TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

SWEN 301 : Scalable Software Development

Process Automation

Convention over Configuration

Jens Dietrich **(jens.dietrich@vuw.ac.nz)**

Overview

- automating builds
- ANT and Maven
- repositories and package managers
- convention over configuration
- continuous integration
- continuous delivery and deployment

Automating Builds: What is the Point ?

- IDEs / CLI compilers are efficient, and often compile in the background
- there is **much more to do**, **tasks** to do in each iteration include:
  - prepare (Clean up, create folders, set variables, interact with repo, ..)
  - (re-) generate code
  - compile
  - run static analysers (e.g.: jdepend, checkstyle, findbugs, pmd)
  - generate documentation (e.g.: javadoc, rev. engineer UML)
  - test, and measure coverage
  - create artefacts (e.g.: create jars/wars with bytecode, resources and metadata)
  - interact with repos (e.g., commit code and reports, release artefacts to Maven repo, ..)
- make builds reproducible to prevent software supply chain attacks (like SolarWinds)

Early Approaches: Make 1976

build tools for C, ca 1976, still widely used

Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, `cc *.o` was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.

— Stuart Feldman, *The Art of Unix Programming*, Eric S. Raymond 2003

https://en.wikipedia.org/wiki/Make_(software)

Early Approaches: ANT

- ant ("Another Neat Tool"), developed @ SUN around 2000
- build tool for Java, uses XML syntax
- core concepts:
  - targets : steps in the workflow, with dependencies between them
  - dependencies: if target A depends on B, B must be executed before A
  - tasks: basic commands used to write targets, comprehensive built-in library
  - staying DRY (=don’t repeat yourself): define variables, reference them
- this is all defined in XML files (usually build.xml)

ANT Example

```
<project>
    <target name="clean">
        <delete dir="build"/>
    </target>
    <target name="compile" depends="clean">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>
    <target name="jar" depends="compile">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
        </jar>
    </target>
</project>
```

named target

task (command) with parameters

dependency between targets, need to run clean first

from https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html, modified

Staying DRY (Do Not Repeat Yourself)

define variables

```
<project>
    <property name="program.name" value="HelloWorld"/>
    <property name="program.version" value="1.0.0"/>
    ..
    <target name="jar" depends="compile">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/${program.name}-${program.version}.jar" />
    </target>
</project>
```

reference variables

from https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html, modified

Dealing with (Library) Dependencies

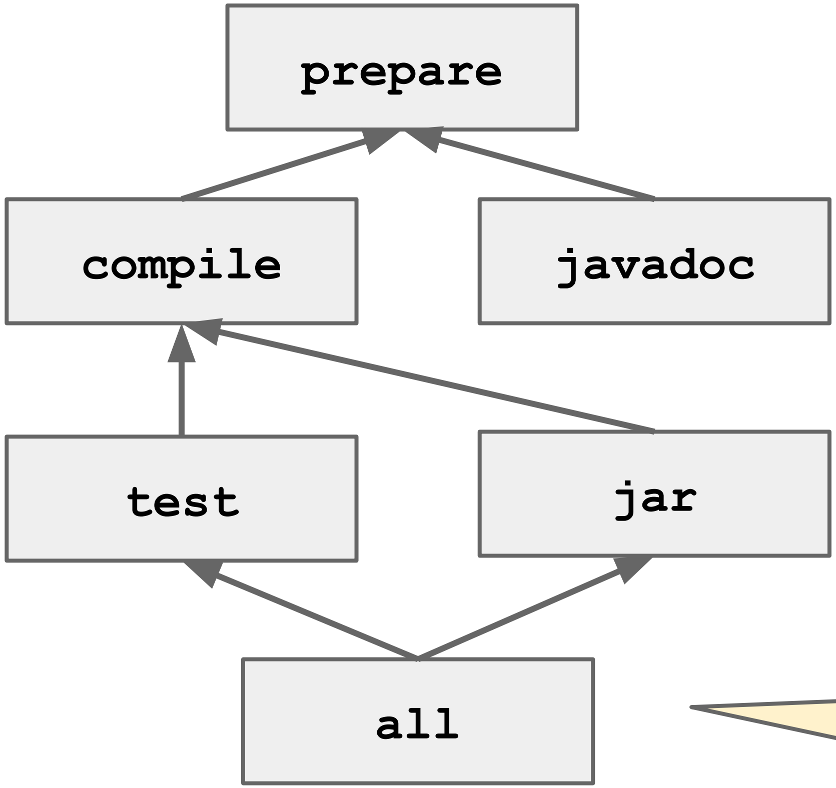
```
<project name="HelloWorld" basedir="." default="main">
    ...
    <property name="lib.dir" value="lib"/>
    <path id="classpath">
        <fileset dir="${lib.dir}" includes="**/*.jar"/>
    </path>
    ...
    <target name="compile">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}"
          classpathref="classpath"/>
    </target>
    ...
</project>
```

define classpath variables:
all jars in /lib
predefined variable type for paths

use classpath to compile

from https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html, modified

Example ANT Structure: Targets and Dependencies



targets only requiring source code and dependencies

targets only requiring byte code (compilation results)

virtual "run all" target - does not do anything by itself, but depends on all other targets, this is usually the default target

Running ANT

- platform independent
- run **`ant`** in folder with buildscripts
- can declare default target, running just **`ant`** runs this
- or use **`ant <target>`**, example: **`ant compile`**
- more features: modularity (invoke scripts), plugins, listeners, ..

ANT Issues

- build scripts are quickly becoming large and complex: https://github.com/apache/ant/blob/master/build.xml 343dff9 has 2141 lines !
- all dependencies must be specified: need to build classpath, manually transitive dependencies: developers must deal with DLL / classpath hell and copy all jars
- basically a scripting language -- no common structure or pattern enforced

Maven -- The Better ANT

- another build tool, also XML-based, also by the Apache foundation
- key differences:

1. enforces and encourages a much more canonical structure how to organise a project -- **convention over configuration**
2. developers only need to declare symbolic dependencies, Maven will fetch the actual jars, and resolve transitive dependencies

Maven Core concepts

- the **POM** is an xml file (**`pom.xml`**) that has the project configuration
- Maven creates **artifacts** - usually jar files of executable (incl. library) code
- artifacts have a composite name consisting of **g**roup id, **a**rtefact name and a **v**ersion (short: GAV)
- an artifact may **depend on** other artifacts
- artifacts are resolved over the network against a **Maven repository** when maven runs, such as https://central.sonatype.com/
- parts of this repository are locally cached, usually in **`~/.m2`**
- the repository is searchable to locate artefacts

Phases

- Maven builds are executed in **lifecycle phase**
- there are dependencies between phases
- syntax (from terminal): **`mvn test`**
- build outputs are stored by default in a **`/target`** folder Maven generates
- classes (bytecode) generated by **`mvn compile`** are in **`target/classes`**
- **`mvn test`** generates a folder **`/target/surefire-reports`** that contains detailed reports on test outcomes

Standard Phases (Selection):

1. **validate**: validate the project is correct and all necessary information is available
2. **compile**: compile the source code of the project
3. **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
4. **package**: take the compiled code and package it in its distributable format, such as a JAR.
5. **verify**: run any checks to verify the package is valid and meets quality criteria

Standard Phases (Selection):

6. **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.
7. **clean**: cleans up artifacts created by prior builds
8. **site**: generates site documentation for this project

A Minimalistic POM

```
<project>
      <modelVersion>4.0.0</modelVersion>
      <groupId>nz.ac.vuw.swen301</groupId>
      <artifactId>helloworld</artifactId>
      <packaging>jar</packaging>
      <version>1.0-SNAPSHOT</version>
      <name>bdd</name>
</project>
```

this is sufficient in order to:

- compile the sources with mvn compile
- run all tests in classes matching any of the following **pattern conventions** : **`Test*`**, **`*Test`**, **`*Tests`** and **`*TestCase`** with mvn test
- build a jar file deliverable with **mvn package**
- **`mvn package`** will also compile the code and run tests, without declaring dependencies

The Maven Project Layout (Simplified)

```
src/main/java              Application/Library sources
src/main/resources         Application/Library resources
src/main/config    Configuration files
src/main/webapp    Web application sources
src/test/java              Test sources
src/test/resources         Test resources
src/site                   Site
LICENSE.txt                Project's license
NOTICE.txt                 Notices and attributions ..
README.txt                 Project's readme
```

https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Convention over Configuration

- for a minimalistic POM to work, users must comply to the following rules:
  - the project layout
  - the test class naming patterns
- this is facilitated by tools, e.g. the generation of initial project templates by **`mvn archetype:generate`**
- this approach to design is called Convention over Configuration
- it was pioneered by Ruby on Rails
- it aims at reducing the number of decisions programmers have to make
- tools can exploit the **semantics encoded in conventions**

Sidetrack:

The Benefits of Convention over Configuration Case Study:

Persisting Object Graphs

The JavaBean Model, aka POJOs (simplified)

- JavaBeans are simple Java classes (aka plain old Java objects - POJOs)
- properties have to follow conventions: matching pairs of setters (**`set*`**) and getters (**`get*`**) (aka accessors and mutators)
- JavaBean classes must also have public constructors without parameters

Bean Conventions Example

```
public class Student {
    public Student() {
        super();
    }
    private java.util.Date dob = null;
    ..
    public java.util.Date  getDob() {
        return dob;
    }
    public void setDob(java.util.Date dob) {
        this.dob = dob;
    }
    ..
}
```

public constructor without parameters (can be omitted only if no other constructor is defined)

matching pairs of setters and getters: types and names must match

Reflection

- reflection is the ability of a computer program to **access** and **modify** its structure and behaviour **at runtime**
- Java has built-in reflection facilities: classes like **`java.lang.Class`** and **`java.lang.reflect.Method`**
- the JavaBean model facilitates reflection
- utility: **`java.beans.Introspector`**

Reflection Examples

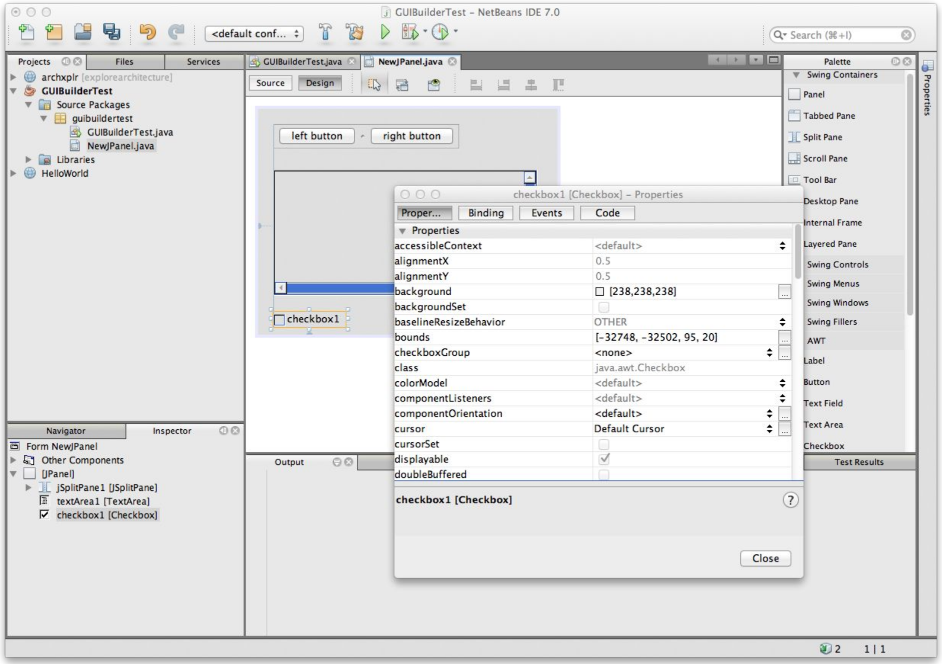
```
public static void main(String[] args) throws Exception {
    Lecturer s = new Lecturer();
    inspect(s);
}
public static void inspect(Object obj) throws Exception {
    BeanInfo beanInfo = Introspector.getBeanInfo(obj.getClass());
    PropertyDescriptor[] properties = beanInfo.getPropertyDescriptors();
    for (PropertyDescriptor property:properties) {
        System.out.print(property.getName());
        Method getter = property.getReadMethod();
        Object value = getter.invoke(obj,new Object[]{});
        System.out.print(" = ");
        System.out.println(value);
    }
}
```

get info for all properties - this will analyse getters/setters

this is a reference to the get method !

execute the get method without parameters

Application of Reflection: UI Builder



checkbox1 [Checkbox] - Properties

Proper... | Binding | Events | Code

Properties	
accessibleContext	<default>
alignmentX	0.5
alignmentY	0.5
background	[238,238,238]
backgroundSet	
baselineResizeBehavior	OTHER
bounds	[-32748, -32502, 95, 20]
checkboxGroup	<none>
class	java.awt.Checkbox
colorModel	<default>
componentListeners	<default>
componentOrientation	<default>
cursor	Default Cursor
cursorSet	
displayable	✓
doubleBuffered	

checkbox1 [Checkbox]

Close

a **property sheet** is used to customise components
this is built dynamically based on the properties discovered, and setters (invoke) are used to **modify** the values of properties

- UI builders can customise components
- this must be kept dynamic: new components can be added

Application of Reflection: Persistency

- Java has two built-in mechanisms to serialises objects: binary and XML serialisation
- use cases: save objects to files, networking
- XML serialisation is based on reflection
- basic idea: read object properties one by one, encode in XML and write to file
- when object is read from file, the class name is read and instantiated: **`Class.forName("classname").newInstance()`**
- this requires the constructor without parameters (and **will fail if it is missing and the convention has been violated!**)
- source code: https://github.com/jensdietrich/se-teaching/tree/main/xml-encoder

XML Encoder

```
Student s = new Student();
s.setName("Max");
s.setFirstName("Dietrich");
XMLEncoder encoder = new XMLEncoder(
    new FileOutputStream("student.xml")
);
encoder.writeObject(s);
encoder.close();
```

object written (serialised) to student.xml

XML Decoder

```
XMLDecoder decoder = new XMLDecoder(
    new FileInputStream("student.xml")
);
Student obj = (Student)decoder.readObject();
decoder.close();
```

object read from student.xml

a **cast** is necessary: this is a general-purpose utility and therefore returns instances of Object

Generated XML

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_33" class="java.beans.XMLDecoder
    <object class="reflection.Student">
        <void property="dob">
            <object class="java.util.Date">
                <long>1344223448903</long>
            </object>
        </void>
        <void property="firstName">
            <string>Bill</string>
        </void>
        <void property="name">
            <string>Clinton</string>
        </void>
    </object>
</java>
```

the class name can be used to create the class (**`Class.forName()`**) and then instantiate it (**`newInstance()`**)

properties are listed here: invoke on the property setters can be used to set the respective values

… back to Maven

Customising Maven

- conventions can be overridden and redefined using archetypes
- an archetype is essentially a project template
- plugins can be used to further customise Maven, in particular to add build functionality (phases) or custom reporting
- plugins are registered in the **`<plugins>`** sections of the pom

Maven Dependencies Example

```
<dependencies>
   <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.11</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>org.json</groupId>
       <artifactId>json</artifactId>
       <version>20160810</version>
   </dependency>
   ..
</dependencies>
```

JUnit is only needed during test phase, not at runtime

XML block copied from Maven repository search (https://mvnrepository.com/)

Dependency Resolution

- Maven manages (transitive) dependencies
- instead of copying libraries into the project, it is sufficient to declare them (using group+name+version)
- references are resolved against a repository
- Maven will fetch the respective artifacts during a build, **and further artifacts they might depend on**
- this means that an initial build can take a long time, but Maven will try to cache artifacts in a local cache
- the repository search function (such as http://search.maven.org/) is used to locate the references (in case of Maven, XML snippets)

Dependencies and Versioning

- Maven also supports flexible dependencies with features like references to version ranges, latest and release versions
- while this sounds like a good idea as it enables *auto-upgrades* when new versions become available in the repository, care must be taken as the new versions often violate API stability
- semantic versioning is a possible solution, but is not widely used (http://semver.org/)

Maven Repositories and Local Dependencies

- using the <repositories> element, it is possible to resolve dependencies against alternative and multiple repositories
- it is also possible to refer to a local repositories
- use cases for custom repos and local deps: closed-source projects

```
<dependency>
    <groupId>nz.ac.vuw.swen301</groupId>
    <artifactId>studentmemdb</artifactId>
    <version>1.0.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/lib/studentmemdb-1.0.0.jar</systemPath>
</dependency>
```

Maven Integrations

- all major IDEs have built-in Maven support, or plugins providing it
- usually, Maven is a project type that can be selected when creating new projects
- Maven then takes over classpath / buildpath management
- Maven needs a network connection (depending on cache status), i.e. proxies need to be configured as required

Scaling Up: Modular Projects

- for large projects, Maven support multi-module projects
- a parent project references several modules
- each module has a separate pom and can be build on its own
- building the parent project will build all modules

Contributing into the Central Repository

- projects can publish open source artefacts into the central repository
- then other projects can use it as dependency
- this requires some additional metadata in the POM, in particular the code hosting side used (**`<scm>`**) and the license (**`<licenses>`**)
- project must be stable to be released, and only depend on artifacts already released and available in the repository
- **`mvn deploy`** can be used to deploy artifacts into the remote rep
- details: https://maven.apache.org/repository/guide-central-repository-upload.html

Maven Alternatives

- there are several other build tools based on the same ideas, and also using artefacts from the Maven repository
- **ivy** is an ANT extension that integrated Maven’s dependency management into ANT
- **gradle** is a popular build tool, the main difference to Maven is that it uses Groovy instead of XML to write build scripts that are shorter and more concise
- https://mvnrepository.com/ search results provide snippets to embed symbolic dependencies into several build scripts

Gradle

- popular Maven alternative
- uses the same repository infrastructure
- main difference: uses Groovy instead of XML

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

Similar Build Tools in other Languages

- repo-based build tools are now mainstream in moist other languages
- https://libraries.io/ is a service that track releases across the most popular package managers
- examples:

language	**build tool(s)**	**package management**
Java	mvn, gradle, ant+ivy	Maven artefacts
JavaScript	npm	NPM
Ruby	rake	RubyGems
Python	pybuilder	PyPI

Build Triggers

- on demand: build is triggered explicitly (e.g., by running **`mvn`**)
- triggered: build starts when a certain event happens (e.g., a commit to the revision control system)
- scheduled: build is performanced periodically (e.g., nightly build)

Continuous Integration

- where should build run ?
- on client: checks for consistency with dependencies, and code on client
- on server (repo): checks for consistency of latest commits

Continuous Integration (ctd)

- the focus is on addressing the risk of integration (parts not matching)
- integration is performed as often as possible, i.e. developers are encouraged to commit into the same branch, or to merge often
- consistency is checked by running builds on the server
- build can be triggered by commits and/or special conditions (e.g. special tags, merge operations, or commit messages matching patterns)
- extensive notification and reporting options
- there are independent CI tools, and tool suits provided by code repo providers (github, bitbucket, gitlab, ..)
- examples: hudson/jenkins, travis, bamboo, ..

Example: Continuous Integration with GitLab

- CI/CD processes are called **pipelines**, they are executed by a **runner**
- add **`.gitlab-ci.yml`** to root folder of project with CI instructions
- file format is YML, json-xml like markup language
- example: run Maven build on commit, save test reports for a week

GitLab CI Result Reporting

- website that is created and automatically updated (streamed), e.g. https://gitlab.ecs.vuw.ac.nz/<project>/pipelines
- has links to specific artifacts created during CI
- email notifications about build status

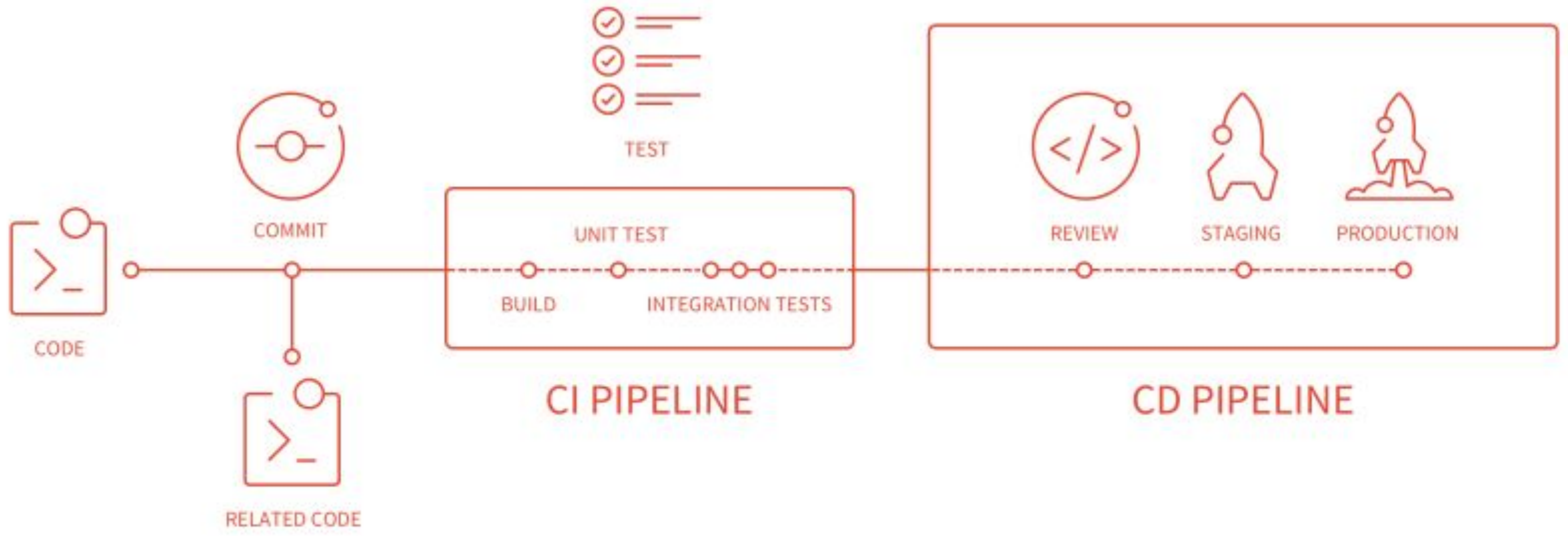
Continuous Delivery

- continuous delivery also automates the process of product delivery
- the pipeline also include releases
- releases are pushed into a **staging system** to prepare them for release into production, perhaps with additional (manual) checks like UI testing and reviews
- this can then be scheduled to release nightly, weekly, etc

Continuous Deployment (CDD)

- continuous deployment is an extreme version of this that skips (automates) staging
- each change (commit) that passes automated quality assurance checks (in particular tests) results in a new product version being delivered and deployed
- deployment can mean:
  - pushing a (new version of) a product into an app store or server
  - and/or adding it to a download page, and notifying clients
- Maven has a built-in CDD feature to push artifacts to the central repository, and additional plugins to deploy web applications on servers, etc

GitLab CI + CD



https://about.gitlab.com/product/continuous-integration/

GitLab CI Example

`.gitlab-ci.yml`

```
mvn-build:
   script:
     - mvn test
   tags:
     - shell
       artifacts:
         paths:
         - target/surefire-reports
         expire_in: 1 week
```

build and run test

keep test reports for one week

GitLab CI Example

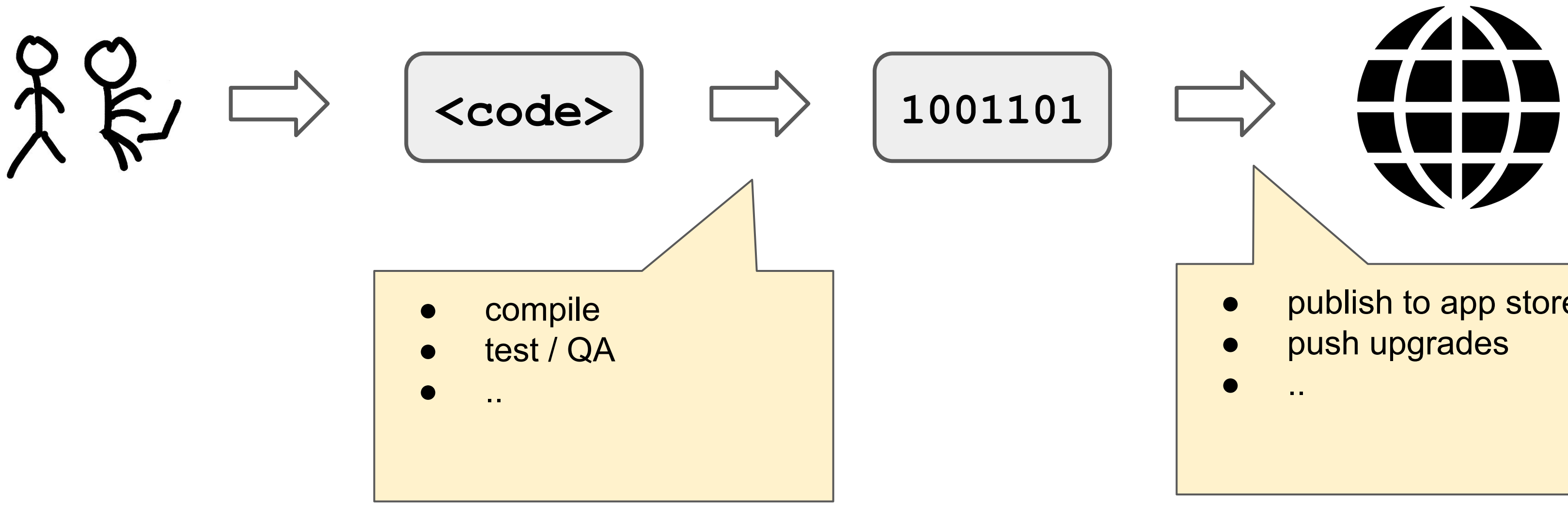
`.gitlab-ci.yml`

```
build:
   script:
     - mvn deploy
     - echo "A new version has been successfully
       released by ${GITLAB_USER_NAME}"  | mail -s
       "release" client@example.com
   tags:
     - shell
```

use mvn to copy to deployment server

example of a custom action to predefined variables

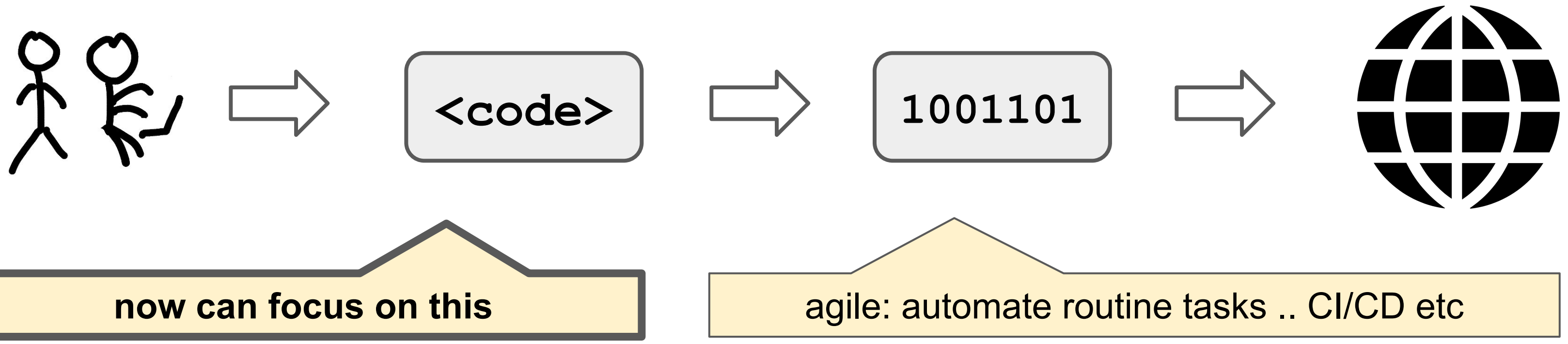
CI/CD High Level



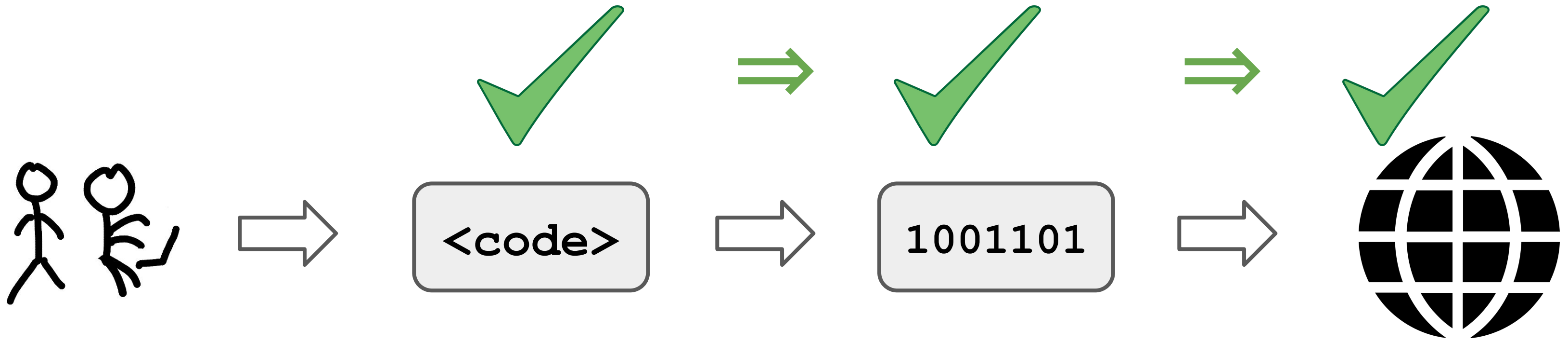
- compile
- test / QA
- ..

- publish to app stores
- push upgrades
- ..

Automation


<code>
1001101
now can focus on this
agile: automate routine tasks .. CI/CD etc

Trusting the Build Infrastructure

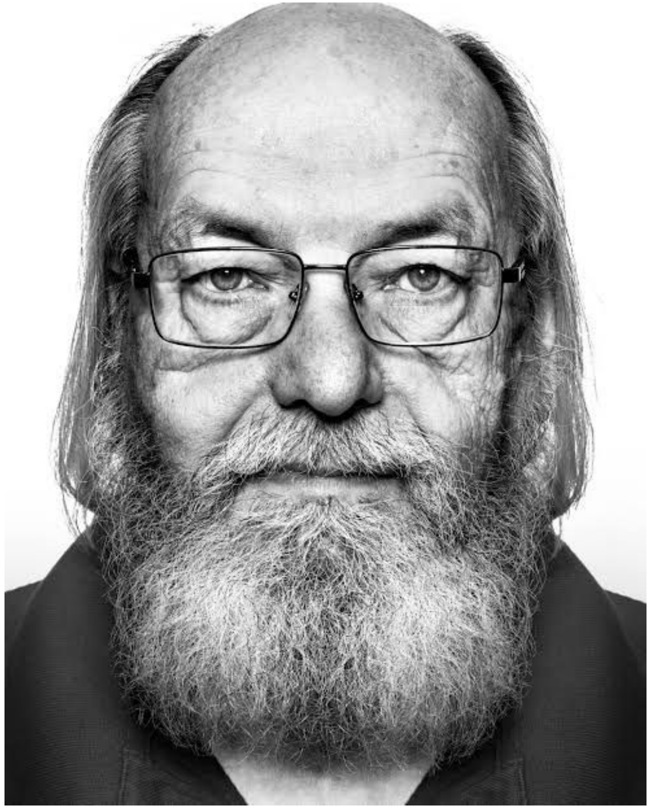


The Ken Thompson Hack (1984)

Compiler injects backdoor into the programs it compiles.

Thompson, Ken. "Reflections on trusting trust."

Communications of the ACM 27.8 (1984): 761-763.



Exploits in the Wild: Solarwinds 2019/20

- Orion product to manage IT resources
- malware watched for builds, inserted a backdoor through a temporary source code file
- attacked DoD, DoJ, Treasury, Cisco, Intel, ..
- executed by Russian Foreign Intelligence Service

https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic

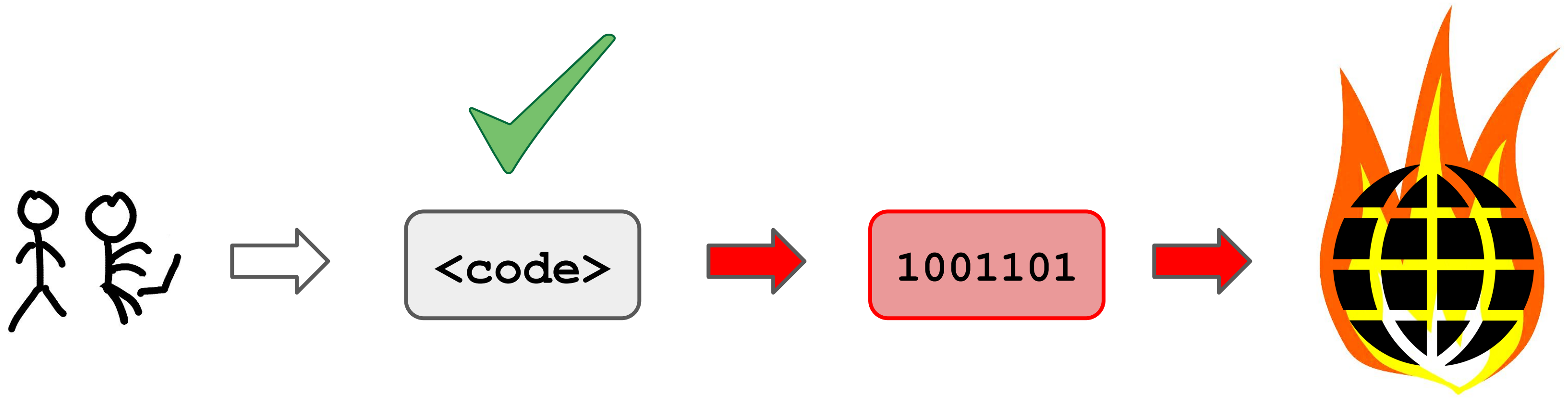
https://kpmg.com/us/en/articles/2023/solarwinds-explainer.html

Exploits in the Wild: Linux XZ - March 24

- open source project XZ – widely used in Linux
- Linux underpins cloud computing and Android
- sophisticated social engineering to take over project from 21
- backdoor inserted by malicious build macro
- disabled some QA tooling to cover up (Google oss-fuzz)

https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/

Trusting the Build Infrastructure



Software Supply Chain Security

- builds should therefore performed in secure environments (e.g. GitHub instead of local PCs) with mature security
- it is is also desirable to collect provenance about the build environment, preferably so that a build can be reproduced to verify the integrity of the original build

Software Supply Chain Security and CI/CD

- several vendors (Google, RedHat, Oracle) now offer repositories with independently build and distributed components from open source
- See https://cloud.google.com/security/products/assured-open-source-software?hl=en
- the SLSA emerging standards defines levels of build maturity to prevent compromised builds