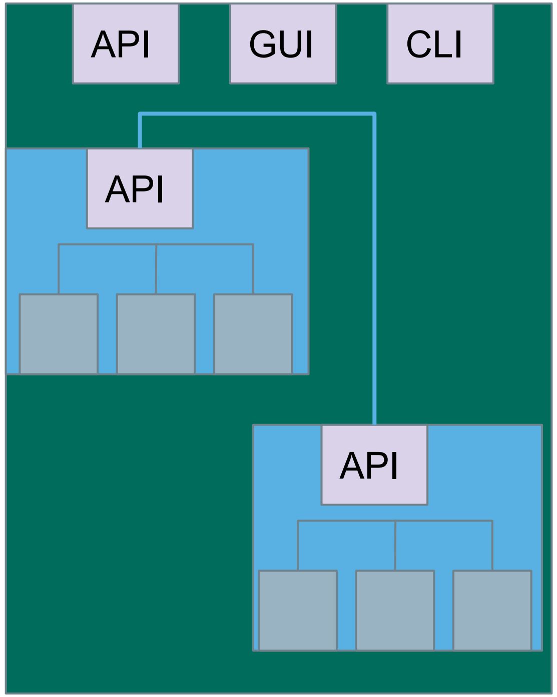
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

# Lecture 7: System (Integration) Testing and Test Automation

Gregory Gay
DIT636/DAT560 - February 9, 2026

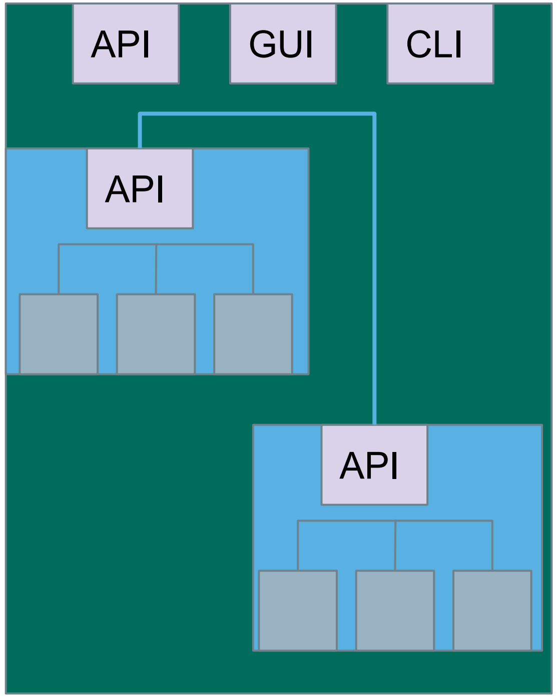
# Testing Stages

- We interact with **systems** through **interfaces**.
  - APIs, GUIs, CLIs
- Systems built from **subsystems**.
  - With their own interfaces.
- Subsystems built from **units**.
  - Communication via method calls.
  - Set of methods is an interface.



## Testing Stages

- **System-level Testing**
  - Tests **whole system** or **independent** subsystems through an **interface**.
  - **Integrates** lower-level components
    - (Subsystem-level) Does unit collaboration work correctly?
    - (System-level) Does subsystem collaboration work correctly?



## Unit vs System Testing

- Unit tests focus on a **single class**.
  - Simple functionality, more freedom.
  - Few method calls.
- System tests **bring many classes together**.
  - Focus on testing through an interface.
  - One interface call triggers many internal calls.
    - Slower test execution.
  - May have complex input and setup.

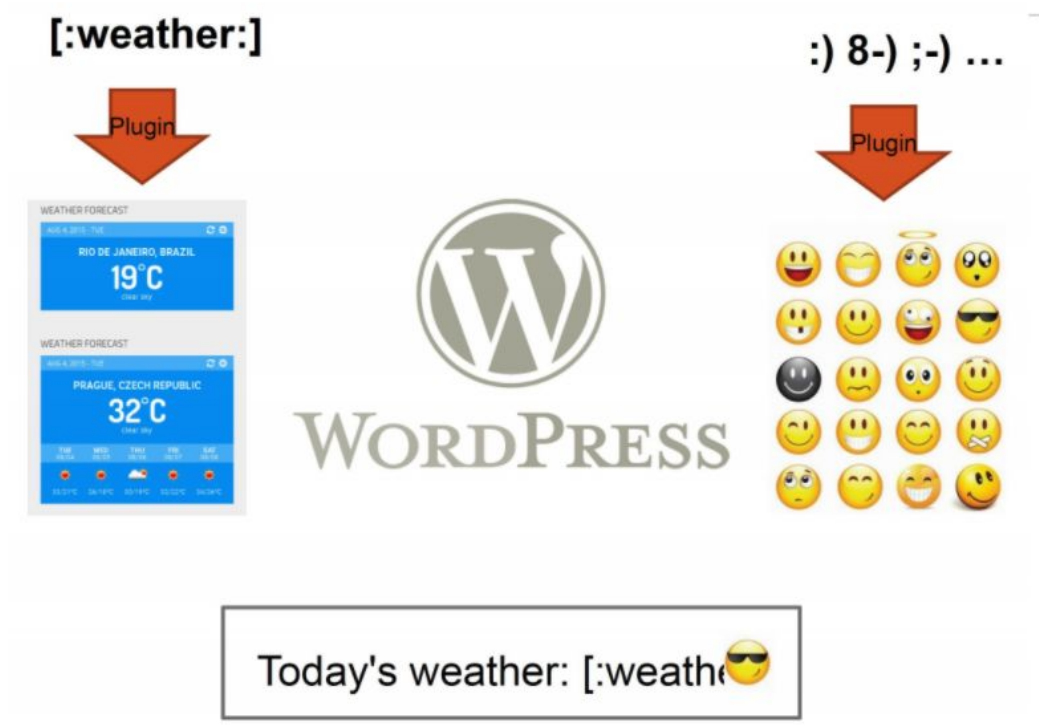
# System Testing

- System components are expected to interact.
  - Usually this is planned!
  - Sometimes unplanned interactions break the system.
  - **We should select tests that thoroughly test component integrations.**

# Fire and Flood Control

- Fire Control activates sprinklers when fire detected.
- Flood Control cuts water supply when water detected on floor.
- **Interaction means building burns down.**

## WordPress Plug-Ins



- Weather and emoji plug-ins tested independently.
- Their interaction results in unexpected behavior.

## Component Interactions

## Interface Types

- Parameter Interfaces
  - Data passed from through method parameters.
  - Subsystem may have interface class that calls into underlying classes.
- Procedural Interfaces
  - Interface surfaces a set of functions that can be called by other components or users (API, CLI, GUI).
  - Integrates lower-level components and controls access.

## Interface Types

- Shared Memory Interfaces
  - A block of memory is shared between (sub)systems.
    - Data placed by one (sub)system and retrieved by another.
  - Common if system architected around data repository.
- Message-Passing Interfaces
  - One (sub)system requests a service by passing a message to another.
    - A return message indicates the results.
  - Common in parallel systems, client-server systems.

## Interface Errors

- Interface Misuse
  - Malformed data, order, number of parameters.
- Interface Misunderstanding
  - Incorrect assumptions made about called component.
  - A binary search called with an unordered array.
- Timing Errors
  - Producer of data and consumer of data access data in the wrong order.

## How to Write System Tests

- As before: choices, representative values, etc.
- If targeting internal code, unit tests can call methods from multiple classes.
- If targeting a dedicated interface:
  - Postman (REST)
  - Selenium (web browser)
  - Bash or Powershell scripts (command line)
  - Espresso (Android)

## Differences from Unit Testing

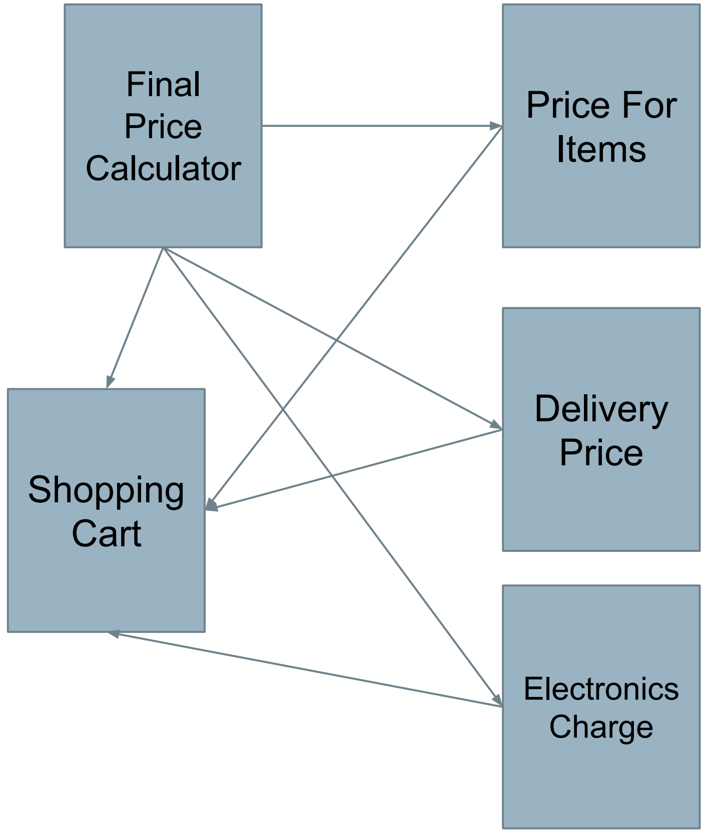
- Test design is more “conceptual”.
  - Based on high-level functionality, not directly traced to low-level code elements.
  - Choices may be tied to multiple code classes.
- Dependencies outside of codebase.
  - May need wrapper code for external dependencies to invoke in test cases that invoke internal code.
- More complex setup and teardown.

## System Testing and Requirements

- Tests can be written early in the project.
    - Can create tests using the requirements.
    - Does not require a detailed design.
- Creating tests supports requirement refinement.
- Tests can be made concrete once code is built.

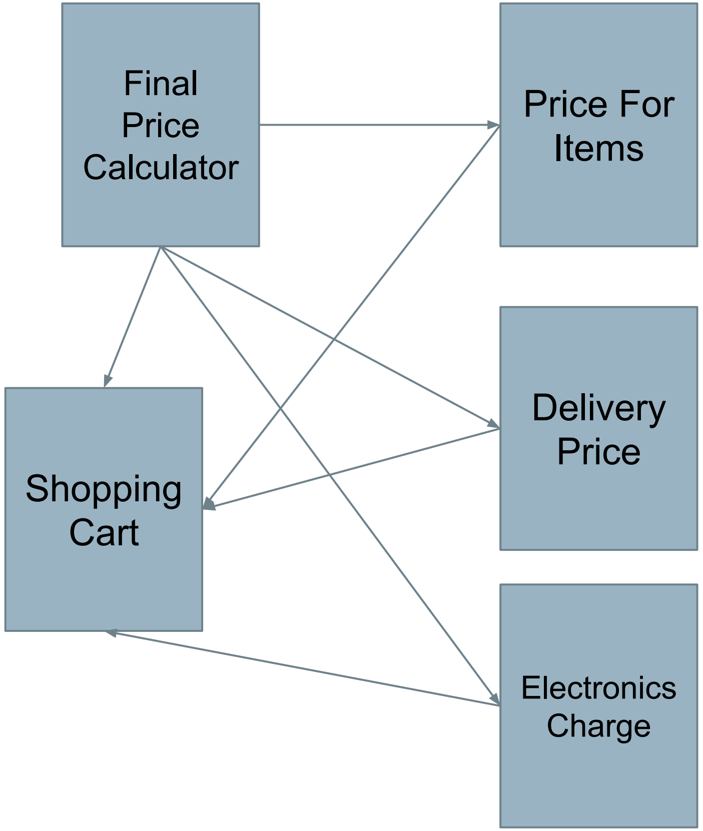
# Example: Shopping

- Final price of each item is calculated as:
  - (price * quantity)
- Delivery costs:
  - 1-3: $5
  - 4-10: $12.5
  - 10+: $20
  - If an item is from electronics category, $7.5 extra.



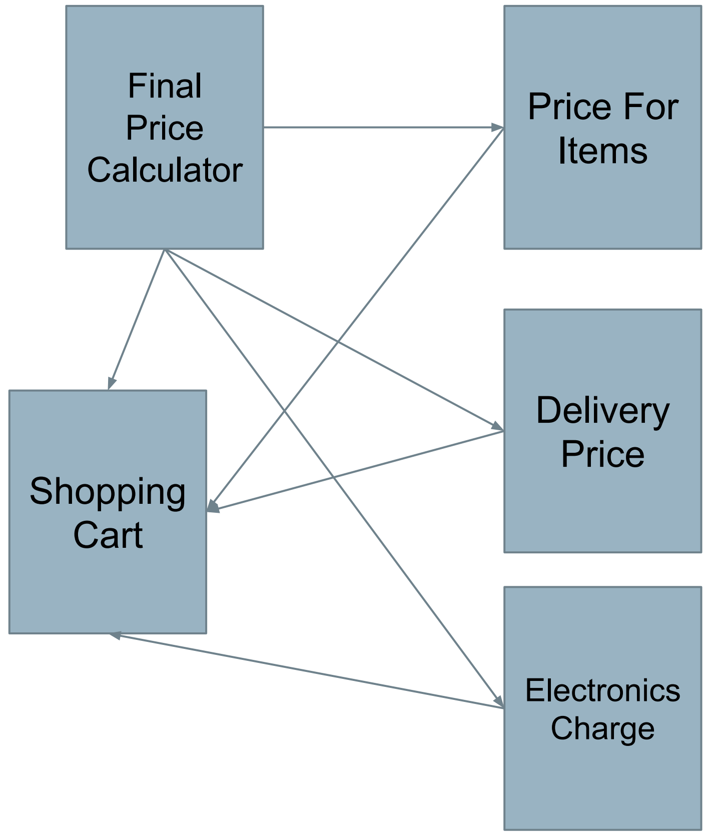
## Example: Shopping

- `ShoppingCart`, "Price Rules" can be tested during unit testing.
- `FinalPriceCalculator` *could* be unit tested, but…
  - ***Should*** be target of integration testing.



## Example: Shopping

- **Choice:** Shopping cart
  - Empty, 1, 2+ items
- **Choice:** Per-item quantities
  - 1, 2+
- **Choice:** Total quantity
  - 1-3 items, 4-10, 10+
- **Choice:** Electronics
  - Yes, no



## User Journeys

- A “business facing” test, designed to simulate **a typical user’s journey** through the system.
  - A user’s entire interaction with a system to achieve some goal - **one path in a use case**.
  - Invokes multiple functions in one test.
- Typical in late stages of system, exploratory testing.
- Also used to demonstrate system to stakeholders.

## Example: Advertising

- Use case: a user wants to create an event and invite appropriate members to it.
- One test case:
  - Create an event.
  - Sort the list of members, filtering by location, age, and gender.
  - Send a message to those users with an invitation to the event.

Message Board
- Events
  - Create
  - Edit
  - Delete
- Get Members
  - Filters
- Messages
  - Compose Message

# Example: Advertising

- Set up database of members.
- Log in.
- Create an event.
- Sort the list of members, filtering by location, age, and gender.
- Send a message to those users with an invitation to the event.
- Check correctness of each step.
- Reset database contents.

# Best Practices

- Implement **separate, reusable** setup and teardown.
  - Pre-testing setup run **before executing the first test**.
    - Restore to this state in teardown of each test.
  - Setup run before **each** test case with a common setup.
  - Simplifies each test case.
  - Easier to maintain.

## Best Practices

- Run each test in a **clean environment**.
    - Resetting database or internal memory.
    - Logging out of accounts.
    - Stopping and restarting system.
- Balance risk against cost of a full restart.
- Can implement a “reset” function in test code.
    - Do not leave in production system.

## Test Automation

# Executing Tests

- How do you run test cases on the program?
  - System level: *could* run code and check results by hand.
  - **Limit how often you do this.**
    - Humans are slow, expensive, and error-prone.
    - **Exception - exploratory and acceptance testing.**
  - Test design requires effort and creativity.
  - Test execution should not.

## Test Automation

- Development of software to separate repetitive tasks from creative aspects of testing.
- Control over ***how*** and ***when*** tests are executed.
  - Control environment and preconditions/setup.
  - Automatic comparison of predicted and actual output.
  - Automatic hands-free re-execution of tests.

## Testing Requires Writing Code

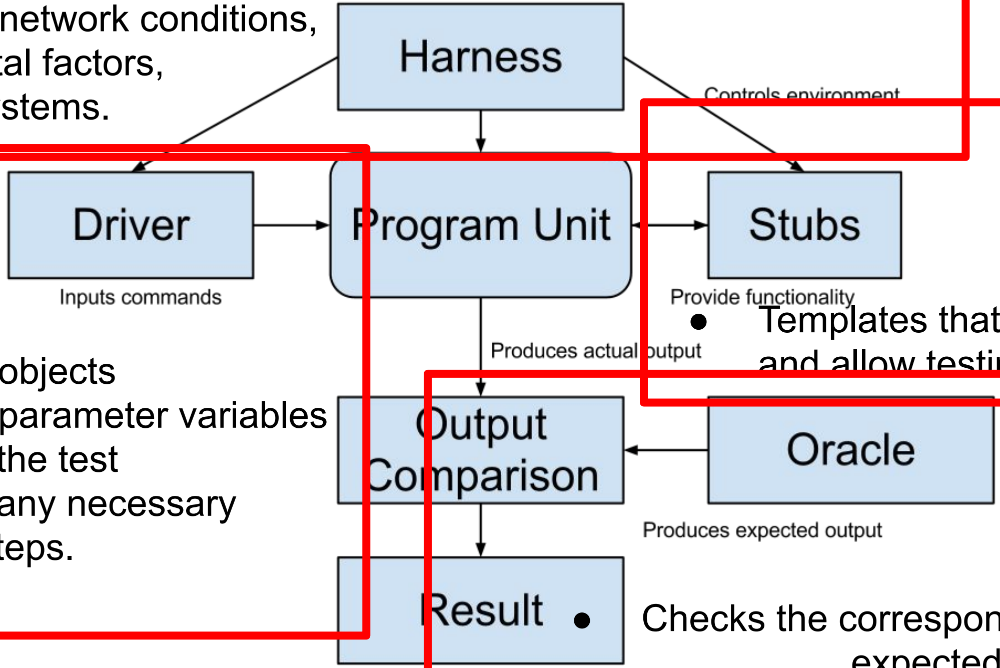
- The component to be tested must be isolated and ***driven*** using method or interface calls.
- Untested dependencies must be ***mocked*** with reliable substitutions.
- The deployment environment must be simulated by a controllable ***harness***.

## Test Scaffolding

- Test scaffolding is a set of programs written to support test automation.
  - Not part of the product, often temporary
- Allows for:
  - Testing before all components complete.
  - Testing independent components.
  - Control over testing environment.

## Test Scaffolding

- Simulates the execution environment.
- Can control network conditions, environmental factors, operating systems.



- Templates that provide functionality and allow testing in isolation.

- Initializes objects
- Initializes parameter variables
- Performs the test
- Performs any necessary cleanup steps.

- Checks the correspondence between the produced and expected output and renders a test verdict.

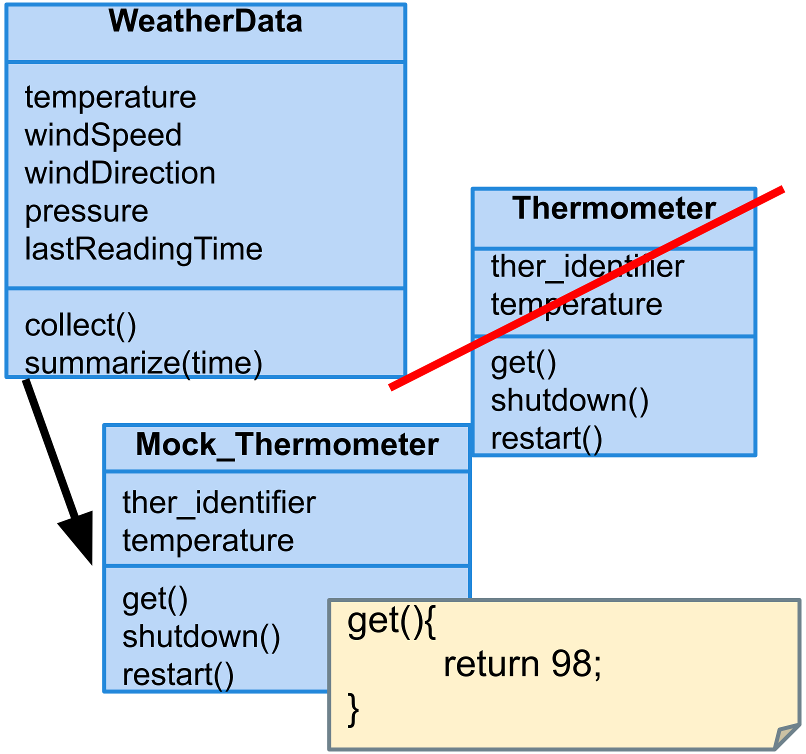
## Scaffolding

- Mock objects and drivers are written as replacements for other parts of the system.
  - May be required if pieces of the system do not exist.
- Scaffolding allows control over test execution and greater observability to judge test results.
  - Simulate dependencies and test components in isolation.
  - Ability to set up specialized testing scenarios.
  - Ability to replace part of the program with a version more suited to testing.

# Unit Testing - Object Mocking

Unit may depend on unfinished (or untested) components. Can **mock** those components.

- Same interface as real component, but hand-created simulation.
- Can be used to simulate abnormal operation or rare events.
  - Ex. Place exact data in database needed to hit special outcome.



```
get(){
        return 98;
}
```

## Mocking Example

- Declare a mock object:

```
LinkedList mList = mock(LinkedList.class);
```

- Specify method behavior:

```
when(mList.get(0)).thenReturn(“first”);
```

  - Returns “first”: `mList.get(0);`
  - Returns null: `mList.get(99);`
    - Because behavior for “99” is not specified.

```
when(mList.get(anyInt()).thenReturn(“element”);
```

  - `mList.get(0)`, `mList.get(99)` both return “element”, as all input are specified.

# Mocking Within a Test

```
@test
public void temperatureTest(){
    Thermometer mockTherm = mock(Thermometer.class);
    when(mockTherm.get()).thenReturn(98);
    WeatherData wData = new WeatherData();
    wData.collect(mockTherm);
    assertEquals(98, wData.temperature);
}
```

## Let’s take a break.

## Build Systems

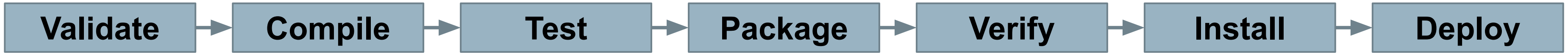
## Build Systems

- Building, running tests, packaging and distributing are very common, effort-intensive tasks.
  - Building and deploying should be as easy as possible.
- **Build systems** ease process by automating as much as possible.
  - Repetitive tasks can be automated and run at-will.

## Build Systems

- Allow control over code compilation, test execution, executable packaging, and deployment.
- Script defines actions that can be automatically invoked at any time.
- Many frameworks for build scripting.
    - Most popular for Java: Ant, Maven, Gradle.
    - Gradle very common for Android projects.

## Build Lifecycle



- **Validate** the project is correct and all necessary information is available
- **Compile** the source code of the project.
- **Test** the source code using a suitable unit testing framework.
    - **Unit tests** and **subsystem integration tests**.
- Take the compiled code and **package** it in its distributable format, such as a JAR.

# Build Lifecycle

$$\begin{tikzcd}\boxed{\text{Validate}} \arrow[r] & \boxed{\text{Compile}} \arrow[r] & \boxed{\text{Test}} \arrow[r] & \boxed{\text{Package}} \arrow[r] & \boxed{\text{Verify}} \arrow[r] & \boxed{\text{Install}} \arrow[r] & \boxed{\text{Deploy}}\end{tikzcd}$$

- **Verify** - run **system tests**.
  - System tests require a packaged executable.
  - This is also when tests of non-functional criteria like performance are executed.
- **Install** the package for use as a dependency in other projects locally.
- **Deploy** the package to production environment.

## Apache Ant

- Simple build system for Java.
- Build scripts define **targets** that can be executed on command.
  - Correspond to lifecycle phases or other automated tasks.
  - Targets can trigger other targets.
  - Build scripts written in XML.
    - Platform neutral, But can invoke platform-specific commands.
    - Human and machine readable.
    - Created automatically by many IDEs (Eclipse).

## A Basic Build Script

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

- File typically named **build.xml**, and placed in the base directory of the project.
- Build script requires **project** element and at least one **target**.
  - Project defines a **name** and a default **target**.
  - This target prints project information.
    - **Echo** prints information to the terminal.

## Targets

```
<target name = "deploy" depends = "package"> .... </target>
<target name = "package" depends = "clean,compile"> .... </target>
<target name = "clean" > .... </target>
<target name = "compile" > .... </target>
```

- A target is a collection of tasks you want to run in a single unit.
  - Targets can depend on other targets.
  - Dependencies denoted using the **depends** attribute.
    - **deploy** will call **package**, which will call **clean** and **compile**.

## Targets

```
<target name = "deploy" depends = "package"> .... </target>
<target name = "package" depends = "clean,compile"> .... </target>
<target name = "clean" > .... </target>
<target name = "compile" > .... </target>
```

- Target attributes:
  - **name** defines the name of the target (required)
  - **depends** lists dependencies of the target.
  - **description** is used to describe the target.
  - **if** and **unless** allow execution of the target to depend on a conditional attribute.
    - Execute target **if** attribute is true, or execute **unless** true.

## Executing targets

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

```
Buildfile: build.xml
info: [echo] Hello World - Welcome to Apache
Ant!
BUILD SUCCESSFUL
Total time: 0 seconds
```

- In the command line, invoke:
  - **ant <target name>**
- If no target is supplied, the default will be executed.
  - In this case, **ant** and **ant info** give same result because info is default target.

# Properties

- XML does not natively allow variable declaration.
  - Instead, create **property** elements, which can be referred to by name.

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property name = "sitename" value = "http://cse.sc.edu"/>
   <target name = "info">
      <echo>Apache Ant version is ${ant.version} - You are at ${sitename} </echo>
   </target>
</project>
```

## Properties

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property name = "sitename" value = "http://cse.sc.edu"/>
   <target name = "info">
      <echo>Apache Ant version is ${ant.version} - You are at ${sitename} </echo>
   </target>
</project>
```

- Properties have a name and a value.
  - Property value is referred to as **${property name}**.
  - Ant pre-defines **ant.version**, **ant.file** (location of the build file), **ant.project.name**, **ant.project.default-target**.

## Property Files

- Can define static properties in a file.
  - Allows reuse of build file in different environments (development, testing, production).
  - Allows easy lookup of property values.
- Called **build.properties** and stored in the same directory as build script.
  - Lists one property per line: `<name> = <value>`
  - Comments can be added using `# <comment>`

## Property Files

- build.xml

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property file = "build.properties"/>
   <target name = "info">
      <echo>You are at ${sitename}, version ${buildversion}.</echo>
   </target>
</project>
```

- build.properties

```
# The Site Name
sitename = http://cse.sc.edu
buildversion = 3.3.2
```

# Conditions

```
<target name = "myTarget" depends =
"myTarget.check" if =
"myTarget.run"> .... </target>
<target name = "myTarget.check">
    <condition property =
"myTarget.run">
        <and>
            <available file =
"foo.txt"/>
            <available file =
"bar.txt"/>
        </and>
    </condition>
</target>
```

- Property value determined by **and/or**.
  - **And** requires that each property is true.
    - foo.txt and bar.txt must exist.
    - (**available** checks for file existence)
  - **Or** requires that 1+ properties true.
    - Calling **myTarget.check** creates property (**myTarget.run**), true if both files present.
    - When **myTarget** called, it will run only if myTarget.run is true.

## Ant Utilities

- **Fileset** generates list of files matching criteria for inclusion or exclusion.
  - ** means that the file can be in any subdirectory.
  - * allows partial file name matches.

```
<fileset dir = "${src}" casesensitive = "yes">
   <include name = "**/*.java"/>
   <exclude name = "**/*Stub*"/>
</fileset>
```

## Ant Utilities

- **Path** is used to represent a classpath.
  - **pathelement** is used to add items or other paths to the path.

```
<path id = "build.classpath.jar">
   <pathelement path = "${env.J2EE_HOME}/j2ee.jar"/>
   <fileset dir = "lib"> <include name = "**/*.jar"/> </fileset>
</path>
```

## Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">
   <property name = "src.dir" value = "src"/>
   <property name = "build.dir" value = "target"/>
   <path id = "master-classpath">
      <fileset dir = "${src.dir}/lib"> <include name = "*.jar"/> </fileset>
      <pathelement path = "${build.dir}"/>
   </path>
</project>
```

- Properties **src.dir** and **build.dir** define where the source files are stored and where the built classes are deployed.
- Path **master-classpath** includes all JAR files in the lib folder and all files in the build.dir folder.

## Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">

   <target name = "clean" description = "Clean output directories">
      <delete>
         <fileset dir = "${build.dir}">
            <include name = "**/*.class"/>
         </fileset>
      </delete>
   </target>
</project>
```

- The clean target is used to prepare for the build process by cleaning up any remnants of previous builds.
  - In this case, it deletes all compiled files (.class)
  - May also remove JAR files or other temporary artifacts that will be regenerated by the build.

## Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">

    <target name = "build" description = "Compile source tree java files">
      <mkdir dir = "${build.dir}"/>
      <javac destdir = "${build.dir}" source = "1.8" target = "1.8">
         <src path = "${src.dir}"/>
         <classpath refid = "master-classpath"/>
      </javac>
   </target>

</project>
```

- The build target will create the build directory, compile the source code (using javac), and place the class files in the build directory.
  - Can specify which java version to target (1.8).
  - Must reference the classpath to use during compilation.

## Creating a JAR File

- The **jar** command creates executable from compiled classes.

```
<target name = "package">
    <jar destfile = "lib/util.jar" basedir = "${build.dir}/classes"
        includes = "app/util/**" excludes = "**/Test.class">
    <manifest><attribute name = "Main-Class" value = "com.util.Util"/></manifest>
</jar>
</target>
```

- **destfile** is the location to place the JAR file.
- **basedir** is the base directory of included files.
- **includes** defines the files to include in the JAR.
- **excludes** prevents certain files from being added.
- The **manifest** declares metadata about the JAR.
    - Attribute Main-Class makes the JAR executable.

## Running Unit Tests

- JUnit tests run using the **junit** command.

```
<target name = "test">
   <junit haltonfailure = "true" haltonerror = "false"
          printsummary = "true" timeout = "5000">
      <test name = "com.utils.UtilsTest"/>
   </junit>
</target>
```

  - **test** entries list the test classes to execute.
  - **haltonfailure** / **haltonerror** will stop execution if tests fail/errors occur.
  - **printsummary** displays number of tests run, number of failures/errors, time elapsed.
  - **timeout** will stop and issue error if the time limit is exceeded.

## We Have Learned

- During system testing, we focus on interactions.
  - Test by calling methods or through an interface.
  - If thoroughly unit tested, failures due to interaction faults.
    - Mistaken assumptions, malformed calls.
  - Tests can focus on one “high-level” function or model full user journeys.

## We Have Learned

- Test automation can lower cost and improve the quality of testing.
- Automation involves creating drivers, harnesses, stubs, and oracles.
  - Test cases written as executable code.
  - Additional support code (mocking, interface manipulation, wrappers for external dependencies) to enable testing.

## We Have Learned

- Testing is not all that can be automated.
  - Project compilation, installation, deployment, etc.
- Project build automation:
  - Automating the entire compilation, testing, and deployment process.
  - Ant is an XML-based tool for automating build process.

## Next Time

- Exploratory Testing
- Exercise Session: Unit Testing
  - (Follow instructions linked on Canvas to set up IDE)

- Assignment 2 due Feb 15.

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY