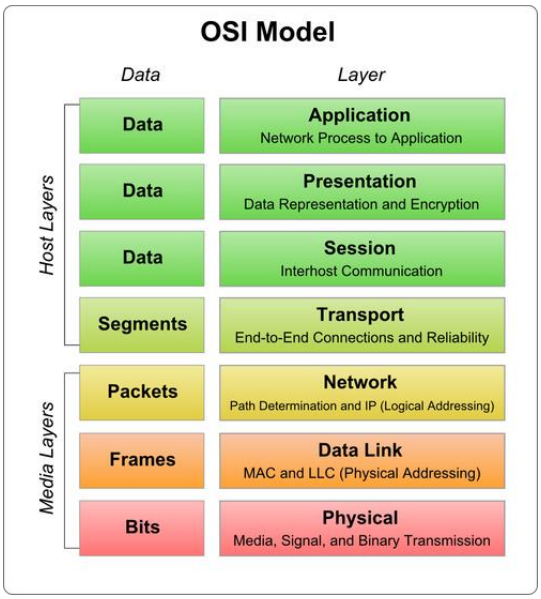
Chapter 10 – Computer Network

The OSI Model

ISO OSI (Open Systems Interconnection) Reference Model since it describes or relates to connecting systems that are open for communication with other systems. The OSI model depicts how data communications should take place. It splits the functions or processes into seven groups that are described as layers.

Each of the layers of the OSI model has a numerical level or layer and plain text descriptor. The Seven Layers of the OSI Model are 1 – Physical, 2 – Data Link, 3 – Network, 4 – Transport, 5 – Session, 6 – Presentation, 7 – Application.

A common mnemonic used to remember the OSI model layers starting with the seventh layer (Application) are: "**A**ll **P**eople **S**eem **t**o **N**eed **D**ata **P**rocessing." The lower two layers of the model are normally implemented through software and hardware solutions, while the upper five layers are typically implemented through the use of software only.



Layer Seven – Application

The Application Layer's function is to provide services to the end-user such as email, file transfers, terminal access, and network management. This is the primary layer of interaction for the end-user in the OSI Model.

Layer Six – Presentation

The Presentation Layer's primary responsibility is to define the syntax that network hosts use to communicate. Compression and encryption fall in the functions of this layer. It is sometimes referred to as the "syntax" layer and is responsible for transforming information or data into format(s) the application layer can use.

Layer Five – Session

The Session Layer establishes process to process communications between two or more networked hosts. Under OSI, this layer is responsible for gracefully closing sessions (a property of TCP) and for session check pointing and recovery (not used in IP). It is used in applications that make use of remote procedure calls.

Layer Four – Transport

The Transport Layer is responsible for the delivery of messages between two or more networked hosts. It handles fragmentation and reassembly of messages and controls the reliability of a given link.

Layer Three – Network

The Network Layer is primarily responsible for establishing the paths used for data transfer on the network. Network routers operated at this layer which can commonly be divided into three sub-layers: Sub network access, Sub network-dependent convergence, and Sub network-independent convergence.

Layer Two – Data Link

The Data Link Layer is primarily responsible for communications between adjacent network nodes. Network switches and hubs operate at this layer which may also correct errors generated in the Physical Layer.

Layer One – Physical

The Physical Layer handles the bit level transmission between two or more network nodes. Components in this layer include connectors, cable types, pin-outs, and voltages which are defined by the applicable standards organization.

Real World Protocols Map to the OSI Model

The following are commonly used or implemented protocols mapped to the appropriate layer of the OSI Model (as best as they can be mapped).

Layer	Name	Common Protocols
7	Application	SSH, FTP, telnet
6	Presentation	HTTP, SNMP, SMTP
5	Session	RPC, Named Pipes, NETBIOS
4	Transport	TCP, UDP
3	Network	IP
2	Data Link	Ethernet
1	Physical	Cat-5, Co-axial or Fiber Optic Cable

TCP/IP Model Functions

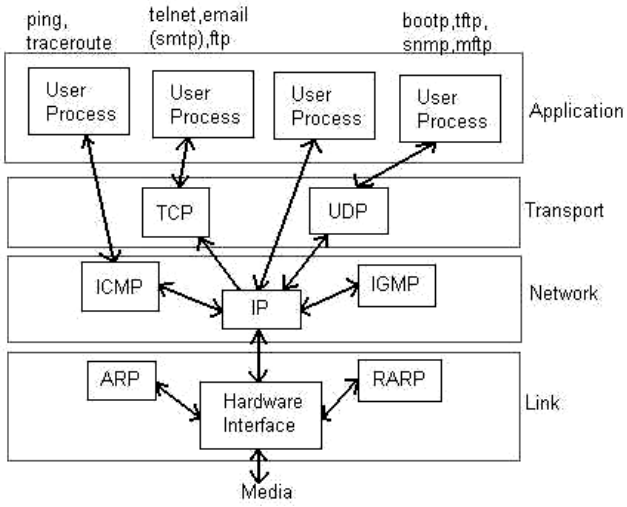
The TCP/IP Model has four functions. Starting from the lowest level, these include the Physical Layer, the Link Layer, the Internet, and the transport layers.

Physical Layer – The Physical Layer consists of purely hardware and includes the network interface card, connection cable, satellite, etc.

Link Layer – Also referred to as the "Network Access Layer." It is the networking scope of the local network connection that a host is attached. The lowest layer of IP, it is used to move data packets between the Internet Layer interfaces of two hosts on the same link. Controlling the process can be accomplished in either the software driver for the network card or via firmware in the chipset. The specifications for translating network addressing methods are included in the TCP/IP model, but lower level aspects are assumed to exist and not explicitly defined. A hierarchical encapsulation sequence is not dictated either.

Internet Layer – Handles the problem of sending data packets to or across one or more networks to a destination address in the routing process.

Transport Layer – The Transport Layer is responsible for end-end message transfer capabilities that are independent of the network. The specific tasks in this layer include error, flow, and congestion control, port numbers, and segmentation. Message transmission at this layer can either be connection-based as defined in TCP, or connectionless as implemented in the User Datagram Protocol (UDP).



The Internet Protocol performs two functions:

1 – Host identification and addressing. This function uses a hierarchical addressing system referred to as the IP address.

2 – Packet routing. This is the task of moving data packets from the source to destination host by sending the information to the next router or network node that is closer to the final destination. Information can be transported that relates to a number of upper layer protocols which are identified by a unique protocol number. Some examples are IGMP (Internet Group Management Protocol) and ICMP (Internet Control Message Protocol) that perform internetworking functions which help show the differences in the TCP/IP and OSI models.

TCP/IP Model Map to Real World Networking

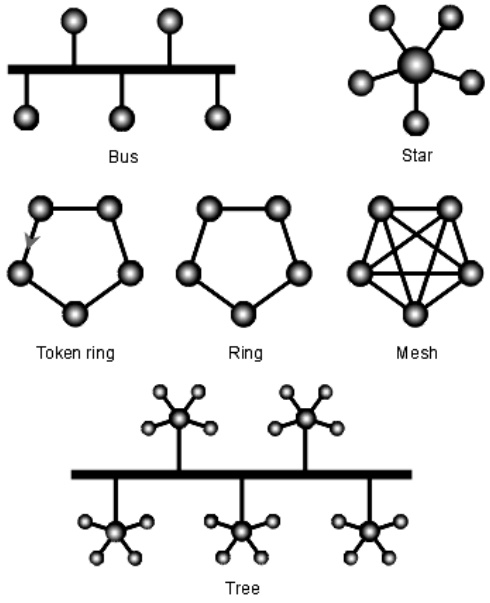
The TCP/IP model has become the defacto standard for real world implementation of networking. Some of the real world protocol mappings to the TCP/IP Model layers are:

TCP/IP Model	
Application Layer	FTP, HTTP, POP3, IMAP, telnet, SMTP, DNS, TFTP
Transport Layer	TCP, UDP, RTP
Internet Layer	IP, ICMP, ARP, RARP
Network Interface Layer	Ethernet, Token Ring, FDDI, X.25, Frame Relay, RS-232, v.36

Network Topology

In communication networks, a topology is a usually schematic description of the arrangement of a network, including its nodes and connecting lines. There are two ways of defining network geometry: the physical topology and the logical (or signal) topology.

The physical topology of a network is the actual geometric layout of workstations. There are several common physical topologies, as described below and as shown in the illustration.



In the bus network topology, every workstation is connected to a main cable called the bus. Therefore, in effect, each workstation is directly connected to every other workstation in the network.

In the star network topology, there is a central computer or server to which all the workstations are directly connected. Every workstation is indirectly connected to every other through the central computer.

In the ring network topology, the workstations are connected in a closed loop configuration. Adjacent pairs of workstations are directly connected. Other pairs of workstations are indirectly connected, the data passing through one or more intermediate nodes.

If a Token Ring protocol is used in a star or ring topology, the signal travels in only one direction, carried by a so-called token from node to node.

The mesh network topology employs either of two schemes, called full mesh and partial mesh. In the full mesh topology, each workstation is connected directly to each of the others. In the partial mesh topology, some workstations are connected to all the others, and some are connected only to those other nodes with which they exchange the most data.

The tree network topology uses two or more star networks connected together. The central computers of the star networks are connected to a main bus. Thus, a tree network is a bus network of star networks.

Logical (or signal) topology refers to the nature of the paths the signals follow from node to node. In many instances, the logical topology is the same as the physical topology. But this is not always the case. For example, some networks are physically laid out in a star configuration, but they operate logically as bus or ring networks.

Chapter 11 – Clustering and Load Balancing

Cluster

A cluster is defined as a group of application servers that transparently run a J2EE application as if it were a single entity. There are two methods of clustering: vertical scaling and horizontal scaling. Vertical scaling is achieved by increasing the number of servers running on a single machine, whereas horizontal scaling is done by increasing the number of machines in the cluster. Horizontal scaling is more reliable than vertical scaling, since there are multiple machines involved in the cluster environment, as compared to only one machine. With vertical scaling, the machine's processing power, CPU usage, and JVM heap memory configurations are the main factors in deciding how many server instances should be run on one machine (also known as the server-to-CPU ratio).

Clustering for Scalability & Failover

Scalability

Clustering typically makes one instance of an application server into a master controller through which all requests are processed and distributed to a number of instances using industry standard algorithms like round robin, weighted round robin, and least connections. Clustering, like load balancing, enables horizontal scalability that is the ability to add more instances of an application server nearly transparently to increase the capacity or response time performance of an application. Clustering features usually include the ability to ensure an instance is available through the use of ICMP ping checks and, in some cases, TCP or HTTP connection checks.

Server Affinity

Clustering uses server affinity to ensure that applications requiring the user interact with the same server during a session get to the right server. This is most often used in applications executing a process, for example order entry, in which the session is used between requests (pages) to store information that will be used to conclude a transaction, for example a shopping cart.

High Availability (Failover)

Clustering solutions claim to provide HA/Failover capabilities, when this failover is related to application process level failover, not high availability of the clustering controller itself. This is an important distinction as in the event the clustering controller instance fails, the entire system falls apart. While cluster-based load-balancing provides high availability for members of the cluster, the controller instance becomes a single point of failure in the data path.

Transparency

Many clustering solutions require a node-agent be deployed on each instance of an application server being clustered by the controller. This agent is often already deployed, so it's often not a burden in terms of deployment and management, but it is another process running on each server that is consuming resources such as memory and CPU and which adds another point of failure into the data path.

Load Balancing

Load balancing (also known as high availability switch over) is a mechanism where the server load is distributed to different nodes within the server cluster, based on a load balancing policy. Rather than execute an application on a single server, the system executes application code on a dynamically selected server. When a client requests a service, one (or more) of the cooperating servers is chosen to execute the request. Load balancers act as single points of entry into the cluster and as traffic directors to individual web or application servers.

Two popular methods of load balancing in a cluster are DNS round robin and hardware load balancing. DNS round robin provides a single logical name, returning any IP address of the nodes in the cluster. This option is inexpensive, simple, and easy to set up, but it doesn't provide any server affinity or high availability. In contrast, hardware load balancing solves the limitations of DNS round robin through virtual IP addressing. Here, the load balancer shows a single IP address for the cluster, which maps the addresses of each machine in the cluster. The load balancer receives each request and rewrites headers to point to

other machines in the cluster. If we remove any machine in the cluster, the changes take effect immediately. The advantages of hardware load balancing are server affinity and high availability; the disadvantages are that it's very expensive and complex to set up.

Load Balancing Algorithms

There are many different algorithms to define the load distribution policy, ranging from a simple round robin algorithm to more sophisticated algorithms used to perform the load balancing. Some of the commonly used algorithms are:

- Round-robin
- Random
- Weight-based
- Minimum load
- Last access time
- Programmatic parameter-based (where the load balancer can choose a server based upon method input arguments)

Chapter 12 – Continuous Integration Process

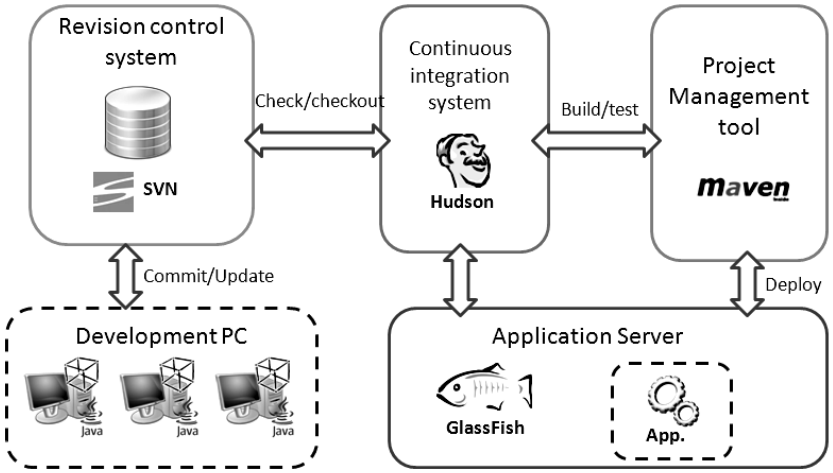
Continuous Integration

CI implements continuous processes of applying quality control. In other words, small pieces of effort applied frequently. CI aims to improve the quality of software, and reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development.

Hudson-Jenkins

Extensible continuous integration server features are:

1. Check-out the source code from the repository
2. Build and test the project. It covers unit testing, code quality reporting, code coverage reporting and build status notification by using tools like PMD, CheckStyle, VeraCode, Cruise Control, Damage Control, Cobertura, SONAR, and so on.)
3. Publish the result.
4. Communicate the results to team members.
5. Extendable with plug-ins (Clover)
6. Monitor the executions of externally run cron or procmail jobs.

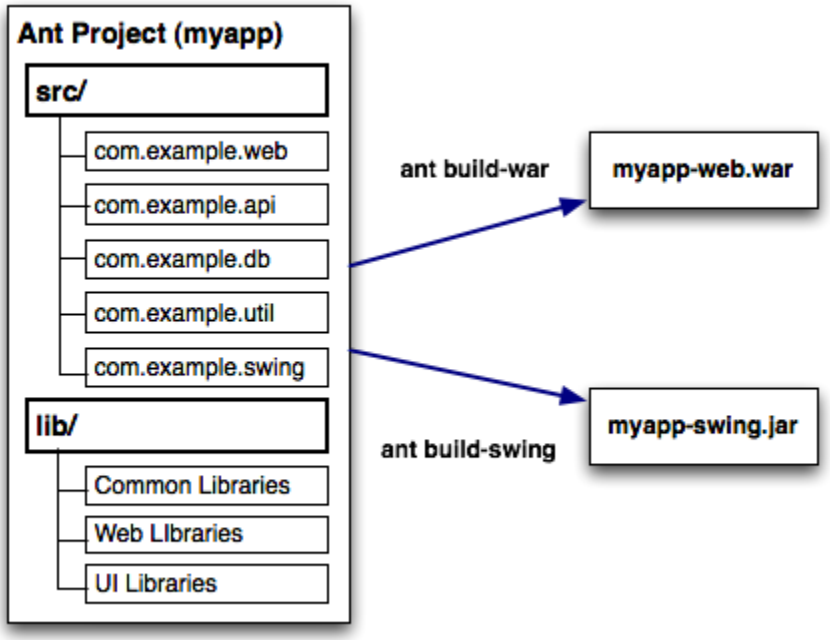


Chapter 13 - Project Management Tools

Ant

Apache Ant ("**A**nother **N**eat **T**ool") is the build tool used by most Java development projects. Ant made it trivial to integrate JUnit tests with the build process; Ant has made it easy for willing developers to adopt test-driven development, and even Extreme Programming. Ant build files are written in XML.

ANT Project Structure



Sample build.xml file

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
  <target name="clean" description="remove intermediate files">
    <delete dir="classes"/>
  </target>
  <target name="clobber" depends="clean" description="remove all artifact files">
    <delete file="hello.jar"/>
  </target>
  <target name="compile" description="compile the Java source code to class files">
    <mkdir dir="classes"/>
    <javac srcdir="." destdir="classes"/>
  </target>
  <target name="jar" depends="compile" description="create a Jar file for the application">
    <jar destfile="hello.jar">
      <fileset dir="classes" includes="**/*.class"/>
      <manifest>
        <attribute name="Main-Class" value="HelloProgram"/>
      </manifest>
    </jar>
  </target>
</project>
```

Within each target are the actions that Ant must take to build that target; these are performed using built-in tasks. For example, to build the compile target Ant must first create a directory called classes (Ant will only do so if it does not already exist) and then invoke the Java compiler. Therefore, the tasks used are mkdir and javac. These perform a similar task to the command-line utilities of the same name.

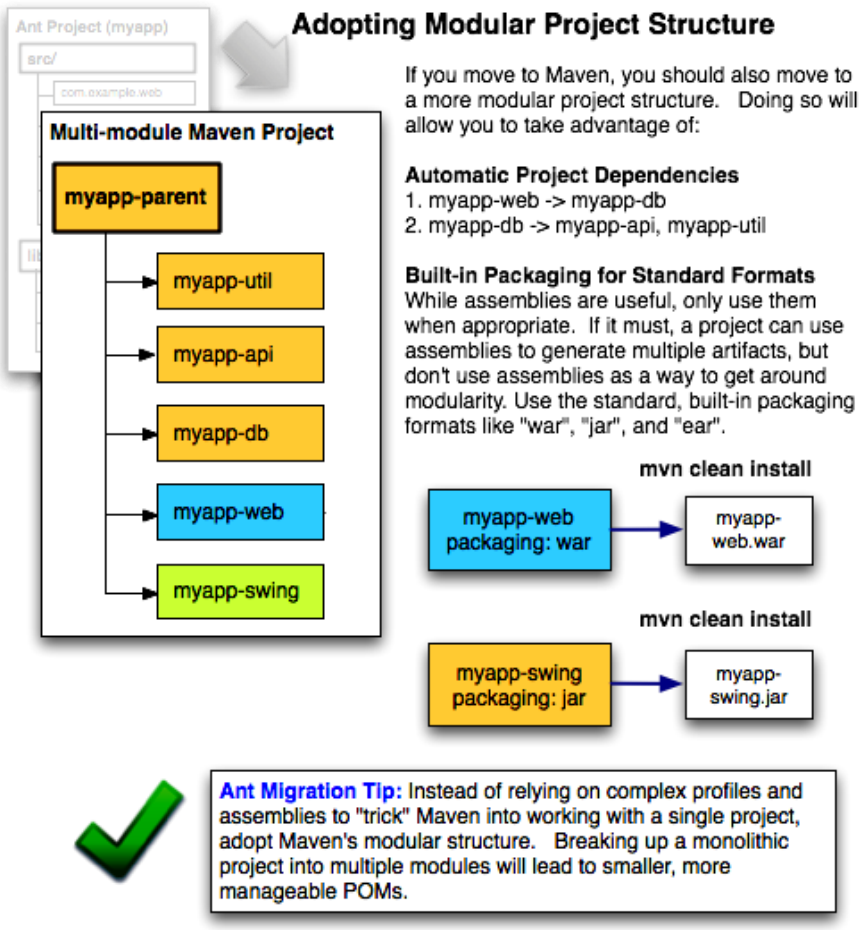
Another task used in this example is named jar:
<jar destfile="hello.jar">

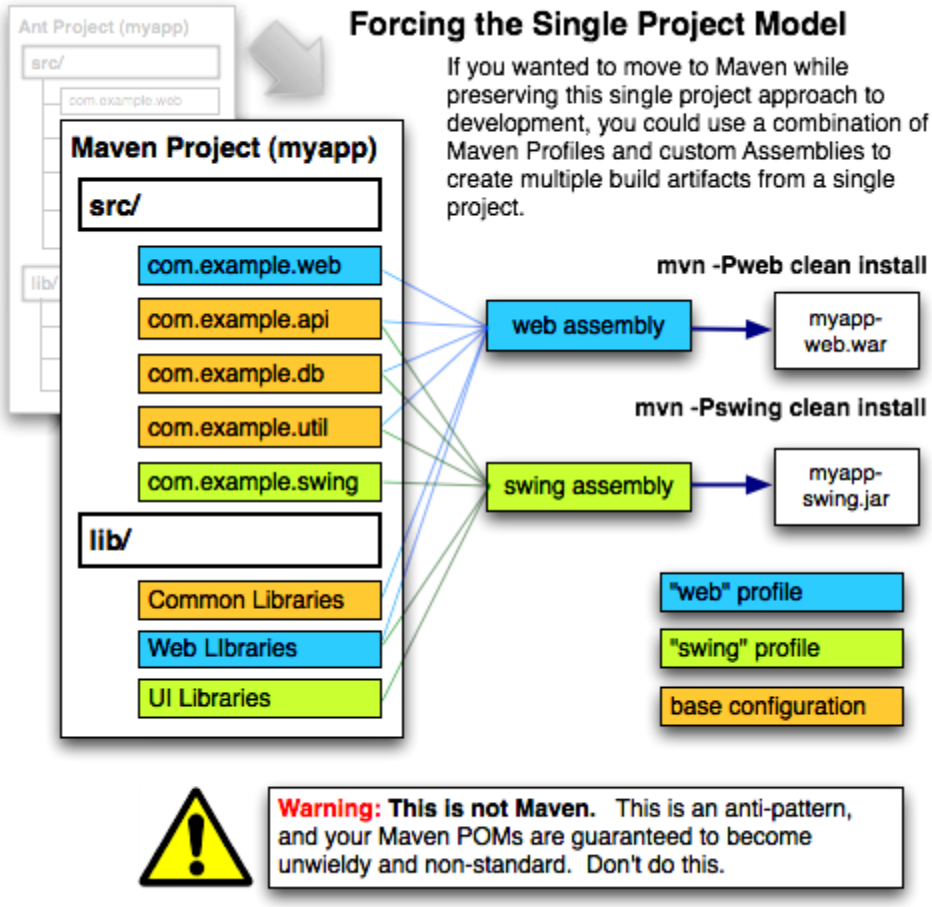
This ant task has the same name as the common java command-line utility, JAR, but is really a call to the ant program's built-in jar/zip file support. This detail is not relevant to most end users, who just get the JAR they wanted, with the files they asked for.
Many Ant tasks delegate their work to external programs, either native or Java. They use Ant's own <exec> and <java> tasks to set up the command lines, and handle all the details of mapping from information in the build file to the program's arguments -and interpreting the return value. Users can see which tasks do this (e.g. <cvs>, <signjar>, <chmod>, <rpm>), by trying to execute the task on a system without the underlying program on the path, or without a full Java Development Kit (JDK) installed.

Maven

Apache Maven is a project management, software comprehension and build automation tool primarily for Java. It can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. Maven serves a similar purpose to the Apache Ant tool, but it is based on different concepts and works in a profoundly different manner. Maven uses a construct known as a **Project Object Model (POM)** to describe the software project being built, its dependencies on other external modules and components, and the build order. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging.

Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository. This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated.
Maven is built using a plugin-based architecture that allows it to make use of any application controllable through standard input. Theoretically, this would allow anyone to write plugins to interface with build tools (compilers, unit test tools, etc.) for any other language. In reality, support and use for languages other than Java has been minimal.




Ant Project (myapp)
src/
com.example.web
lib/
Forcing the Single Project Model
If you wanted to move to Maven while preserving this single project approach to development, you could use a combination of Maven Profiles and custom Assemblies to create multiple build artifacts from a single project.
Maven Project (myapp)
src/
com.example.web
com.example.api
com.example.db
com.example.util
com.example.swing
lib/
Common Libraries
Web Libraries
UI Libraries
mvn -Pweb clean install
web assembly
myapp-web.war
mvn -Pswing clean install
swing assembly
myapp-swing.jar
"web" profile
"swing" profile
base configuration
Warning: This is not Maven. This is an anti-pattern, and your Maven POMs are guaranteed to become unwieldy and non-standard. Don't do this.

Sample pom.xml file

```
<project>
 <!-- model version is always 4.0.0 for Maven 2.x POMs -->
 <modelVersion>4.0.0</modelVersion>
 <!-- project coordinates, i.e. a group of values which
    uniquely identify this project -->
 <groupId>com.mycompany.app</groupId>
 <artifactId>my-app</artifactId>
 <version>1.0</version>
 <!-- library dependencies -->
 <dependencies>
  <dependency>
   <!-- coordinates of the required library -->
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <!-- this dependency is only used for running and compiling tests -->
   <scope>test</scope>
  </dependency>
 </dependencies>
</project>
```

Project Object Model

A Project Object Model (POM) provides the entire configuration for a single project. General configuration covers the project's name, its owner and its dependencies on other projects. One can also configure individual phases of the build process, which are implemented as plugins. For example, one can configure the compiler-plugin to use Java version 1.5 for compilation, or specify packaging the project even if some unit test fails.

Larger projects should be divided into several modules, or sub-projects, each with its own POM. One can then write a root POM through which one can compile all the modules with a single command. POMs can also inherit configuration from other POMs. All POMs inherit from the Super POM] by default. The Super POM provides default configuration, such as default source directories, default plugins, and so on.

Plugins

Most of Maven's functionality is in plugins. A plugin provides a set of goals that can be executed using the following syntax:

mvn [plugin-name]:[goal-name]

For example, a Java project can be compiled with the compiler-plugin's compile-goal by running mvn compiler:compile.

There is Maven plugins for building, testing, source control management, running a web server, generating Eclipse project files, and much more. Plugins are introduced and configured in a <plugins>-section of a pom.xml file. Some basic plugins are included in every project by default, and they have sensible default settings.

However, it would be cumbersome if one would have to run several goals manually in order to build, test and package a project:

mvn compiler:compile
mvn surefire:test
mvn jar:jar

Maven's lifecycle-concept handles this issue.

Build lifecycles

Build lifecycle is a list of named phases that can be used to give order to goal execution. One of Maven's standard lifecycles is the default lifecycle, which includes the following phases, in this order:

1. process-resources
2. compile
3. process-test-resources
4. test-compile
5. test
6. package
7. install
8. deploy

Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

Goals provided by plugins can be associated with different phases of the lifecycle. For example, by default, the goal "compiler:compile" is associated with the compile-phase, while the goal "surefire:test" is associated with the test-phase. When the command ‘mvn test’ is executed, Maven will run all the goals associated with each of the phases up to the test-phase. So it will run the "resources:resources"-goal associated with the process-resources-phase, then "compiler:compile", and so on until it finally runs the "surefire:test"-goal.

Maven also has standard lifecycles for cleaning the project and for generating a project site. If cleaning were part of the default lifecycle, the project would be cleaned every time it was built. This is clearly undesirable, so cleaning has been given its own lifecycle.

Thanks to standard lifecycles, one should be able to build, test and install every Maven-project using the mvn install-command.

Dependencies

The example-section hinted at Maven's dependency-handling mechanism. A project that needs the Hibernate-library simply has to declare Hibernate's project coordinates in its POM. Maven will automatically download the dependency and the dependencies that Hibernate itself needs (called transitive dependencies) and store them in the user's local repository. Maven 2 Central Repository is used by default to search for libraries, but one can configure the repositories used (e.g. company-private repositories) in POM. There are search engines such as Maven Central, which can be used to find out coordinates for different open-source libraries and frameworks.

Maven compared with ANT

Build Configuration Aspects	Maven	Ant
Build File Name	pom.xml	build.xml
Project Name	Manual (line 4)	Manual (line 1)
Project Directory Structure	Automatic	Manual (lines 4-6)
Build Preparation	Automatic	Manual (init target)
Library Dependencies	Manu-matic	Manual (not in build ex)
Compile	Automatic	Manual (compile target)
Testing	Automatic	Manual (not in build ex)
Packaging	Automatic	Manual (dist task)
Versioning	Automatic	Manual (tstamp)
Deployment	Automatic	Manual (not in build ex)
Site Generation	Automatic	Manual (not in build ex)
Clean-up	Automatic	Manual (clean task)

The fundamental difference between Maven and Ant is that Maven's design regards all projects as having a certain structure and a set of supported task work-flows (e.g. getting resources from source control, compiling the project, unit testing, etc.). While most software projects in effect support these operations and actually do have a well-defined structure, Maven requires that this structure and the operation implementation details be defined in the POM file. Thus, Maven relies on a convention on how to define projects and on the list of work-flows that are generally supported in all projects.

This design constraint is more like how an IDE handles projects and it provides many benefits, such as a succinct project definition and the possibility of automatic integration of a Maven project with other development tools such as IDEs, build servers, etc.

The downside is that it requires a user to first understand what a project is from the Maven point of view and how Maven works with projects, because what happens when one executes a phase in Maven is not immediately obvious just from examining the Maven project file. This required structure is also often a barrier in migrating a mature project to Maven because it is usually hard to adapt from other approaches.

In Ant, projects do not really exist from the tool's technical perspective. Ant works with XML build scripts defined in one or more files. It processes targets from these files and each target executes tasks. Each task performs a technical operation such as running a compiler or copying files around. Tasks are executed primarily in the order given by their defined dependency on other tasks. Thus, Ant is a tool that chains together tasks and executes them based on inter-dependencies and other Boolean conditions.

The benefits provided by Ant are also numerous. It has an XML language optimized for clearer definition of what each task does and on what it depends. Also, all the information about what will be executed by an Ant target can be found in the Ant script.

A developer not familiar with Ant would normally be able to determine what a simple Ant script does just by examining the script. This is not usually true for Maven.

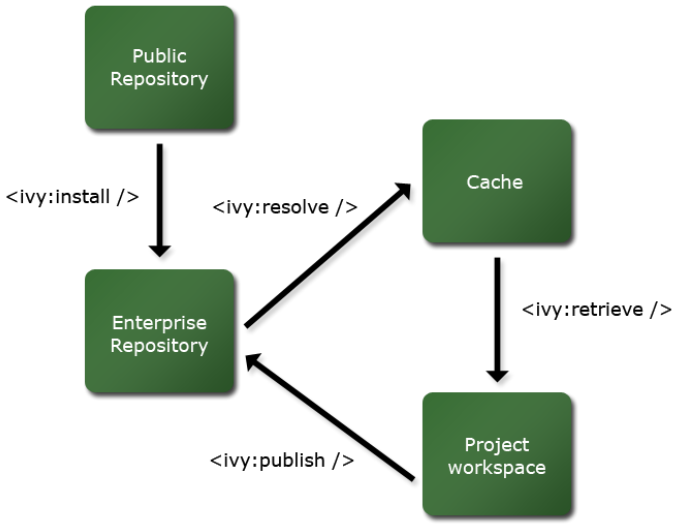
However, even an experienced developer that is new to a project using Ant cannot infer what the higher level structure of an Ant script is and what it does without examining the script in detail. Depending on the script's complexity, this can quickly become a daunting challenge. With Maven, a developer who previously worked with other Maven projects can quickly examine the structure of a never before seen Maven project and execute the standard Maven work-flows against it while already knowing what to expect as an outcome.

It is possible to use Ant scripts that are defined and behave in a uniform manner for all projects in a working group or an organization. However, when the number and complexity of projects rises, it is also very easy to stray from the initially desired uniformity. With Maven this is less of a problem because the tool always imposes a certain way of doing things.
Note that it is also possible to extend and configure Maven in a way that departs from the Maven way of doing things.

Ivy

Apache Ivy is a transitive relation dependency manager. It is a sub-project of the Apache Ant project, with which Ivy works to resolve project dependencies. An external XML file defines project dependencies and lists the resources necessary to build a project. Ivy then resolves and downloads resources from an artifact repository: either a private repository or one publicly available on the Internet.

To some degree, it competes with Apache Maven, which also manages dependencies. However, Maven is a complete build tool, whereas Ivy focuses purely on managing transitive dependencies.



Features:

- Managing project dependencies
- XML-driven declaration of project dependencies and jar repositories
- Automatic retrieval of transitive dependency definitions and resources
- Automatic integration to publicly-available artifact repositories
- Resolution of dependency closures
- Configurable project state definitions, which allow for multiple dependency-set definitions
- Publishing of artifacts into a local enterprise repository

Chapter 14 – Unit Test

JUnit

A unit test is a piece of code written by a developer that tests a specific functionality in the code. Unit tests can ensure that functionality is working and can be used to validate that this functionality still works after code changes.

JUnit is a framework developed in Java for unit test purposes.

- extends junit.framework.TestCase
- protected void setup()
- protected void tearDown()
- private void testSuite()

Assertion Methods

1. assertNull
2. assertNotNull
3. assertEquals
4. assertNotEquals
5. assertSame
6. assertNotSame
7. assertTrue
8. assertFalse
9. fail

You can also use an **inner class with overridden methods** to mock and test the specific functionality in the code.

JUnit 4.x Annotations

1. @Test
2. @Test(expected=ExceptionName.class)
3. @Test(timeout=100)
4. @Before
5. @After
6. @BeforeClass - setup()
7. @AfterClass - tearDown()
8. @Ignore

Easy Mock

Easy Mock provides Mock Objects for interfaces (and objects through the class extension) by generating them on the fly using Java's proxy mechanism. Due to Easy Mock's unique style of recording expectations, most refactoring will not affect the Mock Objects. So, **Easy Mock is a perfect fit for Test-Driven Development.**

Interface Mockups - High Level Steps

1. Creating the mock
2. Defining or Recording the behavior
3. Signal a change into “replay” mode
4. Test with your mock object
5. Verify (optional)

Mock Types

There are three types of mock objects

1) Strict Mock

- The method name is createStrictMock()
- Calling an undefined behavior results in Assertion Error
- The Order matters

2) Mock (Standard)

- The method name is createMock()
- Calling an undefined behavior results in Assertion Error
- The Order doesn’t matter

3) Nice Mock

- The method name is createNiceMock()
- Calling an undefined behavior results in no Assertion Error
- The Order doesn’t matter

Mock Modes

The two modes of a mock are **'Record'** and **'Replay'**.

The initial mode is 'Record. When in record mode, you tell the mock object what to expect and how to react (not relevant for void). Calling a method in record mode defines an expected behavior. Behavior is the method and the parameter values.

In 'Replay' mode, the mock objects act per the behaviors defined while in record mode. The optional 'Verify' confirms that the methods you defined were all called.

Behavior

The methods with return types need an expect(). The methods with return types need a return object defined.

- andReturn()
- andThrow()
- andStubReturn()
- andStubThrow()

The stub methods won't be considered in verify().

Every behavior definition is just ONE definition

- times(int num)
- times(int min, int max)
- anyTimes()
- atLeastOnce()

Flexible Argument Matching

It lets you define the looser requirements on the parameters. The options are:

- eq(X value)
- isNull()
- notNull()
- isA(Class clazz)
- anyInteger()
- anyBoolean()
- anyObject()

Usage:

```
policyUtil.deletePolicyErrors(
     (Policy)EasyMock.notNull());
```

How to write a test case using Easy Mock

- Declare org.easymock.IMocksControl control
- In setup method, contol = createStrictControl()
- Create an instance of Testable class with overridden methods.
- control.createMock()
- expect
- andReturn
- control.checkOrder(Boolean)
- control.replay
- testableObject.invokeMethod()
- assertion
- control.verify
- control.reset
- fail

Sample Code:

```
@Test
public void testPlayWithPolicy2() {
    final FakeClass fakeClass = new FakeClass();

    final Policy policy = Policy.PolicyFactory.create();
    final IPolicyUtil policyUtil =
        EasyMock.createMock(IPolicyUtil.class);

    EasyMock.expect(policyUtil.getCancelledDate(policy)).andReturn(
        Date.getInstance().subtractMonths(1));
    EasyMock.expect(policyUtil.getGaragedZipCode(policy)).andReturn(
        "12345");

    EasyMock.replay(policyUtil);
    assertTrue(fakeClass.playWithPolicy2(policyUtil, policy));
}
```

Partial Class Mockups

- Think of it like extending a class on the fly
- Specify which methods you want to mock
- All other methods function as usual
- Can't mock final classes
- Can't mock private methods

Partial Class Mockups – High Level Steps

1. Creating the mock builder
2. Define the methods to be mocked
3. Creating the mock
4. Defining or Recording the behavior
5. Signal a change into "replay" mode
6. Test with your mock object
7. Verify (optional)

Sample Code:

```
// step 1: create mock builder
IMockBuilder<CommonServicesBean> csMockBuilder =
    EasyMock.createMockBuilder(CommonServicesBean.class);

// step 2: define methods to be mocked
csMockBuilder.addMockedMethod("getMailingStatesMinusNonUS");
csMockBuilder.addMockedMethod("getRegisteredStatesMinusCanada");

// step 3: create mock
CommonServicesBean csBean = csMockBuilder.createNiceMock();

// step 4: define behavior
EasyMock.expect(csBean.getMailingStatesMinusNonUS()).andStubReturn(
    new SelectItem[] {});
EasyMock.expect(csBean.getRegisteredStatesMinusCanada()).andStubReturn(
    new SelectItem[] {});

// step 5: replay
EasyMock.replay(csBean);

// step 6: test
fakeClass.playWithCommonServiceBean(csBean);

// step 7: verify
EasyMock.verify(csBean);
```