Benefits of the Build

A Case Study in Continuous Integration

Kirk Knoernschild

http://techdistrict.kirkk.com

http://www.kirkk.com
pragkirk@kirkk.com

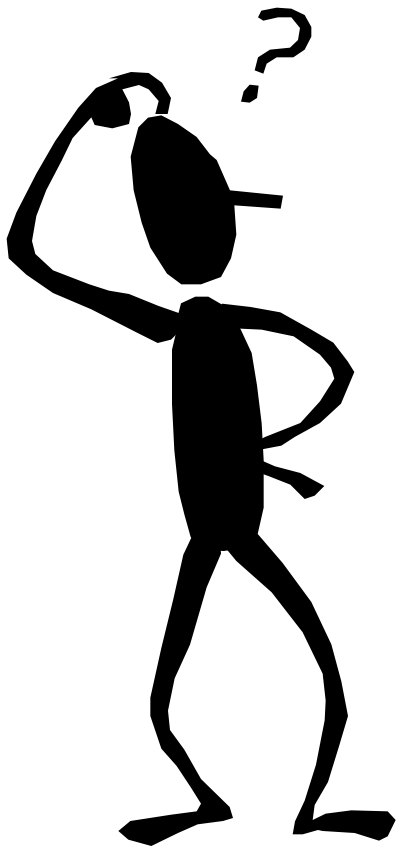
September 21, 2007

Continuous Integration

- ➡The Ground Rules
- The Strategy
- Positive Affects
- Case Study

Continuous Integration

- What does Continuous Integration mean to you?



Defined

- Integrate and build the system … every time a task is completed [Beck in XP Explained]
- Integrate, build, and verify the system as often as feasibly possible
  - Hourly, Daily, On change

The Build

- Must be a clean compile
- Use most current source files
- All files must be compiled
- .jar files created
- All unit tests execute…successfully

Automated and Repeatable

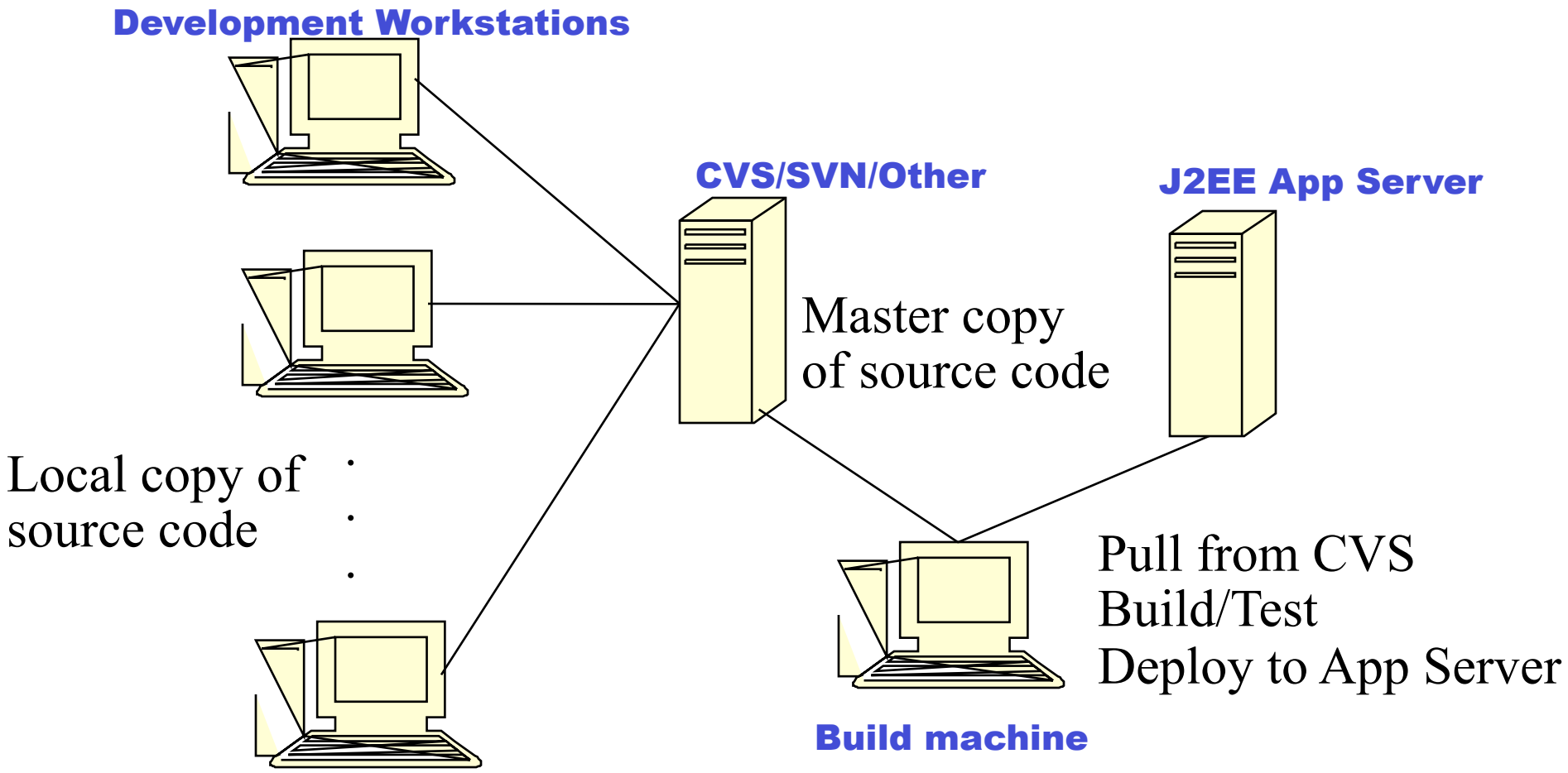
Automated

- Minimizes integration risk
- Build archives enable defect diagnosis
- Time savings
- Improves morale
  - Always have a working product

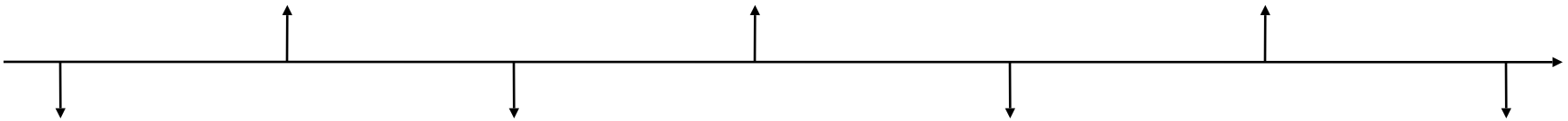
Repeatable

- Avoids tedious, repetitive and error prone manual builds
  - Consistent each time
- Can easily be run anytime you want
- Incorporate metrics

Standard Environment



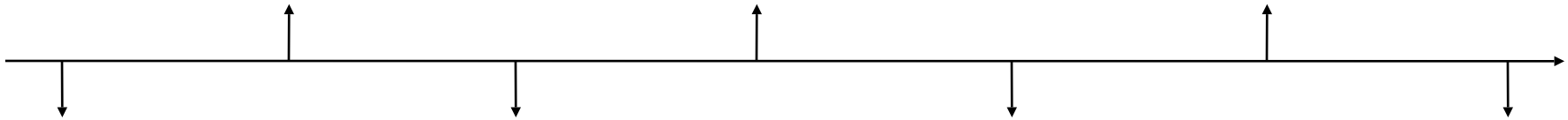
Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

Micro Process

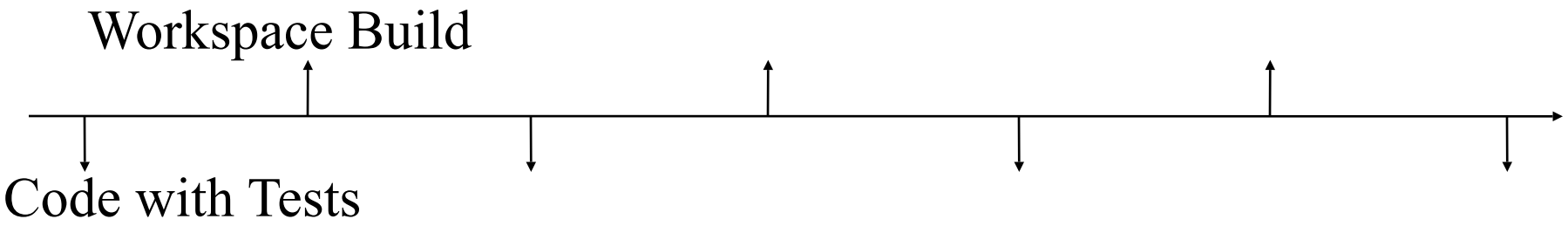


Code with Tests

Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

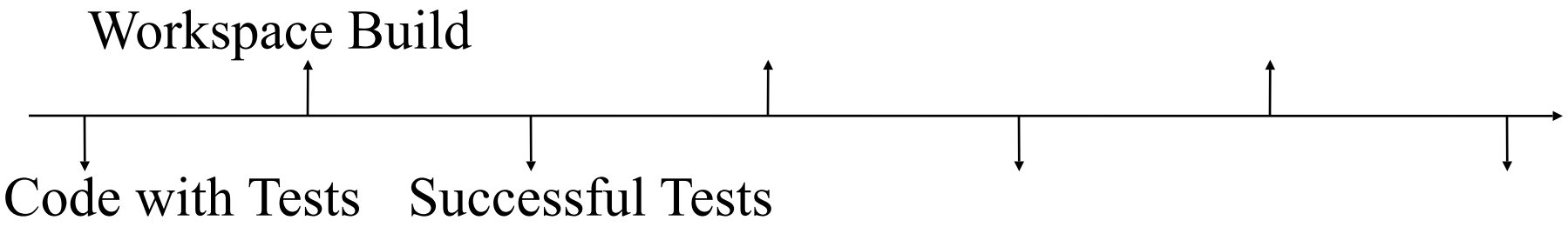
Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

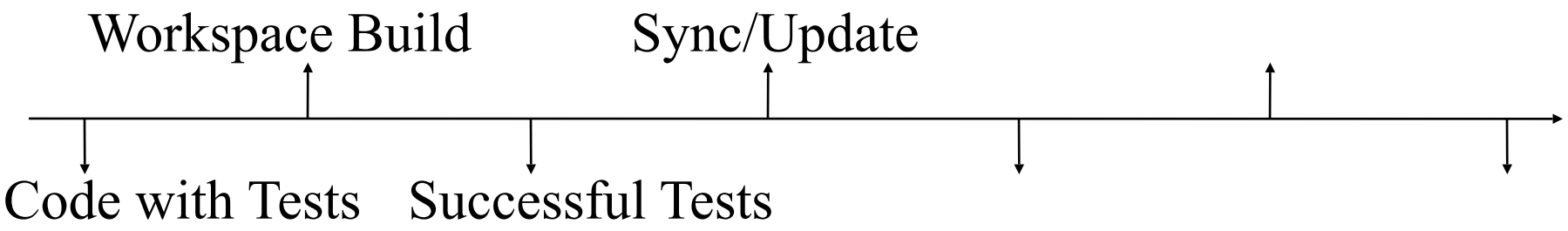
Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

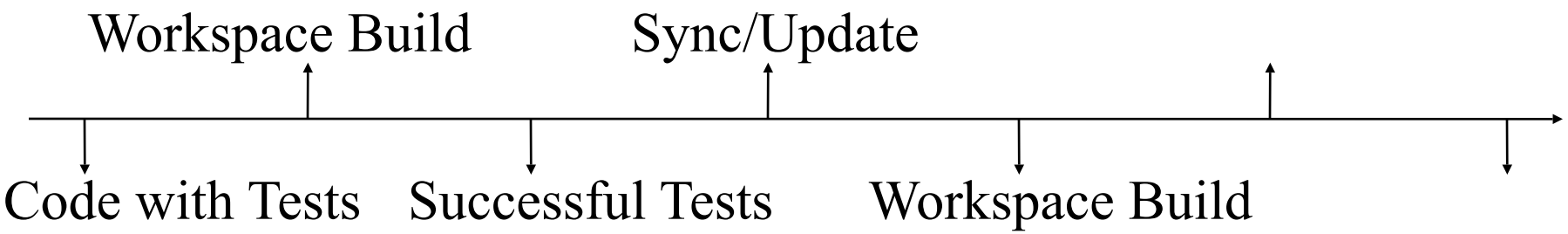
Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

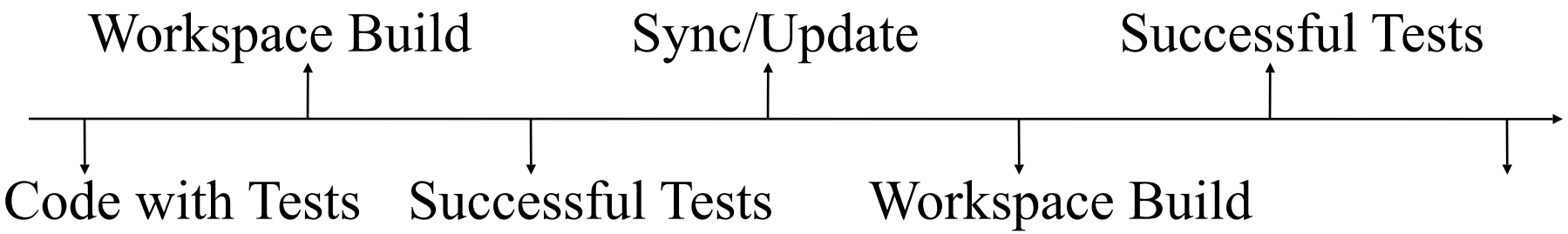
Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

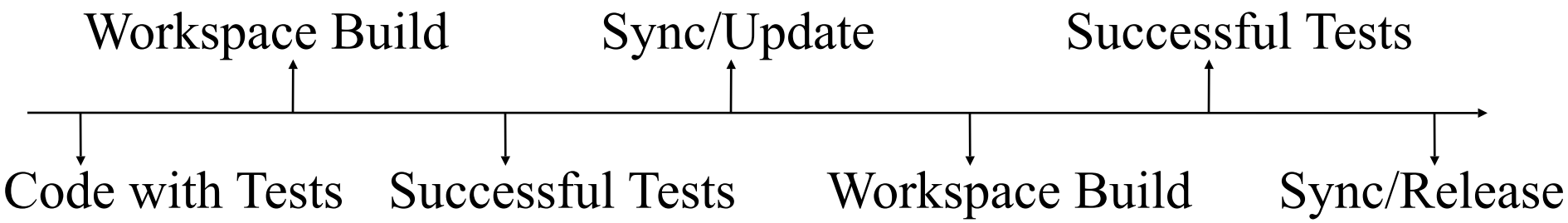
Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

Micro Process



Your Golden Rule

- Stream must always compile
- Test cases must always run successfully
- If either of these two conditions is not true, it must immediately become the focus of all team members to rectify the situation

Fixing Defects

- Identify what you believe is the cause of the defect.
- Create a test that recreates the defect. The test should fail.
- Verify that the test failed due to the suspected defect. If not, start over.
- Correct the defect.
- Run the test. The test should now pass.
- Run all tests. All tests should pass.
- Release your code, including the tests, following the Micro Process.

50% of the solution is identifying the problem correctly.

Impediments

- Resolving conflicts
- Overwriting code
- Slow test execution
- Data dependent tests
- Neglecting to release all files

Continuous Integration

- ✓ The Ground Rules
- ➡ The Strategy
- Positive Affects
- Case Study

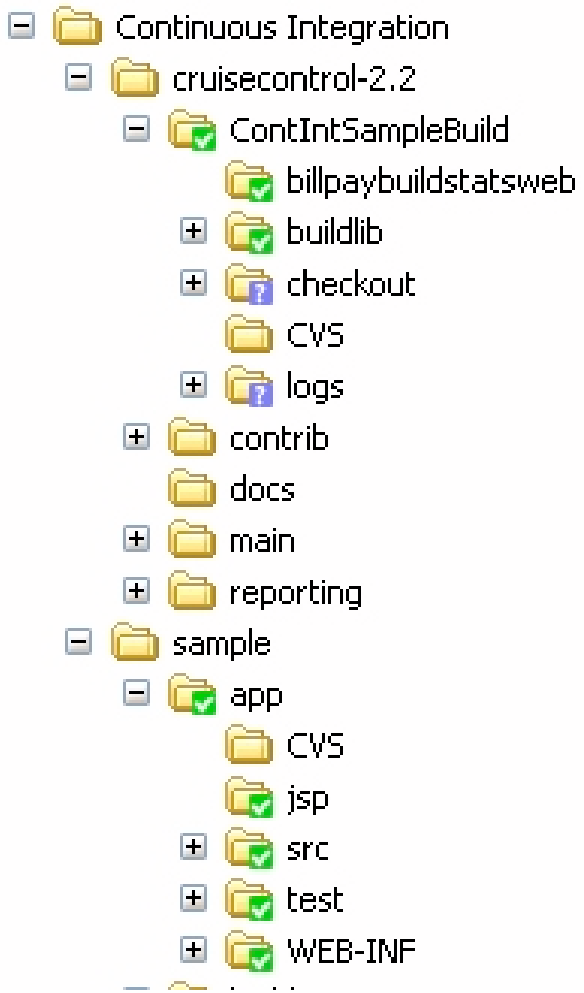
The Steps

- Create a build script
  - Repeatable build
- Incorporate your optional metrics
- Create a build website (dashboard)
- Setup your automated build tool
  - Cruise Control, AntHill

The Build Script

- Checkout from CVS
- Compile the application
- Run all the test cases
- Bundle the application
- Generate the metrics and statistics
- Deploy the application
- Deploy the build statistics application

Directory Structure



CVS Checkout

```
<target name="checkout" depends="init">
        <cvs cvsRoot="c:/repositories"
        package="ContIntSample"
        dest="${basedir}"/>
</target>
```

basedir is the build directory.

Command: ant –buildfile billpaybuild.xml checkout

Result: Checks out the code to the build directory.

Compile Application

```
<target name="billpaycompile" depends="billtestcompile">
        //do some setup stuff here…
        <javac srcdir="${buildsrc}" destdir="${build}">
                <classpath>
                        <pathelement path="${buildsrc}"/>
                        <pathelement location="${bindist}/bill.jar"/>
                        <pathelement location="${bindist}/financial.jar"/>
                        <pathelement location="${bindist}/auditspec.jar"/>
                        <pathelement location="${bindist}/audit1.jar"/>
                        <pathelement location="${bindist}/audit2.jar"/>
                </classpath>
        </javac>
        //do some cleanup stuff here.
</target>
```

Command: ant –buildfile billpaybuild.xml billpaycompile

Run Test Cases

```
<target name="billpaytestcompile" depends="billpaycompile">
        //do some setup stuff here…
        <junit printsummary="yes" haltonfailure="yes">
                <classpath>
                        <pathelement path="${build}"/>
                        //some other jars here….
                </classpath>
                <test name="com.extensiblejava.mediator.test.AllTests"
                        todir="${buildstats}" outfile="billpaytest">
                        <formatter type="xml"/>
                </test>
        </junit>
</target>
```

Command: ant –buildfile billpaybuild.xml billpaytestcompile

Bundle Application

```
<target name="bundle" depends="appcompile">
        <mkdir dir="${deploy}"/>
        <war destfile="${deploy}/billpay.war"
                        webxml="${appdir}/WEB-INF/web.xml">
                <fileset dir="${appdir}/jsp"/>
                <webinf dir="${appdir}/WEB-INF">
                        <exclude name="web.xml"/>
                        <exclude name="lib/servlet-api.jar"/>
                </webinf>
                <lib dir="${bindist}" excludes="**test.jar"/>
                <classes dir="${build}"/>
        </war>
</target>
```

Command: ant –buildfile billpaybuild.xml bundle

Result: Checkout, Compiles, Test, and Bundles the application.

Generate Metrics

```
<target name="jdepend" depends="pmd">
        <jdepend format="xml" outputfile="${buildstats}/jdepend.xml">
                <classespath>
                        <pathelement location="${classes}"/>
                </classespath>
                <classpath location=""/>
        </jdepend>

        <style in="${buildstats}/jdepend.xml"
              out="${buildstats}/jdepend.html"
                   style="${buildlib}/jdepend.xsl">
        </style>
</target>
```

Command: ant –buildfile billpaybuild.xml jdepend

Deploy Application

```
<target name="deploy" depends="undeploy">
        <taskdef name="deploy“ classname="org.apache.catalina.ant.DeployTask">
                <classpath>
                        <pathelement path="${buildlib}/catalina-ant.jar"/>
                </classpath>
        </taskdef>
        <deploy url="http://localhost:8080/manager" path="/billpay"
                war="file:///${deploy}/billpay.war"
                username="admin" password="" />
</target>
```

Command: ant –buildfile billpaybuild.xml deploy

Result: Previous plus Deployes the application.

Deploy Build Statistics

```
<target name="deploystats" depends="undeploystats">
        <taskdef name="deploy"
                classname="org.apache.catalina.ant.DeployTask">
                <classpath>
                        <pathelement path="${buildlib}/catalina-ant.jar"/>
                </classpath>
        </taskdef>
        <deploy url="http://localhost:8080/manager" path="/billpaybuildstats"
                war="file:///${statsdeploy}/billpaybuildstats.war"
                username="admin" password="" />
</target>
```

Build Automation

- Setup CruiseControl
- Define the Project
- Start Cruising

Setup Cruise Control

- Installation
- Defining the directories
- Running Cruise Control

Project Definition

```
<cruisecontrol>
    <project name="ContIntSample" buildafterfailed="false">
        <bootstrappers>
            <currentbuildstatusbootstrapper file="logs/ContIntSampleBuildStatus.txt"/>
        </bootstrappers>
        <modificationset requiremodification="no" quietperiod="60">
            <cvs localworkingcopy="checkout/ContIntSample"/>
        </modificationset>
        <schedule interval="120">
            <ant antscript="c:\ant\bin\ant.bat“ buildfile="billpaybuild.xml" target="deploy"/>
        </schedule>
        <log dir="logs/ContIntSample">
            <merge dir="buildstats"/>
        </log>
        <publishers>
            <currentbuildstatuspublisher file="logs/ContIntSampleBuildStatus.txt"/>
        </publishers>
    </project>
</cruisecontrol>
```

Demonstration

- Start Tomcat
- Start Cruise Control
- Run Sample App and audit Bill 1 (15% discount).
- Modify discount for Audit Façade 2 to 10%. Check-in. Let it build.
- Fix test case. I should have run tests locally first. Let it build
  - AuditFacade2Test and BillTest (why is BillTest flawed?).
- Run Sample App and audit Bill (10% discount).

Continuous Integration

- ✓ The Ground Rules
- ✓ The Strategy
- ➡ Positive Affects
- • Case Study

Build Frequency

- Run on a scheduled basis
  - Hourly, Daily
- Run on repository changes
- Anytime you want
  - Build script executable outside Cruise Control

Consistency

- Build is performed the same way each time.
- Build is automated, but can also be run manually.
- Results are predictable.
- Tests are frequently run.
- Metrics are automatically generated.

Zero Compile Errors

- Project pressures force us to compromise our work.
- Adopt zero tolerance to compile errors and failed tests.
- The Golden Rule.

Enforce Dependencies

- JDepend tests can enforce package dependencies.
- Levelized Build can enforce .jar dependencies.
- JarAnalyzer analyzes jar dependencies.

Drive the Lifecycle

- Application is always functional and ready.
- Frequent demos are possible.
- Frequent customer feedback.
- Acceptance test at any time.
- Performance test
- Load test
- Etc…

Objective Feedback

- Metrics generate feedback.
- PMD, JDepend, JarAnalyzer, Java2HTML, JavaNCSS
- Others may include
  - EMMA (test coverage), JavaDoc, UML

Consistent Development

- Build early in the lifecycle
- Build later in the lifecycle
- Build after product is released

Iterative Development

- Develop in small increments a product that always works.
- Develop, Test, Build, and Deploy frequently.
- Avoid integration nightmares.

Grass Roots Agility

- Culture and politics affect adoption of agile methods.
- Everyone agrees on the benefit of an automated and repeatable build that produces a quality product frequently.
- Each of the above points is a step toward agility.

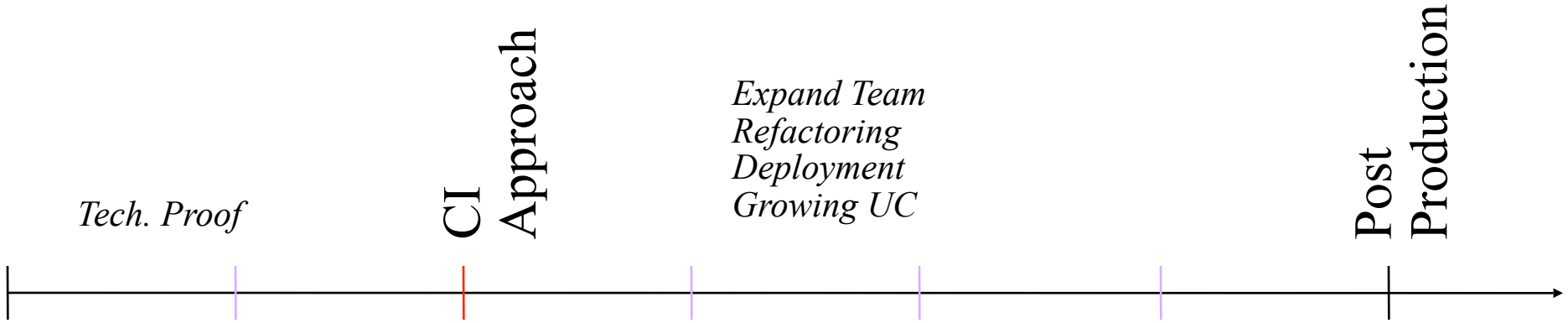
Continuous Integration

- ✓ The Ground Rules
- ✓ The Strategy
- ✓ Positive Affects
- ➡ Case Study

Early Obstacles

- Iterative claim with waterfall execution
- Lack of development infrastructure
- Unproven team
  - Inexperience with the CI approach
- Ill-defined process and few solid practices
- Everyone wanted documentation everywhere

CI Evolution



UC Doc
Client Meetings.

Data Model

Ant Build (twice/wk)
Test Cases
Physical DB
Frequent Demos
Code Reviews
Team Meetings

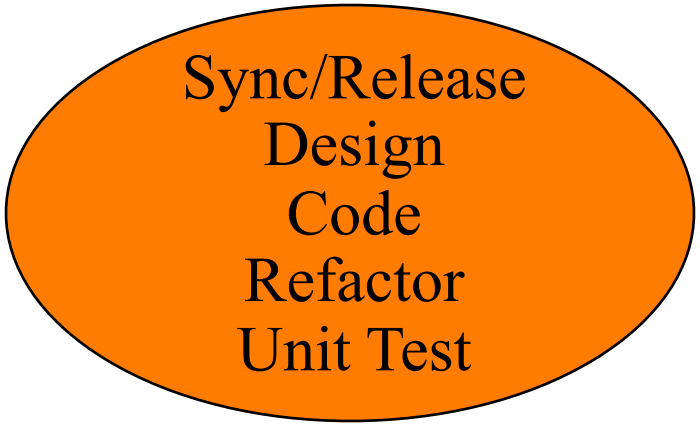
QA testing
Load testing
Defect tracking
Instant Msgr

Little measurable progress
Gain process knowledge
Large amount of doc.
Little dev. infrastructure
Isolated developers

Measurable progress
Heavy development
Heavy collaboration

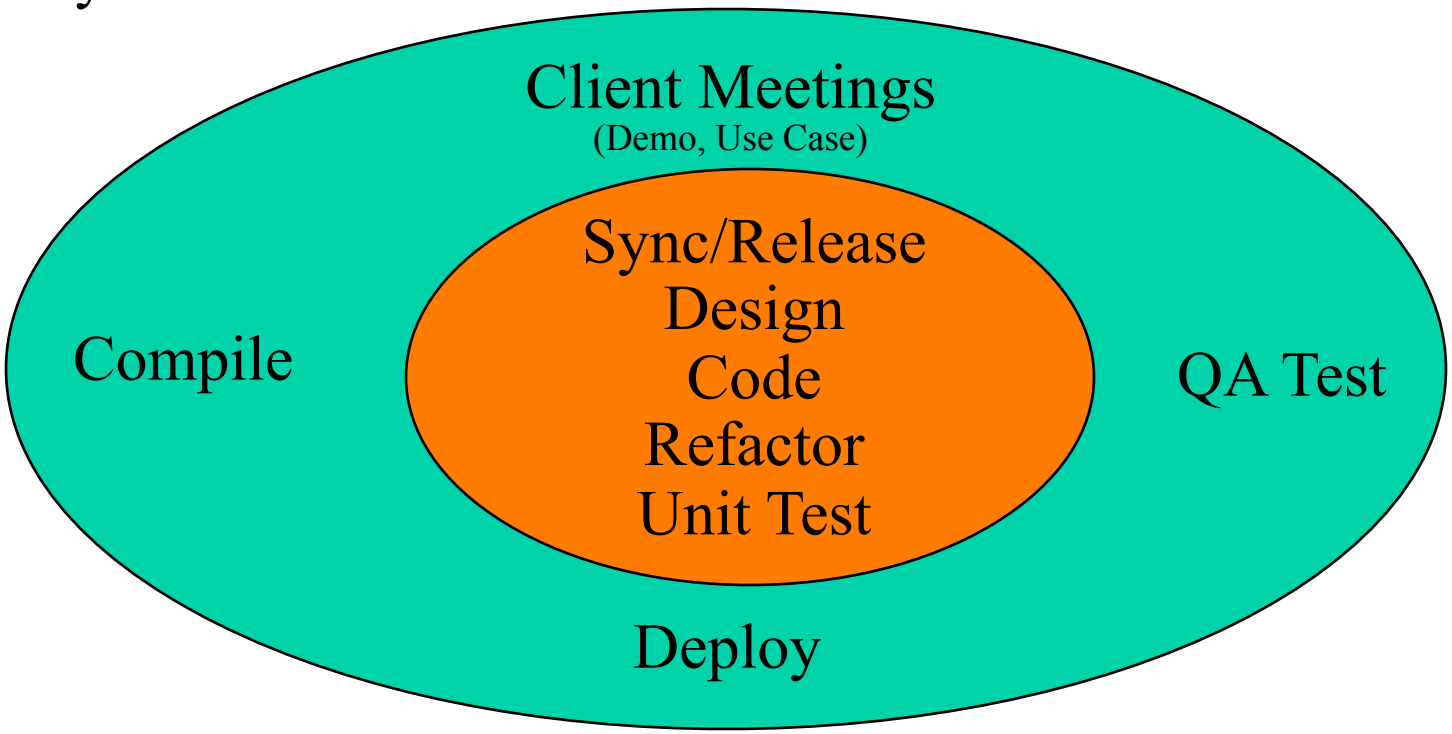
Lifecycle

- Daily Activities



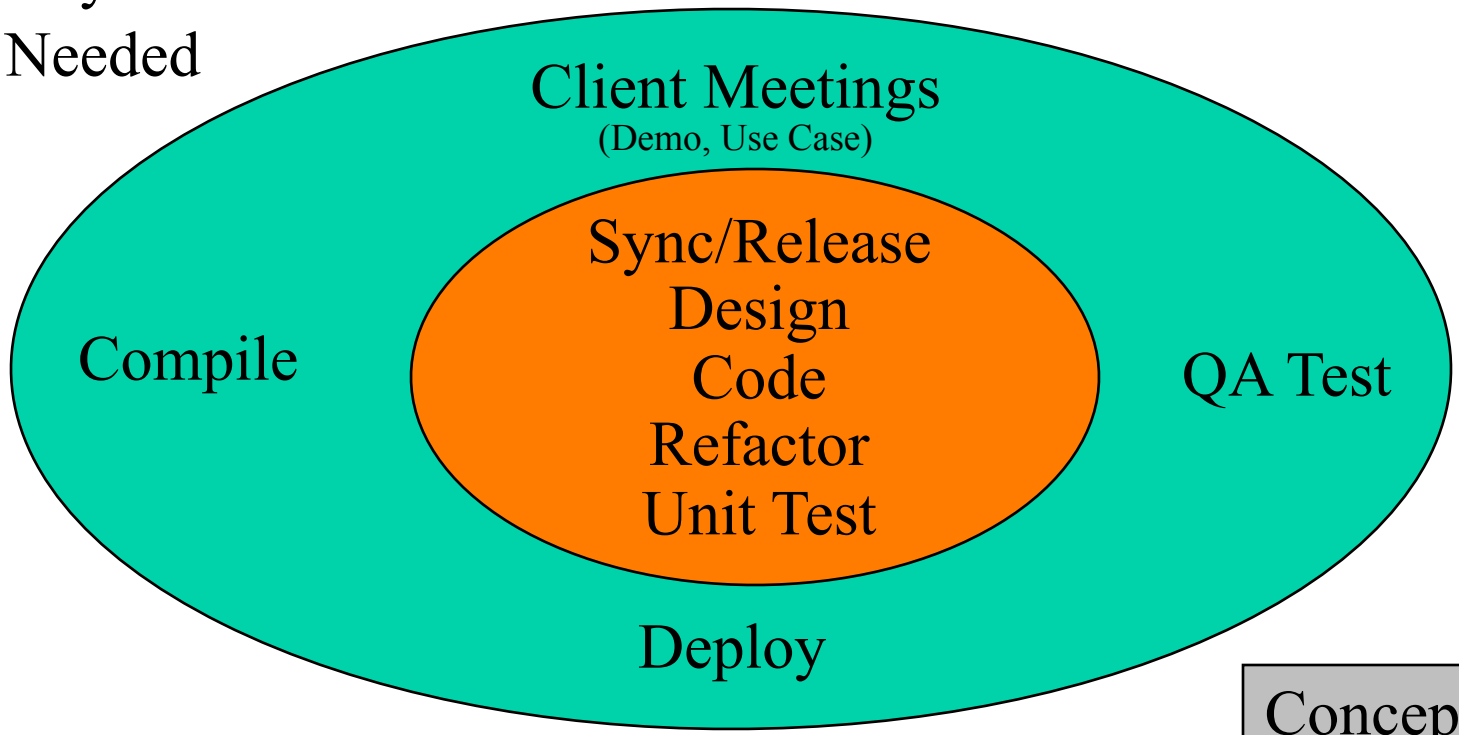
Lifecycle

- Daily Activities
- Weekly Activities



Lifecycle

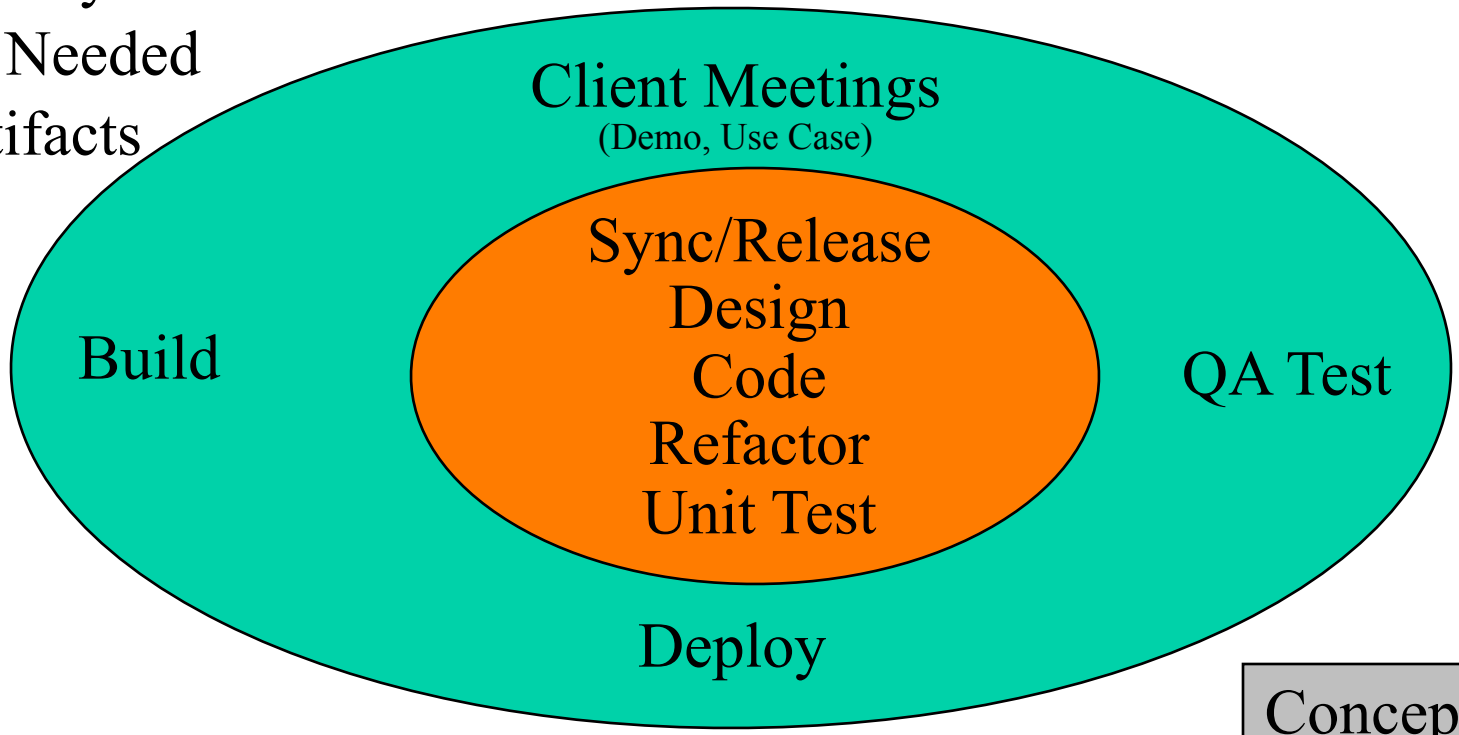
- Daily Activities
- Weekly Activities
- As Needed



Concept Proof
Code Review
Arch. Planning

Lifecycle

- Daily Activities
- Weekly Activities
- As Needed
- Artifacts



Use Case

Concept Proof
Code Review
Arch. Planning

Driving Principle

- Development principles expanded to all team members and activities
- If frequent code/build/test works, do all aspects more frequently

Use Cases

- Never extend/include
- No diagram
- Grew throughout lifecycle (Key Point)
- Maintained by BA
- Updated in meetings
- Features, Flows, and Issues

Modeling

- Only when needed
- Mainly communication
- Short-lived diagrams (Key Point)
  - Little to no maintenance
- High level system model

Design/Coding

- Architectural theme/metaphor **(Key Point)**
- Design sessions when necessary
- Architectural Proof
  - Unproven technology, performance
- Emphasize modularity
  - Physical (ex. Packages and .jars)
  - Logical (ex. IDE Projects)
- End to End development first; rules second

Testing

- JUnit test cases required
  - Verification and test driven design
- Tests must always execute successfully
- QA testing by clients
- Code coverage using Emma

(Key Point)

Builds

- Twice per week (more if needed)
  - Eventually Daily
- Execute full test suite
- Deploy for testing/ensure availability (Key Point)
  - Configuration, performance
- Completely automated & repeatable
- All team members focus on creating a successful build (Key Point)

Team Geography

- All team members on-site **(Key Point)**
- Developers a shout away from each other
- Clients a short walk away
- Instant Messaging
- All communication channels open
  - Project Wiki

Code Reviews

- Verify Compliance
- Identify Bad Practices
- Little emphasis
  - Format, conventions, names, doc
- Major emphasis (Key Point)
  - Exceptions, class responsibility, class relationships, structure, smell
  - PMD Reports run

Client Interaction

- At least weekly meetings
- Each meeting emphasized a Use Case
- From Inception through Post-Deployment **(Key Point)**
- Frequent Demos **(Key Point)**
  - Enabled by build
- Developers heavily involved **(Key Point)**
- Establish UI

Defect Tracking

- Individual defects assigned UID
- Assigned by BA to Developer
- Developer updates defect status
- Project Management Report
- Manage defects and identify change requests

Areas for Improvement

- Code Ownership (per use case)
- Specialization (build master)
- Test cases dependent on external datasources (db and CICS)
- Inconsistent regions
- Automated acceptance tests
- Even more frequent builds

Parting Thoughts

- No process promotion
  - Few members on the team would be able to draw a correlation between our process and RUP/XP
- Those areas where we had the most difficulty were the activities that we performed least frequently