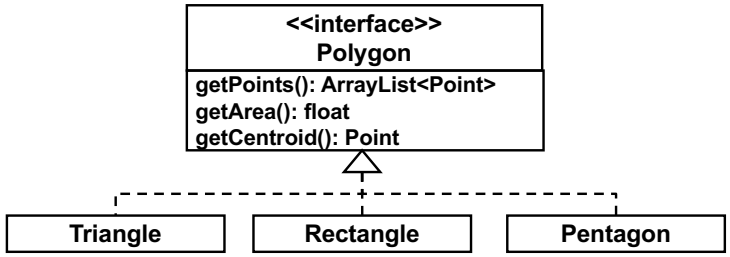
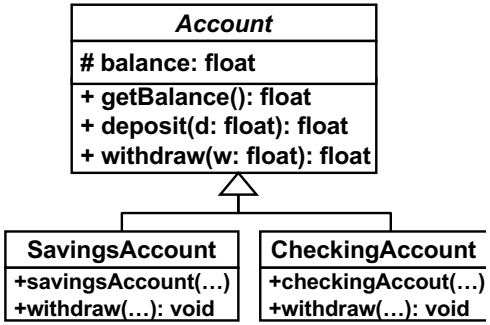
Boundary Testing

- Testing with the extremes (or edges) of the input domain as well as typical values
  - Maximum, minimum, outside boundaries and error values
  - Empty or missing values such as 0, “” and null
  - Division by 0
  - Numerical overflows
- Many (yes MANY) defects come out from these corner cases.

- What if 3D points are passed when 2D points are expected, and vice versa?
- Value range for each axis?
  - Max and min values?
    - What type is used for the values?
  - Positive and negative values?
  - Only positive values?

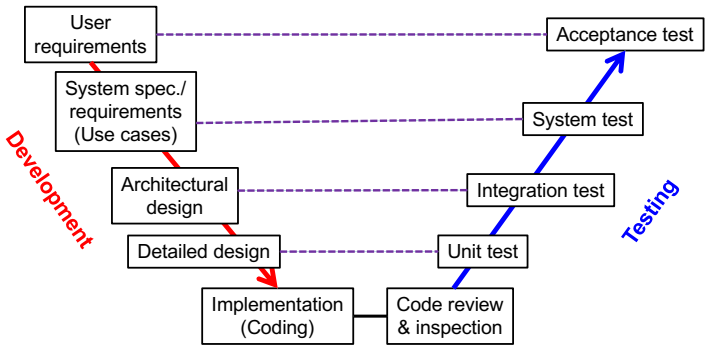


- What is the value range for a balance?
  - Max and min values?
    - Min for a savings account
    - FDIC insurance limit
  - A negative value is allowed?

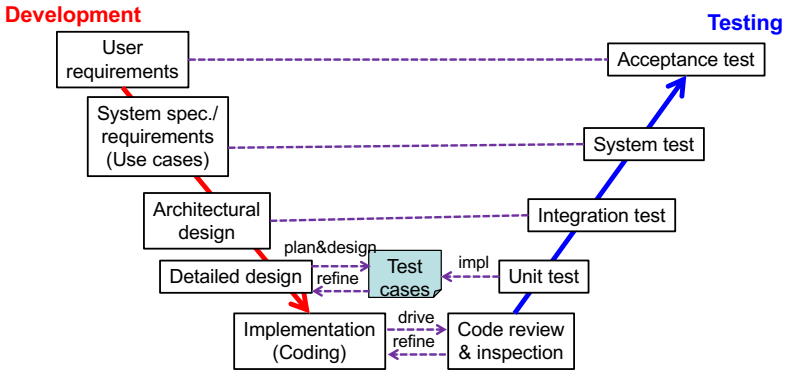


V-Model and Development Process

- Explicitly states which testing phase corresponds to which development phase.
- V-Model can be used to define various iterative development process, beyond waterfall process



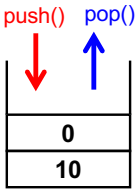
Test Cases in V-Model



Does Planning/Designing Test Cases Help Refine Class Design?

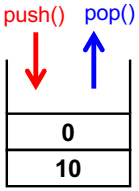
- Assume you are required to implement a stack.
- Detailed design (class design)
  - What's the specification for the Stack class?
    - Stack of int values

Stack
+ pop(): int + push(val: int): void



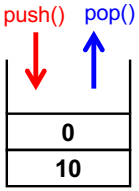
- pop():
  - What kind of unit tests need to be carried out for pop()?
    - Can pop() get the value at the top?
    - What if the stack is empty?
      - Throws an exception
        - » StackUnderflowException

Stack
+ pop(): int {throws StackUnderFlowException} + push(val: int): void

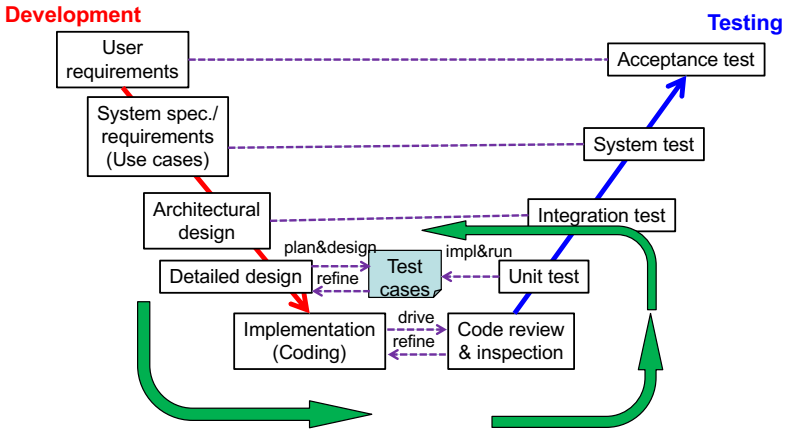


- push():
  - What kind of unit tests need to be carried out for push()?
    - Can push() add a value to the top of this stack?
    - Do we have a limit for the number of values in this stack?
      - int[], ArrayList<Integer>, etc.?
    - What if the stack is full?
      - Throws an exception
        - » StackOverflowException

Stack
- values: int[]
+ Stack(maxSize:int) + pop(): int {throws StackUnderFlowException} + push(val: int): void {throws StackOverFlowException}



Write and Test the Stack Class



Unit Testing for Stack

- Test each of the 3 methods in Stack.
  - Boundary tests for the constructor
    - What if 0, null or a negative value is passed?
  - Boundary tests for pop() and push()
    - Positive and negative tests
  - givenEmptyStackWhenPushingAValueThenStackSizeBecomes1
    - Or, pushingAValueIncrementsStackSize
  - givenFullStackWhenPushingAValueThenStackOverflowExceptionOccurs
    - Or, stackOverflowExceptionOccursWhenPushingAValueToFullStack
  - …etc. etc.

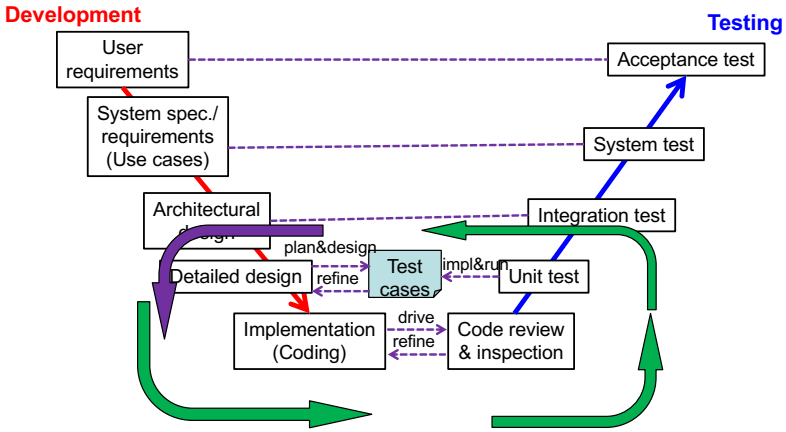
Stack
- values: int[] **- maxSize: int**
+ Stack(maxSize:int) **+ pop(): int** **{throws StackUnderFlowException}** **+ push(val: int): void** **{throws StackOverFlowException}**

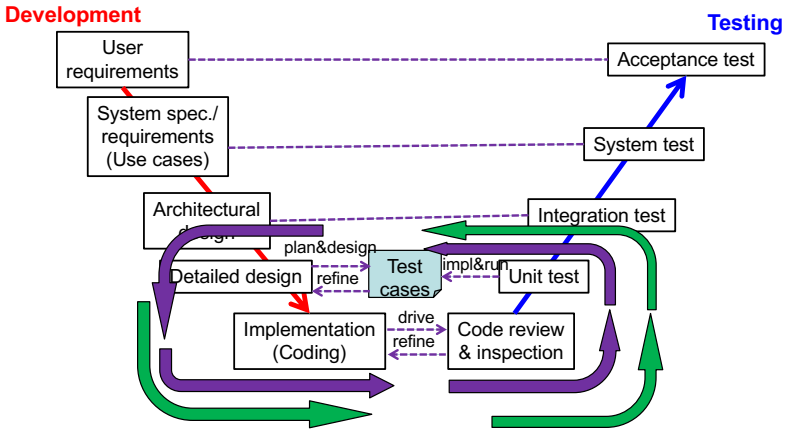
Does Writing/Running Test Cases Help Refine Class Design?

- Do we want size()?
  - Yes!
- Unit testing can trigger design improvement.

Stack
- values: int[] **- maxSize: int**
+ Stack(maxSize:int) **+ pop(): int** **{throws StackUnderFlowException}** **+ push(val: int): void** **{throws StackOverFlowException}** **+ size(): int**

Updating the Design of Stack



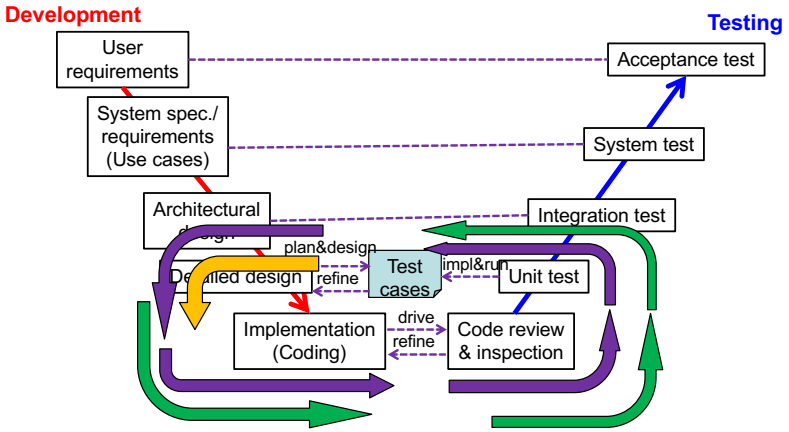


Unit Testing for an Updated Version of Stack

- Write and run a new test case for size().
- Run ALL existing test cases as well
  - to make sure that adding size() does not inject bugs
- Code→test→code→test

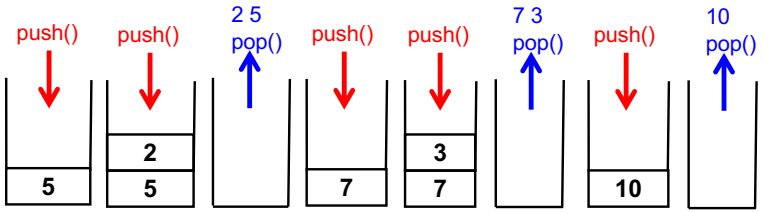
Stack
- values: int[] - maxSize: int
+ Stack(maxSize:int) + pop(): int {throws StackUnderFlowException} + push(val: int): void {throws StackOverFlowException} + size(): int

Write an Application of Stack



RPNCalculator

- Performs numerical calculation based on the reverse Polish notation
  - e.g., 5 + 2 - 3 → 5 2 + 3 –
- Reads operands and operators one by one from the left.
  - Pushes a number to the stack.
  - Upon the arrival of an operator, pops two numbers from the stack, apply the operator on the two numbers and push the result to the stack.
  - Pops a number when no operands and operators remain.



- RPNCalculator
  - Utility class
  - A static method to receive a RPN-based equation as a string.
    - Pops a number from the stack when it has processed all operands and operators.
    - Makes sure that the stack is empty.
      - `if(stack.size()!=0){ Something went wrong! }`

- This empty-or-not check should be common in many client code of Stack.
  - Do we want isEmpty()? Yes!
    - `if(stack.isEmpty()){Something went wrong!}`

      is better (more readable and less error-prone) than

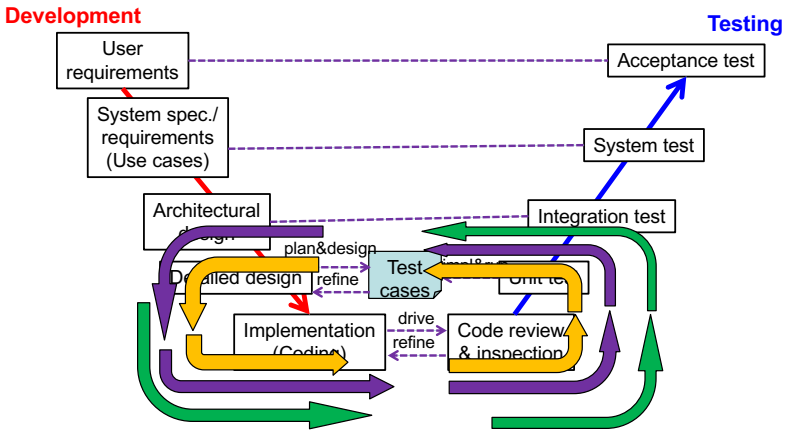
      `if(stack.size()!=0){Something went wrong!}`

Unit Testing for an Updated Version of Stack

- Write and run a new test case for isEmpty().
- Run ALL existing test cases as well
  - to make sure that adding isEmpty() does not inject bugs

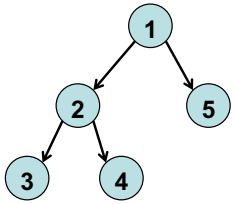
- Code→test→code→test→code→test

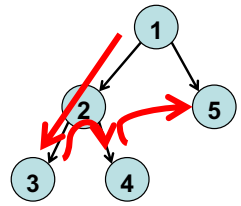
Stack
- values: int[] - maxSize: int
+ Stack(maxSize:int) + pop(): int {throws StackUnderFlowException} + push(val: int): void {throws StackOverFlowException} + size(): int + isEmpty(): boolean



Write Another Application of Stack

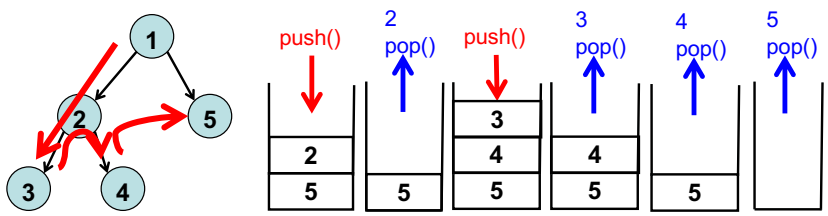
- Use a stack to traverse a tree or graph
  - Tree traversal
    - Required to implement an AI engine for strategy games (e.g. chess)
  - Graph traversal
    - Required to implement navigation applications.
  - Two major traversal algorithms
    - Depth-first and width-first





- A Stack is useful to record the nodes that have been discovered but not visited yet.

- To implement a depth-first tree traversal…
  - Find a child node(s) under the current node.
    - If no children are found, pop a node to move to the node.
  - Push all found nodes. Pop a node to move to the node.



- To decide the next node to visit,
  - you want to know the node at the top in the stack.
- But, you may want to remove the node from the stack after you actually visit it.
  - If an error (or an event) occurs during the move, you may want to abort the move, go back to the last node you have visited, and re-try the move again later.
- pop() does not address this requirement. Another method is necessary.
  - peek(): int

Unit Testing for an Updated Version of Stack

- Write and run a new test case for peek().
- Run ALL existing test cases as well
  - to make sure that adding peek() does not inject bugs
- Code→test→code→test →code→test→code→test
- Finish coding the depth-first algorithm and test it.
- Code→test→code→ test→code→test→ code→test→code→test

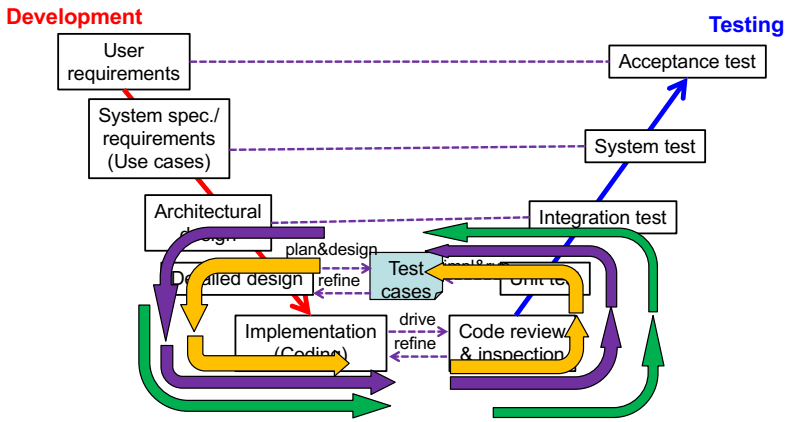
Stack
- values: int[] - maxSize: int
+ Stack(maxSize:int) + pop(): int {throws StackUnderFlowException} + push(val: int): void {throws StackOverFlowException} + peek(): int {throws StackUnderFlowException} + size(): int + isEmpty(): boolean

Continuous (Unit) Testing

- Test your code *early*, *automatically* and *repeatedly*.
  - To maximize the benefits of unit testing.
- Early testing
  - You as a programmer do coding and unit testing at the same time.
- Automated testing
  - Run ALL test cases in an automated way.
    - Never think of selecting and running test cases by hand.
- Repeated testing
  - Run ALL test cases whenever changes are made in the code base.

Continuous Testing and Iterative Development Process

- Continuous testing can be a key to drive iterative (and incremental) development.

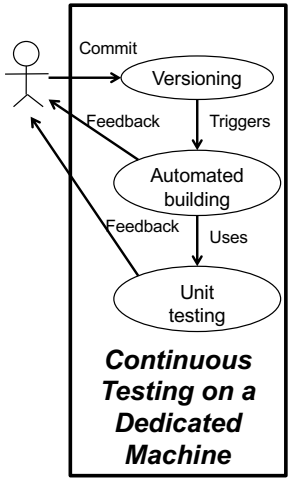


Continuous Testing

- Not everyone is always motivated for testing.
  - Every single test case needs to be executed again and again.
    - If test cases are not executed again and again, you waste your time and effort to write them.
    - If they are executed again and again, the cost you spent will be paid off gradually over time.
- Need a mechanism to drive continuous testing
  - Automatically run tests.
    - Not manually, effortlessly.
  - When and how often?
    - Once code is checked into a shared repository
    - Periodically (e.g. every night)
  - Where?
    - On a dedicated machine (test environment), NOT on developers' machines.

3 Key Elements for Continuous Testing

- Unit testing
  - Regression testing through repeated unit testing
- Automated building
  - Automate the *entire* build process
    - e.g., compilation of all source code, unit testing, coverage measurement, other testing (if any), packaging, deployment and Javadoc generation.
- Versioning
  - When a regression is found upon a change in the code base, you need to know the difference/delta in b/w the current version and a previous version that has passed all unit tests.
    - The regression comes out from that delta.



Automated Building

- Build tools: Ant, Maven, Gradle, etc.
- Ant
  - Basic build tool
    - Sufficient for solo projects and small-scale team projects.
    - Need to write up a project specification (build.xml) from scratch
    - Need to download all external libraries and set them up (e.g., specify the build paths and environmental variables for those libraries).
      - Need to be careful about library-to-library depenency
      - It could take a half day or even a whole day to just set up your development environment.

- Maven
  - Provides basic Ant-like features, plus extra useful features
    - Project templates
      - No need to always write up a project specification from scratch
    - Library management
      - No need to download and set up external libraries. Just specify the libraries you want to use. They will be automatically downloaded.
        - » No need to download a library when it's newer version is available. Just specify a version number you want to use.
      - No need to check library-to-library dependency. Required libraries will be automatically downloaded.
    - It can be very quick to set up your development environment.
    - A large number of plug-ins are available.
      - Good extensibility. Easy to use plug-ins.
      - Not easy to modify existing plug-ins. Not that straightforward to make a new ones.
    - Integration with JUnit, JaCoCo, Covertura, etc.
    - Concurrent/parallel builds
    - Potential "Death by POM (Project Object Model)" or "Death by XML"!
      - Need a POM meister in a large-scale project

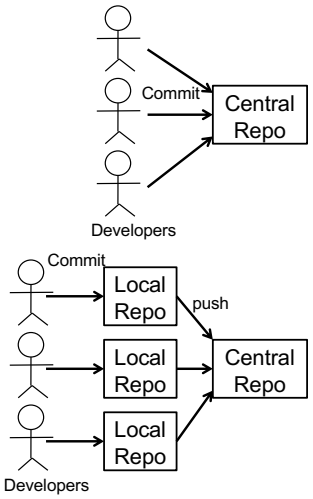
- Gradle
  - Provide basic Ant-like features, plus extra useful features
  - Library management
  - Groovy-based project specifications
    - You can program whatever you want to do in Groovy.
      - No need to find necessary plug-ins.

```
» task HelloWorld{
    doLast{
      println 'Hello World'
    }
  }
```

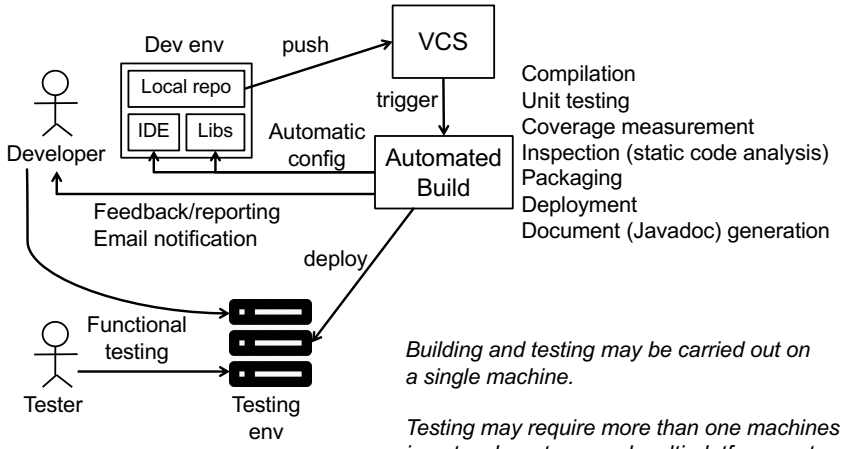
  - Ant migration
    - Can read an Ant scripts (e.g. build.xml) and run it in Gradle.

Versioning w/ Version Ctrl Systems

- Fully centralized: CVS and Subversion
  - Commit code to the central repository.
  - *Fine-grained* (e.g., per-class or per-method) commits are required to avoid potential conflicts among different commits by different developers
  - Unit/regression testing per commit
  - Too many conflicts could occur in each commit.
- Distributed: Git and Mercurial
  - Commit code to the local repository first and then push it to the central repository
  - *Coarse-grained* (e.g., per-feature/functionality or per-use-case) pushes
  - Unit/regression testing per commit to the local repo and per push to the central repo



Continuous Testing



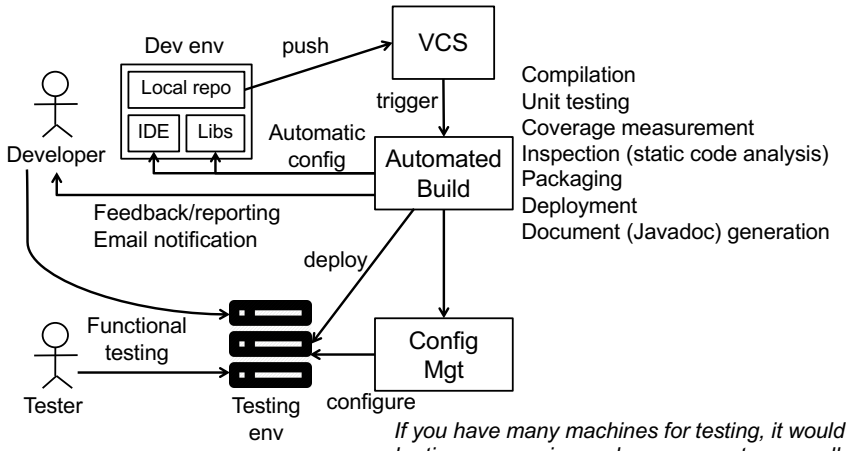
Building and testing may be carried out on a single machine.

Testing may require more than one machines in network systems and multi-platform systems.

Automated Deployment

- Fabric
  - http://www.fabfile.org/
- Capistrano
  - http://capistranorb.com/
- Rundeck
  - http://rundeck.org/
- jclouds
  - https://jclouds.apache.org/
  - VM management for major clouds
    - Amazon EC2, Azure, OpenStack, etc.
  - Automates staring multiple VMs at once and installing software on them.

Continuous Testing w/ Config Mgt



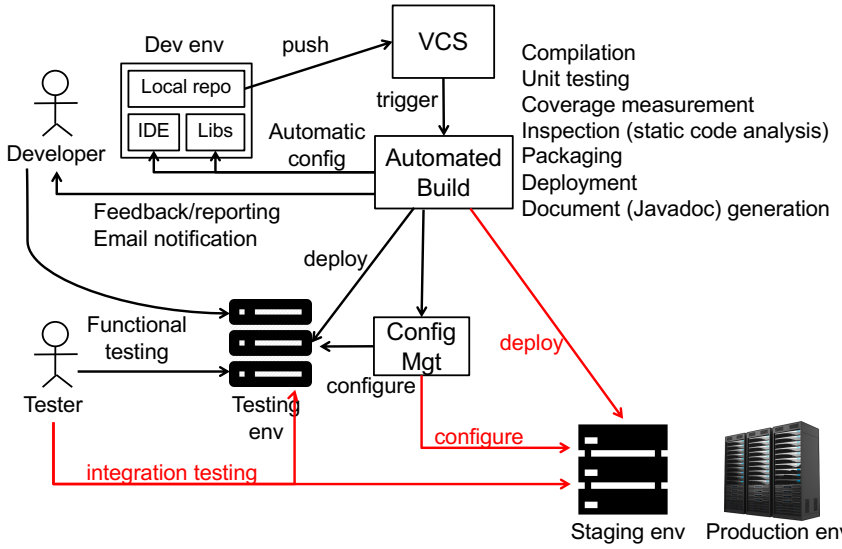
If you have many machines for testing, it would be time-consuming and error-prone to manually configure the machines with particular settings.

A configuration mgt tool can be useful.

Configuration Mgt Tools

- a.k.a. System management tools
- Automate system configurations (e.g., OS settings, app installations) for given machines.
  - May apply the identical configurations to all of the machines
  - May apply different configurations to different machines.
- Open-source tools
  - Puppet
    - http://puppetlabs.com/
    - Jason-like DSL (domain specific language)
  - Chef
    - https://www.getchef.com/
    - Ruby-based DSL

Continuous Integration (CI)



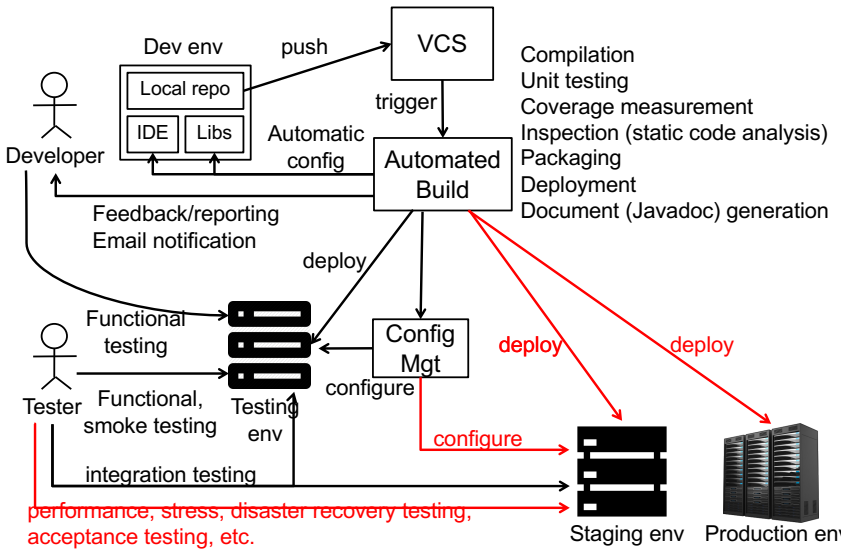
Integration Testing

- Testing the integration of multiple modules/packages.
  - e.g. An output from a module goes to the other module correctly?
  - A module's task starts after the other's task is completed?
- GUI testing
  - e.g., Selenium
    - Automates Web browsing and Web app operations
      - » Page loading, page transitions, data entries, etc.
    - Test cases can be written in HTML, Java, Python, Ruby, C#, etc.
    - Selenium IDE can partially generate test cases.
  - e.g., Android SDK testing framework
    - Automates user activities on the Android emulator
      - » Clicking and other finger gestures, data entries, etc.
- Request-response testing for web apps
  - e.g. AsyncHttpClient

Jenkins: a CI Tool

- Jenkins (http://jenkins-ci.org/)
  - Implemented as a Servlet app
    - Cron-like (timer-based) periodic build and integration testing
  - Advanced GUI-based reporting
  - Not only for Java-based development
    - C/C++, Ruby, Python, JavaScript, etc.
  - Automated process for continuous testing
    - Build with Ant, Maven, Gradle, etc.
    - Unit testing with Junit
    - Coverage measurement with JaCoCo, Cobertura, etc.
    - Code inspection with Findbugs, PMD, CheckStyle, Coverity, etc.
    - App deployment to Tomcat, other Servlet engines, clouds
  - Automated integration testing with Selenium, etc.

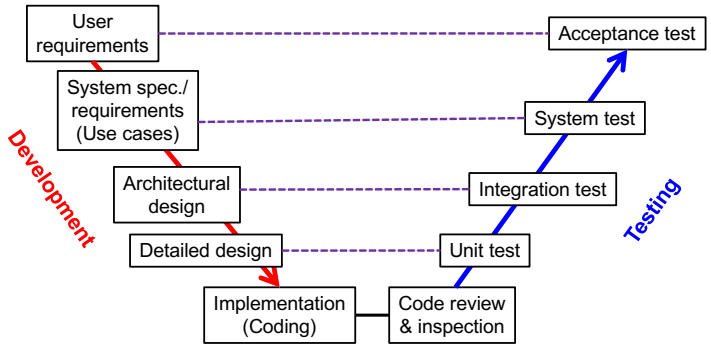
Continuous Delivery



- Automated release/delivery
  - Release any versions of the product
    - Testers can verify the changes between different versions
    - Support engineers can deploy the product on a testing/staging environment and see if a reported defect is reproducible.
    - System operators can deploy the product on an operational environment to perform disaster recovery testing, etc.
    - Customers can be satisfied
  - No more manual procedures for releases
    - Procedure documents
      - Copy this step-by-step time consuming
  - Can rollback to a previous version immediately when critical issues are found in the current version.

V-Model and Development Process

- Explicitly states which testing phase corresponds to which development phase.
- V-Model can be used to define various iterative development process, beyond waterfall process



Code Inspection (Static Code Analysis)

- Analyzing code without actually executing it.
  - Dynamic analysis: analysis performed by executing code
- Static code analyzers
  - Can automatically detect particular (bad) code smells
  - Various tools available for various languages
    - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Static Code Analyzer: FindBugs

- Looks for bugs in Java programs based on bug patterns, which are code idioms that are often error.
  - http://findbugs.sourceforge.net/
  - http://findbugs.sourceforge.net/bugDescriptions.html
  - Can analyze code compiled for any version of Java, from 1.0 to 1.8
  - Can be used from Ant and Eclipse
  - Plug-ins are available for IntelliJ, Maven, Gradle, Jenkins, etc.

Static Code Analyzer: PMD

- Looks for potential problems such as:
  - Possible bugs
    - Empty try/catch/finally/switch statements
  - Dead code
    - Unused local variables, parameters and private methods
  - Suboptimal code
    - Wasteful String/StringBuffer usage
  - Overcomplicated expressions
    - Unnecessary if statements, for loops that could be while loops
  - Duplicate code
    - Copied/pasted code means copied/pasted bugs
  - http://pmd.sourceforge.net/
  - Requires JRE 1.6 or higher to run
  - Can be used from Ant
  - Plug-ins are available for Maven, Eclipse, etc.

Other Static Code Analyzers

- CheckStyle
  - Checks coding conventions
    - http://checkstyle.sourceforge.net/

HW 7

- Run FindBugs on your HW 5 and 6 programs
- Have your Ant script run FindBugs in addition to Junit and JaCoCO
  - Extra points if you run PMD as well as FindBugs.
  - Required to run FindBugs only for this HW
    - Extra point if you run FindBugs in other HWs in the future.

In CS 680/1 and CS 682/3

- In CS 680/1
  - Ant is sufficient.
  - No need to use any VCS
    - Small-scale, solo work
- In CS 682/3
  - Ant? Gradle??
  - Use a VCS: Subversion? Git?