Extracted from:

Pragmatic Project Automation

How to Build, Deploy, and Monitor Java Applications

It claims to be fully automatic, but actually you have to push this little button here.

► Gentleman John Killian

Chapter 3

Scheduled Builds

A one-step build process is a gift that keeps on giving. Every time you push the button that runs a build, it will feel like you're getting something for free. This is the beauty of *commanded automation*. Invest just a wee bit of time and get lots of time back. In this chapter we'll take the next automation step: letting a computer push the build button for us.

Scheduled automation takes the one-step build you created and runs it for you, as often as you want, without you ever lifting a finger. You can still run the build manually if you need to, but in the typical case the computer will do it for you. And it turns out that having a machine running builds continuously does more than just save some typing.

Scheduled builds find both integration (compile time) and failing test (run time) problems quickly because they run at regular intervals. For example, if the schedule says to run a build at the top of every hour, then every 60 minutes you'll know if the build is broken. This makes finding problems easier because you have to look only at changes that occurred in that interval. It also makes *fixing* problems easier because little problems don't have a chance to compound into big problems. And because finding and fixing problems is easier, you're less constrained by fear.

How is a scheduled build any different from, say, all the programmers running the build file every few minutes? Well, I don't know many programmers that want to do that. They've usually got better things to do. The build cycle might take

a few minutes, or even a few hours, and running it interferes with their work. And even if everyone on the team could quickly run a full build, they might deliberately put off doing so because they have a deadline to meet and they're afraid someone else's changes might conflict and cause delays. That is, unlike a scheduled build, programmers typically only build parts of the system at a time rather than testing that the entire system is integrated.

A scheduled build, on the other hand, has nothing better to do than build and test everything. Once you have a one-step build process, you have much to gain by putting it on a schedule for a computer to run. Thankfully, it doesn't cost much to go this extra mile. It *will* end up costing a lot in the end if you don't start scheduling builds early. So let's get cracking!

3.1 Scheduling Your First Build

Scheduling a build is similar to programming the timer that controls your office building's heating system. You want it to start warming up the place before you're out of bed so that you can arrive to a toasty office. In the same way, you want to come into the office with a nice toasty build waiting for you.

Since you can schedule a build to run at a time or frequency of your choosing, why pick just one time every day? You may as well schedule it to run often so you'll know sooner if your world is collapsing. You want to hear those processors grinding as background noise while you're writing code. It's the sound of software being tested. It's the sound of everyone's time being saved. And that's music to our ears.

Scheduling with cron

The easiest way to schedule a build would be to start by writing a script or batch file that does the following:

1. Checks out the current code from version control.
2. Calls your build file to build and test the code.
3. Squirrels away the build results in a log file.

Next you need to run the build script at some predefined time of day (or night). On Unix, the scheduler of choice is cron. To configure cron, type

```
$ crontab -e
```

This pops open your default editor, the computer's subtle way of asking you what you want it to do and when it should be done. Say, for example, you have a build.sh script that runs your Ant build.xml file. You want cron to run that script at 2 a.m. every morning. To appease cron's cryptic syntax, type the following line into your editor and save the file:

```
0 2 * * * $HOME/work/dms/build.sh
```

Each crontab entry is a single line with six fields. The first five fields represent the schedule, starting from the left: the minute (0–59), the hour (0–23), the day of the month (1–31), the month (1–12), and the day of the week (0–6). A * character in any field means to match all possibilities. For example, using * in the third field means that we want it to run every day of the month. The last field specifies the command to run.

If you're on a Windows box, the built-in scheduler is the at command. To schedule the build.bat file to run at 2 a.m. every morning, for example, type the following at the command line:

```
at 02:00 /every: c:\work\dms\build.bat
```

That's really all there is to it! You just scheduled a build. The computer wakes up about the time most authors are going to bed and runs the build, no questions asked.

Picking the Right Tool for the Job

If cron (or at) gets the job done, then why not just use it and move on? It would feel good to check one more thing off the automation checklist. That's a fair question, especially since this is a book about being pragmatic. Creating a continuous build is less about tools than it is about building continuously. We could start with the simplest tool first, then haul out the commercial-grade tools when, and if, we need them.

There's just one problem: Being pragmatic also means using the right tool for the job. And the simplest tool isn't always the right tool. If you start with a simple shell script such

The Cost of Not Integrating Frequently

It seems that many projects don't have, and claim to not be able to afford, a machine dedicated to automatically building and testing their software on a regular interval. Ironically, these same projects can afford to continuously spend time fighting integration and quality problems.

Just how much programmer time does it take to justify the cost of a dedicated build machine? Consider that on average a ten-person development team costs your company at least $500 per hour. If that team spends merely two hours debugging integration problems over the life of the project, you've paid for a respectable build machine fully capable of compiling and testing code. That's a one-time expense. Then when you start to consider that every day your team is debugging integration problems is another day late to market, you just can't afford *not* to have a dedicated build machine.

A dedicated build machine will help your team conserve time for the really important stuff. If you don't already have one on your software project, then you're behind the competition.

as build.sh, it will likely begin as a few commands: check out the project from version control, run the Ant build file, and redirect the build output somewhere useful.

And then you might decide that emailing the build results to the team would be beneficial to let everyone know how things are going. Better yet, why not publish the build result in HTML for viewing in a browser? Oh, and then you will need a web application that shows all previous build results. Before long you're spending more time maintaining your "simple" script than you are writing production code.

That's where being pragmatic comes in again. If you want a build scheduler with all these fancy features, and you can get it for free, then you should use it rather than spend time creating and maintaining your own scheduler. And if that

scheduler is also open source, then you have the option of extending it for any of your special needs later, if necessary.

In that pragmatic spirit, let's take a drive with a scheduler designed to build Java applications. We'll take it one milepost at a time.

3.2 Putting a Build on CruiseControl

CruiseControl[1] is like cron for Ant, but with many bells and whistles. It runs in the background, waking up on cue to run any scheduled Ant targets.

Bear in mind, what CruiseControl does for us isn't rocket science. You could do all this stuff manually if you were bored and didn't mind being pigeonholed as the build guru on your project. It's also nothing a custom build script couldn't do if you wanted to write one and be its maintainer for life. But we're short on time as it is, and maybe even behind schedule. Reinventing all the scheduling features we need that come for free out of the CruiseControl box isn't going to save us any time. CruiseControl isn't the only such tool either, but we'll use it because it meets our needs here.

Choosing a Build Machine

Before installing CruiseControl you need to find a suitable home for it. The machine where you install CruiseControl will be the workhorse for scheduled builds, but it doesn't need to be top-o'-the-line hardware that breaks your bank. You just need it to compile source code and run tests. That's slightly more CPU intensive than reading email and surfing the web, but less so than servicing thousands of concurrent users. If build machines filled out personal want ads, you're looking to hook up with the beautiful bucket of bits described in Figure 3.1 on the following page.

That being said, I realize all the good machines on your project may not be available. If you're lucky enough to find available machines waiting to be put to work, then this decision is easy. Just snag the best one you can and enlist it into service for

[1] http://cruisecontrol.sourceforge.net

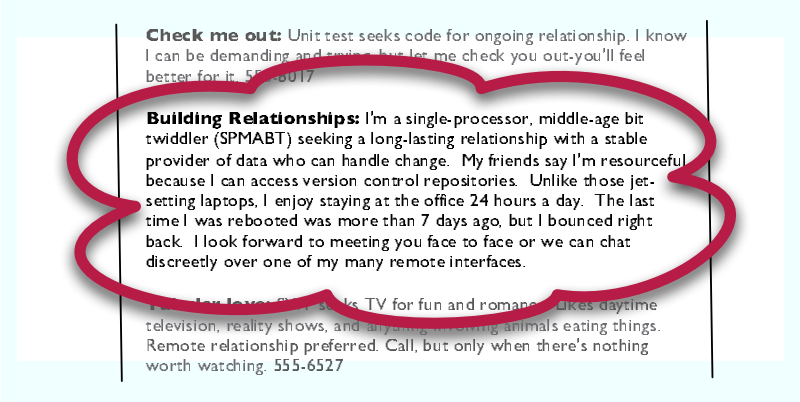

Figure 3.1: WANTED: A DREAM BUILD MACHINE

your project. It's happy to be wanted by someone. If you're not so lucky, then consider two-timing with a machine already in service.[2]

And if you just can't find those spare CPU cycles anywhere on your project, then feel free to mention to your manager how inexpensive good hardware is these days. This will go over better than mentioning how expensive programmers are in comparison.

Installing CruiseControl

Now that you've found a suitable build machine, you're ready to introduce it to CruiseControl. This is somewhat like making a new friend only to turn around and offer him a shovel, but trust that we have good intentions here.

When you download CruiseControl, you get a ZIP file. Extract this file into a directory which we'll refer to as `$CC_HOME`. Then you need to build CruiseControl; on Unix type

```
$ cd $CC_HOME/main
$ sh build.sh
```

[2] To temporarily convert a PC into a Linux box without reconfiguring the PC, check out Knoppix (`http://www.knoppix.net`). It's a Linux distribution that boots and runs completely from a CD. Presto, change-o!

CruiseControl.NET

If you're writing code on the Microsoft .NET platform and using NAnt to build your project, here's another opportunity to follow along. CruiseControl.NET* is a feature port of CruiseControl to the .NET platform. It integrates with the NAnt build tool and the NUnit unit-testing framework. And we'd be remiss if we didn't mention the optional CCTray utility that shows a green or red build status icon in your Windows system tray.

*http://ccnet.thoughtworks.com

Under Windows, the commands are similar.

```
$ cd %CC_HOME%\main
$ build.bat
```

The script then compiles and tests CruiseControl. (Notice that this is commanded automation at work.) When it's done, you'll end up with a file called cruisecontrol.jar in the directory $CC_HOME/main/dist. That file needs to be there to run CruiseControl later.

Preparing a Build Workspace

Next, you need to prepare a workspace on the build machine. This will be the directory from which CruiseControl will run builds and store the results. We'll walk through creating the workspace step by step.

Create the Build Directory

The build workspace is a directory on the build machine. Let's assume we call that directory builds because it's the workspace for all of our scheduled builds. The easiest approach is to create the builds directory in some user's home directory on the build machine. On Unix, log in as that user and type

```
$ mkdir ~/builds
```

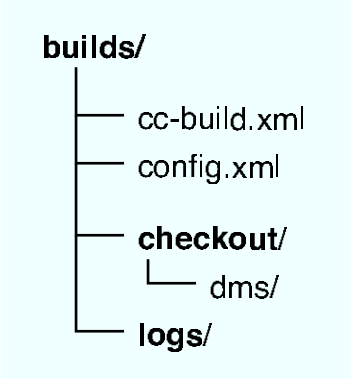

Figure 3.2: THE BUILD WORKSPACE

Check Out the Project

So far, we only have one project to build on a schedule: our *DMS* project. It's safely stored in CVS and needs to be checked out locally for CruiseControl to use. To keep the top-level builds directory tidy, check out the dms module into a checkout subdirectory.

```
$ cd ~/builds
$ mkdir checkout
$ cd checkout
$ cvs co dms
```

This assumes that your CVSROOT environment variable is set to the location of your CVS repository. After running these commands the checkout/dms directory will contain all of the files in the dms project. This is a local copy of the project—a snapshot of the project at this instant of time. We'll use this directory just to prime the scheduled build process.

Create a Log Directory

Finally, create a directory that will contain all of the CruiseControl build log files.

```
$ cd ~/builds
$ mkdir logs
```

Now you have a cozy workspace for scheduled builds. Figure 3.2 shows the directory structure just created. Next, you

need to create the cc-build.xml and config.xml files shown in that directory structure. We'll start by writing the cc-build.xml file.

Writing a Delegating Build File

When your scheduled build runs, it should

1. Delete the last build.
2. Check out the current project from CVS.
3. Run the build.

That is, we want to run a "scorch-the-Earth" build. Starting from scratch each time helps avoid the strangeness that inevitably happens with incremental builds. When a build runs successfully from scratch, you get more confidence that it's complete. And if a machine is going to run the build for us, we can afford to be spendthrifts with its time.

You could put those three steps in a target of the existing build.xml file. But it's a good idea to keep the scheduled build procedure separate from the Ant build file used to run builds manually. To do that, create a separate Ant build file called cc-build.xml in the builds directory. The cc-build.xml file just sets up the checkout directory with a fresh copy of code and then delegates the build procedure to the build.xml file.

builds/cc-build.xml

```
<project name="cc-build" default="build" basedir="checkout">
  <target name="build">
    <delete dir="dms" />
    <cvs command="co dms" />
    <ant antfile="build.xml" dir="dms" target="test" />
  </target>
</project>
```

The syntax of this build file should look familiar. It defines an Ant project with `build` as the default target to run. The `basedir` attribute points to the checkout directory that contains a local copy of the project.

The meat of the cc-build.xml file is the `build` target. It first deletes the copy of the project used during the last build to ensure that the next build starts from scratch.

```
<delete dir="dms" />
```

It then checks out a fresh local copy of the project from the CVS repository into the checkout/dms directory.

```
<cvs command="co dms" />
```

This form of the <*cvs*> task uses the value of the CVSROOT environment variable to locate your CVS repository. Alternatively, you can set the CVSROOT in the cvsRoot attribute of the <*cvs*> task.

Using the repository as the sole source for the build process means that all the build inputs need to be in CVS. The computer will use the lack of any required file as an excuse for not making good builds. For example, it won't tolerate having to find files littered across the filesystem or the network. Using a version control system also means that any machine with access to the repository is a candidate for running builds.

The build target then needs to call the project's build file to compile and test everything.

```
<ant antfile="build.xml" dir="dms" target="test" />
```

This is where having a one-step build process really pays off. The <*ant*> task calls the test target of the build.xml file located in the checkout/dms directory.

Test the Procedure

After writing the cc-build.xml file, it's a good idea to test it before handing it off to a cranky computer. To verify the delegating build file works, type

```
$ cd ~/builds
$ ant -buildfile cc-build.xml
```

Make sure to use the -buildfile option here to specify the cc-build.xml file, since by default Ant will look for a file called build.xml. Alternatively, you can use -f as an abbreviation for -buildfile.

Save the Delegating Build File

You need to store the cc-build.xml file under version control so you don't lose it. This presents a slight conundrum because the build file checks out the project from CVS, and yet it's in CVS itself. But cc-build.xml isn't likely to be updated all that often, so just manually check out cc-build.xml into your

builds directory whenever it's changed. This is another benefit to using a separate build file for CruiseControl builds, rather than just adding a target to the main build file.

All we've done here is created a wrapper around our existing build file: build.xml is wrapped by the cc-build.xml file. This delegating build file checks out the project and builds it, just as you'd do it from the command line.

Configuring the Build Process

Think of the cc-build.xml file as playing the role of any new programmer on the team. They show up with an empty directory, check out the project anew, and build it with the expectation that everything will work. That is, they provide an objective second opinion as to whether the builds are successful. Unfortunately, no one can hire enough new programmers to get build feedback as often as needed in order to keep working confidently. That's where CruiseControl comes in.

Our next step is to tell CruiseControl how and when it should run our build. By default, it looks for a configuration file called config.xml that defines the projects it's responsible for building. We'll write the config.xml file one section at a time. The complete file is shown in Figure 3.3 on page 58.

Define the Project

Create the config.xml file in the builds directory. The first few lines of config.xml set up the project.

builds/config.xml

```
<cruisecontrol>
  <project name="dms" buildafterfailed="false">
```

The name attribute of the <*project*> element identifies this project. Multiple projects can be defined in this file with each project having a unique name.

By default, CruiseControl will continue to attempt to build a project even if the build failed on the last attempt and nothing has changed in CVS since then. This can be useful for projects that have dependencies on external resources that might not be available when the build runs: If at first you don't succeed, try and try again. But it's overkill for this project since everything it depends on is in the CVS repository. Set the value of

the `buildafterfailed` attribute to `false` so that when the build fails the CPUs will get a chance to cool down while you fix the problem.

Bootstrap the Build

Next, define *bootstrappers*—things to be done before the build cycle happens. *bootstrappers*

```
<bootstrappers>
  <currentbuildstatusbootstrapper
    file="logs/dms/currentbuildstatus.txt" />
</bootstrappers>
```

The <*currentbuildstatusbootstrapper*> simply writes a message to the logs/dms/currentbuildstatus.txt file indicating that a build cycle has begun. Running a bootstrapper doesn't mean that a build will be attempted, only that CruiseControl has awakened to check if a build is necessary. Think of it as CruiseControl punching in for work.

Check for Changes

You want to run a build only if something has changed in the CVS repository. After all, there's no sense running builds if all the programmers are away at a conference honing their skills. Next define how CruiseControl checks for changes to determine if a build is necessary.

```
<modificationset quietperiod="60">
  <cvs localworkingcopy="checkout/dms" />
</modificationset>
```

The <*modificationset*> element tells CruiseControl what to watch to see if a build is required. The project is in CVS, so you can use the <*cvs*> element with the `localworkingcopy` attribute pointing at the local copy of the dms module.[3] This means that the local directory will be used to locate the CVS repository to determine if something has changed. This keeps you from having to hard-code the CVSROOT in the config.xml file. The important thing to remember is that a build

[3] ClearCase, Subversion, StarTeam, Visual SourceSafe, and other version control systems are also supported.

will be attempted only if something being watched by the <*modificationset*> has changed.

CVS doesn't support atomic commits, which means if you check in 10 files they are committed in 10 separate steps. What happens when 5 of 10 changes have been committed when CruiseControl wakes up? It will notice that at least 5 things have changed since the last time it looked at the repository. But starting a build at this point would be problematic because not everything has made it into the repository.

To give you a chance to get everything checked in before a build starts, set the value of the <*modificationset*> element's `quietperiod` attribute to 60 seconds. This means the CVS repository must be quiet (inactive) for 60 seconds before a build is attempted. If CruiseControl wakes up and detects that changes have been made to the repository during the quiet period, it will go back to sleep and check again later.

Dial In the Build Interval

Finally, define the build interval and how a build should be attempted.

```
<schedule interval="60">
  <ant buildfile="cc-build.xml" target="build" />
</schedule>
```

The <*schedule*> element tells CruiseControl *when* to attempt a build. Here, we set the `interval` attribute to 60 seconds. This means CruiseControl will wake up every minute to check to see if any changes have been made as indicated by the results of the <*modificationset*> element. In other words, the dms module of CVS will be polled every minute for differences. If changes were made, but not within the quiet period, then a build will be attempted.

The <*ant*> element tells CruiseControl *how* to run a build. In this case, we want it to invoke the `build` target of our delegating build file—cc-build.xml. Recall that this build file will delete the last build, check out a fresh copy of dms from CVS, and then run the `test` target of the checkout/dms/build.xml file.

To recap what we've done here: Every minute CruiseControl will check to see if something in the project has changed. If

Joe Asks...

How Frequently Should a Build Run?

The only limiting factor to how often you can run the build is the length of your build cycle. Some projects may not even finish the compile step in under a minute. But if we can build the entire project in less than five minutes, for example, then why not build every five minutes?

Remember, if nobody changes code, the build just doesn't run. But if somebody does change code, then wouldn't it be nice to know as soon as possible if all of the tests still pass? If they didn't pass, then you'd only have to look at the last five minutes worth of changes to diagnose what went wrong.

On a real-world project you'll probably have different types of tests: unit tests, acceptance tests, performance tests, etc. You don't want to wait for all of those tests to run just to see if your unit tests passed. To avoid that, each type of test would have a different Ant target and you'd configure CruiseControl to run each target on a different schedule.

Schedule build targets to run based on how often you want feedback about your system. For example, you might run all the unit tests every five minutes, all the acceptance tests every hour, and all the performance tests once a day. It's a game of confidence and this computer is here to help you feel better.

so, the system will be rebuilt and all of our tests will be run, using the latest code. Now that's automation!

Save the Logs

CruiseControl generates a log file every time it attempts a build. It's a good idea to save those files so that you can check on the build results later.

```
<log dir="logs/dms">
  <merge dir="checkout/dms/build/test-results" />
</log>
```

We'll use the logs directory created earlier as the dumping ground for log files. The dms subdirectory will be created to hold the dms project's log files. That is, the build log files are stored in a directory that isn't deleted every build cycle.

In addition to the log files that CruiseControl generates, you also want each build log to include the results of JUnit tests. Unfortunately, the test output is currently being displayed only on the console. You need to create a new `test` target that, when run, will also output the JUnit test results as XML files in the build/test-results directory of the project. This directory is used as the value of the `dir` attribute of the <*merge*> element. CruiseControl will then merge the contents of that directory into the build log.

Generate Test Results As XML

When we ran our tests from the command line in the previous chapter, they output messages to the console. But when a scheduled build is run, nobody will be watching the console. We need to capture the test results in a format that can be displayed to us later in the CruiseControl build log.

Let's revisit the build.xml file and define a new build target that will run the tests and send the output to XML files.

```
<target name="test" depends="compile-tests">
  <delete dir="${test.xml.dir}"/>
  <mkdir dir="${test.xml.dir}"/>
  <junit errorProperty="test.failed"
         failureProperty="test.failed">
    <classpath refid="project.classpath" />
    <formatter type="brief" usefile="false" />
    <formatter type="xml" />
    <batchtest todir="${test.xml.dir}">
      <fileset dir="${build.test.dir}"
        includes="**/*Test.class" />
    </batchtest>
    <sysproperty key="doc.dir" value="${doc.dir}" />
    <sysproperty key="index.dir" value="${index.dir}" />
  </junit>
  <fail message="Tests failed! Check test reports."
    if="test.failed" />
</target>
```

This `test` target is similar to the `test` target from the last chapter, but has a few important differences. First, it always

creates an empty directory to hold the JUnit test results.

```
<delete dir="${test.xml.dir}"/>
<mkdir dir="${test.xml.dir}"/>
```

The `test.xml.dir` property, defined in the properties section of the build.xml file, points to the project's build/test-results directory. This is the directory that CruiseControl uses as the source for merging test results into the build log.

Instead of halting on the first test failure, the <*junit*> task sets a `test.failed` property on either an error or a failure.

```
<junit errorProperty="test.failed"
       failureProperty="test.failed">
```

This makes sure that all the tests results—successes and failures—are collected in XML files. Notice that later in the file we use the `test.failed` property in the <*fail*> task to alert us if one or more tests failed.

To output test results to the console and to XML files, define both a `brief` and an `xml` formatter.

```
<formatter type="brief" usefile="false" />
<formatter type="xml" />
```

The <*batchtest*> task needs to be changed to include a `todir` attribute. This attribute defines the output directory for the XML files generated by the XML formatter.

```
<batchtest todir="${test.xml.dir}">
  <fileset dir="${build.test.dir}"
    includes="**/*Test.class" />
</batchtest>
```

Now we have a new `test` target that generates XML files, in addition to showing test results on the console. CruiseControl will use those XML files when it generates a build log. This feature will come in handy later when we send the build status to the team.

Publish Build Results

Finally, back in the config.xml file, you need to specify *publishers*—things to be notified after the build cycle happens.

publishers

```
<publishers>
  <currentbuildstatuspublisher
    file="logs/dms/currentbuildstatus.txt" />
</publishers>
```

```
<cruisecontrol>
  <project name="dms" buildafterfailed="false">
    <bootstrappers>
      <currentbuildstatusbootstrapper
        file="logs/dms/currentbuildstatus.txt" />
    </bootstrappers>
    <modificationset quietperiod="60">
      <cvs localworkingcopy="checkout/dms" />
    </modificationset>
    <schedule interval="60">
      <ant buildfile="cc-build.xml" target="build" />
    </schedule>
    <log dir="logs/dms">
      <merge dir="checkout/dms/build/test-results" />
    </log>
    <publishers>
      <currentbuildstatuspublisher
        file="logs/dms/currentbuildstatus.txt" />
    </publishers>
  </project>
</cruisecontrol>
```

builds/config.xml

Figure 3.3: CRUISECONTROL CONFIGURATION FILE

The *<currentbuildstatuspublisher>* publisher simply writes a message to the logs/dms/currentbuildstatus.txt file indicating that the build cycle has finished. Similar to bootstrappers, the publishers are run regardless of whether a build was actually attempted. Think of this as CruiseControl punching out after a hard interval's work.

You've passed the test. You're now licensed to drive on CruiseControl! Figure 3.3 shows the complete config.xml file.

OK, so that configuration exercise wasn't a leisurely Sunday drive, especially compared to the one-liner you wrote for cron. But from this point, you can easily get a lot more than cron offers. Moreover, now that you've configured CruiseControl for the first time, you can apply the same steps to put your other projects on a schedule.

3.3 Running CruiseControl

With the configuration file that tells CruiseControl everything it needs to know to run our build process in hand, we're ready

to see some action! First, navigate to the builds directory that contains the config.xml and cc-build.xml files. Then run the CruiseControl script. On Unix, the commands are

```
$ cd ~/builds
$ $CC_HOME/main/bin/cruisecontrol.sh
```

Under Windows, the slashes swing around.

```
$ cd \builds
$ %CC_HOME%\main\bin\cruisecontrol.bat
```

CruiseControl will start up, read the config.xml file, and go right to work.

Starting Up

When CruiseControl starts up, the output can be verbose. It likes to let us know it's doing something useful as a result of our configuration effort. Here's the important information:

```
projectName = [dms]
Project dms:  reading settings from config file
              [/Users/mike/builds/config.xml]
Project dms starting
Project dms:  next build in 1 minutes
Project dms:  idle
```

If the output you see doesn't look so hopeful, then perhaps you need to tweak your config.xml file. Thankfully, you don't have to restart CruiseControl to change the configuration. It will reload the config.xml file every time a build cycle starts. You can make any necessary changes and simply wait another minute for it to notice.

Then You Wait...

Now wait patiently as 60 long seconds go by. CruiseControl then wakes up on schedule to check if there's any work.

```
Project dms:  in build queue
Project dms:  reading settings from config file
              [/Users/mike/builds/config.xml]
Project dms:  bootstrapping
Project dms:  checking for modifications
Project dms:  2 modifications have been detected.
Project dms:  now building
```

When it wakes up, it first reloads the config.xml file. Then it checks the CVS repository and finds that something has been modified. This being the first build cycle, CruiseControl may

or may not detect changes in your repository. It needs to establish a baseline and you may have to change a file in your repository to force CruiseControl to run a build. Assuming it detects a change, it's ready to run the build.

...Until a Build Is Attempted

This is where you finally get to experience the fruits of your labors. At long last, you will see the one-step build process get run automatically by the computer.

```
Buildfile: cc-build.xml

build:
[delete] Deleting directory /Users/mike/builds/checkout/dms
[cvs] Using cvs passfile: /Users/mike/.cvspass
[cvs] U dms/README
[cvs] U dms/build.xml
...
prepare:
[mkdir] Created dir: /Users/mike/builds/checkout/dms/build/prod
[mkdir] Created dir: /Users/mike/builds/checkout/dms/build/test

compile:
[javac] Compiling 4 source files to /Users/.../dms/build/prod

compile-tests:
[javac] Compiling 3 source files to /Users/.../dms/build/test

test:
[junit] Testsuite: com.pragprog.dms.SearchTest
...
BUILD SUCCESSFUL
```

A lot happened here. The `build` target of the cc-build.xml file ran. It deleted the checkout/dms directory and then re-created it by checking out the dms project from CVS.

Then the `test` target of the build.xml file ran. That Ant target has dependencies on other targets, such as the `compile` target. As you'd expect, all the dependent targets are run prior to running the tests. And miracle of miracles, the project built successfully!

Having run the build, CruiseControl records the results in a log file, notifies the publishers that indeed it showed up for work on time, and then promptly goes back to sleep.

```
Project dms:  merging accumulated log files
Project dms:  publishing build results
Project dms:  idle
Project dms:  next build in 1 minutes
```

Once CruiseControl is started, it keeps running regardless of whether the last build succeeded or failed. It awakens on cue to check if a build is necessary, and if so goes about the

Joe Asks...

What About Anthill?

Another build scheduler that's definitely worth exploring is Anthill.* It's available in either an open-source version (Anthill OS) or, for those who need some chrome under the hood, there's Anthill Pro.

Opinions vary as to whether CruiseControl or Anthill is easier to install and configure. It really depends on what you consider easy. To run Anthill you deploy a WAR file into your favorite servlet engine and then configure it through a web interface. CruiseControl, on the other hand, can be configured and run via the command line without ever firing up a servlet engine. It's easier to demonstrate scheduled builds using CruiseControl as it doesn't require a servlet engine.

Remember, the choice of a tool isn't as important as getting your build scheduled as soon as possible. So use whatever tool helps you do that.

*http://www.urbancode.com/projects/anthill

business of attempting a build. Then it goes to sleep until the next build interval. Rinse and repeat. It's a pretty dull life, which is exactly why we're happy not to be doing it ourselves.

Now It's Your Turn

CruiseControl is now in its rhythmic build loop waiting for us to do what we're paid to do. Every minute it wakes up, notices that we haven't touched anything in CVS, and goes back to sleep.

```
Project dms:  No modifications found, build not necessary.
Project dms:  idle
Project dms:  next build in 1 minutes
```

And it's happy to just keep doing this and enjoying a life of leisure. But we're not going to stand for that kind of lackadaisical behavior—we want to see if it's really watching the CVS repository and not asleep at the switch.

In the ~/work directory, there is a checked-out local copy of the dms project. Now we'll change a Java source file. But suppose in our haste we unknowingly introduce a bug. Worse yet, we forget to run our unit tests before checking in the modified source file.

```
$ cd ~/work/dms
$ emacs src/com/pragprog/dms/Search.java
(Hack, hack, hack)
$ cvs commit -m "I'm too busy to test"
```

Now we wait around for the build timer to pop. When it does, CruiseControl checks for work. Again, the output is verbose, but it vaguely resembles the following:

```
1 modification has been detected.
Project dms:  now building
Buildfile: cc-build.xml
build:
[delete] Deleting directory /Users/mike/builds/checkout/dms
[cvs] Using cvs passfile: /Users/mike/.cvspass
[cvs] U dms/README
[cvs] U dms/build.xml
...
prepare:
...
compile:
...
compile-tests:
...
test:
[junit] Testsuite: com.pragprog.dms.SearchTest
[junit] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 1.957 sec
[junit] Testcase: testTitleSearch(com.pragprog.dms.SearchTest): FAILED
[junit] expected:[2]  but was:[0]
...
BUILD FAILED
```

Uh oh! We just got busted. CruiseControl can't do much for us other than record the failure in the log file and tell us to fix things before the next build interval. Rest assured, we won't have to spend our days monitoring the build machine. We'll automate the notification of a build failure through email a bit later and explore advanced monitoring techniques in Section 6.1, *Monitoring Scheduled Builds*, on page 125.

What a Scheduled Build Is Good For

We got sloppy. It happens to the best of us from time to time, so we need somebody looking over our shoulder. In this case, that somebody is CruiseControl. It noticed that a modification was made to the CVS repository, and it attempted to run the tests against those changes. But the test is expecting one

value and got another, so it fails. It's not the ideal situation, but at least you now know there's a problem and you can fix it before it turns into a costly problem later.

Now before you do anything else short of breathing, you need to get the build back to a steady state. Make the necessary changes to the local copy of the project in the ~/work/dms directory. And run the tests before checking in this time! Then sit back and wait for the next build interval.

A minute later CruiseControl builds the project and confirms that indeed you're still the world's greatest programmer. Better yet, it will continue watching for changes and running all the tests while you're off doing what you're good at—writing programs.

3.4 Publishing the Build Status

The build is now running on a schedule, but you're missing something important. Unless a real, live human watches the console output of CruiseControl, you won't know when the build breaks.

When a build fails, we'd like something to send up a flare, sound the alarm, and start brewing a fresh pot of coffee. Failing all that, an email will do.

Sending Build Results via Email

We have a lot of options when it comes to who gets what kind of email, but let's keep it simple. We're interested only in getting an email when the build fails and when it has been fixed. And once we get an email that tells us the build has failed, we don't care to continue getting more email until we're back on stable ground. Less is more in this case. If we're constantly being bombarded with build email, we'll stop reading them. It's like signing up for a newsgroup. All the posts are interesting...for the first day.

Notification by email is relatively easy with CruiseControl.[4] Just add an email publisher.

[4] See [HL02] for details on how to send a build failure email using Ant.

```
<htmlemail mailhost="your.smtp.host"
  returnaddress="cruisecontrol@clarkware.com"
  defaultsuffix="@clarkware.com"
  buildresultsurl="http://localhost:8080/cruisecontrol/buildresults/dms"
  css="/Users/mike/tools/cruisecontrol/reporting/jsp/css/cruisecontrol.css"
  xsldir="/Users/mike/tools/cruisecontrol/reporting/jsp/xsl"
  logdir="logs/dms">
  <map alias="manager" address="bigcheese@clarkware.com" />
  <map alias="mike" address="mike@clarkware.com" />
  <map alias="fred" address="fred@somewhere.com" />
  <always address="manager" />
  <failure address="mike" reportWhenFixed="true" />
  <failure address="fred" reportWhenFixed="true" />
</htmlemail>
```

Add this <*htmlemail*> element inside the <*publishers*> element of config.xml. Even though the build is failing, we'd like the email to be nicely formatted HTML. Figure 3.4 on the next page shows what arrives in our inbox. Notice that it includes test failure details because we merged our JUnit test results into the CruiseControl log. It also lists all the modifications that were made—and who made those modifications!—since the last successful build. Perhaps you know where the guilty party lives.

In the interest of sanity, we're going to gloss over the details of email configuration here. Most of it is self-explanatory. However, there are a few things worth noting, starting from the top:

```
<htmlemail mailhost="your.smtp.host"
  returnaddress="cruisecontrol@clarkware.com"
  defaultsuffix="@clarkware.com"
  buildresultsurl="http://localhost:8080/cruisecontrol/buildresults/dms"
  css="/Users/mike/tools/cruisecontrol/reporting/jsp/css/cruisecontrol.css"
  xsldir="/Users/mike/tools/cruisecontrol/reporting/jsp/xsl"
  logdir="logs/dms">
```

The `logdir` directory points to the directory CruiseControl uses for saving each build log. To format the email, it applies a style through the formatting wonders of the `css` and `xsldir` attributes to the latest build log. If you don't particularly like the default email format, you have the power of CSS and XSLT at your fingertips.

Next, create email aliases for each user that should receive an email.

```
<map alias="manager" address="bigcheese@clarkware.com" />
<map alias="mike" address="mike@clarkware.com" />
<map alias="fred" address="fred@somewhere.com" />
```

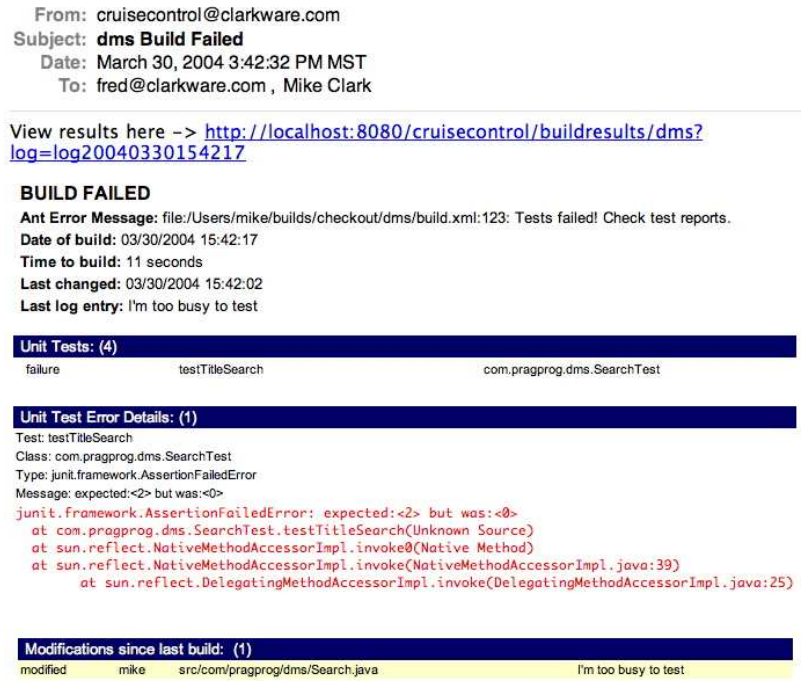

Figure 3.4: BUILD FAILURE EMAIL

Using the <*map*> element, each member of our team is associated with their corresponding email address. Without any mappings in place, CruiseControl will use the CVS username and the value of the `defaultsuffix` attribute. In this example, it's not necessary to map "mike" to "mike@clarkware.com" if "mike" is a CVS user. That's taken care of when the email publisher applies the value of the `defaultsuffix` attribute. Our manager needs to have an email address mapped because he wants email, but he's not a CVS user. And Fred wants his email sent to an address different from that in the `defaultsuffix` attribute, so we have to define a specific mapping for him.

By default, CruiseControl sends email on a success or a failure to those folks who checked stuff in since the last successful build. We can get a bit more control by defining exactly what kind of email each mapped user receives.

```
<always address="manager" />
<failure address="mike" reportWhenFixed="true" />
<failure address="fred" reportWhenFixed="true" />
```

Using the <*always*> element, we make sure our manager gets an email for both successful and failed builds. That just happens to be his preference. He doesn't make changes to CVS, so we need to explicitly declare him as an email recipient.

All the programmers should know when the software *isn't* building. As a team, *we* need to get it fixed pronto. (Oh, and a little peer pressure goes a long way on some teams.) Use the <*failure*> element to list programmers as recipients of email when the build fails. It's also important for the programmers to know when the build is fixed, so set the value of the `reportWhenFixed` attribute to `true` to get those emails as rewards for fixing the build. You may want to set up an alias in your email system for all the developers on your team and send an email to that alias when the build fails and when it's fixed.

You may have noticed that the build status email includes a "View results here" hyperlink at the top. Let's see what that's all about.

Pulling Build History from a Web Page

It's nice to have the build status forwarded via email. But when it comes time to debug build failures, it's also convenient to have a historical record of all the builds. When you need that information, you can pull it from a web page.

The standard CruiseControl distribution includes an optional web reporting project in the $CC_HOME/reporting/jsp directory. Building this project creates a WAR file that can be dropped into your favorite servlet engine, such as Tomcat.

Build and Deploy the Web Application

First, you need to define three properties that tell the web application where to find files and directories in your build workspace. In the $CC_HOME/reporting/jsp directory, create a file called override.properties that defines the following properties (substitute your absolute builds directory):

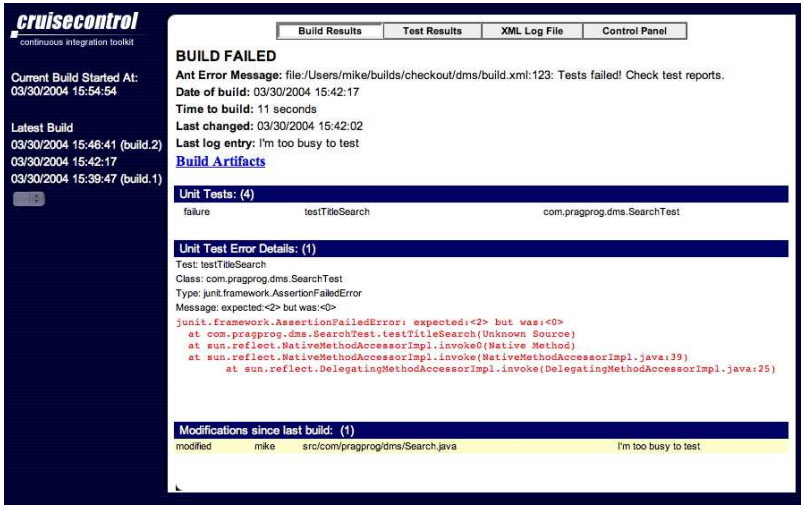

Figure 3.5: BUILD HISTORY WEB PAGE

```
user.log.dir=/Users/mike/builds/logs
user.build.status.file=currentbuildstatus.txt
cruise.build.artifacts.dir=/Users/mike/builds/logs
```

Next, build the web application. On Unix type the following:

```
$ cd $CC_HOME/reporting/jsp
$ sh build.sh war
```

This incantation creates a cruisecontrol.war file in the directory $CC_HOME/reporting/jsp/dist. Deploy this WAR file into your server. If you're using Tomcat on Unix, for example, type

```
$ cp dist/cruisecontrol.war $TOMCAT_HOME/webapps
```

View the Build History

With your server running and the CruiseControl web application deployed, click the hyperlink at the top of a build results email or browse to

```
http://buildmachine:port/cruisecontrol/buildresults/dms
```

This will take you to a web page similar to the one shown in Figure 3.5.

Along the left side of this page is a list of all the builds that were attempted. Clicking any build shows the details you see in the right area. This is the same information you'll see in emails sent by the <*htmlemail*> publisher.

Now you have build results being pushed via email, and anybody with access to the web server can actively pull a detailed history of builds from a web page. That's a good start, but in Section 6.1, *Monitoring Scheduled Builds*, on page 125 we'll explore how to get feedback about builds in other cool and exciting ways.

3.5 Scaling Up

If while reading this chapter you've been wondering if CruiseControl can handle all the code in your Java project, then wonder no longer. Here's a glimpse of CruiseControl on a massive, real-world project:

CruiseControl on a Large Scale
by Jared Richardson, Software Manager, SAS Institute

Many people think that open-source projects can't scale to the enterprise level, but CruiseControl is an example of one that does. This is our success story of how flexible and extensible CruiseControl is.

We have approximately 800 developers working on more than 250 projects with five million lines of Java code. Some of these projects are very low-level components, some are portlets, and some are end-user solutions. We were able to get all five million lines of Java code under continuous integration using CruiseControl relatively easily. In fact, as I type this, we are covering three code branches, so we are really covering 15 million lines of code, and the CruiseControl box is a single CPU x86 machine.

We used a few tricks to get CruiseControl running at the enterprise level. First, we multithreaded CruiseControl ourselves. (Those changes should be in the next release of CC.) This is one of the advantages of working with an open-source project!

Next, instead of using the regular CVS modification set, we are using the *compound modification set*. It contains a *trigger* that initiates the build and a *target* that is used to actually get the

> file changes. For our trigger, we use the filesystem modification set. When a project changes in CVS, a CVS trigger touches a single file that CruiseControl is monitoring. This prevents CruiseControl from trying to poll CVS every ten minutes for changes in 15 million lines of code. Once it sees that a project trigger file has changed, it uses the regular CVS modification set—the target in the compound modification set—to see exactly what changes were made.
>
> Will Gwaltney, another SAS employee, wrote the compound modification set, and we contributed it back to the CruiseControl project. Now anyone can use a compound modification set, and you can use any of the CruiseControl modification sets as either triggers or targets.
>
> We use one trick that isn't stock. We have a build grid at SAS that has a number of machines behind it. We are able to ask it to do the builds for us, and it finds an available machine. This keeps the load of building the systems off the CruiseControl box.
>
> All in all, CC was very easy to roll out and is now part of the standard Java development experience at SAS. With very little effort, you can get this same type of coverage at your company, no matter the size of the code base.

That's right, Jared, no project is too big to be built on a schedule! Indeed, the more code you have, the more you need continuous integration to keep it in check. After all, would you want to be running builds of that proportion manually to get confidence that it's always working? You might have to use a few clever tricks, but it's well worth it in the end. And with CruiseControl, you already have a powerful scheduler that's free. Of course, this is just the beginning of what CruiseControl can do. To learn more, visit the CruiseControl wiki.[5]

What We Just Did

We've come a long way in this chapter. We started with a one-step build process that we previously ran manually from the command line. Then we scheduled that command to run at regular intervals so that the project is continually integrated

[5] `http://confluence.public.thoughtworks.org/display/CC/Home`

and tested. We can even schedule multiple Ant targets, each running on a different schedule. The build scheduler alerts us when the build breaks by sending email and recording the build history on a common web page. All this makes finding and fixing problems easier so that we have more time to do the really exciting stuff.