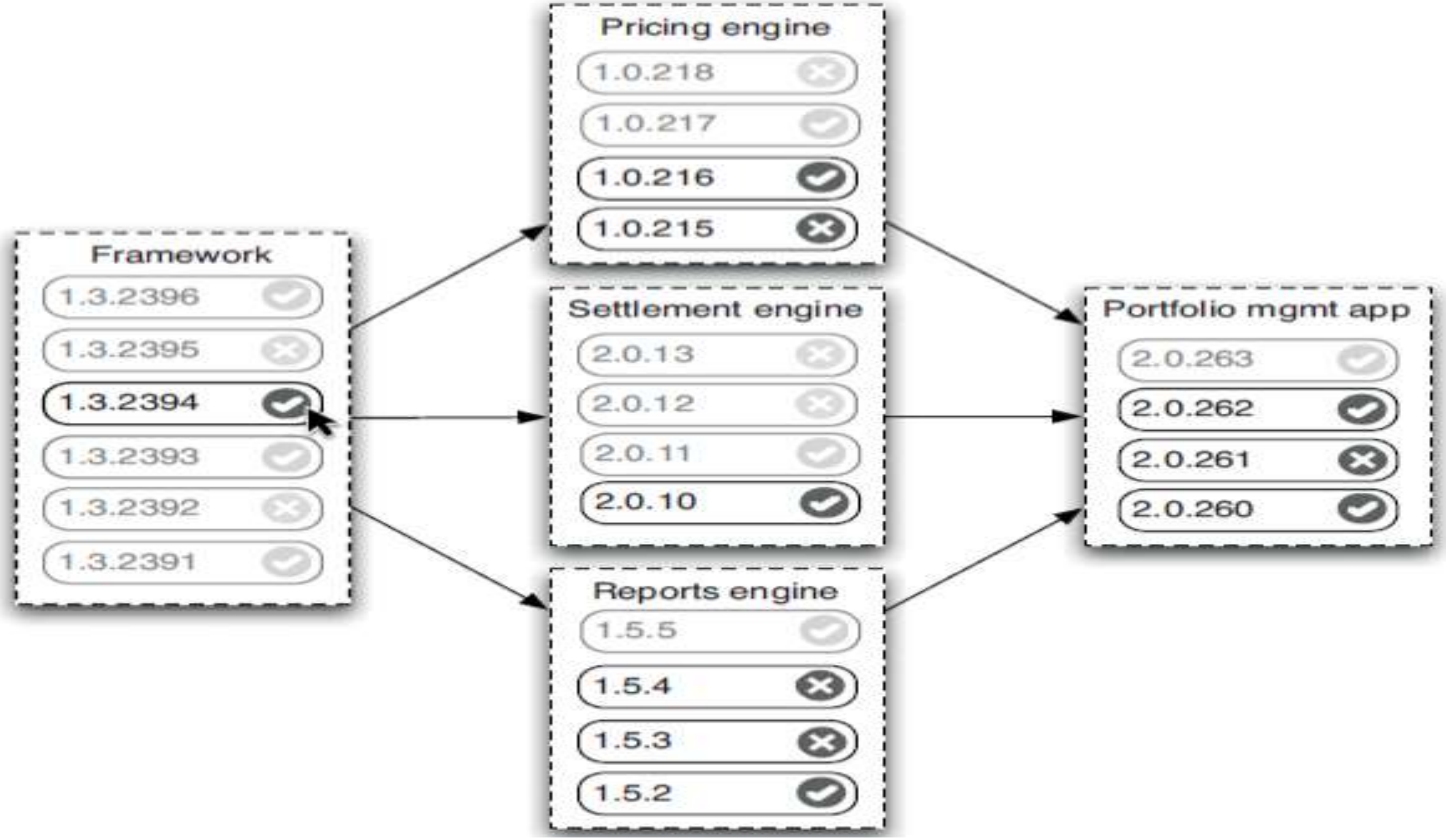
Managing Component

Visualizing upstream dependencies



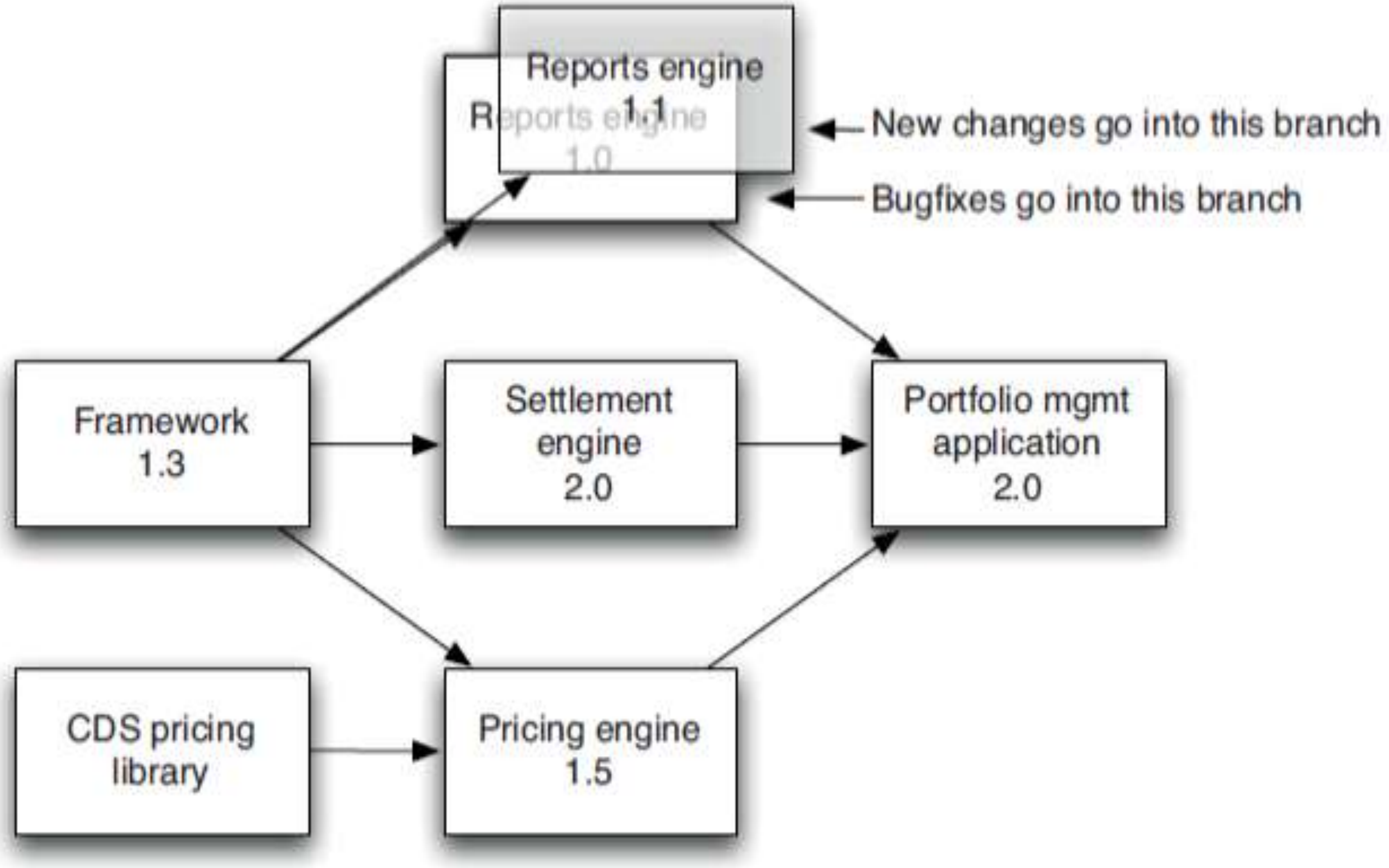
Managing Component

Visualizing downstream dependencies



Managing Component

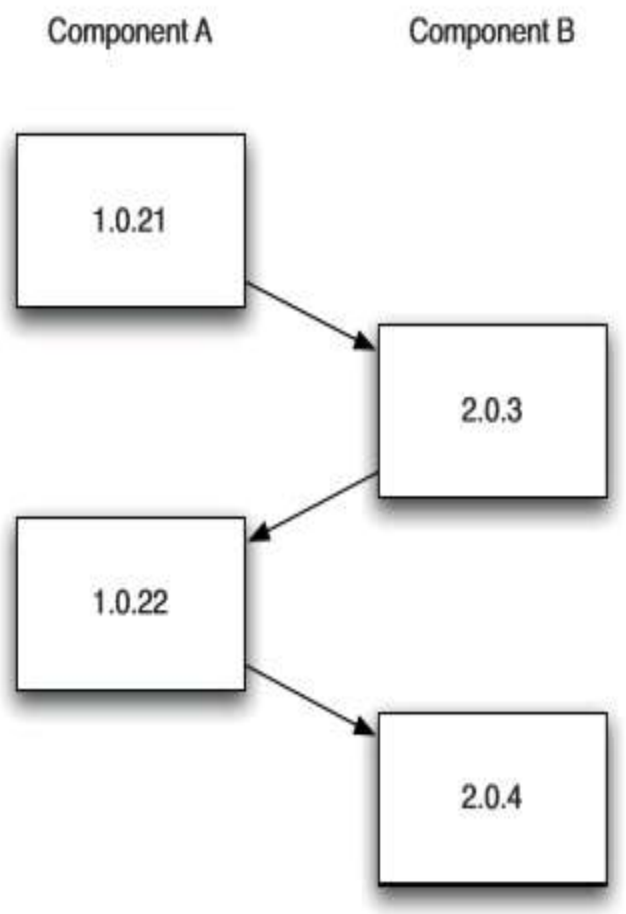
Branching Components



Managing Component

Circular Dependencies

- This occurs when the dependency graph contains cycles
- No build system supports such a configuration out of the box, so you have to hack your toolchain to support it



Circular dependency build ladder

Build automation

Build tools

- Steps of a build process

Compile the source code

Running & evaluating the unit test

Processing existing resource files (configurations)

Generating artifacts (WAR, JAR,..)

Build automation

Build tools

- Additional steps that are often executed in the a build process:

Administering dependencies

Analyzing code quality (static code analysis)

Running additional tests

Archiving generated artifacts and packages in a central repository

Automating Build Process

Build Tools

- Maven
- Gradle

Technology	Tool
Rails	Rake
.Net	MsBuild
Java	Ant, Maven, Buildr, Gradle
C,C++	SCons

Maven

What is Maven

Maven

What is Maven

Build Tool
- One artifact (Component, JAR, even a ZIP)
- Manage Dependencies

Management Tool
- Handles Versioning / Releases
- Describe Project
- Produce Javadocs / Site information

Maven

Who owns Maven

Apache Software Foundation
Maven official site is built with Maven
Open Source

Maven

Maven Structure

- src/main/java : by default maven looks for a src/main/java directory underneath of project)
- target folder : compiles all our source code to target directory
- pom.xml : maven compile source code in way provided by pom.xml

- Different language : src/main/groovy or src/main/resources
- Unit testing: src/test/java

- Target directory : Everything get compile

  Even your test gets run and validated

  Packaged Conents (like JAR, WAR or ZIP depending on what we have provided in pom.xml

Lets Demo

Maven

Pom.xml

- Maven uniquely identifies a project using
- groupID :
  - The groupid is often the same as our package
  - Ex: com.bits or com.maven.training
  - gorupId is like, business name or application name as you would reference it as a web address
- artifactId :
  - Same as name of your application
  - Ex. HelloWorld, OnDemandService etc.
- version :
  - Version of project
  - Format {Major}.{Minor}.{Maintenance} if it is RELEASE
  - '-SNAPSHOT ' to identify in development
  - Ex: 1.0-SNAPSHOT

Maven

Pom.xml

- packaging :
    - Packaging is how we want to distribute our application
    - Ex. JAR file, a WAR file, RAR file or an EAR file
    - The default packing is JAR
- dependencies :
    - Just add it to our dependency section of POM file
    - Need to know our three things for dependency i.e. groupId, artifactId, and version
- plugins
    - Just add it to plugins section of POM file

Maven

Pom.xml example

pom.xml > project > build > plugins > plugin > version

```xml
<project >
  <groupId>com.bits</groupId>
  <artifactId>HelloWorld</artifactId>
  <version>1.0-SNAPSHOT</version>
  <modelVersion>4.0.0</modelVersion>
  <packaging>jar</packaging>

  <dependencies>
      <dependency>
          <groupId>org.apache.commons</groupId>
          <artifactId>commons-lang3</artifactId>
          <version>3.8.1</version>
      </dependency>
  </dependencies>

  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.7.0</version>
            <configuration>
                <target>10</target>
                <source>10</source>
                <release>10</release>
            </configuration>

        </plugin>
    </plugins>
</build>

</project>
```

Maven

Maven Goals

- clean
- compile
- package
  - First run compile goal
  - Then runs unit test
  - Generate artifact/package as per provided in pom.xml
  - Ex. JAR, WAR
- install
  - First run package goal
  - Then install the package in local repository
  - Default it is .m2 folder
- deploy
  - Runs install goal first
  - then deploy it to a corporate or remote repository
  - Like file sharing

Maven

Project Inheritance

- Pom files can inherit configuration
  - groupId, version
  - Project Config
  - Dependencies
  - Plugin configuration
  - Etc.

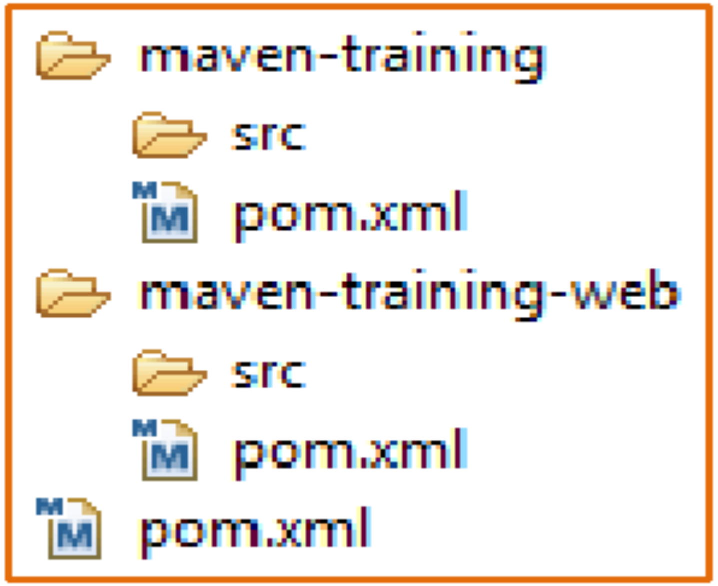
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <parent>
        <artifactId>maven-training-parent</artifactId>
        <groupId>org.lds.training</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>maven-training</artifactId>
    <packaging>jar</packaging>
</project>
```

Maven

Multi Module Projects

- Maven has 1st class multi-module support
- Each maven project creates 1 primary artifact
- A parent pom is used to group modules

```
<project>
    ...
    <packaging>pom</packaging>
    <modules>
        <module>maven-training</module>
        <module>maven-training-web</module>
    </modules>
</project>
```



Gradle

What is Gradle

What is Gradle

Open source build automation tool
Gradle build scripts are written in Domain Specific Language [DSL]

Why Gradle

High performance
- runs only required task; which have been changed
- Build cache helps to reuse tasks outputs from previous run
- ability to have shared build cache within different machine

JVM foundation
- Java Development Kit is prerequisite
- It is not limited to Java

Gradle

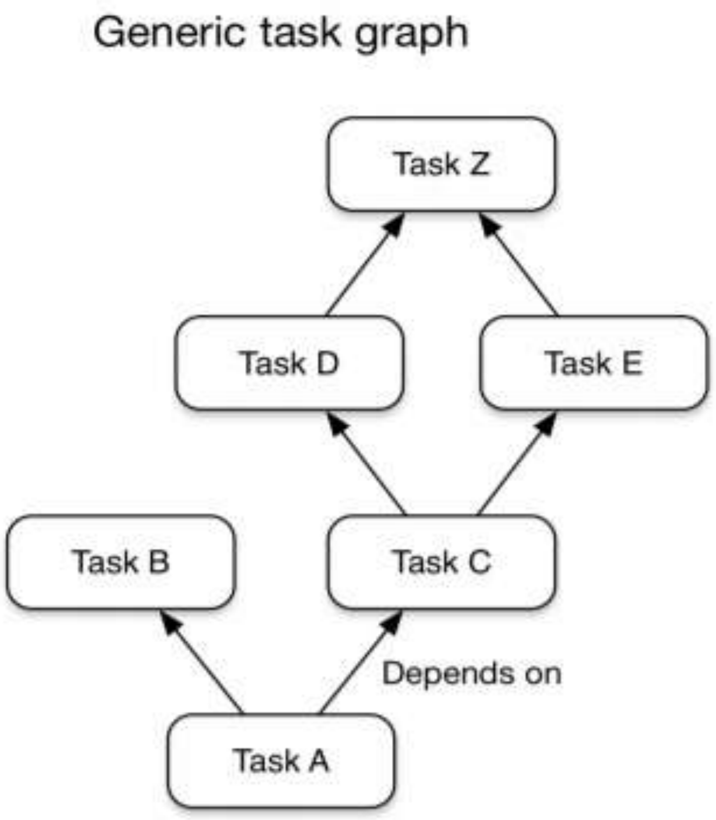
Core concepts

- Tasks and the dependencies between them
- Gradle calculates a directed, acyclic graph to determine which tasks have to be executed in which order
- Graph can change through custom tasks, additional plug-ins, or the modification of existing dependencies
- Plug-ins allows to work with other programming languages like Groovy, kotlin C++ etc.,

Gradle

Core model

- The core model is based on tasks
  - Directed Acyclic Graphs (DAGs) of tasks
- Example of a Graph
- Tasks Consists of:
  - Action : To perform something
  - Input : values to action
  - Output : generated by



Build Tool

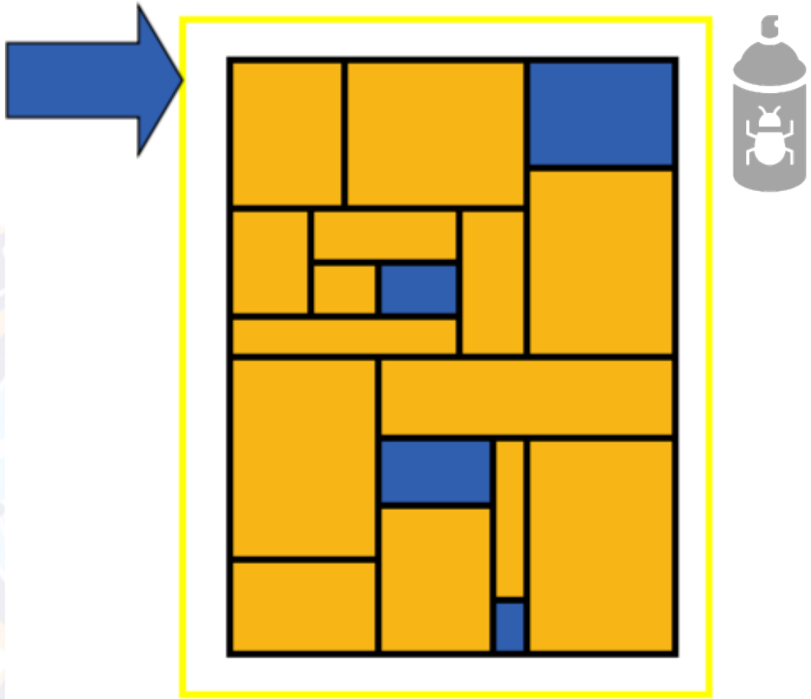
Summary

Gradle	Maven
Flexibility: • Flexibility on Conventions • User Friendly & Customized	**Flexibility:** • No flexibility on Conventions • It is rigid
Performance: • It process only the files has been changed • Reusability by working with Build Cache • Shipping is faster	**Performance:** • It process the complete build • No Build Cache concept • Shipping is slow as compared to Gradle
User Experience: • IDE Support : Is in evolving Stage • CLI : Modern CLI	**User Experience:** • IDE Support : Is mature • CLI solution is classic in comparison with Gradle

Unit Testing

Traditional Testing

- Test the system as a whole
- Errors go undetected
- Isolation of errors difficult to track down



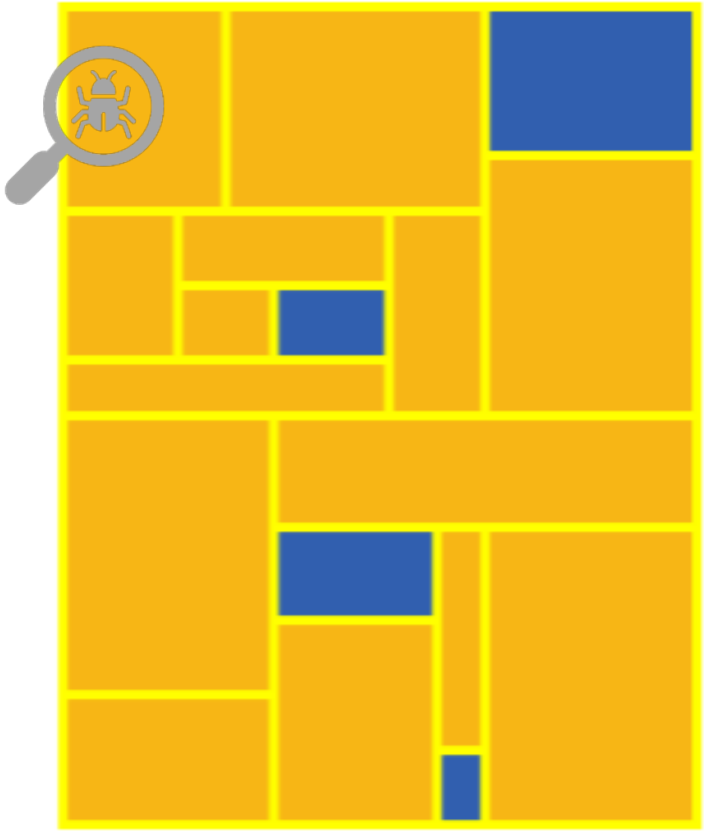
Traditional Testing Strategies

- Print Statements
- Use of Debugger
- Debugger Expressions
- Test Scripts

Unit Testing

What is Unit Testing

- Is a level of the software testing process where individual units/components of a software/system are tested
- Each part tested individually
- All components tested at least once
- Errors picked up earlier
- Scope is smaller, easier to fix errors
- Typically written and run by software developers
- Its goal is to isolate each part of the program and show that the individual parts are correct



Unit Testing

Why Unit Testing

Concerned with

- Functional correctness and completeness
- Error handling
- Checking input values (parameter)
- Correctness of output data (return values)
- Optimizing algorithm and performance

- Faster Debugging
- Faster Development
- Better Design
- Excellent Regression Tool
- Reduce Future Cost

Unit Testing

Benefits

- Unit testing allows the programmer to refactor code earlier and make sure the module works correctly
- By testing the parts of a program first and then testing the sum of its parts, i.e. integration testing becomes much easier
- Unit testing provides a sort of living documentation of the system

Unit Testing

Guidelines

- Keep unit tests small and fast
- Unit tests should be fully automated and non-interactive
- Make unit tests simple to run
- Measure the tests
- Fix failing tests immediately
- Keep testing at unit level
- Keep tests independent
- Name tests properly
- Prioritize testing

Q&A

Thank You!

In our next session: