



Build Engineering Whitepaper

Software Quality Through Repeatability

August 2009

Confidential Information & Copyright

This document contains information that is proprietary to Odecee Pty Ltd. All information within this document relating to the business of Odecee and is the property of Odecee Pty Ltd and deemed to be confidential information of Odecee Pty Ltd. Unauthorised disclosure of this information will be considered a breach of confidentiality resulting in material damage to Odecee Pty Ltd.

Introduction	4
SDLC Overview.....	5
Continuous Integration.....	6
Overview	6
Build	6
Testing	7
Unit Testing.....	7
Integration Testing.....	7
Test Coverage.....	8
Static Testing.....	8
The Build Process.....	9
Process Overview	9
Component Process.....	10
Build Pipeline – Bringing it all together.....	12
Dependency Management.....	15
Overview	15
Component Hierarchies.....	15
Dependency Descriptors.....	16
Direct and Transitive Dependencies	17
Versioning and SCM	17
Artefact Management.....	18
Repository Scoping	18
Public.....	19
Shared.....	19
Local.....	20
Version Management.....	21
Team Structure	23
Architecture	23
Build Engineering	23
Development	23
Environments & Infrastructure.....	24
Return on Investment.....	25
Reduced End-To-End Build Times.....	25
Consistent Repeatable Results.....	25
Long-Term Availability of Assets.....	25
Reduced Duplication of Effort.....	25
Re-Usable Project Accelerators	26

Further Considerations.....27

Introduction

Agility to respond and adapt to pressures from competitors and unforeseen factors in the business landscape are putting greater pressure on software engineering projects than ever before. Agility of software engineering teams are critical in ensuring the business can rapidly change their business system to compete in this landscape. For a software team to meet these pressures and deliver to these demands, patterns and methods of the Software Development Life Cycle must be capable of producing high quality outputs and ensure minimum rework in downstream Quality Assurance tasks. For this reason, the build and deploy cycles must be well thought out and executed.

One of the commonly used build and deployment patterns in the software development life cycle are based on manual processes. These manual processes typically involve the hand configuration of infrastructure and software artifacts. Manual processes suffer from a number of quality issues which manifest into errors in deployments and system configuration. For this reason, the rework factors in manual based artifact creation and configuration is very large, resulting in significant rework in problem isolation and identification tasks.

Build Engineering, one of the core patterns of a software development life cycle, is geared to manage the build, deployment and testing of software components and their dependencies to ensure the software build cycles are well managed. By defining a framework and pattern which covers the building, testing, publication and deployment of these artifacts in an automated manner, software quality through repeatability can be achieved. This whitepaper will present the processes for build engineering and the patterns to implement and deploy this vital function into your SDLC.

SDLC Overview

There are many methods defined for a SDLC. The typical approach can be defined as the well-known Waterfall Model, where the process of development will typically follow a process like:

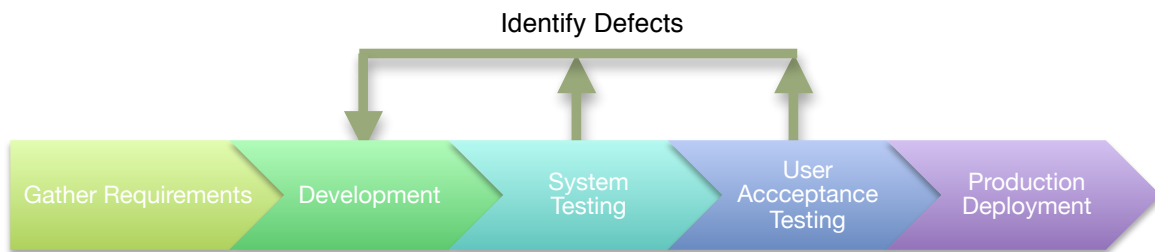


Figure 1

As illustrated in Figure 1, requirements are defined, developed, tested then finally deployed to a target environment. There is feedback from the System and User Acceptance Testing components of the process to perfect the requirements defined early in the process.

As requirements are typically developed before the system is developed, the development phase of the SDLC is tasked with implementing the system to realise the business requirements through an architected technology stack. This technology stack is typically built in an iterative manner and released to system testing cycles for verification. Software quality, following this processes, is attained using numerous methods; the key one being rigorous testing in the development phase. Facilitation of development testing is achieved through repetition of testing cycles made possible by employing an automation solution to build and deploy the system into a test environment. Using automated testing methods, albeit Unit and Integration, will facilitate rapid testing cycles and provide a means to test more regularly and frequently.

Frequently testing a system will naturally increase the quality of the software artifacts and provide quality software outputs to downstream testing cycles, like Product Integration Testing and User Acceptance Testing cycles. Furthermore, including identified defects in the development unit and integration testing cycles will decrease the likelihood of defects (re)appearing in the testing cycles, further cementing the quality of the application through build and release cycles.

Continuous Integration

Overview

A proven software quality enabler, Continuous Integration (or CI) is a process which is adopted by the development teams to automate the build and unit & integration testing of the system. The process is simple: each developer in the team will complete a functional component of the solution and build the appropriate unit and integration tests. Once the components are built, the code will be committed into a source code repository. A build scheduler (also known as the Continuous Integration Server or Build Server) is connected to the same source code repository and has visibility of all changes which are committed. The build server, will check-out or read the newly committed changes and launch a verification process. This typically involves:

1. Compilation and building of the artefacts – this ensures the committed source code passes the compilation requirements.
2. Unit and Integration tests executed – ensuring the system is tested and the newly committed code has integrated with the code base.
3. Compiled artefacts are packaged and published – ensures the availability of deployable artefacts and also tests the packaging concerns of the system.

A resulting green light from the compilation, testing and packaging process will signal a successful integration from the developer. A red light will indicate a failure in integration where the developers recently committed code was not sociable with the code base, requiring immediate attention from the developer. In the red light scenario, the developer will check the failure reasons and remediate the failure by making changes to the codebase and re-committing the changes. Once the remediated code is recompiled and retested, the system will eventually become operational again. This process provides visibility of the systems operational state and enforces ownership of non-sociable code so the system remains in a “working” state and minimises the amount of time the system is “broken” or in a non-operational state.

Build

Building of the system in the Continuous Integration process occurs at two location; the developers machine and the build server. The developers machine will typically contain all source code which is checked-out from the Source Code Management (SCM) system, which includes all build scripts that are used to compile and test the system under development. Once the developer builds their functional component(s), unit tests and integration tests are built in addition to the developed components to ensure the components are tested to assert their correct operation. Before committing to the SCM system, the developer will build and test the system locally (typically executing all unit tests; ignoring the integration tests) to increase the likelihood of a successful integration.

The second build is executed by the CI server and will execute both the unit and integration tests to provide a complete coverage view of the newly integrated source code. Integration tests, being the more time consuming test conditions, are typically executed on the CI server as developer optimisation in a view to increase the commits performed into the SCM system.

Testing

Unit and Integration tests are compiled and executed to assert the correct functional operation of the system. Depending on the technology stack, various frameworks can be used to implement the testing solution. A common framework which is widely used in Java based systems is JUnit, which is intended to provide a simple scalable testing solution. Depending on your testing patterns and requirements, other auxiliary testing frameworks such as JMock, EasyMock and Selenium can be used to provide a robust and feature rich selection of API's to decrease the testing build effort and increase the coverage of the testable conditions.

The goal of testing during development is to facilitate change through refactoring and provide assurance of module functionality through rapid identification of faults for the purposes regression testing. Change confidence is large for systems with broad unit testing coverage.

Unit Testing

Unit testing is described as the process of isolating each part of the program to prove the individual parts (or units) of the program are correct; typically through functional assertions. A unit test provides a strict, written contract that the piece of code must satisfy. It also serves as documentation for the system through a set of assertions which explicitly describes the expected behavior of the unit.

There are many unit testing methodologies and approaches. Selecting the correct methodology and approach is not in scope for this whitepaper, however, for the purposes of example, a typical unit testing methodology will have no dependencies on system components (like a DBMS) or deployed components (like Servlets). Simply put, a unit tests runtime does not include a container, e.g. a Servlet container. Achieving a runtime configuration without these container service dependencies can be challenging and support of testing frameworks like JMock can provide the ideal dependencies to achieve container or service avoidance.

In the CI construct, unit tests are built by the developer and are committed into the SCM system. The CI server will build the committed components and execute the unit tests outside of a container. This way, the CI server will not depend on a deployment to simplify the testing effort. However, unit testing cannot provide a complete picture of quality and sociability of all components in the system. Further testing is required to ensure this coverage is attained.

Integration Testing

The purpose of integration testing is to verify functional and reliability requirements placed on major components of a system. These components (or groups of units), are exercised through their interfaces using a black box testing approach, where success and error cases being simulated via appropriate data inputs. Simulated usage of system components through inter-process communication or front-end testing is used to test individual subsystems or complete system

assemblages. Test cases are typically constructed to test the complete collection of components within the system can interact using the expected behavior.

Integration tests are typically service dependent, e.g. they depend on an RDMS or other deployed components of the system. For this reason, they are classified as in-container tests and require a deployment cycle before the tests can be executed.

In the CI construct, integration tests require the deployment of compiled and assembled components, typically executed by the CI system. For this reason, the developer build cycles will not include the execution of integration tests as the overhead of the deployment cycles can be large.

Test Coverage

Testing coverage is important in identifying areas of the system which are not unit or integration tested. This provides a valuable input into the unit and integration test build cycle to target system components which are missing or light on testing. A coverage view will provide valuable information about the regression testing capabilities of the unit and integration testing framework. Tooling is a key consideration in this domain, and must be tied into the CI build process to generate a coverage report post build and test cycle.

Static Testing

Having a unit and integration test view of the system is not sufficient in covering all quality aspects of the code base. Additionally, static testing is required to complete the quality picture by enforcing some coding standards and increase the early identification of performance related issues. Tools which check code quality can be employed to ensure developer written code is simple and free from programmatically identifiable errors and warnings. Checkstyle is a commonly used tool and can be configured to includes static checks like code complexity (n-path and cyclomatic), general style and class data abstraction coupling.

The Build Process

The build process is the end-to-end process that begins when source is committed to the version control repository and completes with a production deployment. To support this process, any non-trivial program of work will utilise a large number of technical skills, possessed by many team members, with well-defined handover points between each.

When defining processes for a program of work, it is important to consider what level of formality is required. On the one hand, well defined processes may assist teams to interact effectively and avoid duplication of effort improving efficiency. On the other hand, process can stifle creativity and independent thought within teams. The processes described in this document should not be seen as a one size fits all strategy to solve all problems, but rather as a set of ideas that can be selected and augmented as required to solve the needs of the program.

Process Overview

No two software projects are the same. However most projects follow some form of SDLC and as a result there are a number of common stages in the production of software. The stages most relevant to build engineering are shown at a high-level below to provide context for the subsequent discussion topics.

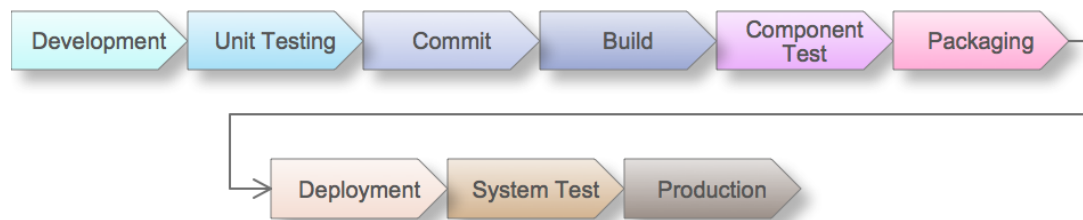


Figure 2 – Software Creation Stages

Each stage performs an important role described below:

- Development – The source describing the logic and behaviour of the software is created at this step. Source may consist of other material than source code such as configuration data, media, etc.
- Unit Testing – Unit testing is the simplest form of testing and is performed by those most intimate with the software, typically developers themselves.
- Commit – When source has reached a usable point, it is committed to version control where it becomes available outside the workstation that was created on. Once committed, source lives forever as part of the history of the version control system and allows the evolution of the source to be traced.
- Build – Building the software is the process of taking human readable source, and compiling it into a machine usable form. This is typified by a software compiler taking source code and

turning it into machine code, but may perform other functions such as building database reference data, compressing media artefacts, etc.

- **Component Test** – This is an umbrella term used to describe various forms of testing. A component is a term loosely describing something larger than an individually testable item, but smaller than an overall system. Component testing may consist of several types of testing including automated regression unit tests, integration testing with a database, integration testing inside an application server, service remoting via web services or various forms of IPC, etc.
- **Packaging** – Software packaging takes build artefacts and augments them so that they can be deployed to the final environment. This packaging may consist of simple compression archives with manual instructions, or utilise comprehensive automated installation systems such as Microsoft Installer packages (e.g. MSI) or Linux packages (eg. RPM, DEB, etc).
- **System Test** – The final gate check for software quality is the system test stage. This stage integrates all software components into a single environment and verifies that they work correctly. System Test here loosely refers to the various forms of testing that may occur on the complete system and includes specific phases such as User Acceptance Testing. Like all forms of testing this can be automated or manual, but is typically the hardest to automate.
- **Production** – The final stage is deployment and usage of the software in the production environment. The software that reaches this point should have passed through all previous stages and been verified in configurations closely matching production.

While the above stages may be common to all projects and should be familiar to most readers, they may be implemented in many ways. Stages may be automated, divided up among various teams, consist of many sub-stages, and iterated over many times during the course of a project.

Component Process

The build process follows the progress of the software through several phases of the SDLC from development to deployment. However the build process requires a complete eco-system of tools and components to support the creation of the software and these also warrant attention. The artefacts produced by different technical teams will perform drastically different functions and it is common for teams to be arranged in silos with their own unique processes and ways of managing their work.

Despite the apparent diversity between teams, there is also much in common which can be exploited to better manage the assets produced by these teams. The advantages of following a common approach to asset management are many and include improved communication between teams, improved visibility of assets, minimised duplication of effort and increased long term maintainability through consistent access to all assets. The benefits are only enhanced over time as original team members move off the project.

The following stages are common to all teams producing technical components regardless of where they fit in the build process, and allow common tools and processes to be built to support them:

1. **Source Creation** – Each team produces source material. This may consist of source code, media files, automation scripts, configuration files, etc.
2. **Artefact Generation** – The source material is then compiled into a consumable form. In the case of source code this may be turned into executables, automation scripts may be

incorporated into installation packages, media files may be compressed into distributable form, etc.

3. Component Testing – Some form of testing must be performed on the compiled artefact. This may consist of unit tests for code, deployment tests for deployment scripts, and even static code quality checks such as code style.
4. Baselining – The final stage is versioning the artefact in such a way that it can be readily identified by multiple teams, and referred to uniquely over time.

A Component Creation Process that aligns with the Component Lifecycle and introduces the concepts of version management, artefact management and dependency management is illustrated below.

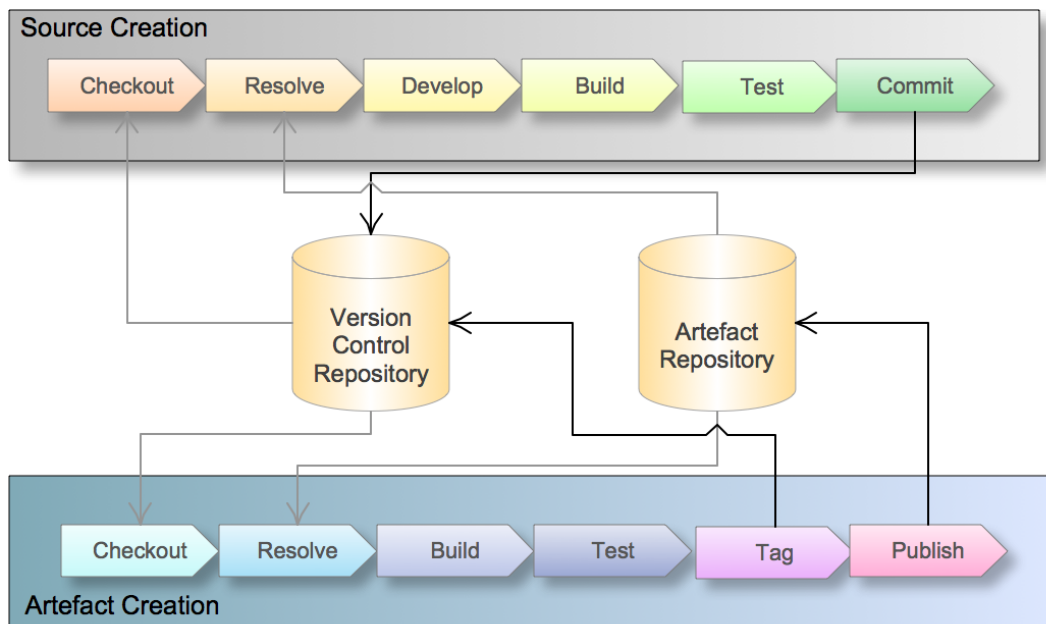


Figure 3 – Component Creation Process

The process has been split into two halves, the source creation process, and the artefact creation process. There are several reasons for this:

- The production of artefacts should be performed using a separate build workspace. Using a developer workspace to produce artefacts is dangerous because it may contain un-versioned changes, or local modifications that will lead to inconsistent and unrepeatable results.
- The artefact creation process should aim to be automated to the maximum extent that is practicable. Human error is a prime source of defects during the build process and can be largely eliminated via automation. Note that many steps in the source creation process can also be automated and ideally the two processes should share build scripts, but source creation is by its very nature a largely manual process.

This process is the same regardless of the type of artefact being produced, the differences lie in the implementation of each step. The steps are:

- Checkout – Retrieving the current source from the version control repository.
- Resolve – Obtaining the dependencies required to build the artefact.
- Develop – Updating source with changes. These changes will typically be to introduce new functionality, or to fix defects. Note that source doesn't necessarily refer to source code, it may consist of many things including source code, deployment scripts, reference data spreadsheets, XML files, visual images, HTML documents, documentation, etc.
- Build – Produce a new instance of the artefact. This will differ drastically depending on the type of artefact being produced.
- Test – Verify that the artefact matches expected criteria. Appropriate testing is program and artefact specific and may take the form of unit tests, integration tests, static code analysis, checklist comparisons, or manual peer reviews.
- Commit – Adding a new revision to the version control system with the new source changes.
- Tag – Marking the current state of the version control system so that it can be identified and retrieved at a later date.
- Publish – Uploading the produced artefacts into the artefact repository where they can be obtained by other components.

Following sections of this document will provide further detail on how this process is used in practice, how processes utilised by different teams are coordinated, and ways of automating the build process to minimise manual effort.

Build Pipeline – Bringing it all together

The build pipeline is defined as the definition and process adopted in the SDLC to generate and deploy a set of application artifacts. As the development team create source code and integrate into the source code repository, a set of artifacts will typically be generated from the source code for the purposes of deployment. Depending on the technology stack, the artifacts can vary, so for the purposes of example, the following section will discuss a typical JEE stack and artifacts relating to the specification.

Using Figure 3 as the model that illustrates the source and artifact creation process, a build pipeline can be defined as an instance view of the Component Process. We define the build pipeline as a process used to build and deliver artifacts for deployment. The process is broken up into a number of steps, where each step can be simplified into three categories; (i) asset creation, (ii) asset verification and (iii) asset identification and publication.

Asset creation is the process in which source artifacts are transformed into a deployable configuration for the purposes of execution. Asset verification is the process of testing and verifying compiled (created) artifacts for the purposes of quality assurance. And finally, asset identification and publication is the process of marking artifacts for the purposes of identification for retrieval.

The build pipeline requires coordination of three teams; development, build engineering and infrastructure. Each team have a critical part to play in the overall process. Figure 4 illustrates the build pipeline and divides each responsibility into swim lanes to increase the clarity and definition.

The process starts by the development team writing source code to implement features of the system. All source code artefacts need to be compiled in a compilation step which typically create a runtime version of the source code artefacts. At this stage, the generated artefacts are roughly grouped into components and are packaged to create an instance artefact. Using the JEE example, the source files are compiled into class files and aggregated into JAR, WAR, EJB-JAR and EAR artefacts.

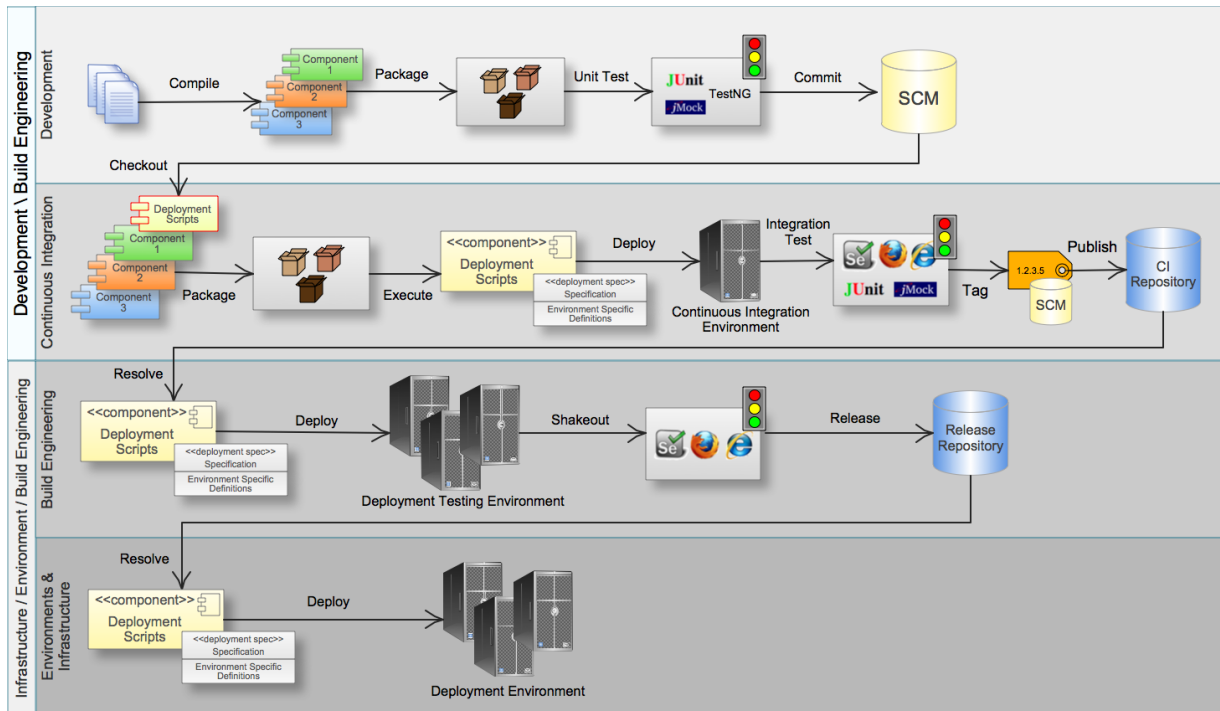


Figure 4 – Build Pipeline

Once the artefacts are compiled and packaged, unit testing is executed by the developer to ensure the system meets the contractual obligations set for the unit. Technologies like JUnit or JMock can be used to assist in this activity. This is the first point of test failure detection. A developer will not be able to commit any source code into the SCM without achieving a successful integration. Once the components are functionally verified using the unit testing suite, the developer will make the components available to the overall system by committing into the source code repository.

Newly committed source code will be detected by the Continuous Integration system and automatically built. As previously described, CI will build and test the artefacts using a number of possible methods and approaches. One major difference in the CI testing cycle is the inclusion of a deployment step to enable in-container testing, e.g. WebServices testing. In this step, the applications artefacts created in a previous build step will be deployed and configured into a “smaller scale” test environment; the Continuous Integration Environment. The major reason for deploying the newly built artefacts into scaled-back environment is to increase the efficiency of the deployments and facilitate a greater number of deployments throughout a business day. This way, we can test the developer integrations far more frequently.

At the completion of the testing step in the CI processes, the application stability is measured and presented via a dashboard. This is where, for example, the unit and integration test pass rate can be presented (as a percentage) using a dashboard. At this point, the builds can also be failed if the minimum testing requirements have not been met. In the test failure scenario, the development and build engineering teams can investigate the reasons for failure and remediate; resulting in a system which remains functionally operational and stable. As in the workstation unit testing step, this is the second point in the build pipeline where the build can fail; stopping the release of software components which fail verification.

At the completion of a successful software component verification, the SCM system is tagged to ensure functionally verified software components are easily identified. This way, retrieval of a “verified” system configuration is simplified.

By the end of the tagging process, all artefacts that have been functionally verified are published into the CI artefact repository. The CI artefact repository will, therefore, only contain functionally verified artefacts.

Progressing the artefacts into downstream environments plays a big part of the build pipeline. For this reason, the build engineering team will execute a resolve step to target a version of the application for deployment. Once the resolution is complete, the artefacts will be deployed to a deployment testing environment (which is near-scale or fully-scaled). The deployment testing environment is different from the scaled-back CI environment in a number of ways. For example, clustering would not be configured in the CI environment, whereas the deployment testing environment would contain a clustering configuration, in a view to test the automated deployment implementation. This is the key and critical point of deployment testing; ensuring the generated artefacts and deployment scripts are functioning as expected.

Once the build engineers deploy the application into the deployment testing environment, a verification step is required for to ensure the automated deployment implementation is functioning correctly. This is where a set of automated screen tests (typically called shakeout tests) are executed and results presented in a dashboard. This is the final verification process of the build pipeline. Once a deployment has been verified in the deployment testing environment, the artefact is published into the release repository; completing the deliverables of the Build Engineering team.

Artefacts in the release repository are available for distribution. This is where the environments team source artefacts for deployment into all downstream environments, like system testing, performance & volume testing and production.

Dependency Management

Overview

All non-trivial software projects must deal with dependency management. Modern software projects typically utilise large numbers of re-usable components. Understanding graphs of dependencies between components is a tedious and error-prone task. Some of the tasks that become difficult when the dependency graph grows are:

- Tracking down all dependencies of your libraries.
- Determining which versions of libraries to use.
- Resolving version mismatches between dependencies of included libraries.
- Auditing use of libraries within software applications.
- Distinguishing between libraries required for compilation, for testing, for deployment, etc.

Fortunately, tools exist to manage these dependencies. Two tools in common use today in the Java world are Maven and Ivy. These tools provide functionality for defining, identifying, and locating component dependencies.

Component Hierarchies

Before the advent of dependency managers, project source trees were typically organised into rigid hierarchies in order to group components together. Automated build scripts needed to have intimate knowledge of this build tree structure in order to manage dependencies between components. The build was brittle because minor changes in the tree structure would require constant updates to the build. The dependencies between each component were handled manually, and each type of component used a different mechanism for handling its dependencies.

Consider a software solution including the following components:

- Integration Project – Set of adapters for connecting to a remote system. Final artefact is a jar file.
- Service Project – A set of business services containing logic utilising the integration components. Final artefact is a jar file.
- Web Project – A web user interface exposing the services to an end-user. Final artefact is a war file.
- Deployment Project – Scripts for deploying a complete web project to an application server. Final artefact is a zip file.

Each of the above projects has a dependency on the previous project. Each project must obtain the output of the previous project in order to build itself. A hard-coded build script approach will directly copy artefacts between projects as each component is built.

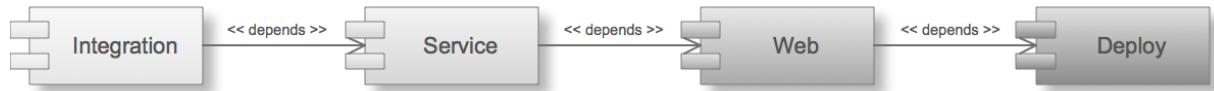


Figure 5 – Hard-Wired Dependencies

A dependency manager breaks the direct links between projects and manages the artefacts in a dependency repository.

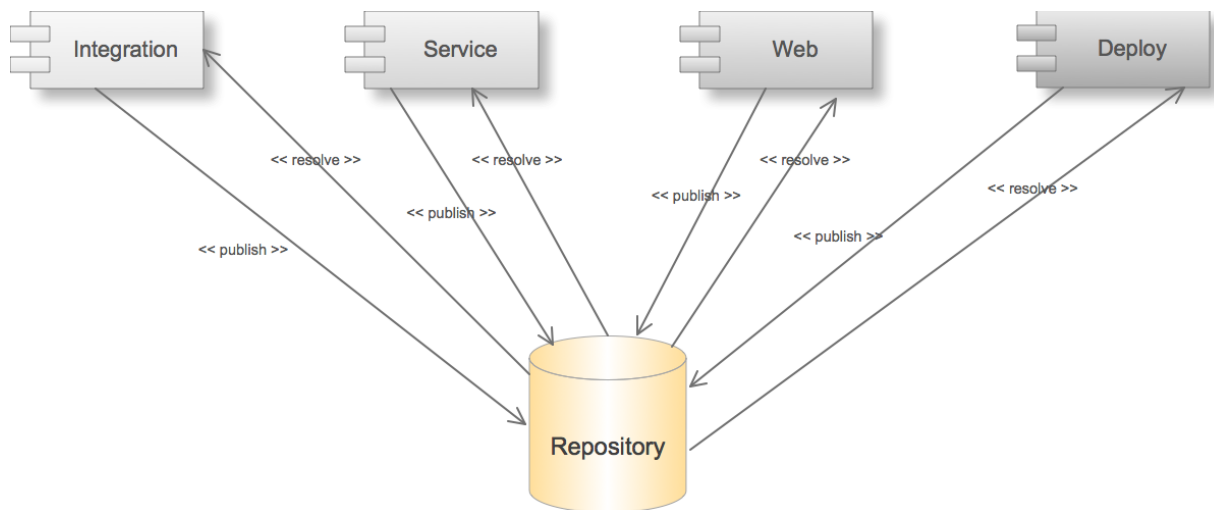


Figure 6 – Managed Dependencies

Managed dependencies publish all artefacts into a central repository where they can be accessed by other projects. Each project first “resolves” its libraries, then builds its artefacts, then “publishes” its artefacts into the dependency repository.

The dependency manager allows arbitrary component hierarchies to be defined. The component artefacts may be of any type, therefore all artefacts produced during the complete build pipeline should be managed using the same mechanism and all teams should access a single unified dependency management repository.

Dependency Descriptors

In order for the dependency manager to do its job, each project must define two key pieces of information:

1. The dependencies that the project has on artefacts produced by other projects. This includes the unique identifier of the artefact, and the version number required.
2. The artefacts that this project will produce. Similar to dependencies, this also defines unique artefact names and their versions.

This dependency information is defined in the dependency descriptor. Due to the strong coupling between dependencies and the current state of a project, this descriptor takes the form of a definition file stored along with the project in the source code repository. This ensures that as the project evolve and dependencies change, the dependency descriptor is kept tightly in sync.

Direct and Transitive Dependencies

Direct dependencies are those dependencies that are required by a project in order to function. These are explicitly defined in the project deployment descriptor. However, many of those dependencies will in turn have their own dependencies which must also be made available. These so called transitive dependencies must also be made available to the project.

A key function of a dependency manager is to not only make available the direct dependencies, but to traverse the entire dependency tree and resolve the complete set of dependencies to be included.

Versioning and SCM

The version control repository and the dependency management repository have different purposes but are strongly related to one another. Software Configuration Management procedures control both of these repositories.

When defining a new version, the version control repository is tagged so that the source used to produce an artefact can be located at any point in the future. Once an artefact is successfully produced from this tagged version, the versioned artefact is published to the dependency management repository where it can be consumed by other projects.

There are several rules to be followed:

- An artefact version must uniquely identify a particular instance of that artefact. Multiple instances of the same artefact cannot exist with the same version. This is to ensure that when a project depends on a specific version of an artefact, the result is predictable and repeatable.
- All artefacts must be produced by a project managed in version control. This project will contain the dependency descriptor, and any build scripts and source required by the project.
- Every artefact version must exist as a tag in the version control repository. This ensures that the source defining that artefact can be located at any point in the future.

Artefact Management

For dependency management to work effectively, artefacts need to be organised and made available in a consistent manner. Where dependency management deals with dependency graphs and the relationship between software packages, artefact management deals with the storage and cataloguing of artefacts themselves.

An artefact management strategy must take the following considerations into account:

- The dependency repository naming hierarchy must utilise a clear and consistent naming policy so that artefacts can be browsed and located efficiently.
- The status of an artefact must be able to be identified. It must be possible to distinguish between a newly built package and a package that has completed an official release after suitable testing.
- Short-lived artefacts produced during development iterations must be isolated from artefacts produced by build servers. Developers need to be able to utilise dependency management in their builds and produce artefacts without impacting the stability of “blessed” software releases made available to a wider audience.
- Externally sourced artefacts such as off the shelf and open source libraries must be managed in much the same way as generated artefacts. Guidelines must define who and under what circumstances new libraries can be introduced. Small teams may get by using verbal agreement amongst team members, but repositories consumed by larger teams may need to restrict access and require artefacts to meet entry criteria.
- It is often necessary to track the consumers of libraries. This may be due to licensing considerations of proprietary software, open source licence compatibility auditing, or to determine which consumers need to be notified of a library upgrade. For these reasons a mechanism should be created to report and/or visualise dependency graphs within dependency repositories.
- A mechanism for authentication and authorising users will be required on any repository. This should be kept as simple as possible to minimise administration, but it will often be necessary to limit the users allowed to connect, and the users allowed to make changes to the repository. In cases where some artefacts contain sensitive information (e.g. Users and passwords in deployment artefacts), it may be necessary to introduce finer grained authorisation.

Repository Scoping

It is not practical to use a single repository shared between all users and systems relying on dependency management. Instead repositories are typically organised into a hierarchy with different repositories having varied levels of scope and authority.

There is a de-facto standard for repository scoping that is followed by tools such as Maven and Ivy by default. While Ivy in particular can be heavily customised, the following repository hierarchy should be used as a starting point and only made more complex when requirements demand it.

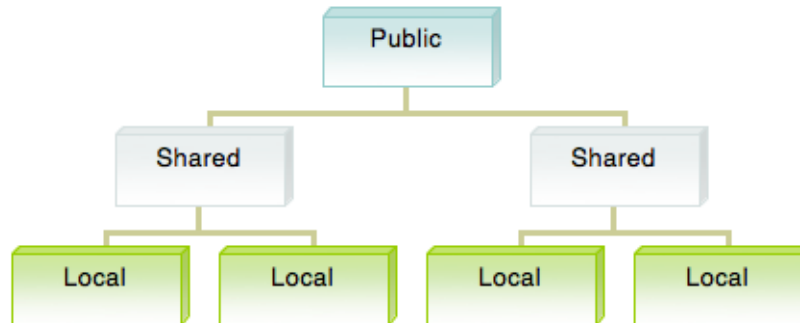


Figure 7 – Repository Hierarchy

The diagram above shows the standard way of defining a repository hierarchy. There are three levels, with the public repository visible by all, the shared repository visible by a group, and a local repository visible by an individual or single system. The hierarchy doesn't have to be a strict tree (e.g. multiple public repositories may be used), but a tree should be used if possible to keep things simple.

When searching for artefacts, a dependency manager will first begin at the bottom of the tree in the Local repository, and then move up the tree to parent repositories when it cannot find a match.

Public

The public repository is the repository that contains global artefacts visible by everybody. “Everybody” can mean different things to different organisations. Internet-based open source communities will usually rely on repositories such as the public Maven repository (<http://repo1.maven.org/maven2/>) because that is available to all users that download their software and wish to build from source. In-house software development teams should create and manage their own public repository. By creating a public repository, control can be placed over which libraries are made available to software teams. To keep things simple, an in-house public repository should be kept as closely aligned to external public repositories as possible, however they will typically contain a smaller set of libraries and avoid some of the inconsistent quality of external repositories.

A public repository should typically contain only artefacts that have been sourced externally. These may be open source libraries downloaded from the Internet, or other purchased libraries. It is rare for developed artefacts to be placed into a public repository.

The public repository should only be accessible by everybody, but in larger teams only a limited set of people should be allowed to make updates.

Shared

A single developer developing software in isolation may have no use for a shared repository relying only on their personal local repository, but in a complex environment the shared repository becomes the primary repository used for distributing developed artefacts between teams. It is the shared repository infrastructure that scales along with the size of the team.

While a public repository typically contains externally sourced artefacts and a local repository is a transient working area, the shared repository is the repository used for managing baselined artefacts. Software that is deployed to production will be sourced from the shared repository by the deployment packaging processes.

All artefacts in the shared repository must be traceable back to tags in the version control repository. Artefacts should not be placed into the shared repository unless all source used to produce that artefact can be later retrieved.

Access control to the shared repository may be quite complex. The artefacts published to this repository may be produced by a large number of teams, and have widely varying contents. For this reason this repository is the most likely to require fine-grained access permissions. It may be necessary to provide different groups with access to different portions of the repository.

If the three-level public/shared/local repository is not sufficient in a complex environment, the shared repository may be split into a sub-hierarchy. If teams across an organisation wish to exchange artefacts, it may be necessary to have a team-level shared repository attached to a cross-organisation shared repository.

Unlike the public and local repositories which can typically be re-built (public by re-obtaining external artefacts, the local directly from source), the shared repository should be considered an important asset. The shared repository contains the final baselined versions of software, and as such is irreplaceable. The effort of rebuilding a shared repository from version controlled tags can be completely impractical, and there is no guarantee that the entire original infrastructure will still exist to do so. Furthermore, there is no guarantee that rebuilt artefacts will match the originals exactly. As such it needs to be carefully managed with procedures such as backups given high priority.

Local

The local repository is the most limited in scope, but the most integral to the build process. A local repository is only visible to a single user or build system. A developer may choose to create a local repository per build system or may have a single local repository on their workstation, but the contents of the repository are only visible to them. A continuous integration server uses a local repository in much the same way.

During a software build process, the artefacts of each project are published into the local repository after they are built and made available to subsequent projects being built. It is common for large numbers of artefacts to be published and retrieved from the local repository during a build consisting of multiple projects.

The local repository does not bear any resemblance to the version control repository. Artefacts placed into the local repository may be built from source that will never be checked into source control, much less tagged as baselined versions. The local repository can be considered a working area with transient artefacts that are short-lived and easily replaced.

Due to the transient nature of a local repository, very little special attention needs to be paid to procedures such as backups. If a local repository begins to be used to hold artefacts, the process needs to be reworked so that these artefacts are moved into a more carefully maintained shared repository.

Version Management

Defining project dependencies in a complex environment can be a challenging problem. This task is made easier if project artefacts utilise a consistent and easily comprehensible versioning scheme. It is not difficult to create an effective versioning scheme, the bigger challenge is ensuring it is applied consistently. A lack of version management can lead to incorrect dependencies being selected, and lost productivity due to time wasted in troubleshooting.

Version management should follow a few simple rules:

- Version numbers must always increase. Selection of artefacts is greatly simplified if a higher number always means a later version. It is important to resist introducing new version numbering schemes unless they are absolutely essential.
- No two builds should share a version number. Every published artefact should use a unique version number. This rule can be bent in a development environment, but as soon as a build leaves a development workstation it should be given a unique version number. This allows troubleshooting to identify the exact source used to produce the artefact.
- Don't reflect project hierarchies in version numbering schemes. Just because an artefact is part of an overall software project, doesn't mean the overall project version should be included in the artefact version. For example, if the release is 1.0, the artefact should not be versioned 1.0.x.x. Dependency management is a more effective way of managing project hierarchies. Including project version numbers in artefact versions restricts re-usability, and leads to long-term version number consistency problems.
- Version numbers should reflect technical status, not enveloping project status. Minor changes should result in minor version number updates, major changes should result in major version number updates. An exception to this rule may be in overall release bundles grouping many deployable artefacts together, these may need to reflect the project status.
- Version numbering is more important for low level artefacts than high level artefacts. Low level artefacts tend to have longer lifetimes and get less attention therefore it is important that they follow consistent and easily identifiable version numbering over their lifetime. High level artefacts tend to only be relevant for a short time and can therefore follow the political trends of the project rather than the technical status.

There is no one true way to implement version numbering. The version numbering scheme will differ depending on whether the artefact has strong compatibility rules such as an API, versus library used internally to a project. A simple and effective strategy for version numbering of rigid API artefacts is defined below:

<Major>.<Minor>.<Build>

Where:

- <Major> – This number should be incremented when major backwards incompatible changes are introduced. It may also be incremented for major enhancements are made, although this should be limited. Projects depending on this artefact will have to make changes if this number is increased.

- <Minor> – This number should be incremented when changes to the API are made that remain backwards compatible but not forwards compatible. Projects depending on this artefact will not have to make changes if this number is increased unless they wish to take advantage of new features.
- <Build> – This element is free form but must always increment. It may contain sub-elements where necessary although this should be limited for simplicity. Artefacts produced for development or continuous integration may include some form of identifier (eg. <major>.<minor>.dev.250) to make it clear that they aren't intended to be official releases.

Artefacts that don't have rigid compatibility rules should follow similar rules to the above, but rules around which element should be incremented for a particular change are relaxed.

Team Structure

Ensuring the processes of the build pipeline are achievable and easy to maintain, the project must ensure teams are setup to own individual concerns and processes of the pipeline. Having the correct teams focussing on areas of the build pipeline will enable a successful outcome to achieve the return on investments through build engineering and automating the build and deployment process.

Architecture

The architecture team are ultimate owners of the build pipeline process as they will define and ensure the correct guidance is provided to the development and build engineering teams. Selection of key components in the technology stack to enable the build pipeline is critical in realising the artifact build and verification pattern, and therefore the responsibility of the architecture team.

Typically, the architecture team will liaise with the development and build engineering teams to provide direction in technology selection and configuration. This way, the application components under development are accurately governed to ensure compliance to development, test and build policies.

Build Engineering

Tasked with automating the artefact creation, testing and publication process', the build engineering teams are responsible for owning the continuous integration and deployment testing concerns of the program. The build engineers will lease with the architecture team and set the direction for the implementation of the build pipeline, where decisions around scheduling tools, tagging, versioning and dependency management are made.

Secondly, the build engineering team will liaise with development teams to understand the deployment considerations and steps for the developed artefacts. This way, the build engineering team can ensure the automated deployment components are scripted in parallel to the system artefacts.

Thirdly, the build engineering team will lease with the environment and infrastructure team to understand the deployment configurations for the purposes of automating the application deployment.

Lastly, the build engineering team will execute and own the deployment testing aspects of the solution. As defined in the build pipeline, the build engineers will deploy the solution to a fully scaled environment for the purposes of testing the deployment steps. Any issues which are identified in this step are treated as defects and managed through the defect management process.

Development

Development teams are tasked with building the business requirements on a given technology stack and platform. In the context of the build pipeline, the development team are the custodians of the CI process. The development teams concerns end at the unit and integration test point, where the deployment of the solution is typically built and extended by the build engineering team.

Environments & Infrastructure

Infrastructure and environments teams are the consumers of the deployment scripts developed by the build engineering team. By the time the environments team execute a deployment into a target environment, the automated deployment scripts are fully functional and fit-for-purpose. This way, the environments team will not be concerned with troubleshooting deployment related issues, given the deployment implementation has been verified on a system configuration similar to the target environment deployed to by the environments team.

Furthermore, the build engineering team will support the environments team for all deployment related issues.

Return on Investment

The processes described in this document have the potential to provide significant savings for projects that implement them. There is an up-front cost associated with establishing the tools and infrastructure to implement the processes described in this document, but over the course of a project this is more than offset by the returns.

Reduced End-To-End Build Times

An automated build process that can produce and deploy artefacts in minutes instead of hours provides enormous flexibility in how projects are coordinated. Reduction of end-to-end build times is achieved primarily through automation. Blocking defects can have fixes deployed to an environment several times a day if necessary minimising test team downtime. Fast iterative development involving multiple applications becomes possible and greatly reduces the time taken to achieve seamless integration across all touch points. The ability to do fast builds is a key enabler for projects to adopt Agile approaches to development with many iterations and regular integration.

Consistent Repeatable Results

Two major costs to any program of work are system testing involving multiple applications, and troubleshooting deployment problems. These can both be reduced if the software is built and deployed in a consistent repeatable manner. Inconsistent builds cost time and money due to system testing downtime, defect regressions that have to be detected and managed, and broken deployments that utilise valuable team members who could be more productively employed elsewhere. This is an area that is taken for granted when it works well; the costs only become apparent when things go wrong. Consistent and repeatable build processes are an essential part of a well run project.

Long-Term Availability of Assets

Most of the costs of a software program are in maintenance. Budget and timeline pressures during initial development mean that it is easy to take a short term view, but this should be strongly resisted when it comes to asset management. The primary assets of a software program are the source material, and the generated artefacts. For a project to be supported efficiently in the long term, maintenance staff need ready access to all assets produced by the program. As a program moves to business as usual operations, there will tend to be less staff supporting it, and costs will be reduced if the asset repositories are minimised and easily accessed.

Reduced Duplication of Effort

Teams sharing common processes can use common tools and infrastructure. This leads to direct cost savings for the project. It allows consolidation of project roles so that a single team can be responsible for managing the infrastructure and ensuring that important processes such as regular backups are performed correctly.

Re-Usable Project Accelerators

An advantage of using common processes is that a program can leverage the tools built by previous projects. A significant portion of the build process automation software, asset management software, and the infrastructure to support it can be re-used between projects. This re-use has a number of advantages including skill availability, shortened project start-up times, and less risk introduced by unproven tools.

Further Considerations

This document defines processes that allow source and artefacts to be created and managed more effectively. However this only covers those artefacts used in the creation and deployment of software. An area that is not addressed directly is environment configuration management. Consistent building and configuration of environments is an area which is outside the scope of this document, but is a major source of problems and requires similar techniques to be employed. The process is defined in a separate Odecee whitepaper titled Environment Configuration Automation Whitepaper – Patterns for Environment Consistency.