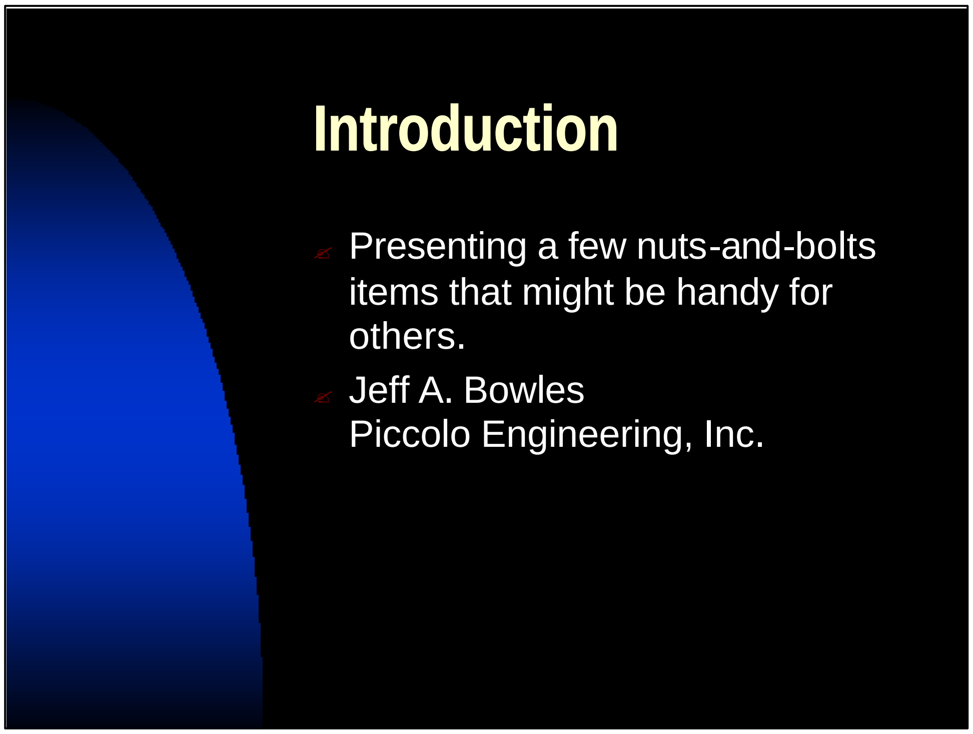
Tricks and Tools...

A few things a build-guy picked up along the path…

(There are notes included in the ‘notes pages’ here so that you can have access to some of the things Jeff said during the talk. But if you really want more verbiage, don’t forget the conference paper itself, which is in MS-Word and also in PDF in the Perforce Conference proceedings.)

This talk was done on 9/14/00 in San Francisco, at the Perforce User’s Conference, by Jeff Bowles of Piccolo Engineering. You’re welcome to use the materials and ideas in this talk for your own work, but if you use it outright please give me credit – including my e-mail address (“jab@pobox.com”). It might drum up business…



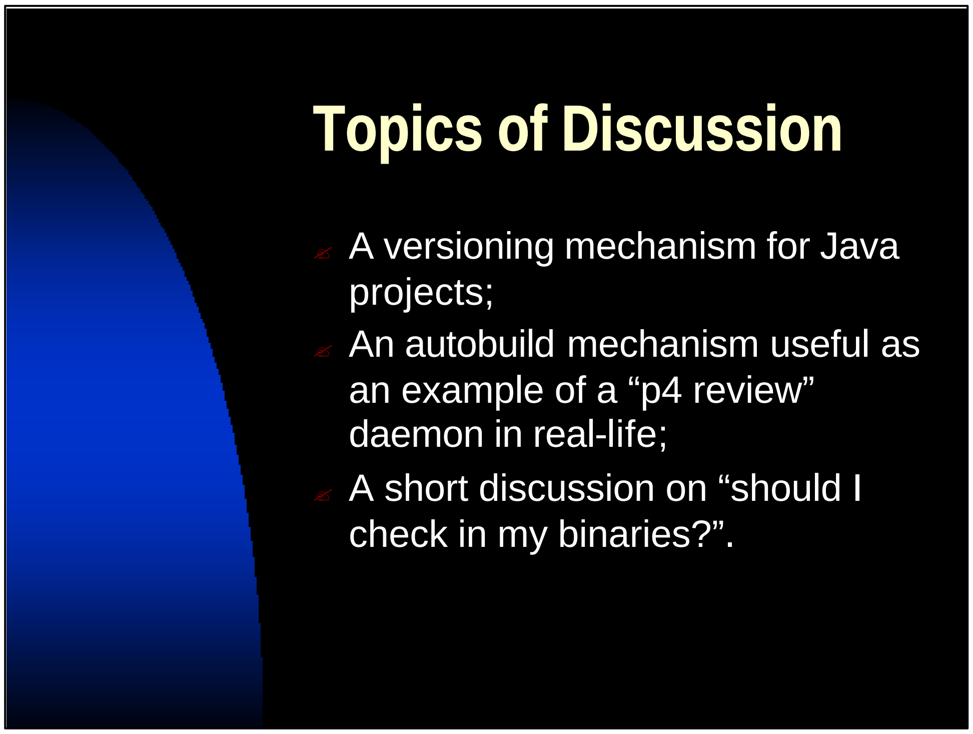
Introduction

- Presenting a few nuts-and-bolts items that might be handy for others.
- Jeff A. Bowles
  Piccolo Engineering, Inc.

We're presenting here a few simple items that anyone can implement using Perforce – and some can be implemented using other S.C.M. systems.

The upshot is that you don't have to aim for a 12-month project for a SCM integration, and people really never do. They start with something simple, that grows as your needs and experience grow.

Here are some of those simple projects that start you along the path…



Topics of Discussion

- A versioning mechanism for Java projects;
- An autobuild mechanism useful as an example of a “p4 review” daemon in real-life;
- A short discussion on “should I check in my binaries?”.

We’ll cover three things:

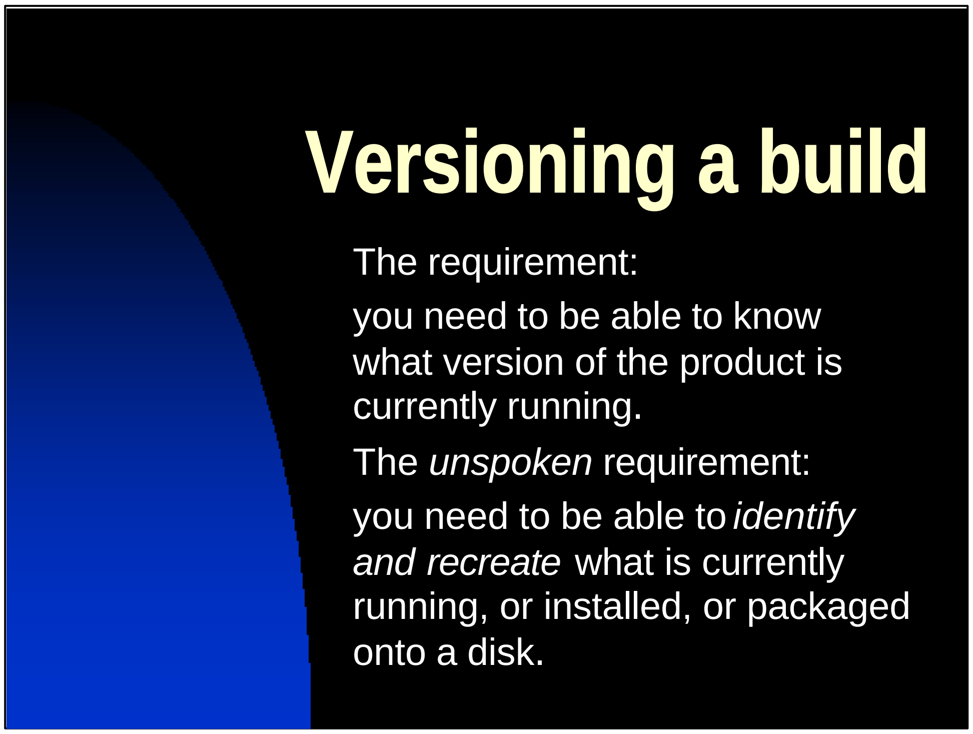
• A Java build mechanism that makes

```
java –classpath yourproduct.jar  yourproduct.PrintVersion
```

print out the version string (“label” or “changenum”) for your product. If you deliver more than one Java archive file (“JAR file”) you can version them independently.

•Several simple examples of “p4 review” (post-submit trigger) scripts, including one for propogating changelist information to a bug database and one for doing autobuilds.

•A discussion on “should I check in my binaries?”. (I try to be even-handed enough that you cannot tell which side of the argument I’m on. I was complimented by one person afterwards who said “you’re for checking in binaries, right?” after the talk, when – in fact – I’m against it.)



Versioning a build

The requirement:

you need to be able to know what version of the product is currently running.

The *unspoken* requirement:

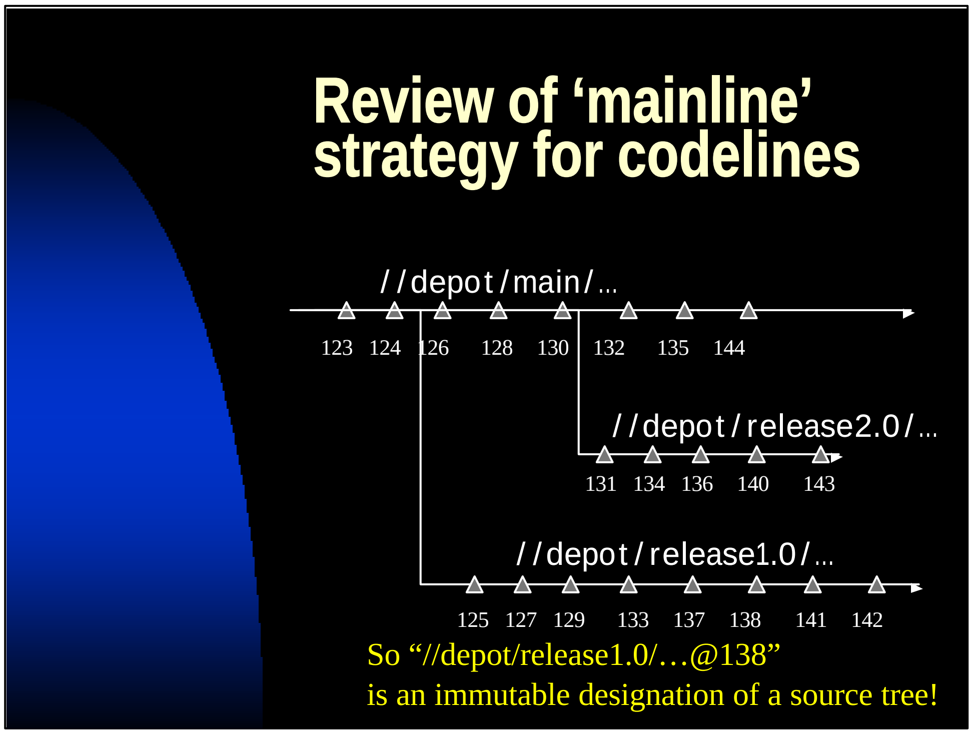
you need to be able to *identify and recreate* what is currently running, or installed, or packaged onto a disk.

This slide has a slight bit of animation:

•“The requirement…” appears first, then

•“The *unspoken* requirement” appears.

It reads straight from the slide.



Review of 'mainline' strategy for codelines

So "//depot/release1.0/…@138" is an immutable designation of a source tree!

This is a straight-forward presentation of the first part of the Perforce white paper located at

http://www.perforce.com/perforce/bestpractices.html

A summary is:

•You have a 'main' codeline.

•Release codelines are branched from that (or from a child of the main).

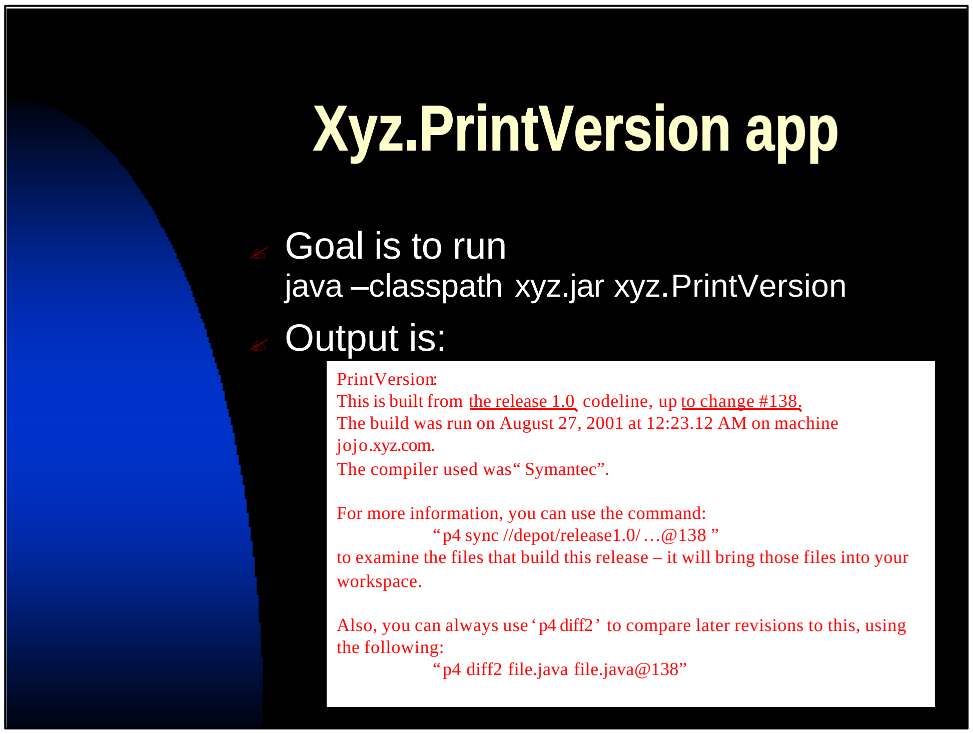
•Bug fixes/patches are made in the release line and ALWAYS integrated to their parent and its parent and so on, up to the main. (The exception might be "I fixed something in release1.0 but we'll do it differently in 'main' and 'release2.0'.")

•You try to avoid sibling/sibling merges when possible, to avoid the "I fixed it in 1.0 and pushed it to 2.0, but forgot to include it when we made a 3.0 codeline and now it's broken again" issue.

I bring this model up to point out that

(codeline, change number)

Describes a source-set in an immutable way. (Assuming you pull down and build the entire thing!)



Xyz.PrintVersion app

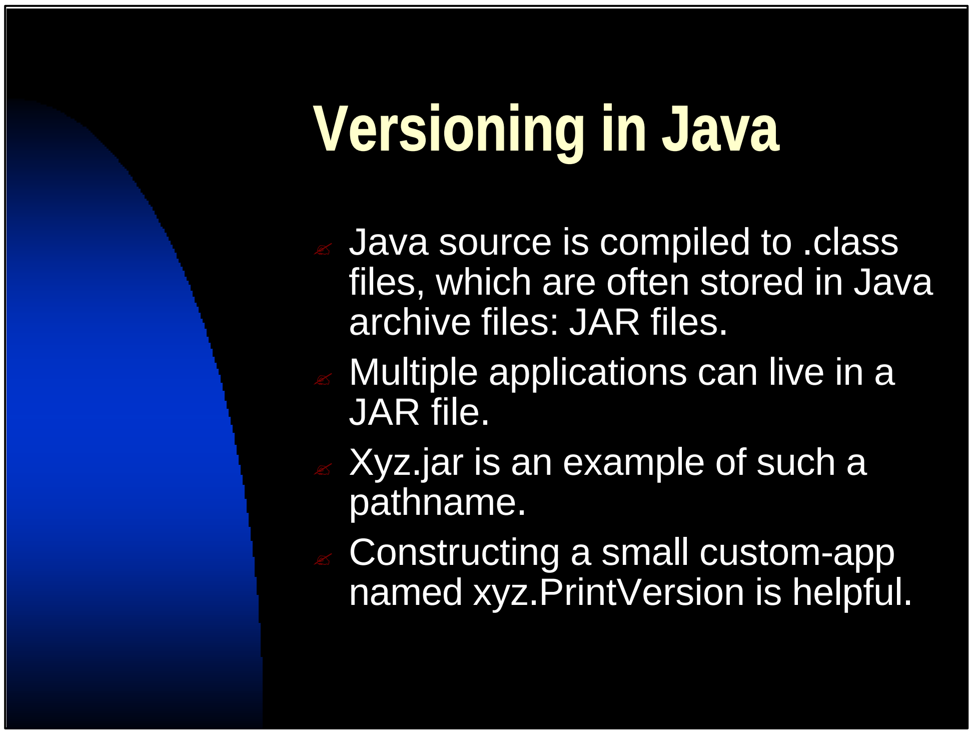
- Goal is to run
  java –classpath xyz.jar xyz.PrintVersion
- Output is:

```
PrintVersion:
This is built from the release 1.0 codeline, up to change #138.
The build was run on August 27, 2001 at 12:23.12 AM on machine
jojo.xyz.com.
The compiler used was“ Symantec”.

For more information, you can use the command:
                “p4 sync //depot/release1.0/…@138 ”
to examine the files that build this release – it will bring those files into your
workspace.

Also, you can always use ‘p4 diff2’ to compare later revisions to this, using
the following:
                “p4 diff2 file.java file.java@138”
```

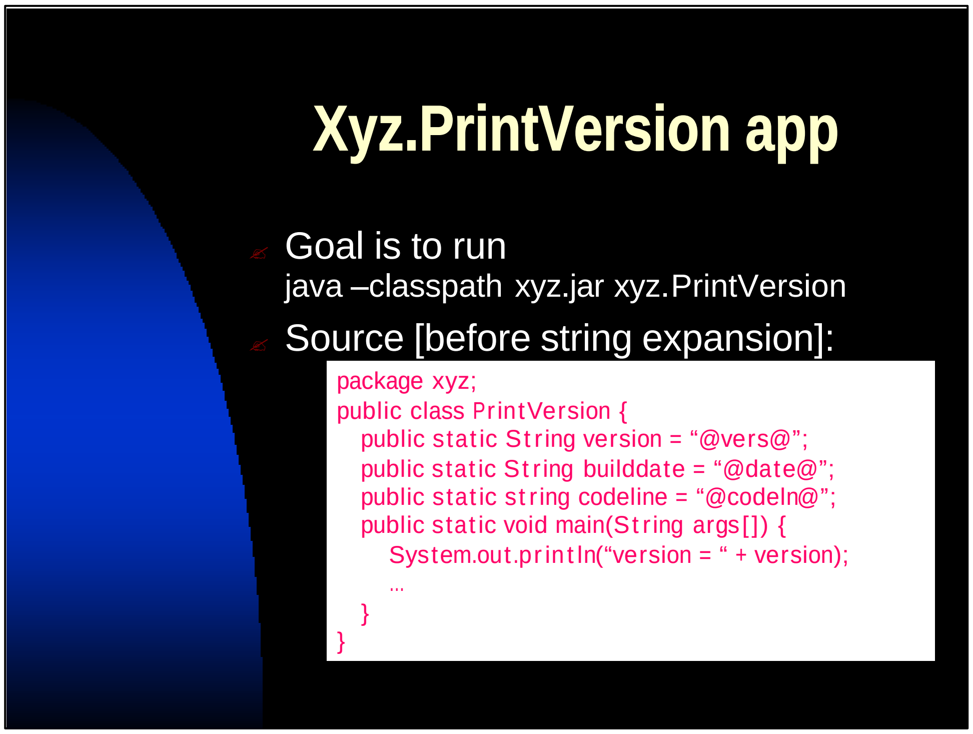
So the goal is to have a Java program embedded into the Java archive file (“JAR file”) that is a whole lot of “println” statements. Sample output is above – note how verbose it’s allowed to be.



Versioning in Java

- Java source is compiled to .class files, which are often stored in Java archive files: JAR files.
- Multiple applications can live in a JAR file.
- Xyz.jar is an example of such a pathname.
- Constructing a small custom-app named xyz.PrintVersion is helpful.

This reads straight from the slide.



Xyz.PrintVersion app

- Goal is to run
  java –classpath xyz.jar xyz.PrintVersion
- Source [before string expansion]:

```
package xyz;
public class PrintVersion {
   public static String version = “@vers@”;
   public static String builddate = “@date@”;
   public static string codeline = “@codeln@”;
   public static void main(String args[]) {
      System.out.println(“version = “ + version);
      ...
   }
}
```

The stuff that’s substituted (“@vers@”) is something that is really easy for the build tool “ant” to replace – you can always use ‘sed’ if you’re using “make”.

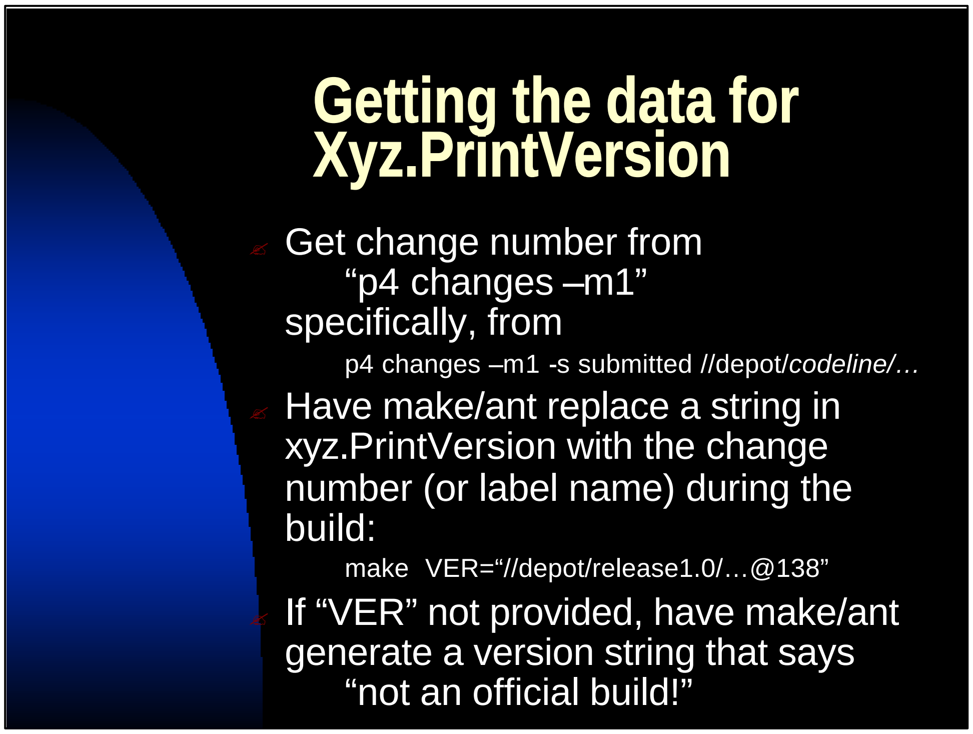
For example, the “make” target I used to generate this looked like this:

```
generated_printversion:
        $(MAKEPRINTVERSION)    \
                iafc/common/default/PrintVersion.template \
                -o iafc/common/PrintVersion.java \
                -v "$(VER)" -d "$(BUILDDATE)" \
                -c "$(JAVAC)“
```

But a simpler one might be:

```
generated_printversion:
        sed   “s/@vers@/$(VER)/”  \
                < src/PrintVersion.template   \
                ✍src/PrintVersion.java
```

Then you’d make your compiles depend on “generated_printversion”.



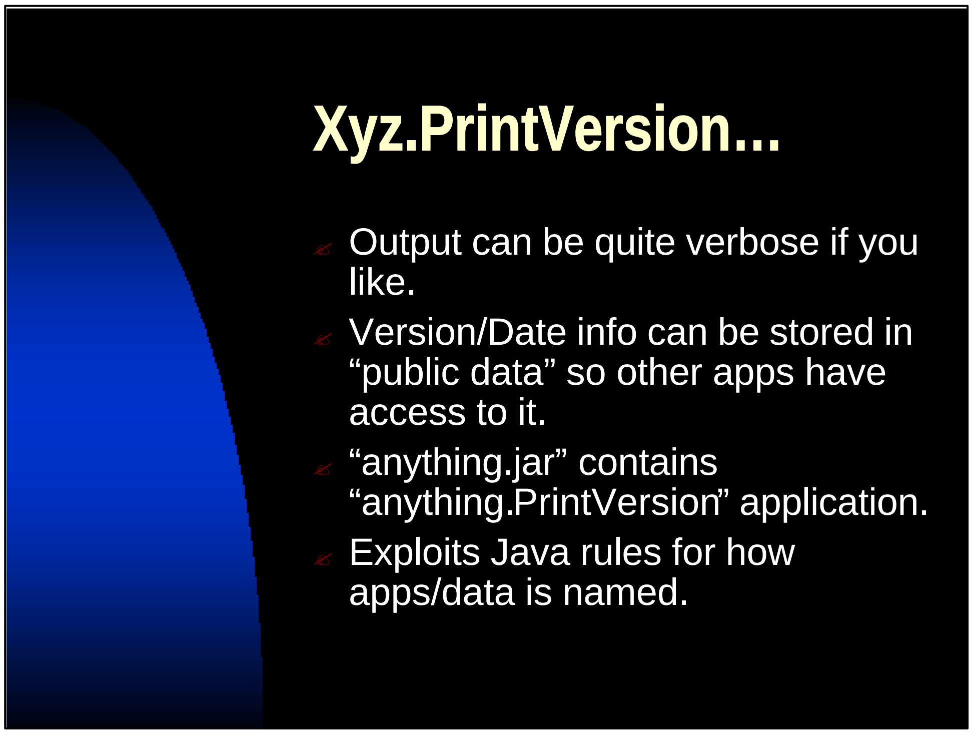
Getting the data for Xyz.PrintVersion

- Get change number from
  "p4 changes –m1"
  specifically, from
  p4 changes –m1 -s submitted //depot/*codeline/…*
- Have make/ant replace a string in xyz.PrintVersion with the change number (or label name) during the build:
  make VER="//depot/release1.0/…@138"
- If "VER" not provided, have make/ant generate a version string that says
  "not an official build!"

Read carefully the stuff about "p4 changes –s submitted" – you need to make sure that you build what you think you have. The two models are:

- Build and make a label from "#have", e.g.
  p4 labelsync –l labelname #have
- Make a list of the files (label, changenum) and then
  1. Clear the client of any files ("p4 sync #none")
  2. Clear the client of any object/class/generated files (you will use "rm" or "del /s" here.)
  3. Sync to the list (label, changenum) and do the build.

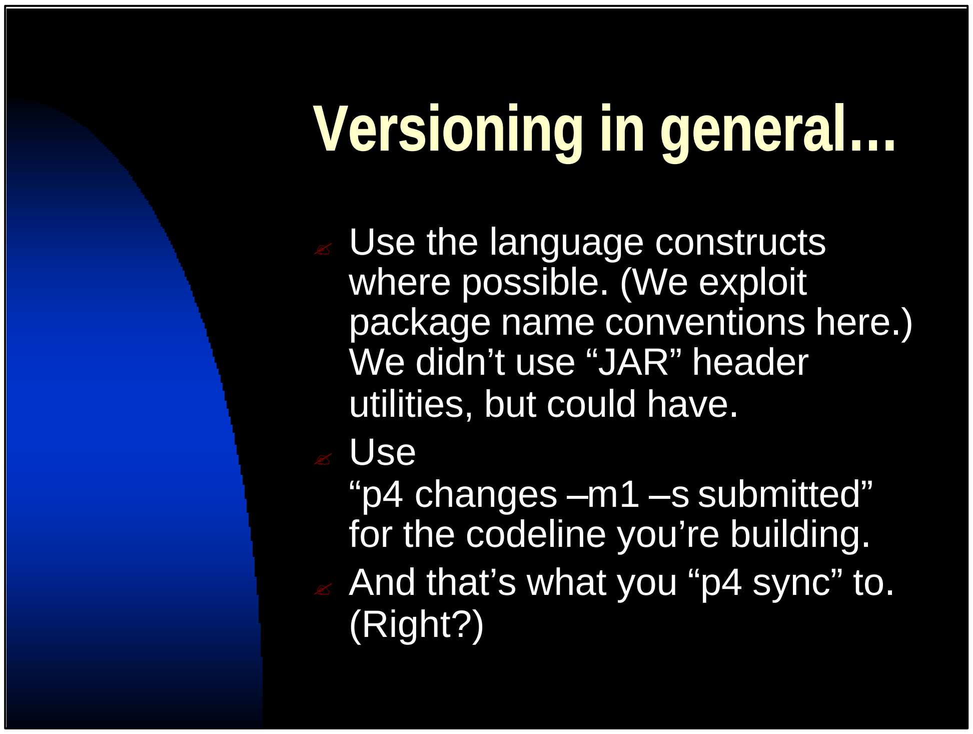
I strongly recommend this second method – it guarantees that you built what you thought you were supposed to. Fewer surprises.



Xyz.PrintVersion…

- Output can be quite verbose if you like.
- Version/Date info can be stored in “public data” so other apps have access to it.
- “anything.jar” contains “anything.PrintVersion” application.
- Exploits Java rules for how apps/data is named.

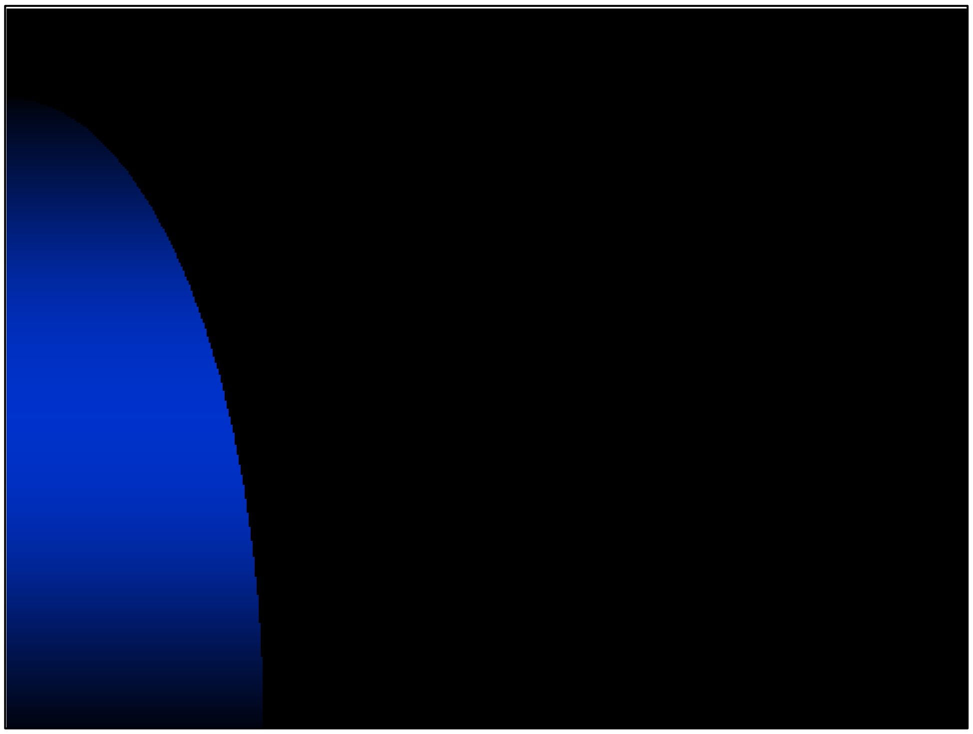
This reads straight from the slide. We could’ve used JAR header utilities to store the version information, but Java .class files are sometimes delivered as non-archived trees (in which case no header exists) or sometimes the .class files are delivered in .ZIP files.

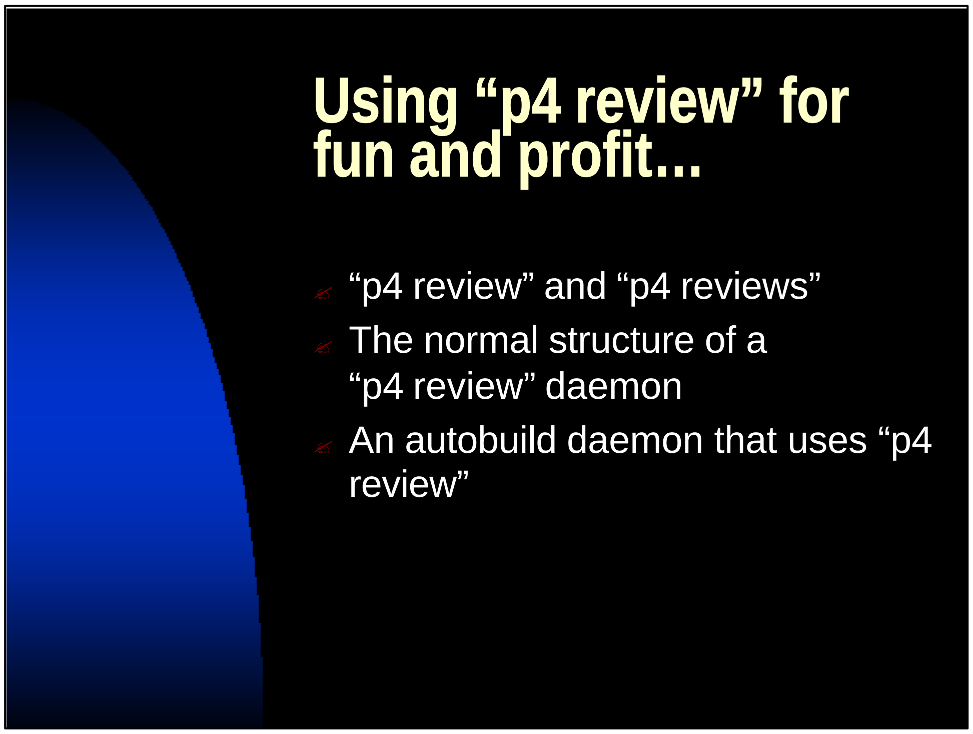


Versioning in general…

- Use the language constructs where possible. (We exploit package name conventions here.) We didn't use "JAR" header utilities, but could have.
- Use "p4 changes –m1 –s submitted" for the codeline you're building.
- And that's what you "p4 sync" to. (Right?)

Reads straight from the slide.





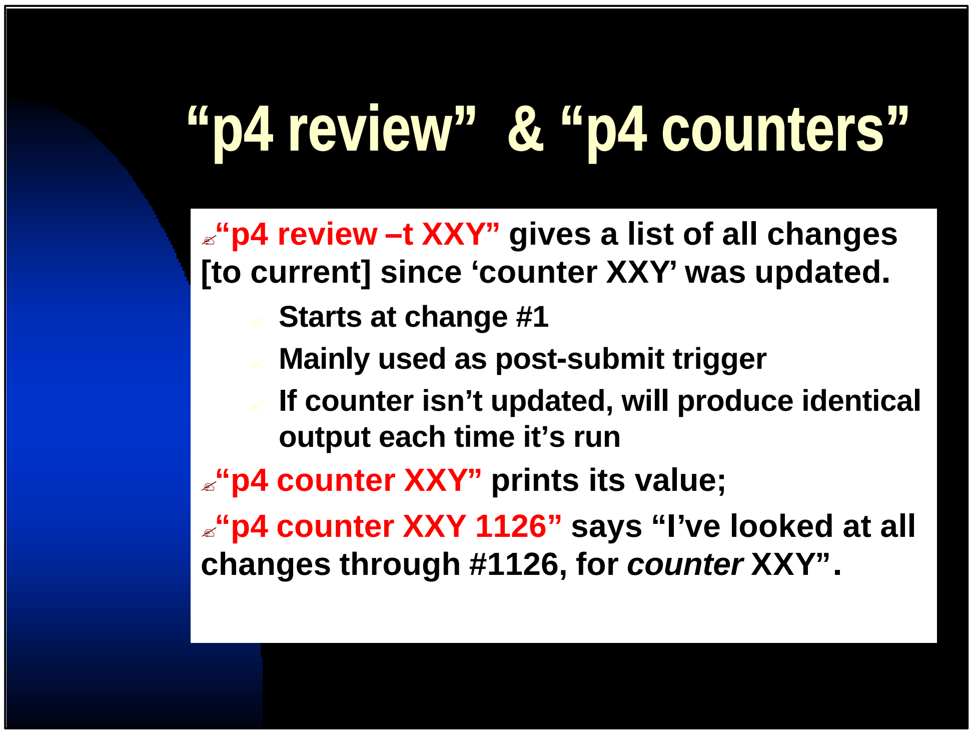
Using “p4 review” for fun and profit…

- “p4 review” and “p4 reviews”
- The normal structure of a “p4 review” daemon
- An autobuild daemon that uses “p4 review”

Here we talk about pre-submit and post-submit triggers. This is a talk about post-submit mechanisms, but I spend a bit of time reminding people that:

•“Presubmit” triggers don’t affect data – they mustn’t, because if six (6) triggers are to be run and the fourth (4th) fails, the first three think everything is okay. What if they’d updated a database or done something like sending e-mail saying a change was checked in? That action would’ve been inappropriate because it failed the check-in after all.

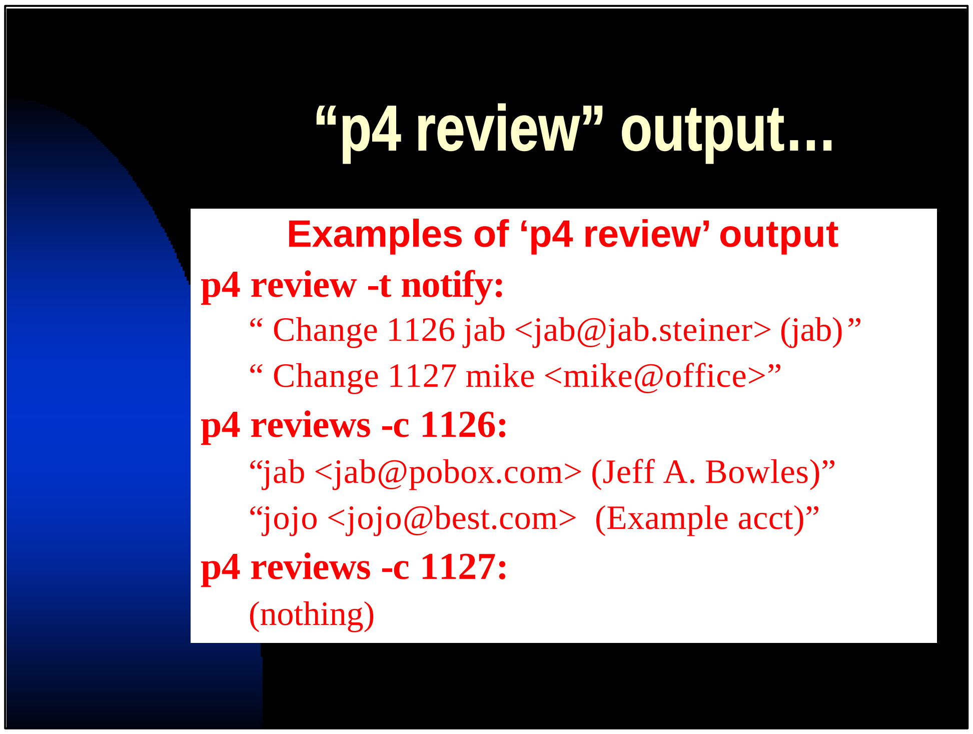
•“Post-submit triggers” actually aren’t run directly by the server – they’re scripts that poll the server using “p4 review” and and do anything they choose including submitting changes and the like.



“p4 review” & “p4 counters”

- “p4 review –t XXY” gives a list of all changes [to current] since ‘counter XXY’ was updated.
  - Starts at change #1
  - Mainly used as post-submit trigger
  - If counter isn’t updated, will produce identical output each time it’s run
- “p4 counter XXY” prints its value;
- “p4 counter XXY 1126” says “I’ve looked at all changes through #1126, for *counter* XXY”.

Brief explanation of “p4 review” and “p4 counter”.



“p4 review” output...

Examples of ‘p4 review’ output

p4 review -t notify:

```
“ Change 1126 jab <jab@jab.steiner> (jab) ”
“ Change 1127 mike <mike@office>”
```

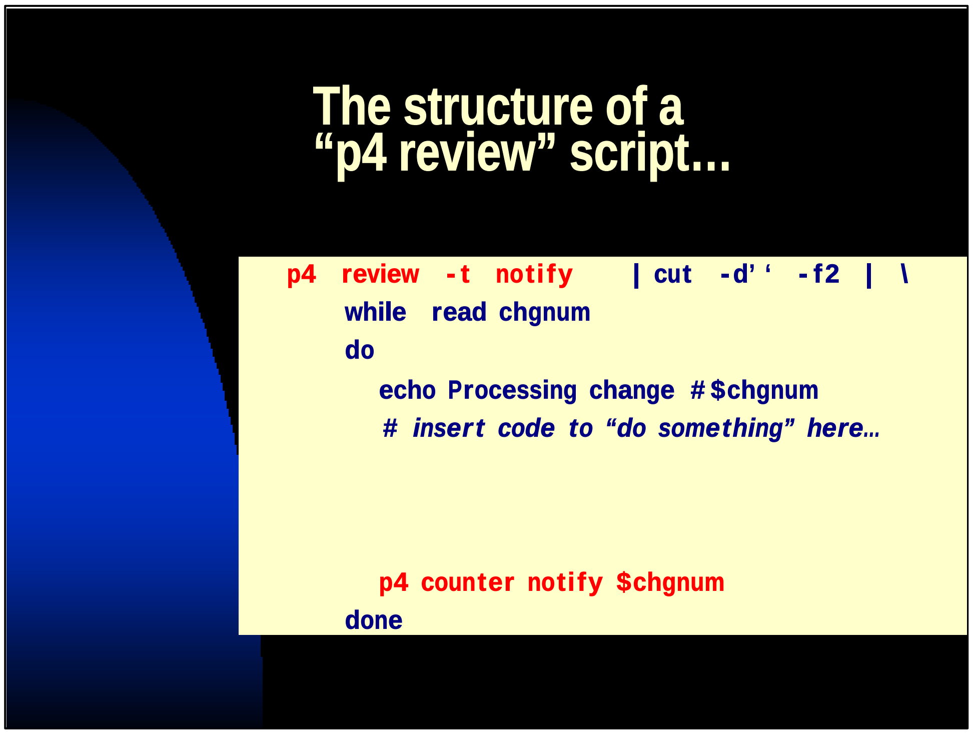
p4 reviews -c 1126:

```
“jab <jab@pobox.com> (Jeff A. Bowles)”
“jojo <jojo@best.com>  (Example acct)”
```

p4 reviews -c 1127:

(nothing)

Reads from slide. “p4 reviews” uses the “p4 user” information (for each user) to tell you who will get mail for each change.



The structure of a “p4 review” script…

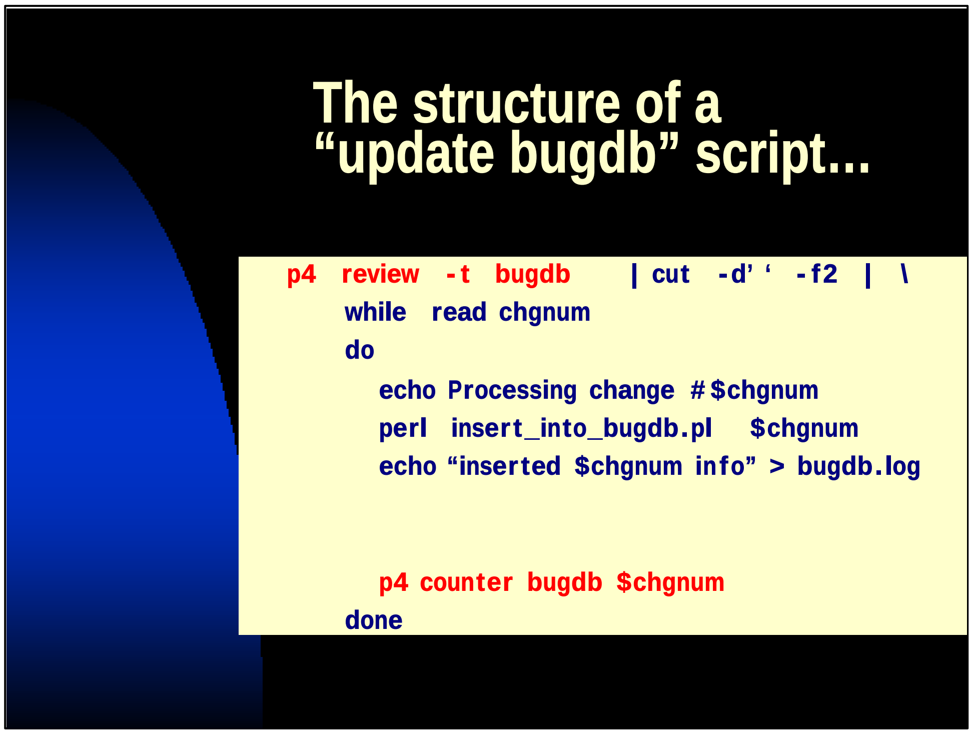
```
p4 review -t notify    | cut -d' ' -f2 | \
    while read chgnum
    do
       echo Processing change #$chgnum
       # insert code to “do something” here…



       p4 counter notify $chgnum
    done
```

I repeat these words many times: “any post-submit ‘review’ script will look like this program.”

Here’s the uber-script.

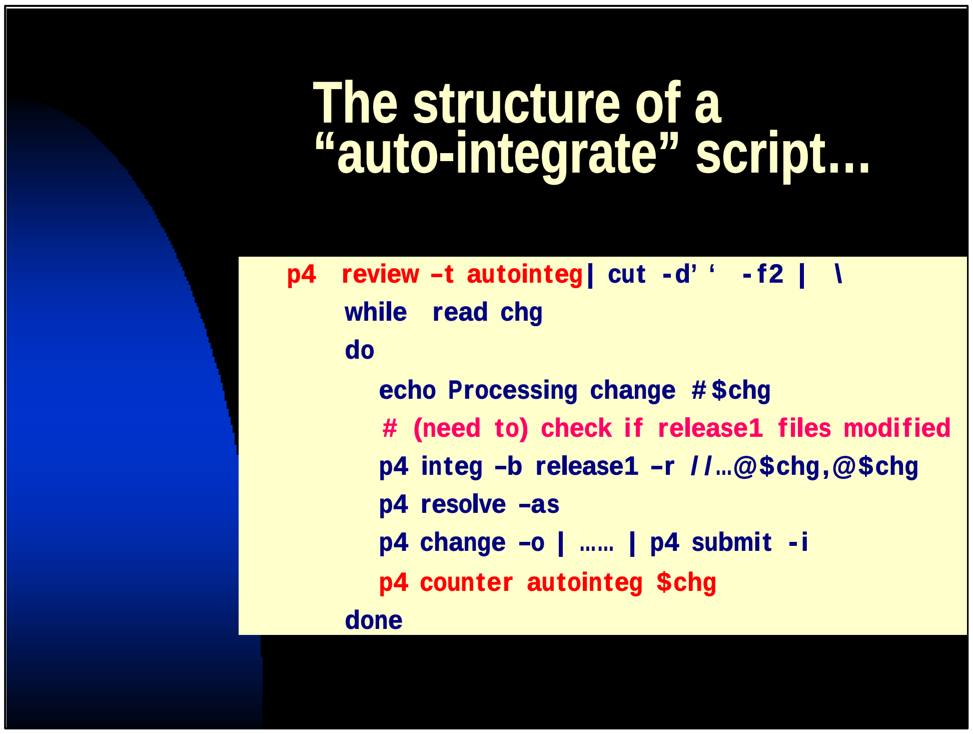


The structure of a "update bugdb" script...

```
p4 review -t bugdb    | cut -d' ' -f2 | \
   while read chgnum
   do
      echo Processing change #$chgnum
      perl insert_into_bugdb.pl  $chgnum
      echo "inserted $chgnum info" > bugdb.log


      p4 counter bugdb $chgnum
   done
```

The same script with a slight customization: a different counter, a different function.



The structure of a “auto-integrate” script...

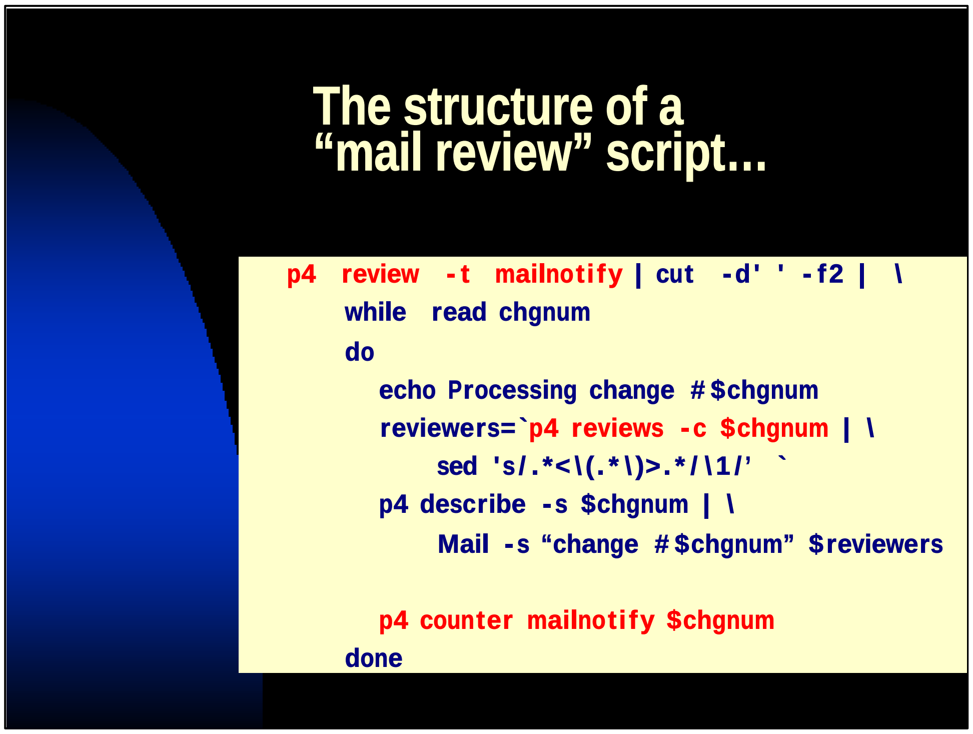
```
p4  review –t autointeg| cut -d’ ‘  -f2 |  \
    while  read chg
    do
       echo Processing change #$chg
       # (need to) check if release1 files modified
       p4 integ –b release1 –r //…@$chg,@$chg
       p4 resolve –as
       p4 change –o | …… | p4 submit -i
       p4 counter autointeg $chg
    done
```

Another different counter for a different function.

You could optimize a bit. For example,

```
   FileList=`p4 files //depot/release1/…@$chgnum,@$chgnum`
   [ “$FileList” = “” ] && continue
```

(this means “if no files modified in that changelist are part of the //depot/release1/…, don’t bother auto-integrating this change.”)



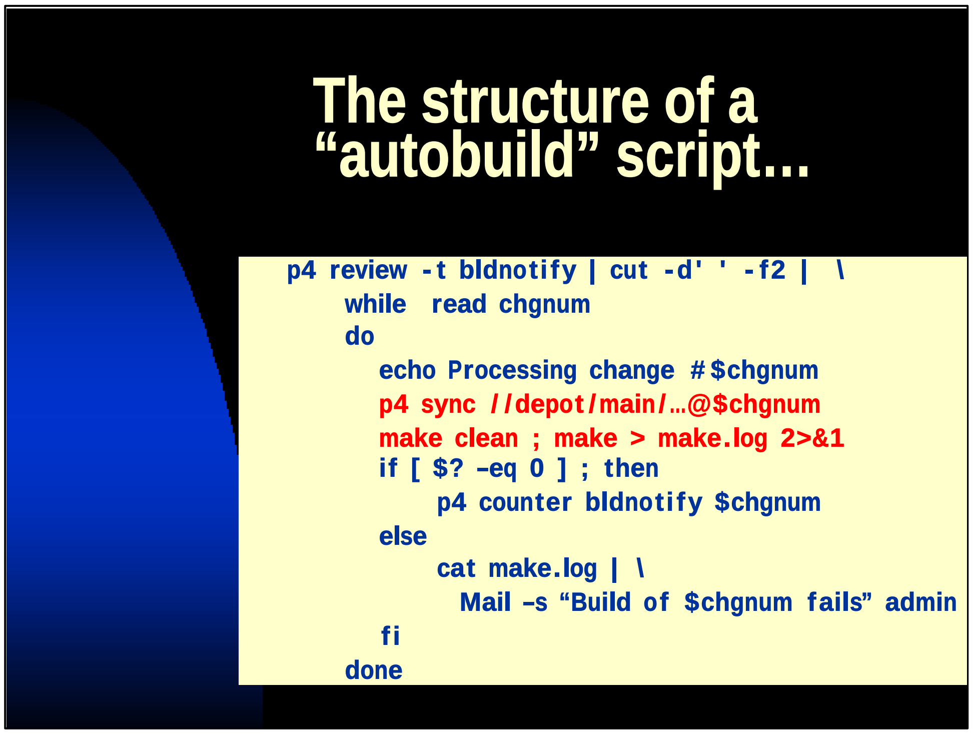
The structure of a “mail review” script…

```
p4  review  -t  mailnotify | cut  -d' ' -f2 |  \
    while  read chgnum
    do
       echo Processing change #$chgnum
       reviewers=`p4 reviews -c $chgnum | \
            sed 's/.*<\(.*\)>.*/\1/'  `
       p4 describe -s $chgnum | \
            Mail -s “change #$chgnum” $reviewers

       p4 counter mailnotify $chgnum
    done
```

And if you were writing your own mail notification mechanism, it would be like this.

However, I wouldn’t bother – the Python one at the Perforce site is more robust and very good.



The structure of a “autobuild” script…

```
p4 review -t bldnotify | cut -d' ' -f2 |  \
    while  read chgnum
    do
       echo Processing change #$chgnum
       p4 sync //depot/main/…@$chgnum
       make clean ; make > make.log 2>&1
       if [ $? –eq 0 ] ; then
            p4 counter bldnotify $chgnum
       else
            cat make.log | \
              Mail –s “Build of $chgnum fails” admin
       fi
    done
```

A few notes:

•You could optimize a bit. For example,

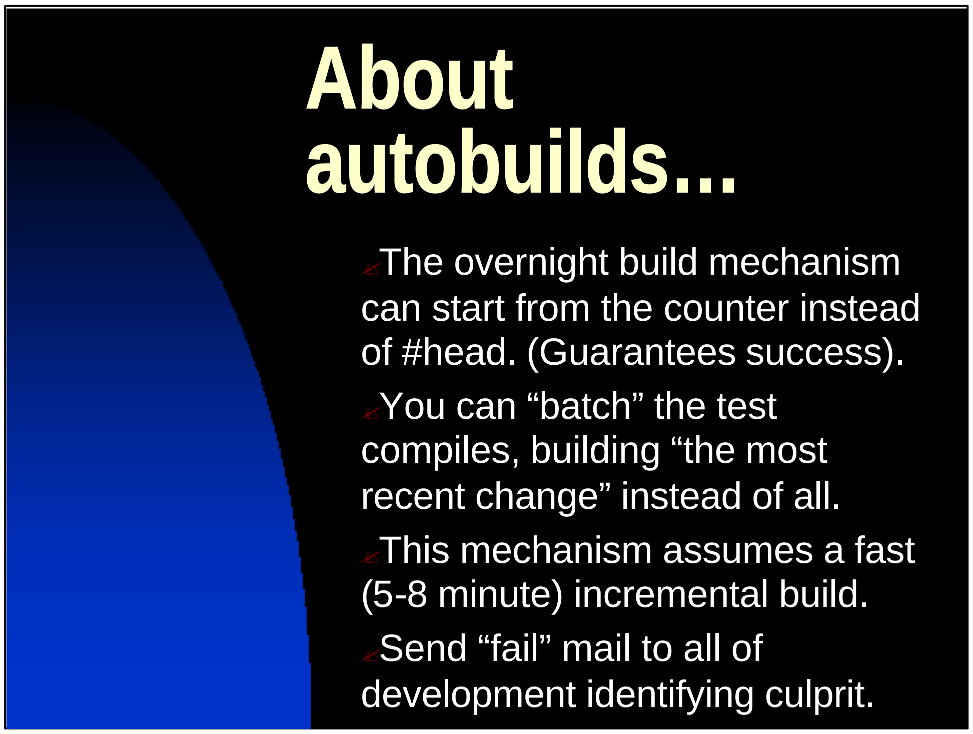
```
FileList=`p4 files //depot/main/…@$chgnum,@$chgnum`
 [ “$FileList” = “” ] && continue
```

(this means “if no files modified in that changelist are part of the //depot/main/…, don’t bother test-building this change.”)

•The “make clean” is optional. For this test, if you have good dependencies in the make files (or jam or ant) it’s unnecessary.

•You can be as nasty as you want in the e-mail you generate.

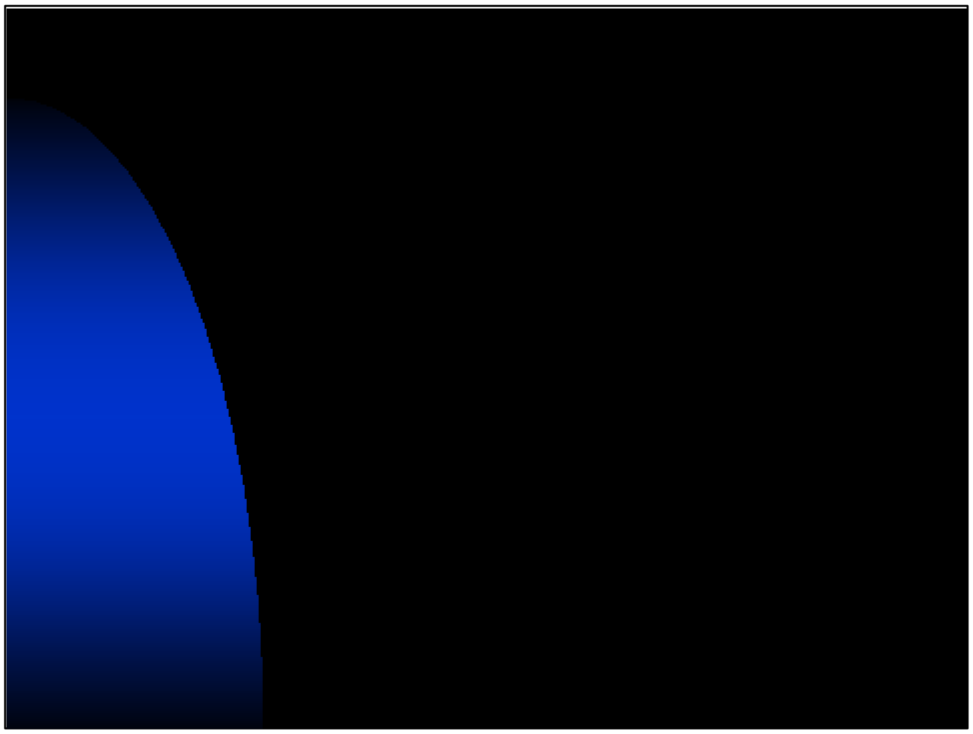
•It’s possible to have two counters: last-tried and last-good-compile. The model of the script doesn’t change too much to accommodate that.

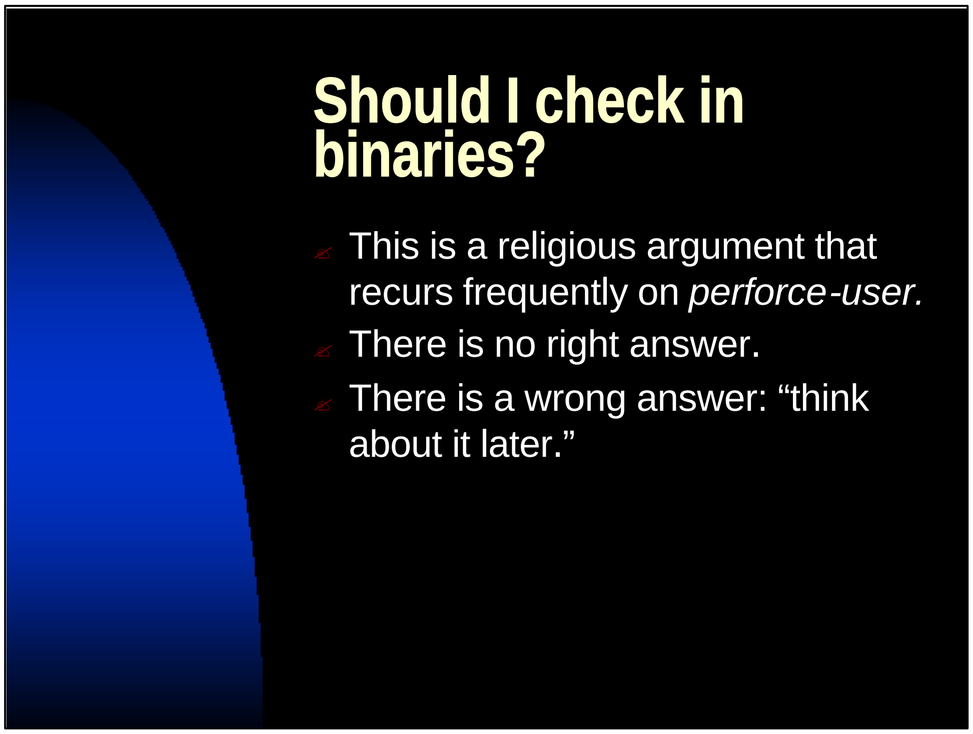


About autobuilds…

- The overnight build mechanism can start from the counter instead of #head. (Guarantees success).
- You can “batch” the test compiles, building “the most recent change” instead of all.
- This mechanism assumes a fast (5-8 minute) incremental build.
- Send “fail” mail to all of development identifying culprit.

This reads straight from the slide.

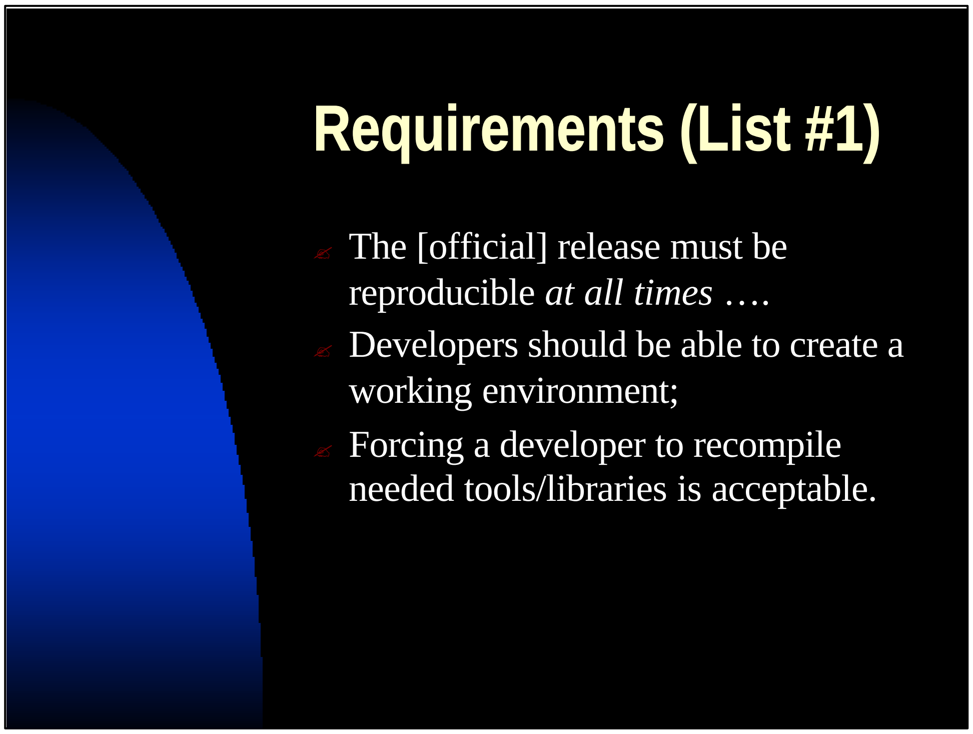




Should I check in binaries?

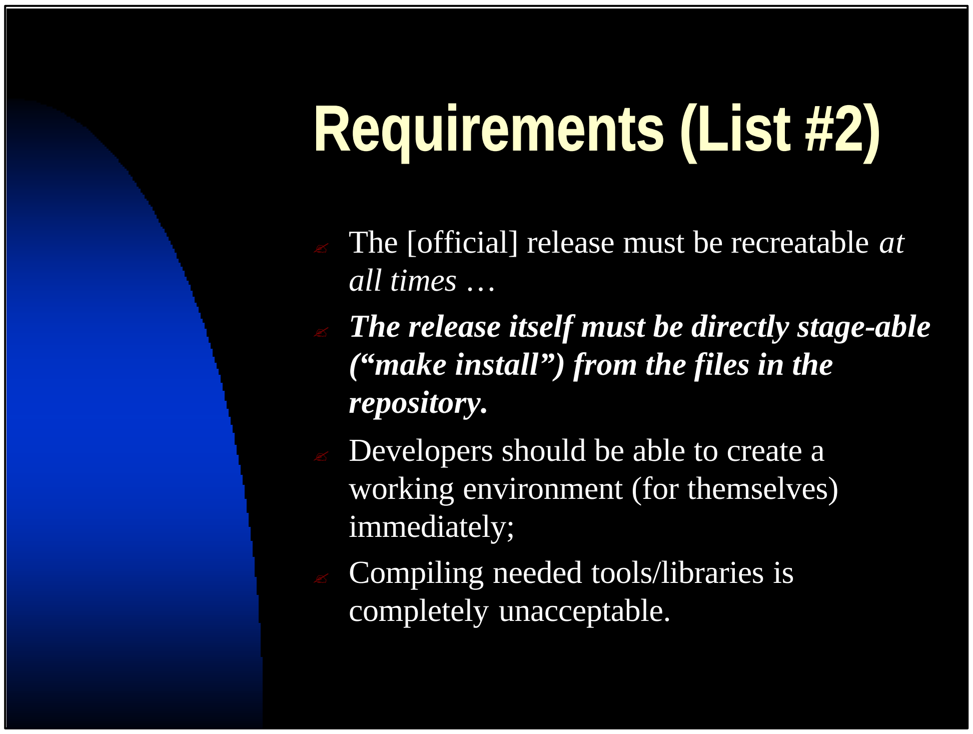
- This is a religious argument that recurs frequently on *perforce-user.*
- There is no right answer.
- There is a wrong answer: “think about it later.”

A very good bit of verbiage is in the paper for this – just go to the proceedings for this discussion.



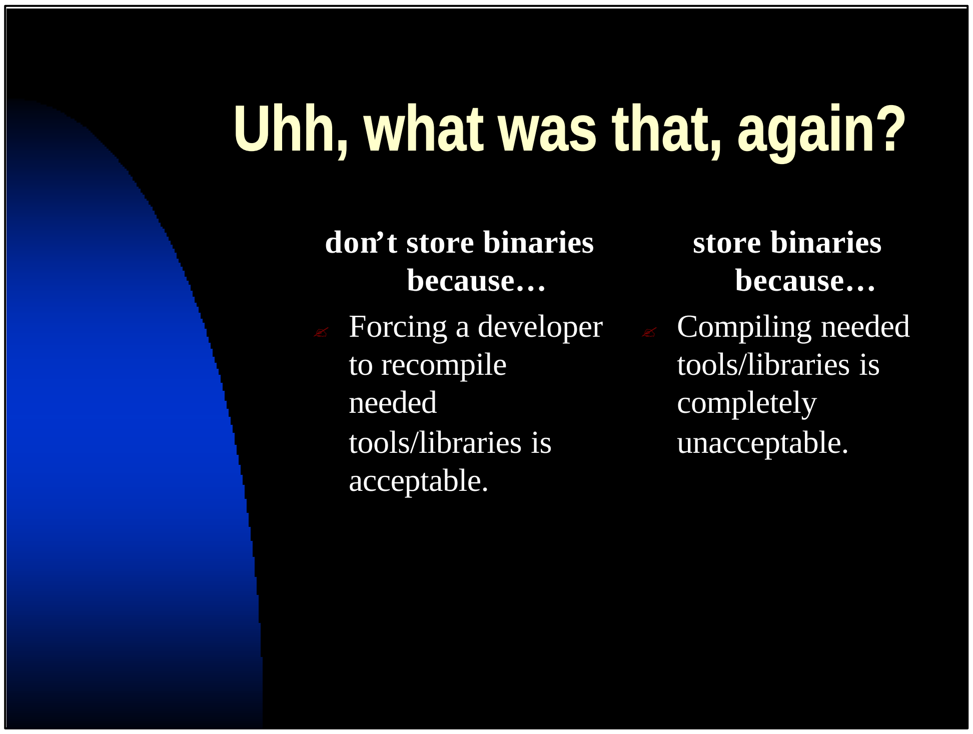
Requirements (List #1)

- The [official] release must be reproducible *at all times* ….
- Developers should be able to create a working environment;
- Forcing a developer to recompile needed tools/libraries is acceptable.



Requirements (List #2)

- The [official] release must be recreatable *at all times …*
- ***The release itself must be directly stage-able (“make install”) from the files in the repository.***
- Developers should be able to create a working environment (for themselves) immediately;
- Compiling needed tools/libraries is completely unacceptable.



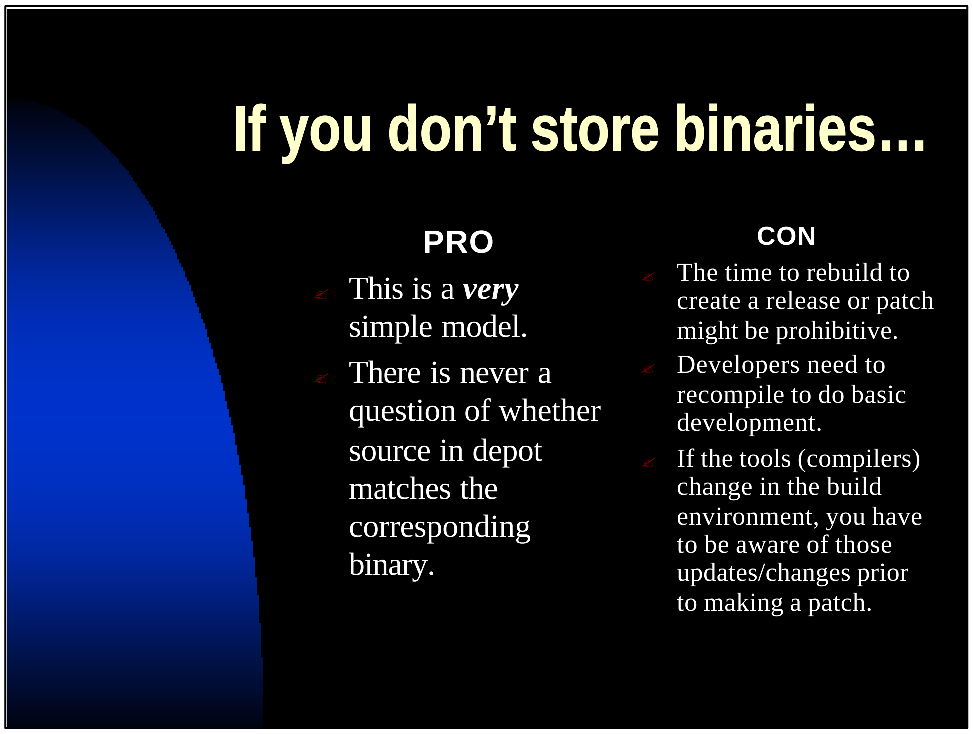
Uhh, what was that, again?

don’t store binaries because…

- Forcing a developer to recompile needed tools/libraries is acceptable.

store binaries because…

- Compiling needed tools/libraries is completely unacceptable.



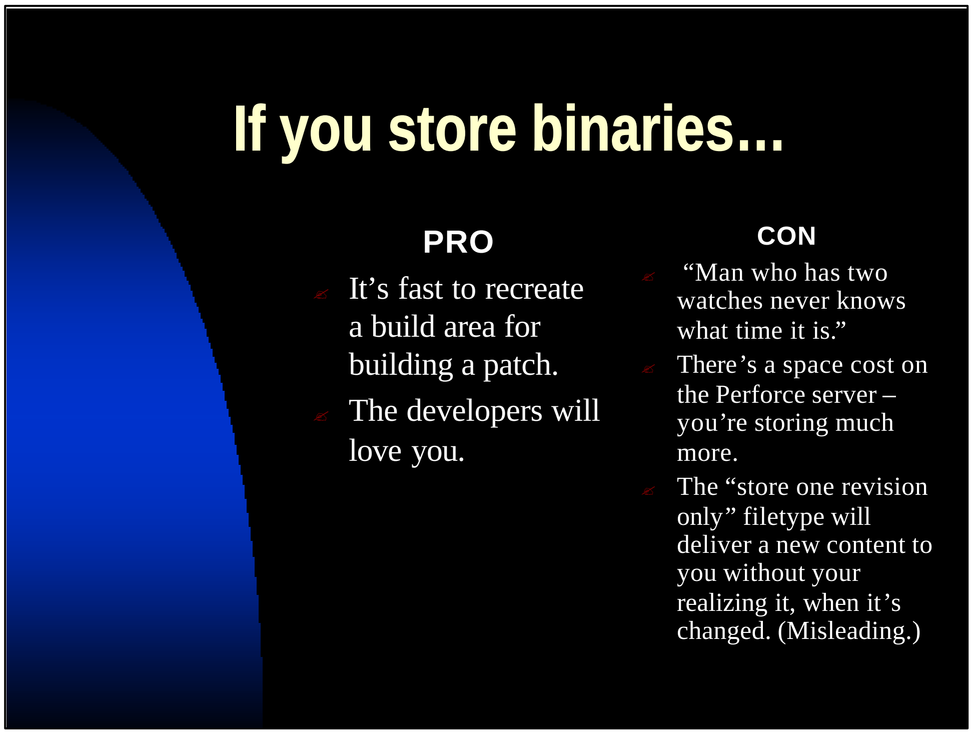
If you don't store binaries…

PRO

- This is a ***very*** simple model.
- There is never a question of whether source in depot matches the corresponding binary.

CON

- The time to rebuild to create a release or patch might be prohibitive.
- Developers need to recompile to do basic development.
- If the tools (compilers) change in the build environment, you have to be aware of those updates/changes prior to making a patch.



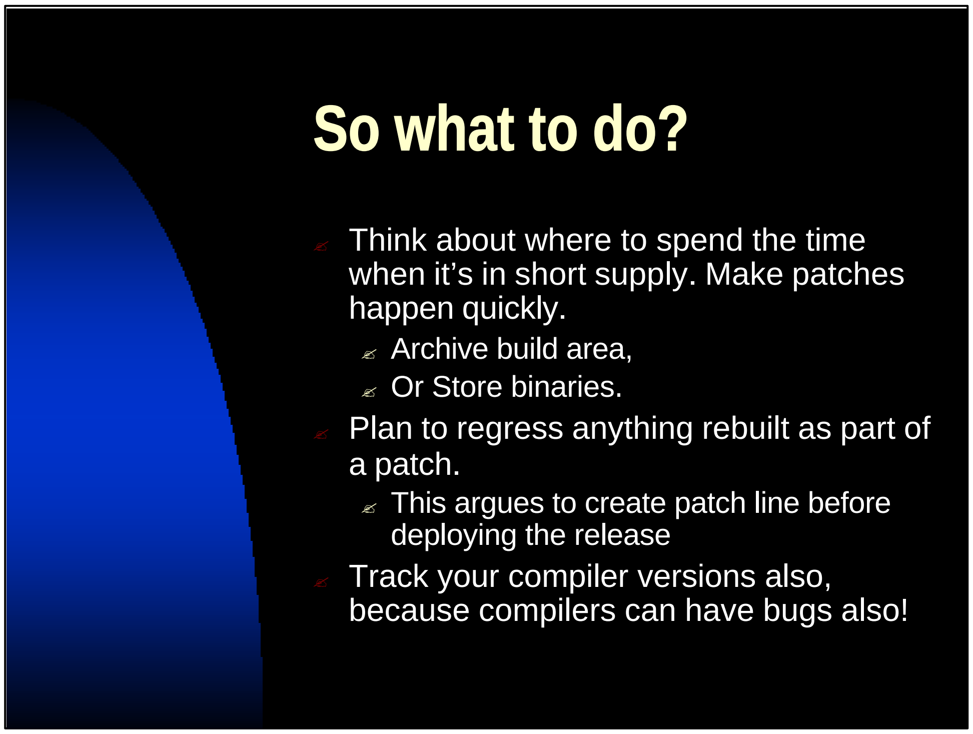
If you store binaries…

PRO

- It’s fast to recreate a build area for building a patch.
- The developers will love you.

CON

- “Man who has two watches never knows what time it is.”
- There’s a space cost on the Perforce server – you’re storing much more.
- The “store one revision only” filetype will deliver a new content to you without your realizing it, when it’s changed. (Misleading.)



So what to do?

- Think about where to spend the time when it’s in short supply. Make patches happen quickly.
  - Archive build area,
  - Or Store binaries.
- Plan to regress anything rebuilt as part of a patch.
  - This argues to create patch line before deploying the release
- Track your compiler versions also, because compilers can have bugs also!

Tricks and Tools...

A few things a build-guy picked up along the path…