Pair programming

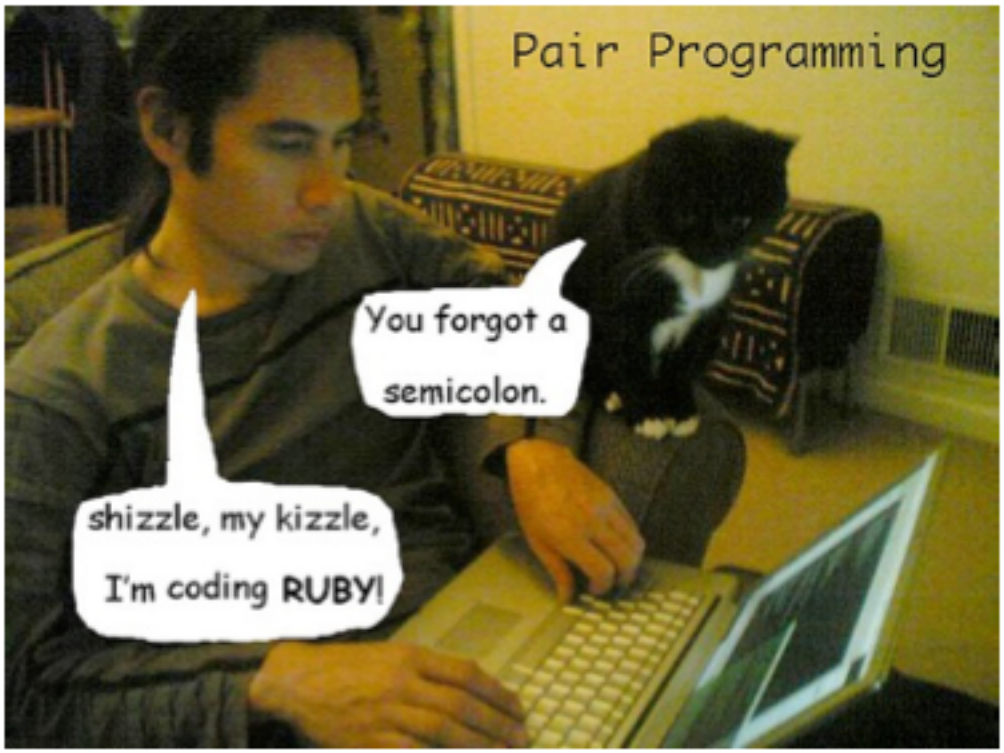
- In pair programming, programmers sit together at the same workstation to develop the software.
- Pairs are created dynamically so that all team members work with each other during the development process.
- The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

Advantages of pair programming

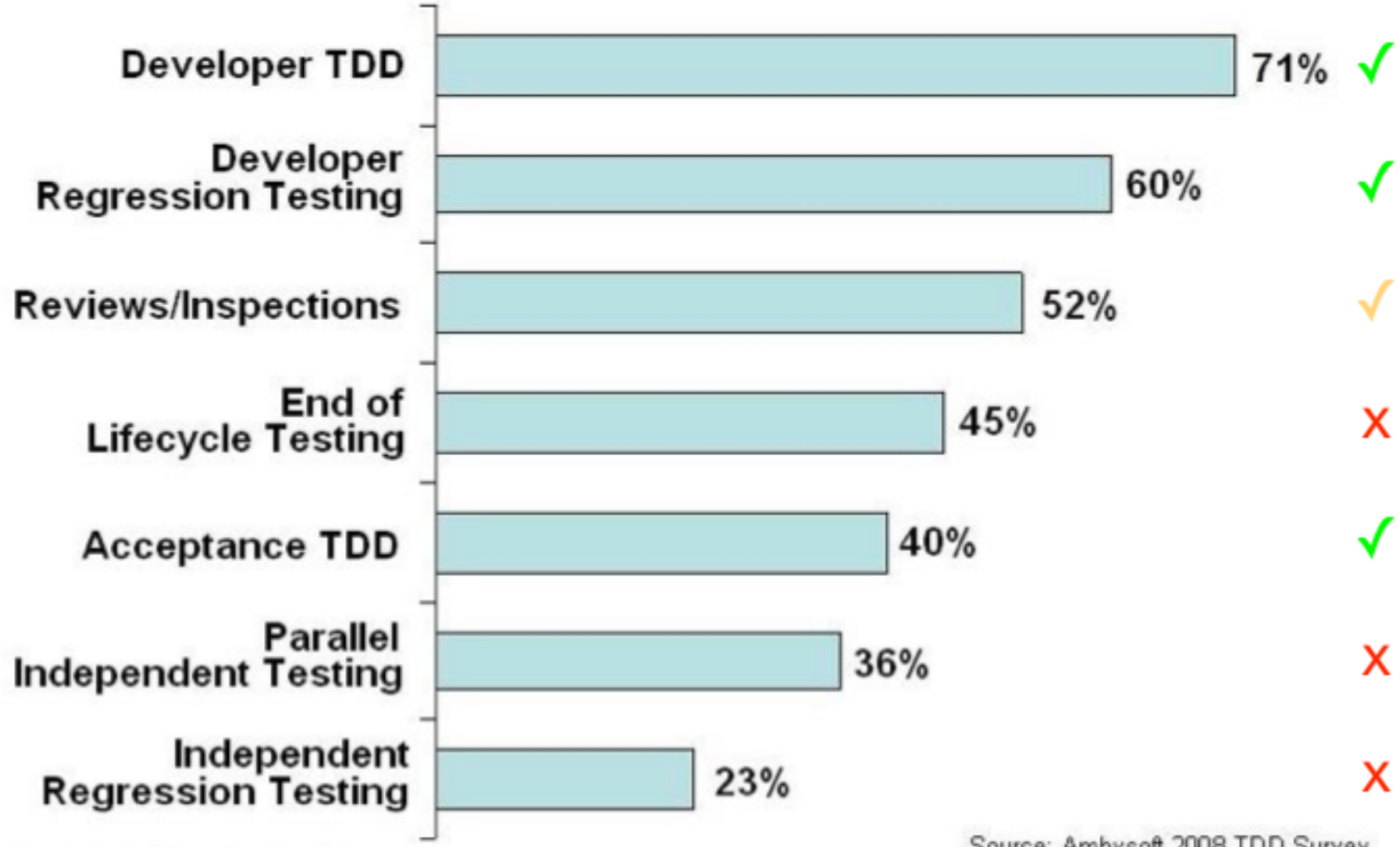
- It supports the idea of collective ownership and responsibility for the system
  - Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- It acts as an informal review process because each line of code is looked at by at least two people.
- It helps support refactoring
  - Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Doing Pair Programming

- **2x4 Pair Programming Rotation**
  - http://www.youtube.com/watch?v=TzUNGOVrhWs

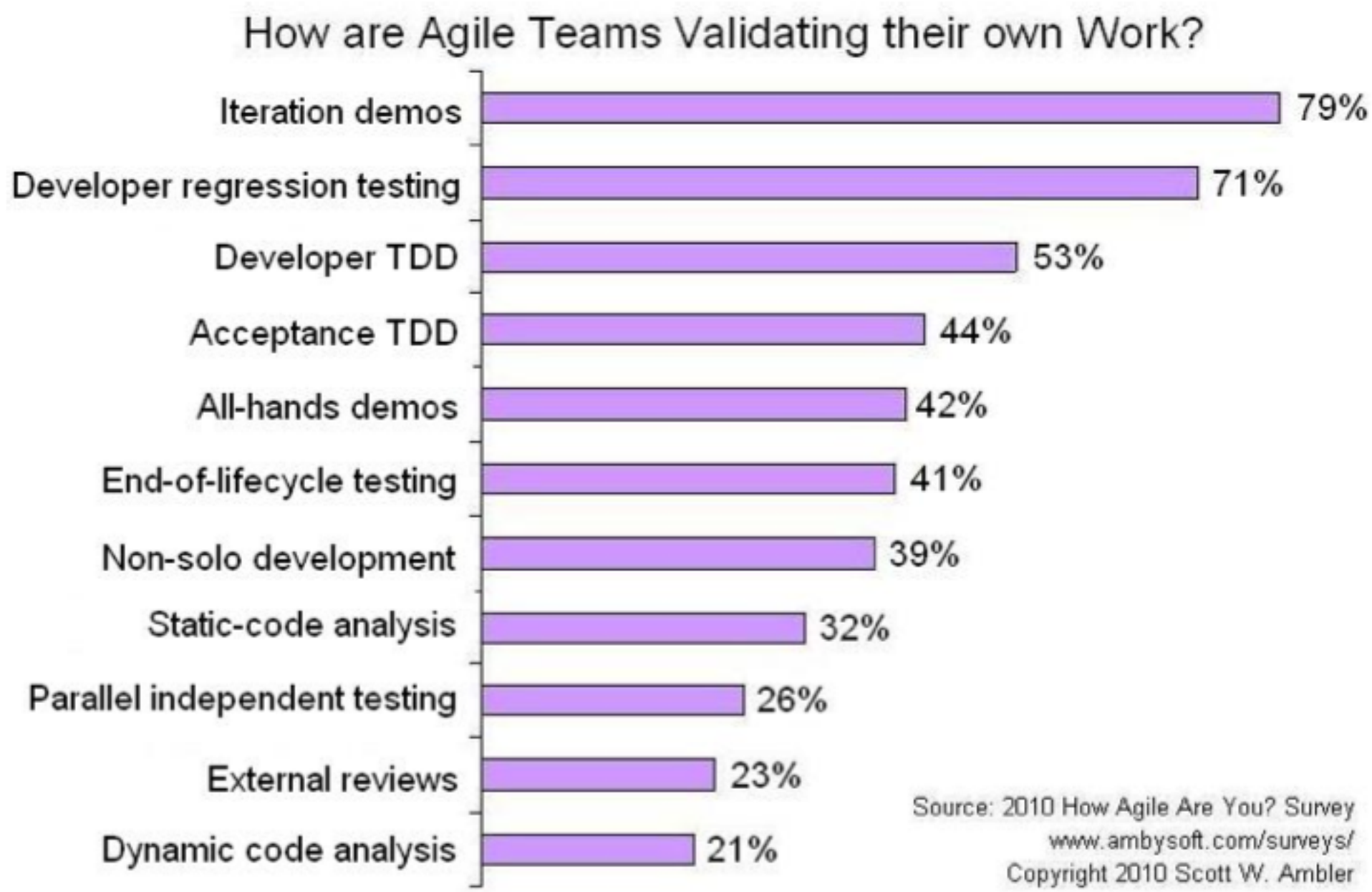


Testing/Validation Practices Amongst Agile Developers

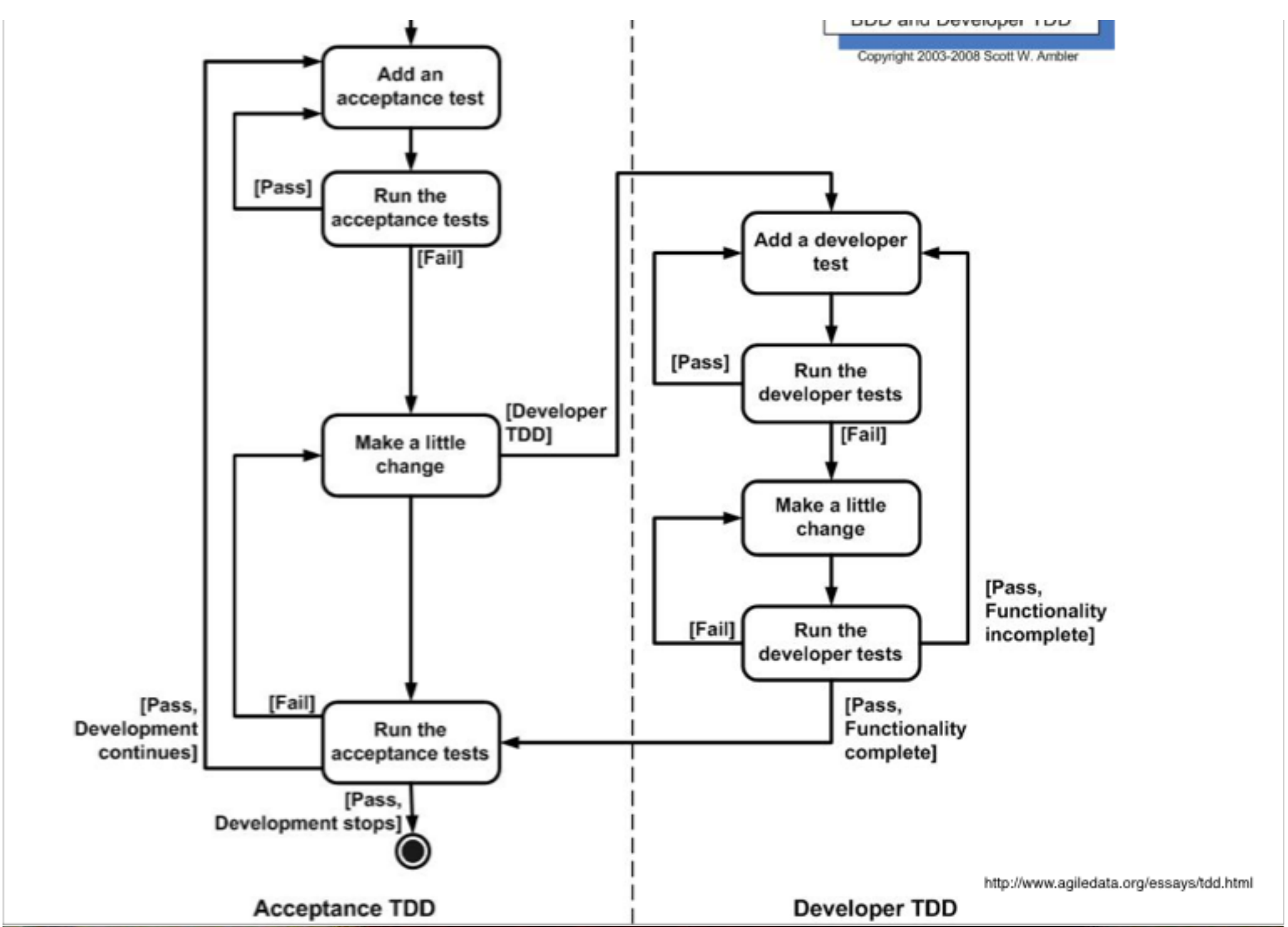


Source: Ambysoft 2008 TDD Survey
www.ambysoft.com/surveys/tdd2008.html

Survey of TDD adherents



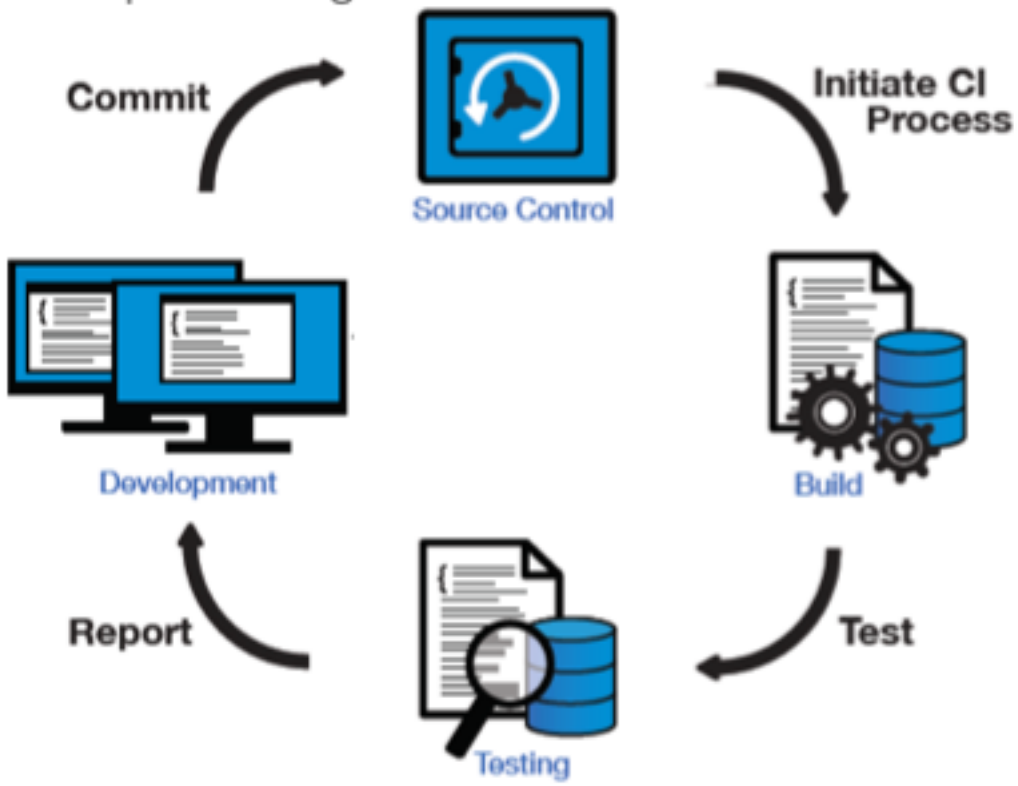
Survey of TDD adherents, 2 years later


Copyright 2003-2008 Scott W. Ambler
Add an acceptance test
[Pass]
Run the acceptance tests
[Fail]
Make a little change
[Developer TDD]
[Pass, Development continues]
[Fail]
Run the acceptance tests
[Pass, Development stops]
Add a developer test
[Pass]
Run the developer tests
[Fail]
Make a little change
[Fail]
Run the developer tests
[Pass, Functionality incomplete]
[Pass, Functionality complete]
http://www.agiledata.org/essays/tdd.html
Acceptance TDD
Developer TDD


Continuous Integration (CI)
cc Jan Krutisch

Objectives of CI

- minimize duration and effort of each integration episode
- be able to deliver a product version suitable for release at any moment
- any effort related to producing intermediate releases should be included in CI



Elements of CI

- use of a version control tool
- automated build and product release process
- build triggers unit and acceptance tests any time any change is submitted
- build alerts team when any test fails, so resolution can be found
    - must be resolved immediately

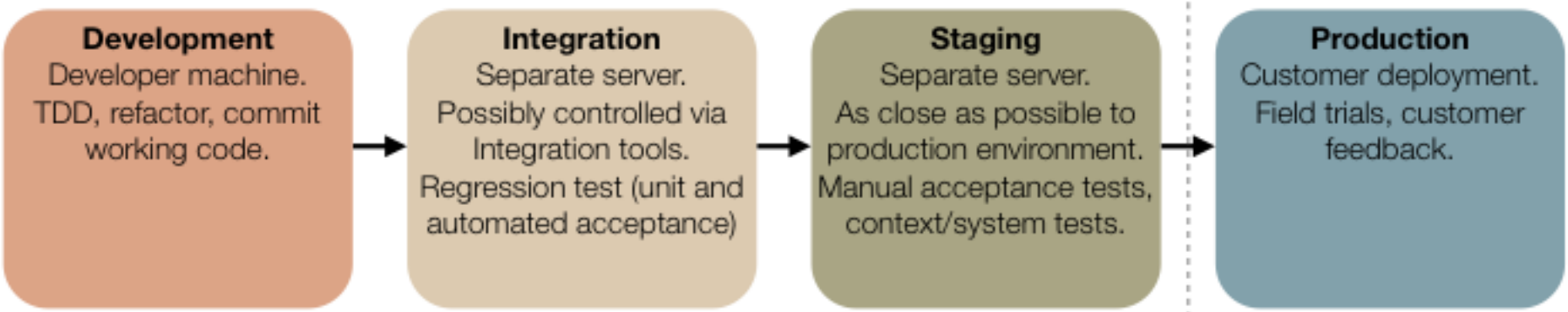
Why regular builds?

- Build the software and regression test it at regular intervals
- Make sure the new code does not compromise pre-existing functionality
- Build frequency often depends on phase of project
  - Weekly builds early on (*Agile practices can have more frequent builds*)
  - Daily builds at tail end to deal with last-minute changes/additions (and bug fixes!)
  - Agile emphasizes ***continuous, regular, evenly-paced*** builds.

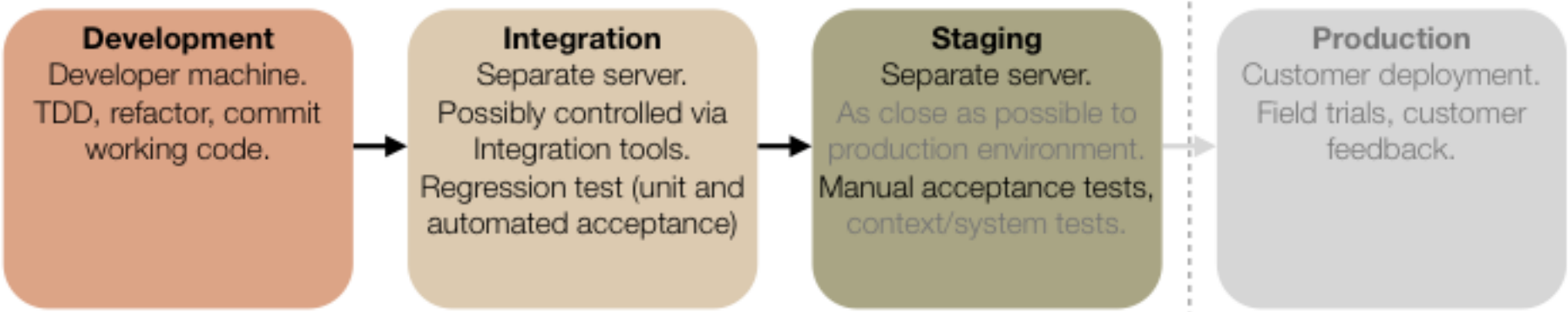
Committing code in CI

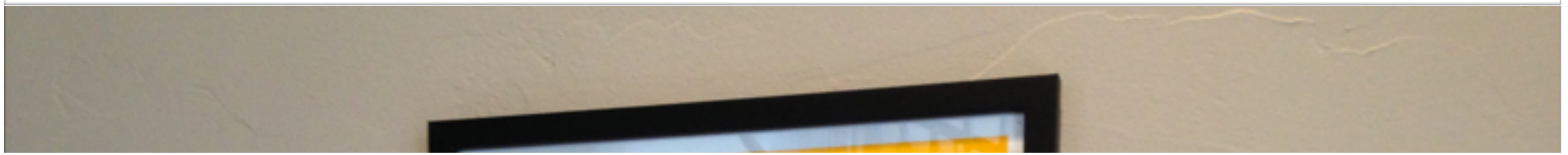
- Small amounts of code added to the baseline on a very frequent basis (perhaps daily)
- Newly integrated code may not even be at level of a completed method/class
- Everyone should commit code to the baseline regularly, quickly identifying conflicts
- As long as code is unit-tested and introduces no errors, it can be integrated into the baseline

Continuous Integration, Continuous Deployment



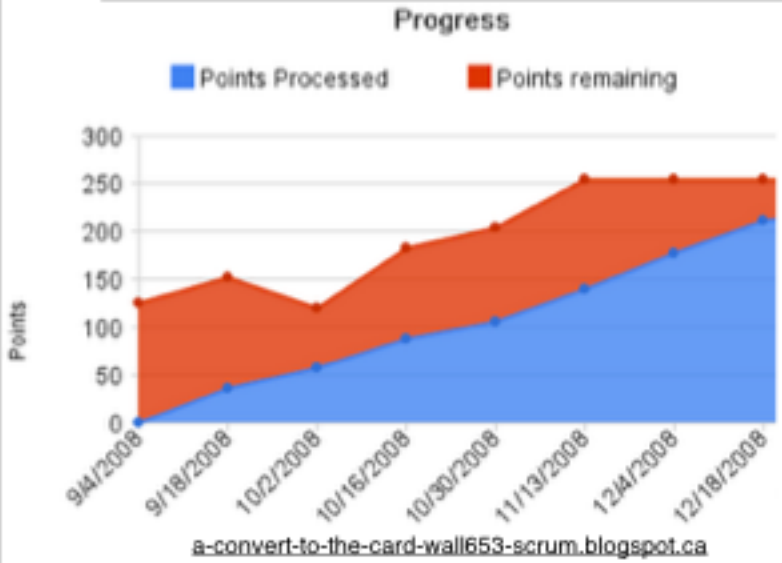
Continuous Integration, CSCI 3130 project



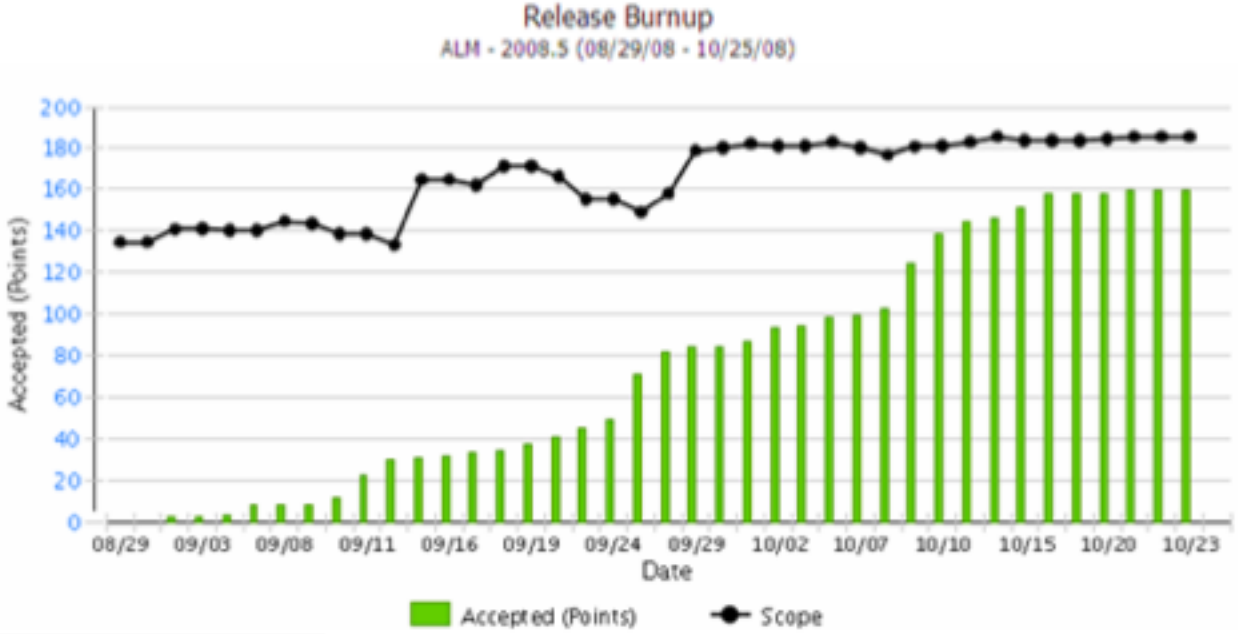


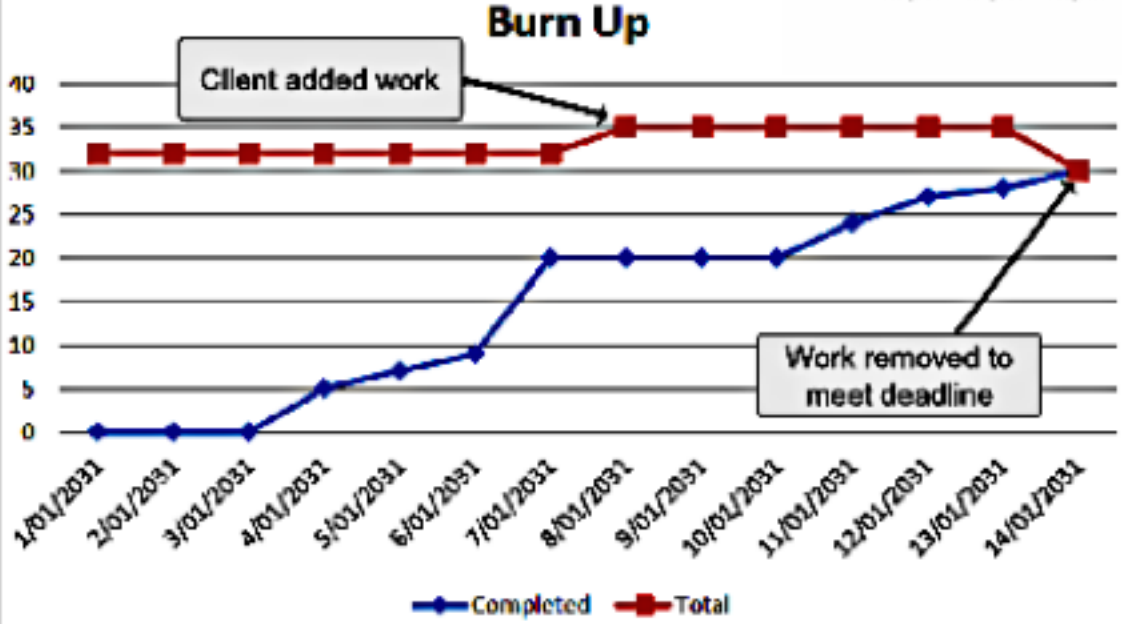
Charting progress

Burn-up



a-convert-to-the-card-wall653-scrum.blogspot.ca



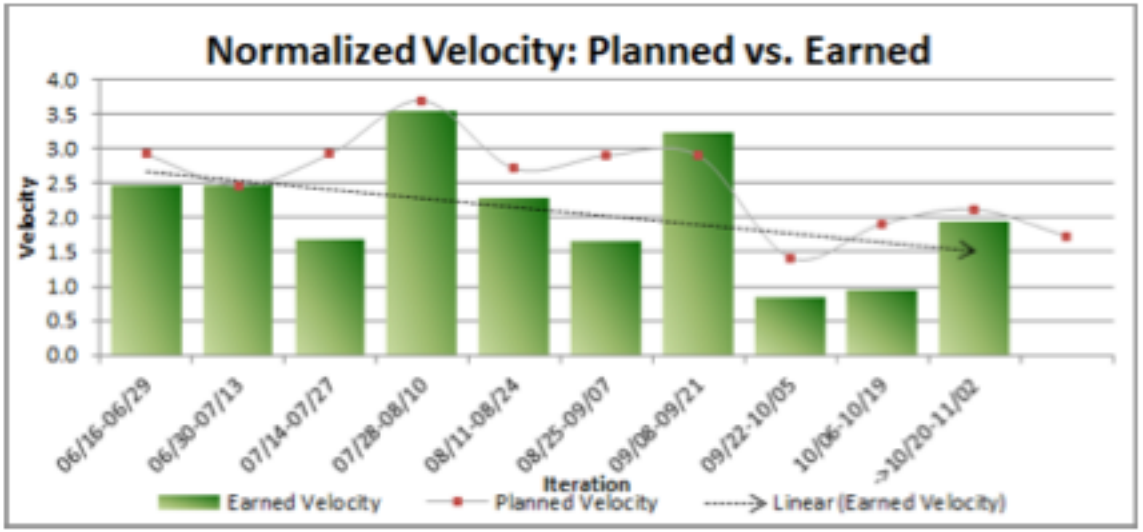


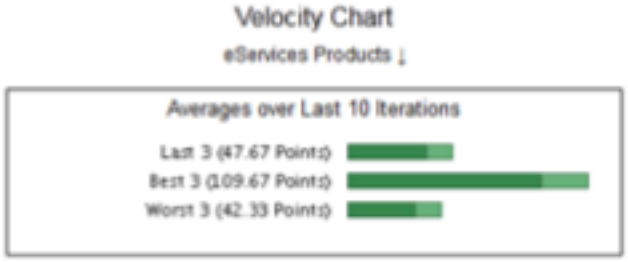
http://www.clariostechnology.com/images/blog/burnup.png

http://al106653-scrum.blogspot.ca/2013_05_01_archive.html

Velocity

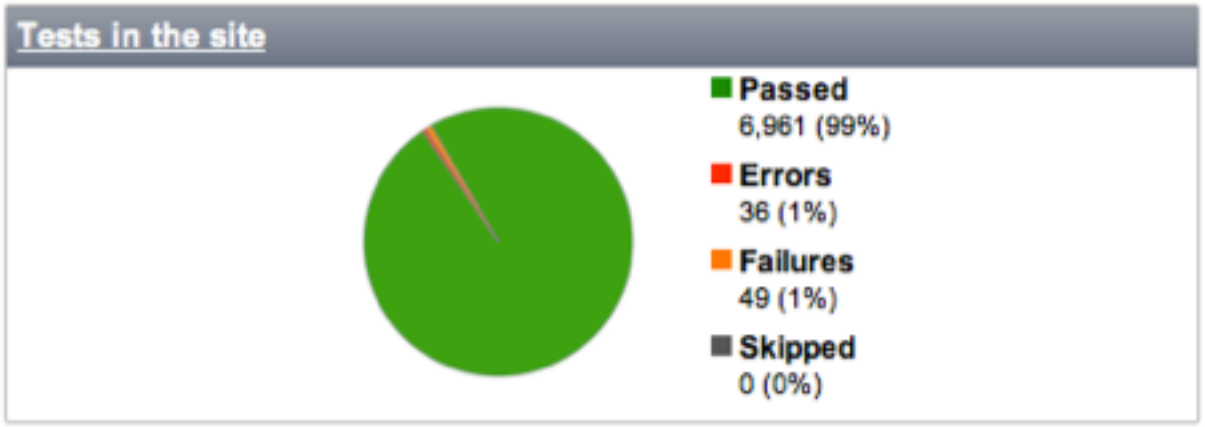
Velocity

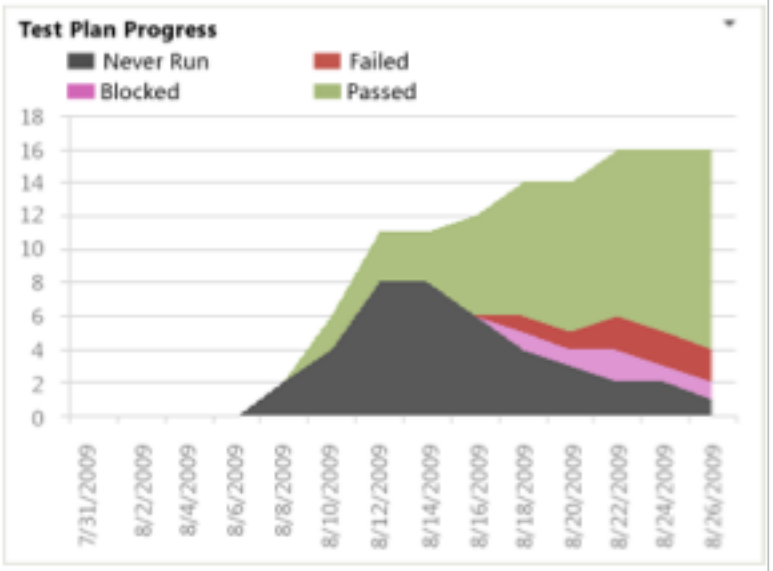




Test

Test





https://msdn.microsoft.com/en-us/library/dd420548.aspx

https://msdn.microsoft.com/en-us/library/dd420548.aspx

Exercise: Agile Charts

Exercise: Agile Charts

- break into groups of 2-3 (one must have a laptop or tablet)
- open the data sheet SeafoodMarketIteration1 — link on course website: https://docs.google.com/spreadsheets/d/1tdmeG2jOi_UHVfHavoccEAK09qIrO9nXnf2f0EfWYI0/edit?usp=sharing
- generate burnup, velocity, and test charts using the data (you can use a spreadsheet program or sketch them out by hand)
- 20 mins

```
/**
 * Simple HelloButton() method.
 * @version 1.0
```

```
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnList

  // use the JFrame type until support for t
  // new component is finished
  JFrame frame = new JFrame( "Hello Button"
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();
  frame.show();                // display the fra
}
```

Code Documentation

interface documentation, self-documenting code, inline comments

Documenting Interfaces: what?

Documenting Interfaces: what?

- classes and interfaces
  - general description, @author
- public and protected data members
  - identify what it represents, how to change if read/write
- public and protected methods
  - identify what it does, describe parameters (@param), return values (@return), and exceptions thrown (@throws)

Documenting Interfaces: how?

```
/**
 * Reads a line from this console, using the specified prompt.
 * The prompt is given as a format string and optional arguments.
 * Note that this can be a source of errors: if it is possible that your
 * prompt contains {@code %} characters, you must use the format string {@code "%s"}
 * and pass the actual prompt as a parameter.
 *
 * @param format the format string (see {@link java.util.Formatter#format})
 * @param args
 *             the list of arguments passed to the formatter. If there are
 *             more arguments than required by {@code format},
 *             additional arguments are ignored.
 * @return the line, or null at EOF.
 */
public String readLine(String format, Object... args) {
    synchronized (CONSOLE_LOCK) {
        format(format, args);
        return readLine();
    }
}
```

Documenting Interfaces: how?

```
/**
* Returns the number of days in the given date range
* [startDate, endDate) that fall on the
* provided dayOfWeek.
* @param startDate beginning of the date range, inclusive
* @param endDate the end of the date range, exclusive
* @param dayOfWeek the day of week
* @return the total number of days in the range
* @throws IllegalArgumentException if any parameter is
* null, or if startDate >= endDate
*/
public int daysInRange (java.util.Date startDate,
       java.util.Date endDate,
       Day dayOfWeek)
       throws IllegalArgumentException;
```

Self-documenting code

- careful choice of class/interface, method and variable names
- careful choice of enumeration values
- composing larger methods out of many smaller, well-named methods
- composing classes out of other classes
- one-responsibility rule: if you can't name something easily, it might need to be split into smaller parts (e.g. complex classes)
- you should still document interfaces (e.g. Javadoc)!

Inline comments, some recommendations

- don’t use comments to explain poorly written code
    - refactor first, comment later
- use “TODO”, “FIXME”, “BY” sparingly, if at all
    - a better place for TODO is in a work plan, FIXME or BY in a commit message and/or issue tracker, especially for large projects
    - at least put it in both places (searching code for these messages can work for small projects with few developers)
- consider whether better self-documentation can improve readability before adding an inline comment
- ok, then comment. Make it short. Keep it up to date.

Building with Ant

Ant…

- Is configured with a build file that is an XML document
- Is generally platform independent
- Can take care of archiving/versioning, compilation, execution, documentation, deployment, and more

A simple Ant build file

```
<project default="hello">

  <target name="hello">

  <echo message="Hello, World"/> </
  target>

</project>
```

Execution :

```
$ ant
Buildfile: build.xml
   hello: [echo] Hello, World
   BUILD SUCCESSFUL
Total time: 2 seconds
```

Construction of a build file

- Default name for a build file is build.xml
- Root element must be the ‘project’ element
  - The ‘default’ attribute is required
    - Specifies the default target to use
  - Other attributes available (e.g. name, basedir)

Construction of a build file (continued)

- Targets contain zero or more tasks
  - The ‘name’ attribute is required
- Tasks are the smallest units of work
  - Ant Tasks located in ant.jar or ant classpath

Build file specifics

- There is only one ‘project’ per build file
- As many different ‘target’ elements as needed (at least one)
- Each target can contain as many tasks as are needed
- Comments are normal XML comments `<!--comment-->`
- Properties can be set or included from a file

Project element

Attribute	*Description*
name	the name of the project.
default *	the default target to use when no target is supplied.
basedir	the base directory from which all path calculations are done.

* Denotes required field

Target element

Attribute	*Description*
name *	the name of the target.
depends	a comma-separated list of names of targets on which this target depends.
if	the name of the property that must be set in order for this target to execute.
unless	name of the property that must not be set in order for this target to execute.
description	a short description of this target's function.

* Denotes required field

Task element

- A piece of code that can be executed
- All have common structure:
  `<name attribute1="value1“ attribute2="value2" ... />`
- Large set of built-in tasks and optional tasks
  - Complete listing/defs at ant.apache.org/manual
- Write your own tasks in Java

A more typical example

A Java class :

A Java class :

```
public class hello {
    public static void main( String[] args )
    {
        System.out.println( "Hello World" );
    }
}
```

An Ant build file named hello.xml :

```
<project default="compile">
  <target name="compile">
    <javac srcdir="." />
  </target>
</project>
```

Console command :

```
$ ant -f hello.xml compile
```

Or (using default) :

```
$ ant -f hello.xml
```

Or (saved as build.xml, using default) :

```
$ ant
```

Typical example extended

```
<project default="compile">
```

```
<project default="compile">
  <target name="compile">
    <javac srcdir="." />
  </target>

  <target name="jar" depends="compile">
    <jar destfile="hello.jar"
         basedir="."
         includes="**/*.class"
         />
  </target>
</project>

$ ant -f hello.xml jar
```

javac task

- Over 30 attributes available

- Over 30 attributes available
- Only the **`srcdir`** attribute is required
  - Unless nested **`src`** elements are present
- Some attributes (e.g. **`srcdir`** & **`classpath`**) are path-like structures and can also be set via nested elements
- You can specify additional command line arguments for the compiler with nested **`<compilerarg>`** elements.

`java` task

- Over 20 attributes available

- Over 20 attributes available
- Either `jar` or `classname` attribute is required
- Arguments to class specified in nested `<arg>` elements
- If odd things go wrong when you run this task, set `fork="true"` to use a new JVM

More complete example

```
<project name="MyProject" default="dist" basedir=".">
```

```
    <description> simple example build file </description>

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
     <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
      description="compile the source" >
  <javac srcdir="${src}" destdir="${build}"/>
  </target>
```

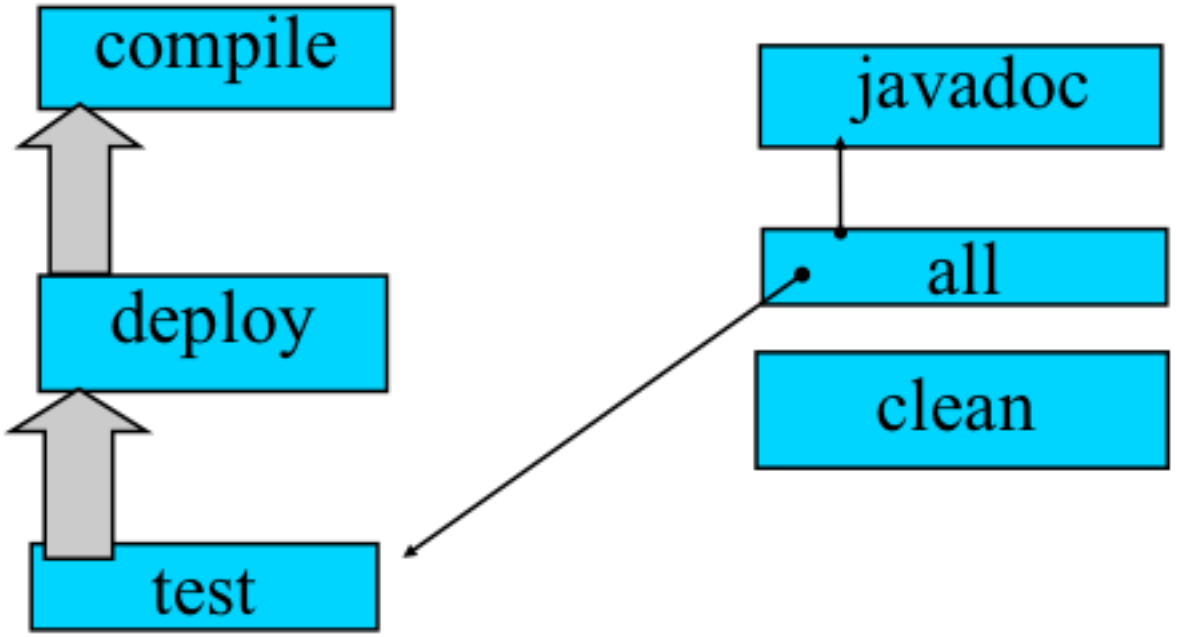
Complete example contd.

```
  <target name="dist" depends="compile"
```

```
        description="generate the distribution" >
    <mkdir dir="${dist}/lib"/>
    <tstamp/>
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="$
    {build}"/>
    </target>

    <target name="clean" description="clean up" >
    <delete dir="${build}"/>
        <delete dir="${dist}"/>
    </target>
</project>
```

Typical Dependences



Properties

- A property is a name-value pair that can be referenced

in an Ant script like variables

- Many ways of defining/importing properties in Ant
- You get some built-in properties for free
  - **`basedir, ant.file, ant.version, ant.project.name, ant.java.version`**
  - all Java system properties
    - http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#getProperties()
  - some tasks make properties available (e.g. **`tstamp`**)

Defining Properties

- With a properties file

- Define `name=value` pairs in a text file (usually of extension `.properties`)
  - Watch out for escape sequences.
  - Example:

```
bin.dir=${basedir}/bin
HelloWorldStr=Hello world.
```

- Load the properties file using the `<property>` task
- Example:

```
<project name="build" default="all">
    <property file="build.properties" />
    ...
```

Defining Properties (cont.)

- In the build script using the **`name`** and **`value | file | refid`** attributes of the **`<property>`** task

the **<property>** task

- Example:

```
<project name="build" default="all">
    <property name="HelloWorldStr"
              value="Hello world." />
    ...
```

- Via command-line when calling Ant
  - Example: **ant -Ddate=050206**

Using Properties

- Example:

```
<echo message="${HelloWorldStr}" />

<delete dir="${bin.dir}" quiet="yes" />
```

- What happens when the first line is run?
  - If the property **`HelloWorldStr`** is defined, then **`${HelloWorldStr}`** will be replaced by the value
  - Otherwise, the string "${HelloWorldStr}" is echoed!

Filesets

- Fileset is one of the fundamental structures in Ant

- User can provide rules to include and/or exclude assets in a directory, for example:
  - Including only .java files in your build directory
  - Excluding any backup files (.bak, .tmp, etc.) generated by your text editor
  - Copy only the files specified in a list

Using Filesets in Ant Tasks

- Using **<fileset>** to delete .bak files:

```
<delete quiet="yes">
```

```
<delete quiet="yes">
    <fileset dir="doc" includes="**/*.bak" />
</delete>
```

or

```
<delete quiet="yes">
    <fileset dir="doc">
        <include name="**/*.bak" />
    </fileset>
</delete>
```

- … and you can nest many filesets in **<delete>**

Implicit Filesets

- Some tasks are implicit filesets

- Example: javac, tar, zip, etc.

- Just treat the task like a genuine **<fileset>**

- Example – compiling Java files in the **org.apache.ant** package:

```
<javac srcdir="src" destdir="bin">
    <include name="org/apache/ant/**/*.java" />
</javac>
```

Multiple Build Scripts

- Usually not necessary, but it is possible to use multiple Ant scripts in the build process

- The **<ant>** task can call another Ant script
  - Example:

```
<ant antfile="other.xml" />
```

  - This will run the default target of other.xml while inheriting all properties from the caller

Helper Target

- Just like helper methods – available for calling with different parameters

- The **<antcall>** task can call targets specifying parameters using **<param>**
- **<param>** can override global properties

Helper Target (cont.)

- Example (cont.):
  - Helper target:

```
<target name="echoTwice">
    <echo message="${message}" />
    <echo message="... and again: ${message}" />
</target>
```

- ... and calling it:

```
<antcall target="echoTwice">
    <param name="message" value="Hi." />
</antcall>
```

References - refid

- Example, repeated elements of a fileset:

```
<project ... >
  <target ... >
    <rmic ...>
```

```
    <rmic ...>
      <classpath>
        <pathelement location="lib/"/>
        <pathelement path="${java.class.path}/"/>
        <pathelement path="${additional.path}"/>
      </classpath>
    </rmic>
  </target>

  <target ... >
    <javac ...>
      <classpath>
        <pathelement location="lib/"/>
        <pathelement path="${java.class.path}/"/>
        <pathelement path="${additional.path}"/>
      </classpath>
    </javac>
  </target>
</project>
```

References - refid

- Example, rewritten using refid:

```
<project ... >
  <path id="project.class.path">
    <pathelement location="lib/"/>
```

```
    <pathelement location="lib/"/>
    <pathelement path="${java.class.path}/"/>
    <pathelement path="${additional.path}"/>
  </path>

  <target ... >
    <rmic ...>
      <classpath refid="project.class.path"/>
    </rmic>
  </target>

  <target ... >
    <javac ...>
      <classpath refid="project.class.path"/>
    </javac>
  </target>
</project>
```

One-Step Build Process

- Ant provides a lot of useful tasks:
  - **<svn>** can check out and update repository

- **<copy>** and **<delete>** can rearrange files
- **<javac>** and **<java>** can compile and run Java code
- **<junit>** can run test suites
- **<exec>** can run any command-line executable
  - Compromises portability
- **<zip>** and **<jar>** can help package the software
- … and you can write your own tasks!

Build Notification – Ant Task

- One way is to send it with the build script
- **<mail>** can send e-mail

- Example:

```
<mail mailhost="smtp.myisp.com"
        subject="Build Successful">
    <from address="me@myisp.com" />
    <to address="developers@myisp.com" />
    <message>Today's build is done.</message>
</mail>
```

Write your own task (the basics)

1. Create a Java class that extends **`org.apache.tools.ant.Task`**
2. For each attribute, write a setter method

3. If the task contains other tasks as nested elements, you must implement the interface **`org.apache.tools.ant.TaskContainer`**

4. If the task should support character data, write a

   `public void addText(String)` method

5. For each nested element, write a **`create`**, **`add`** or **`addConfigured`** method

6. Write a **`public void execute`** method, with no arguments, that throws a **`BuildException`**

Exercise: Ant

- break into groups of 2-3 (one must have a laptop or tablet)

- download the “ant and javadoc mini exercise” pdf from the website, and the

download the "ant and javadoc mini exercise" pdf from the website, and the "AntSample.zip" file

- (optional, for testing) install Ant on your machine
- complete (a) and (b), in that order
- 20 mins