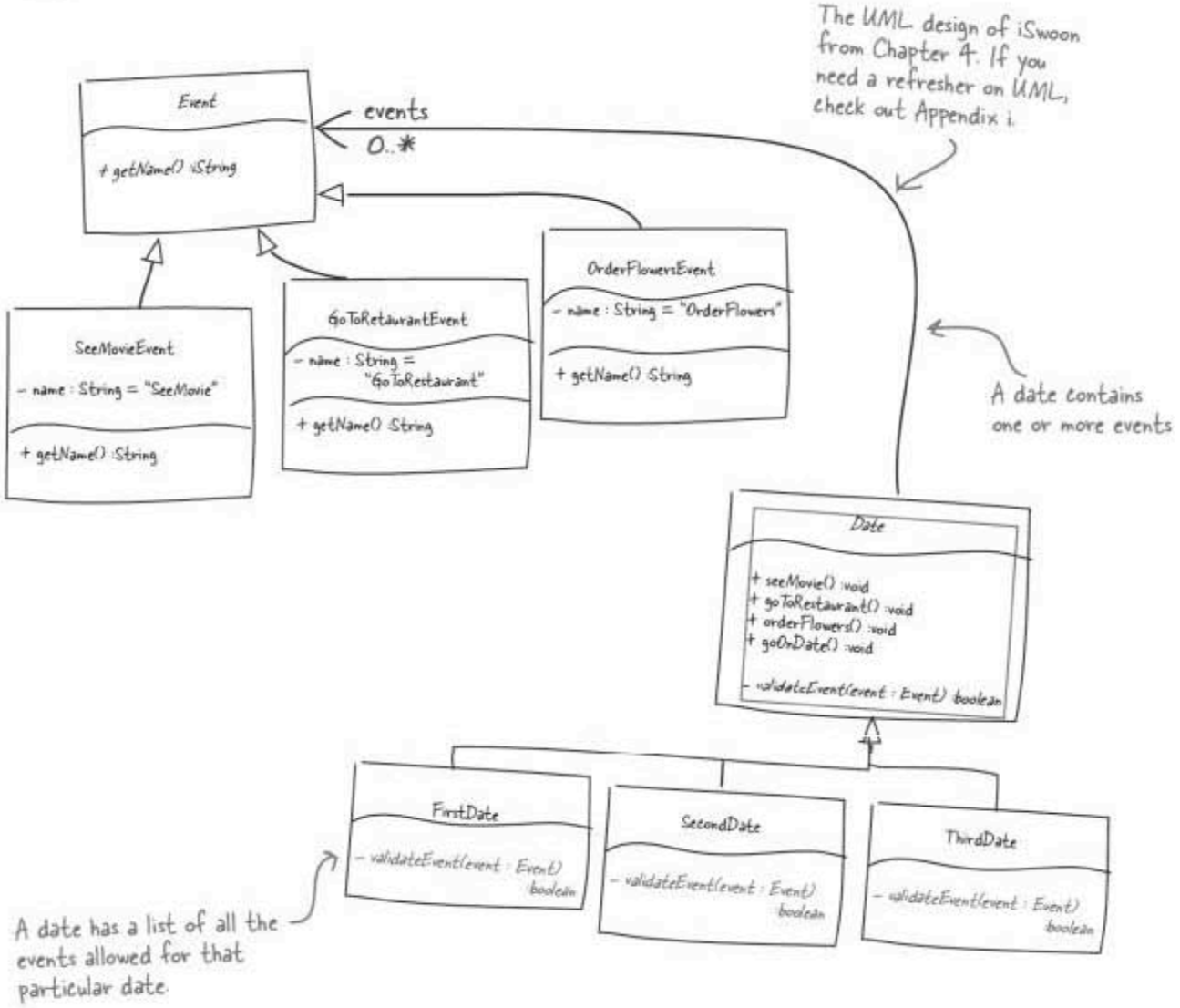
Getting it done with great design

Good design helps you deliver. In the last chapter things were looking pretty dire. A bad design was making life **hard for everyone**, and, to make matters worse, an unplanned task cropped up. In this chapter you'll see how to **refactor** your design so that you and your team can be **more productive**. You'll apply **principles of good design**, while at the same time be wary of striving for the promise of the ***"perfect design."*** Finally you'll **handle unplanned tasks** in exactly the same way you handle all the other work on your project using the big project board on your wall.

iSwoon is in serious trouble...

In the last chapter things were in pretty bad shape at iSwoon. You had some refactoring work to do to improve your design that was going to impact your deadlines, and the customer had piped in with a surprise task to develop a demonstration for the CEO of Starbuzz. All is not lost, however. First let's get the refactoring work done so that you can turn what looks like a slip into a way of speeding up your development work. The current design called for lots of changes just to add a new event:



Sharpen your pencil

Write down the changes you think would be needed if...

...you needed to add three new event types?

What would you have to do to the software to implement these changes?

...you needed to add a new event type called "Sleeping over," but that event was only allowed on the third date?

The validateEvent( ) method will certainly come in handy here.

...you changed the value of the name attribute in the OrderFlowersEvent class to "SendFlowers"?

Sharpen your pencil Solution

You were asked to write down the changes you think would be needed if...

...you needed to add three new event types?

We'd need a new event class for each of the new types. Three new methods, one for each type of event, would need to be added to the abstract parent Date class. Then, each of the date classes, however many there are, will need to be updated to allow (or disallow) the three new types of event, depending on if the event is allowed for that date.

...you needed to add a new event type called "Sleeping over," but that event was only allowed on the third date?

A new event class would be added, called something like "SleepingOverEvent." Then a new method called "sleepOver" needs to be added to the Date class so the new event can be added to a date. Finally, all three of the existing date classes would need to be updated in order to specify that only the third date allows a SleepingOverEvent to be specified.

...you changed the value of the name attribute in the OrderFlowersEvent class to "SendFlowers"?

All three of the different concrete classes of Date would need to be updated so that the logic that decides if a particular event is allowed now uses the new name in regards to the OrderFlowersEvent class's name attribute value change. Also, the value of OrderFlowerEvent's name will need to change from "OrderFlowers" to "SendFlowers," then finally the class name will need to be changed to SendFlowersEvent so it follows the naming convention we're currently using for date events.

Well-designed classes are singularly focused.

The problem here is that for any particular behavior—like sending flowers—the logic for that behavior is spread out over a lot of different classes. So what seems like a simple change, like the name in `OrderFlowersEvent` being changed to "SendFlowers," turns into a multi-class mess of modifications.

This design breaks the single responsibility principle

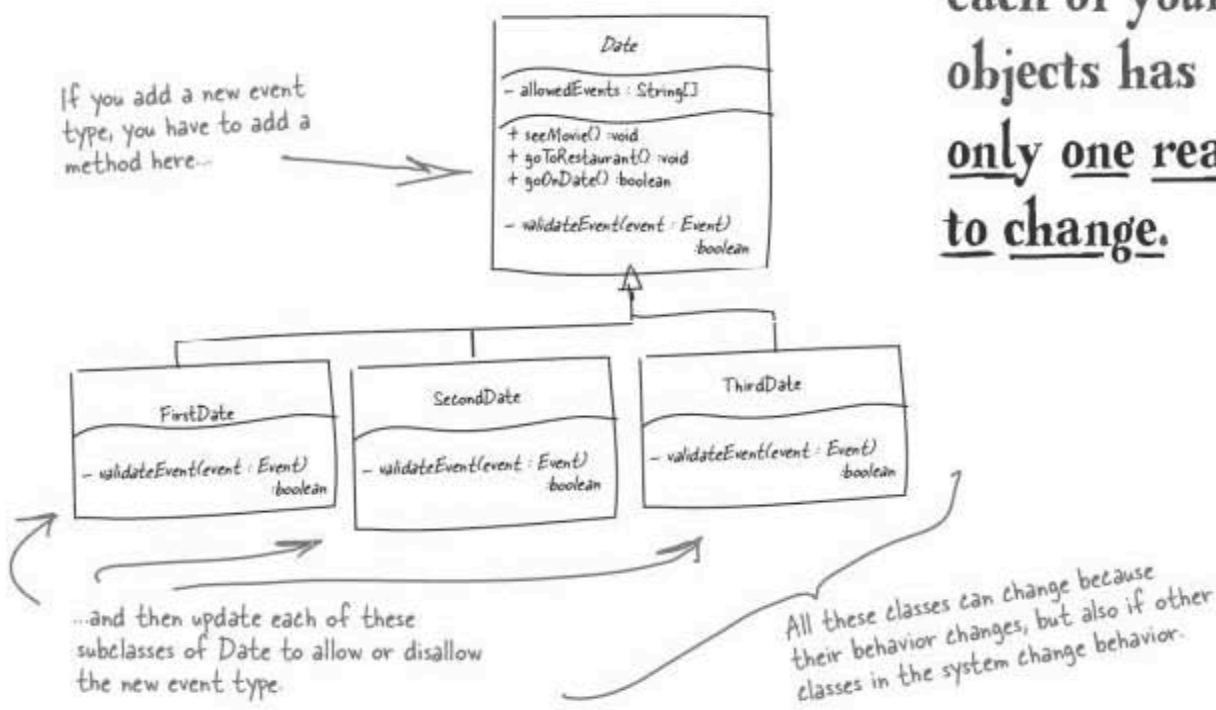
iSwoon is such a headache to update because it breaks one of the fundamental principles of good object oriented design, the **single responsibility principle** (or **SRP** for short).

Single responsibility principle

*Every object in your system should have a **single responsibility**, and all the object's services should be focused on carrying out that single responsibility.*

Both the Date and Event class break the single responsibility principle

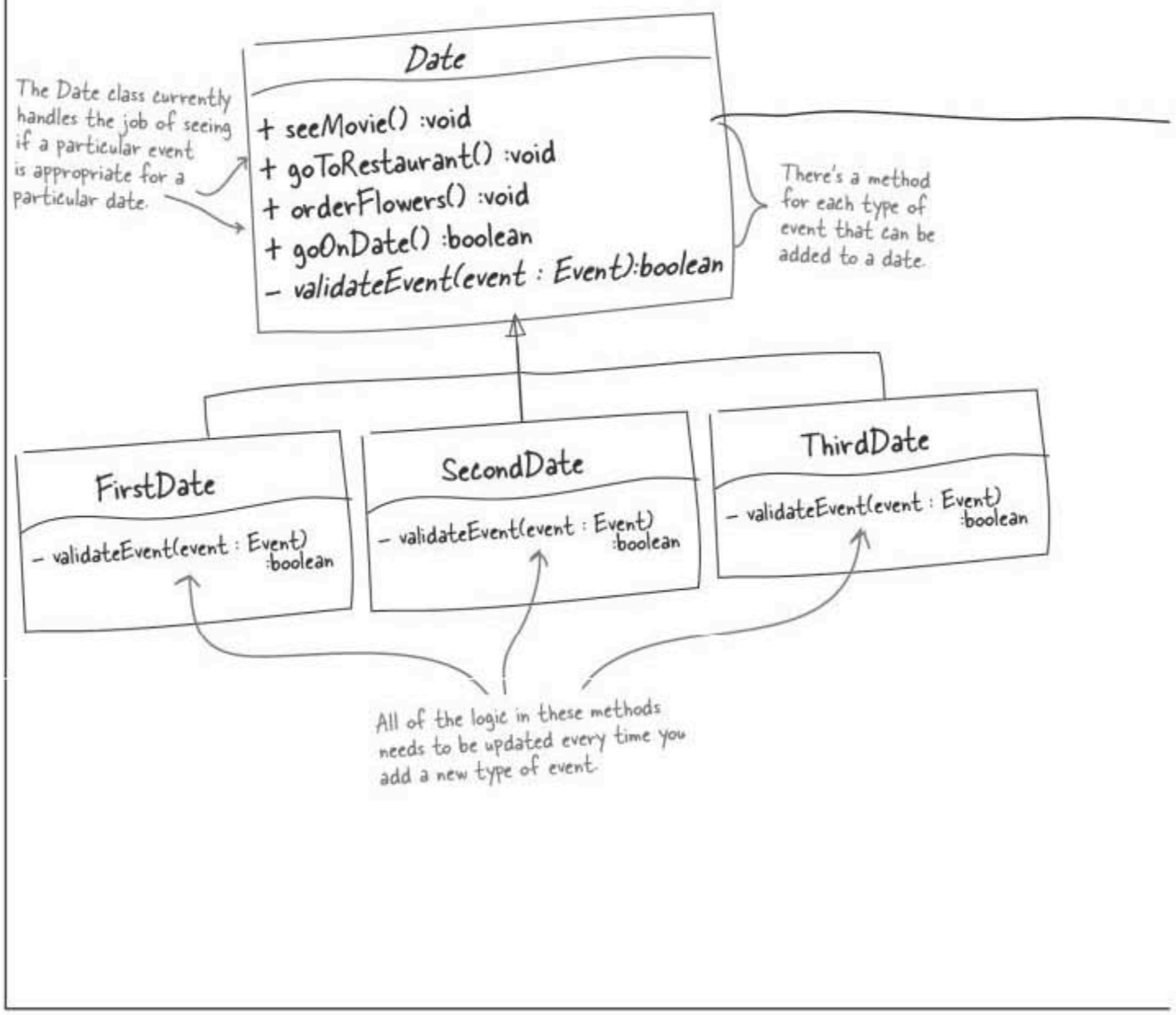
When a new type of event is added, the single responsibility principle states that all you should really need to do is add the new event class, and then you're done. However, with the current design, adding a new event also requires changes in the `Date` class ***and*** all of its subclasses.

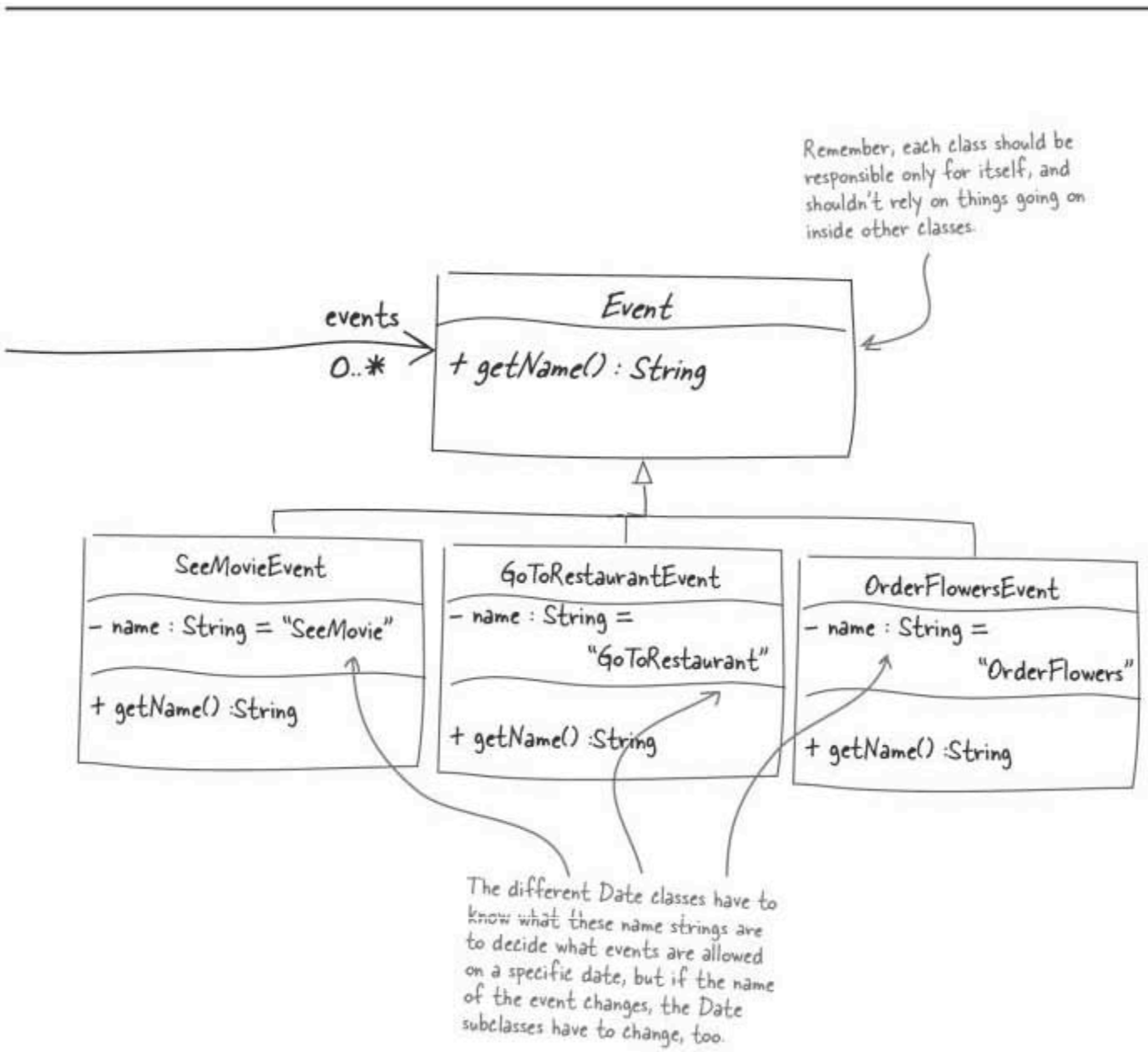


You've implemented the single responsibility principle correctly when each of your objects has only one reason to change.

Long Exercise

Your design at the moment makes it hard work to add events, change event names, and even deal with additional dates. Take a look at the current design and mark up what changes you'd make to apply the single responsibility principle to the iSwoon design (and in the process, make it easier to add new events and dates).





Answers on page 162.

If you're feeling stuck, turn the page for more on the single responsibility principle...

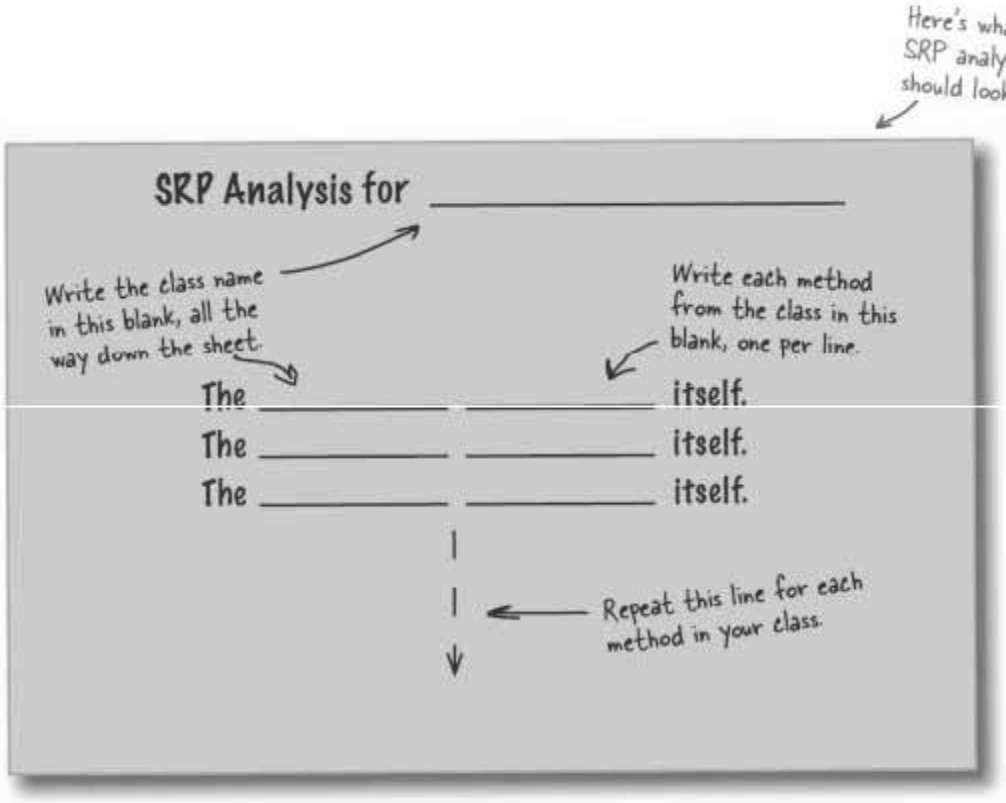
Spotting multiple responsibilies in your design

Most of the time, you can spot classes that aren't using the SRP with a simple test:

1. On a sheet of paper, write down a bunch of lines like this: The [blank] [blanks] itself. You should have a line like this for every method in the class you're testing for the SRP.
2. In the first blank of each line, write down the class name. In the second blank, write down one of the methods in the class. Do this for each method in the class.
3. Read each line out loud (you may have to add a letter or word to get it to read normally). Does what you just said make any sense? Does your class really have the responsibility that the method indicates it does?

If what you've just said doesn't make sense, then you're probably violating the SRP with that method. The method might belong in a different class—think about moving the method.

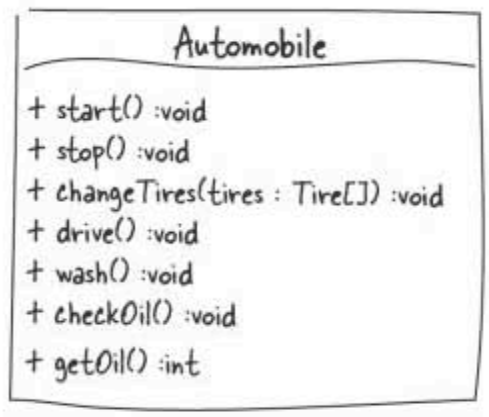
Here's what your SRP analysis sheet should look like.



Sharpen your pencil

Apply the SRP to the Automobile class.

Do an SRP analysis on the Automobile class shown below. Fill out the sheet with the class name methods in Automobile, like we've described on the last page. Then, decide if you think it makes sense for the Automobile class to have each method, and check the right box.



SRP Analysis for Automobile

The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.
The _____ _____ itself.

Follows SRP	Violates SRP
☐	☐
☐	☐
☐	☐
☐	☐
☐	☐
☐	☐
☐	☐

If what you read doesn't make sense, then the method on that line is probably violating the SRP.

Apply the SRP to the Automobile class.

Your job was to do an SRP analysis on the Automobile class shown below. You should have filled out the sheet with the class name methods in Automobile, and decided if you think it makes sense for the Automobile class to have each method.

It makes sense that the automobile is responsible for starting and stopping. That's a function of the automobile.

An automobile is NOT responsible for changing its own tires, washing itself, or checking its own oil.

SRP Analysis for Automobile

				Follows SRP	Violates SRP
The	Automobile	start[s]	itself.	☑	☐
The	Automobile	stop[s]	itself.	☑	☐
The	Automobile	changesTires	itself.	☐	☑
The	Automobile	drive[s]	itself.	☐	☑
The	Automobile	wash[es]	itself.	☐	☑
The	Automobile	check[s] oil	itself.	☐	☑
The	Automobile	get[s] oil	itself.	☑	☐

You may have to add an "s" or a word or two to make the sentence readable.

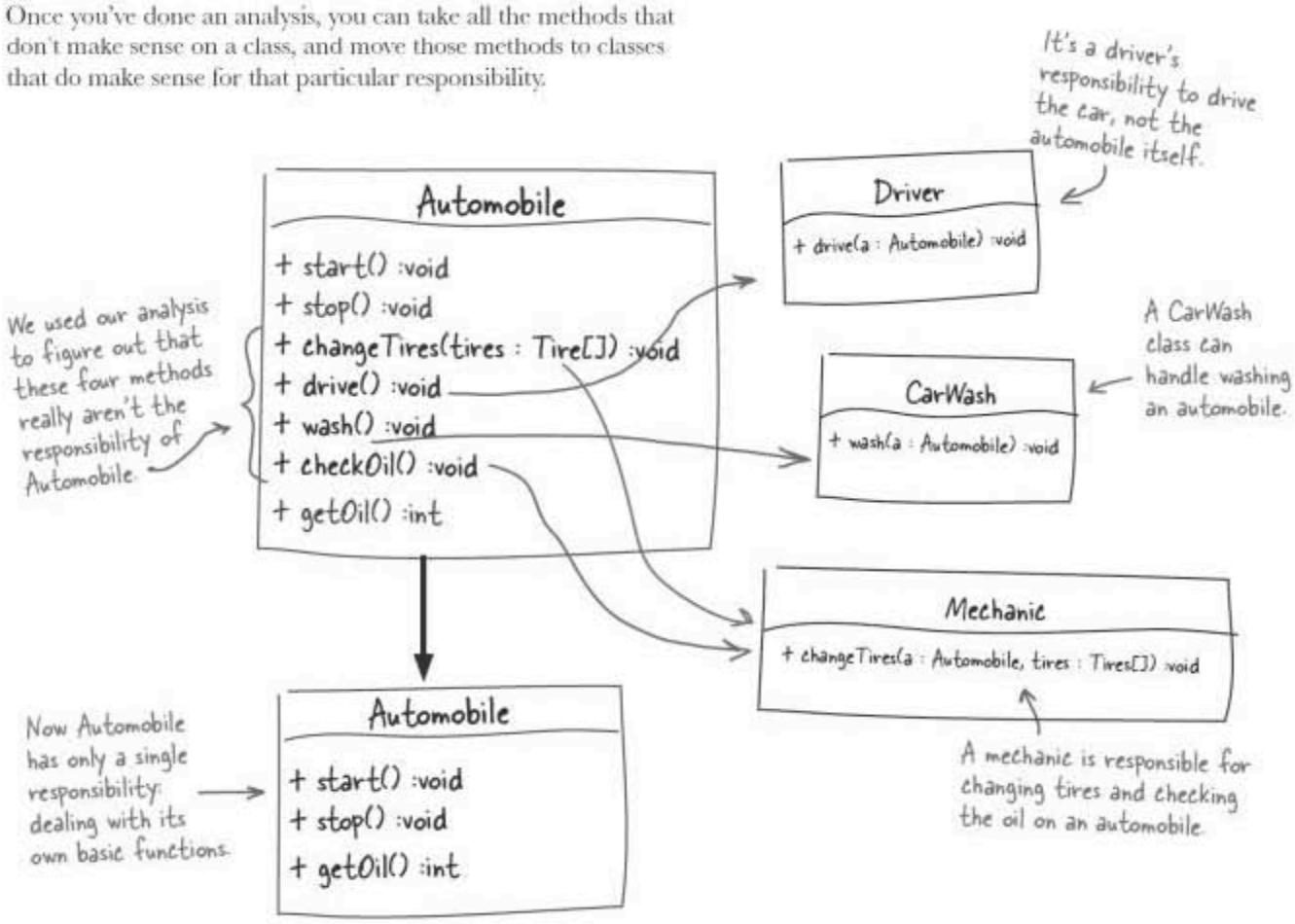
This one was a little tricky—we thought that while an automobile might start and stop itself, it's really the responsibilty of a driver to drive the car.

You should have thought carefully about this one, and what "get" means. This is a method that just returns the amount of oil in the automobile—and that is something that the automobile should do

Cases like this are why SRP analysis is just a guideline. You still are going to have to make some judgment calls using common sense and your own experience.

Going from multiple responsibilies to a single responsibility

Once you've done an analysis, you can take all the methods that don't make sense on a class, and move those methods to classes that do make sense for that particular responsibility.



there are no Dumb Questions

Q: How does SRP analysis work when a method takes parameters, like wash(Automobile) on the CarWash class?

A: Good question! For your SRP analysis to make any sense, you need to include the parameter of the method in the method blank. So you would write "The CarWash washes [an] automobile itself." That method makes sense (with the `Automobile` parameter), so it would stay on the `CarWash` class.

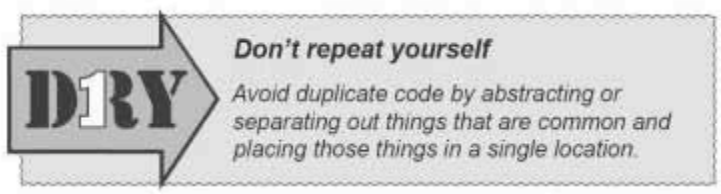
Q: But what if CarWash took in an Automobile parameter as part of its constructor, and the method was just wash()? Wouldn't SRP analysis give you a wrong result?

A: It would. If a parameter that might cause a method to make sense, like an `Automobile` for the `wash()` method on `CarWash`, is passed into a class's constructor, your SRP analysis might be misleading. But that's why you always need to apply a good amount of your own common sense and knowledge of the system in addition to what you learn from the SRP analysis.

Your design should obey the SRP, but also be DRY...

The SRP is all about responsibility, and which objects in your system do what. You want each object that you design to have **just one responsibility** to focus on—and when something about that responsibility changes, you'll know exactly where to look to make those changes in your code. Most importantly you'll avoid what's called the **ripple effect**, where one small change to your software can cause a ripple of changes throughout your code.

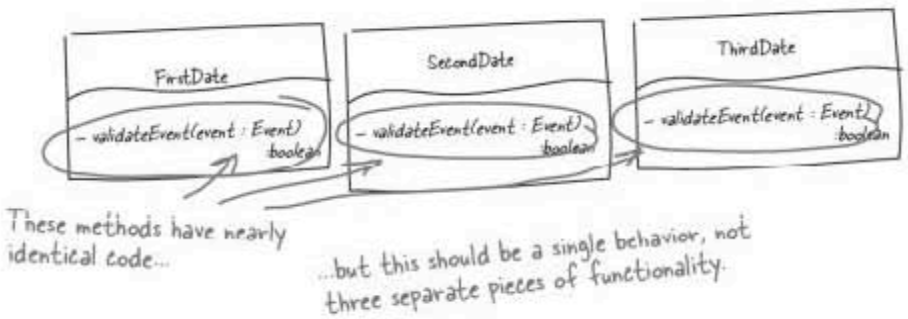
But there's a principle that goes hand in hand with SRP, and that's DRY:



The different Date classes are not DRY

Each of the different `Date` classes (`FirstDate`, `SecondDate`, `ThirdDate`) have almost identical behavior in their `validateEvent()` methods. This not only breaks the SRP, but means that one change in logic—like specifying that you can actually Sleep Over on the second date—would result in changes to the logic in all three classes.

This quickly turns into a maintenance nightmare.



DRY is about having each piece of information and behavior in your system in a single, sensible place.

there are no Dumb Questions

Q: SRP sounded a lot like DRY to me. Aren't both about a single class doing the one thing it's supposed to do?

A: They are related, and often appear together. DRY is about putting a piece of functionality in a single place, such as a class; SRP is about making sure that a class does only one thing, and that it does that one thing well. In well-designed applications, one class does one thing, and does it well, and no other classes share that behavior.

Q: Isn't having each class do only one thing kind of limiting?

A: It's not, when you realize that the one thing a class does can be a pretty ***big*** thing. For example, the `Event` class in iSwoon and its subclasses only store and manage one thing, the details of the specific event. Currently those details are only the name of the event, but those classes could store any of a host of details about an event, such as times, dates, notifications and alarms, even addresses. However all this extra information is still only about ***one thing***, describing an event. The different `Event` classes do that one thing, and that's all they do, so they are great examples of the SRP.

Q: And using SRP will help my classes stay smaller, since they're only doing one thing, right?

A: Actually, the SRP will often make your classes bigger. Since you're not spreading out functionality over a lot of classes—which is what many programmers not familiar with the SRP will do—you're often putting more things into a class.

But using the SRP will usually result in fewer classes, and that generally makes your overall application a lot simpler to manage and maintain.

Q: I've heard of something called cohesion that sounds a lot like this. Are cohesion and the SRP the same thing?

A: Cohesion is actually just another name for the SRP. If you're writing **highly cohesive software**, then you're correctly applying the SRP. In the current iSwoon design, a `Date` does two things: it creates events *and* it stores the events that are happening on that specific date. When a class is cohesive, it has **one** main job. So in the case of the `Date` class, it makes more sense for the class to focus on storing events, and give up the responsibility for actually creating the events.

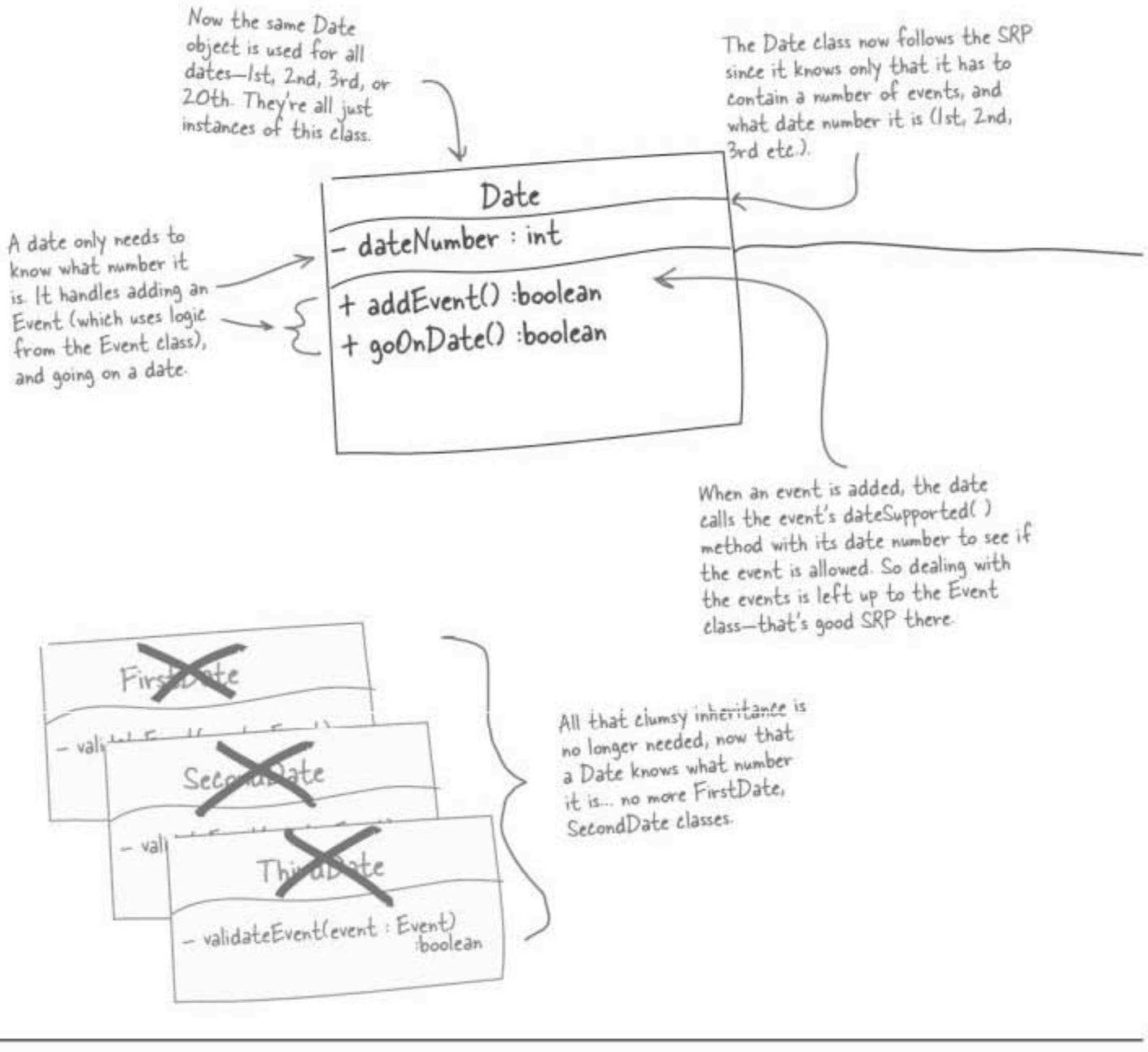
Want to know more about design principles? Check out Head First Object-Oriented Analysis and Design.

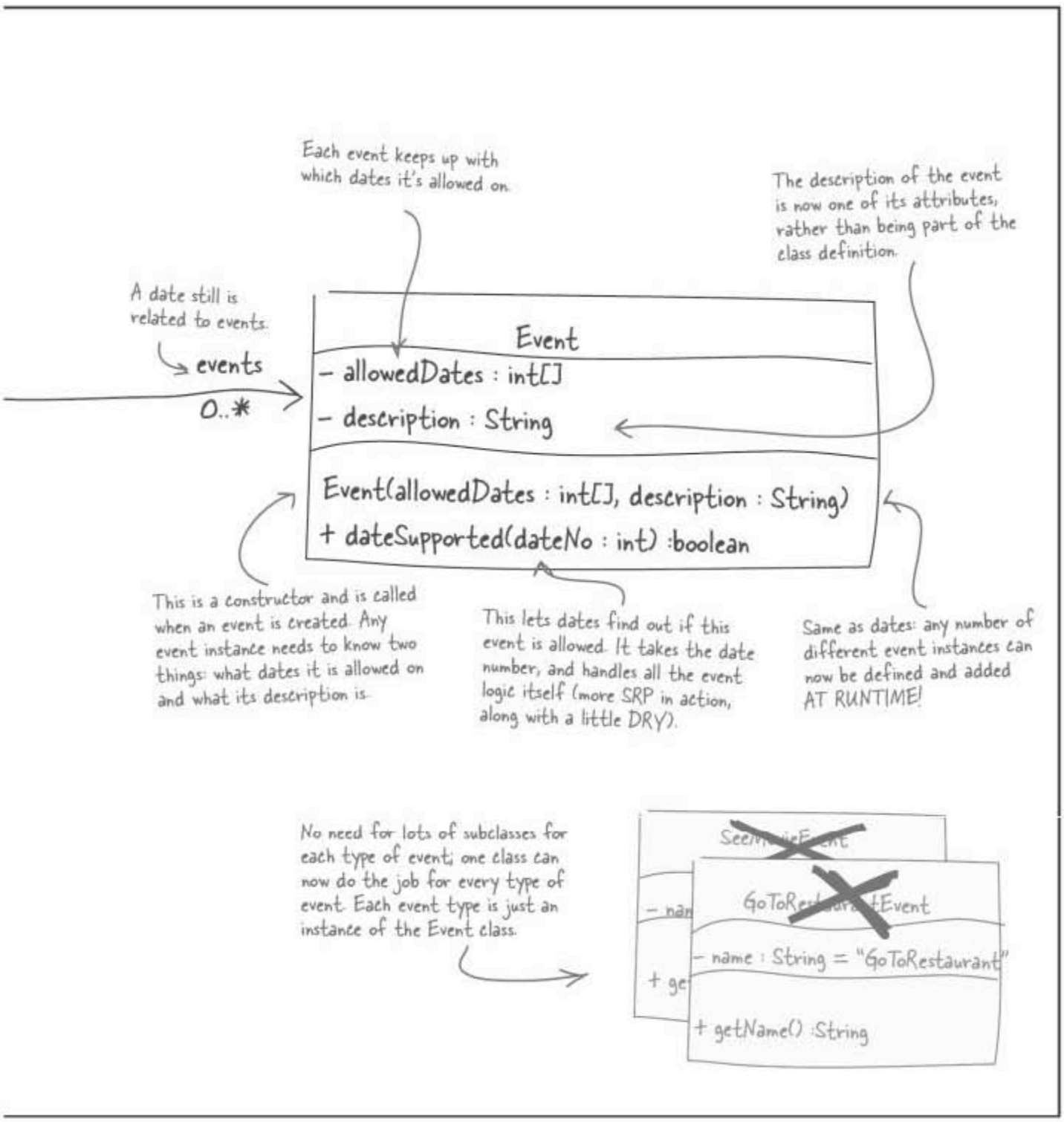
You've been loaded up with hints on how to make the iSwoon design, now make sure you've worked through and solved the exercise on pages 154 and 155 before turning the page...

For extra points, try to apply DRY as well as SRP to come up with a really great design.

Long Exercise Solution

You were asked to take a look at the current design and mark up what changes you'd make to apply the single responsibility principle to the iSwoon design to make it a breeze to update your software.




Each event keeps up with which dates it's allowed on.
The description of the event is now one of its attributes, rather than being part of the class definition.
A date still is related to events.
events
0..*
Event
- allowedDates : int[]
- description : String
Event(allowedDates : int[], description : String)
+ dateSupported(dateNo : int) :boolean
This is a constructor and is called when an event is created. Any event instance needs to know two things: what dates it is allowed on and what its description is.
This lets dates find out if this event is allowed. It takes the date number, and handles all the event logic itself (more SRP in action, along with a little DRY).
Same as dates: any number of different event instances can now be defined and added AT RUNTIME!
No need for lots of subclasses for each type of event; one class can now do the job for every type of event. Each event type is just an instance of the Event class.
- name : String = "GoToRestaurant"
+ getName() :String

The post-refactoring standup meeting...

Bob: Got it all done, we now have a really flexible piece of software that can support any number of different types of dates and events.

Laura: That's great! Sounds like the extra work might pay off for us; we've got a ton of new events to add...

Bob: Oh, it will. Now we can just write one or two lines of code, and, boom, the new event is in the system. We allowed between two and five days for each event, and now it only takes a day, at most.

Mark: You're not kidding. I've already added all the new events. And I'm sure we could make some more improvements as well...

Laura: Wait, just hang on a sec. For now the software is *more* than good enough, actually. Let's not starting making more changes just because we can.

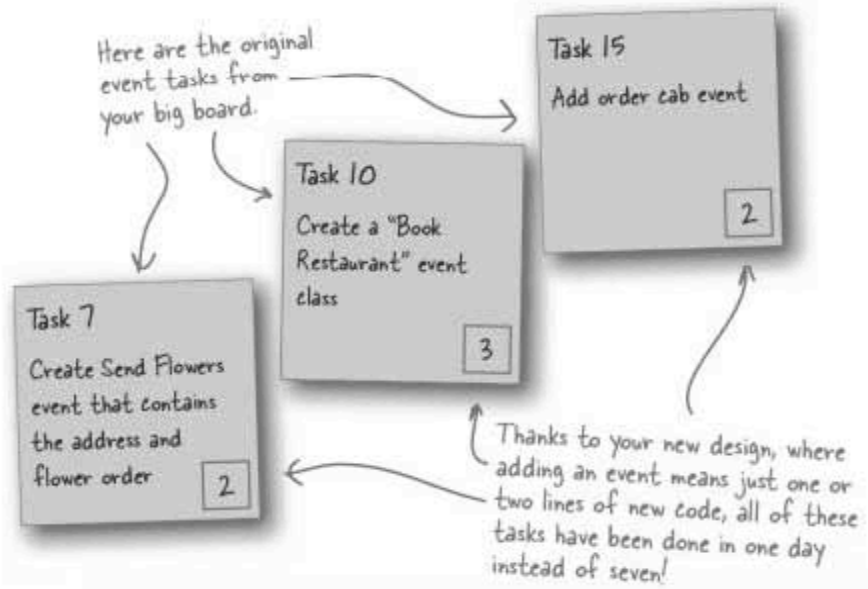
Mark: So what's next?

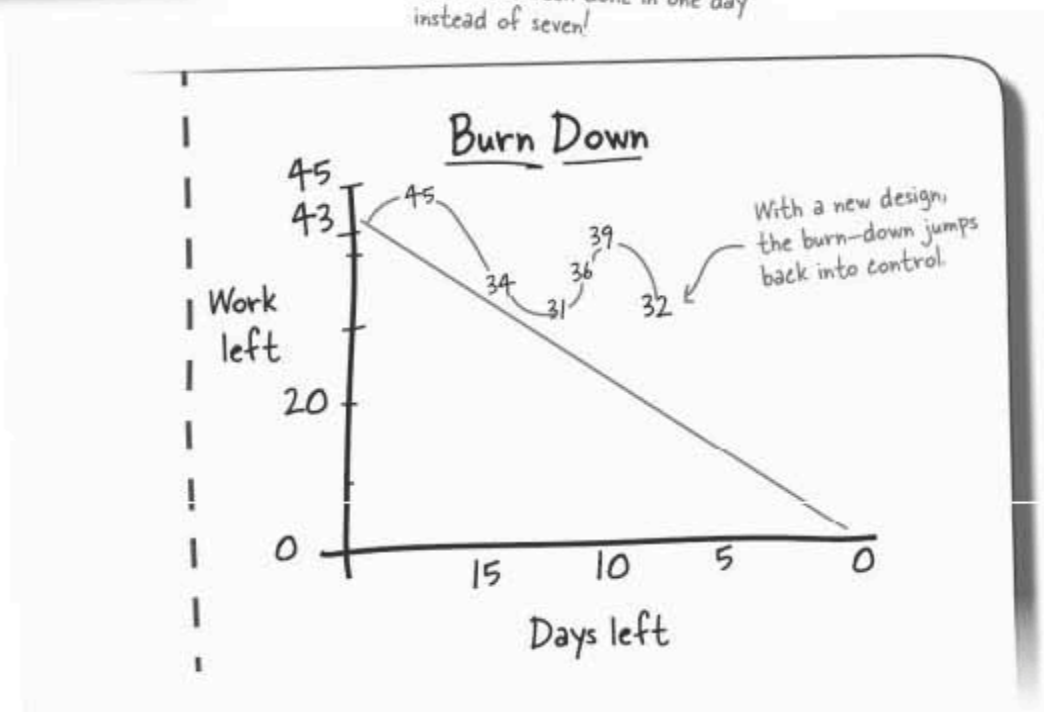
Bob: Well, now that I've got the refactoring done, it looks like we have some time to focus on the demo that the Starbuzz CEO wanted...

there are no Dumb Questions

Q: When Laura says that the code is good enough, what does she mean?

A: Good question! We'll talk a lot more about testing in Chapters 7 and 8 and how you can be confident, and prove that your code does what it should.

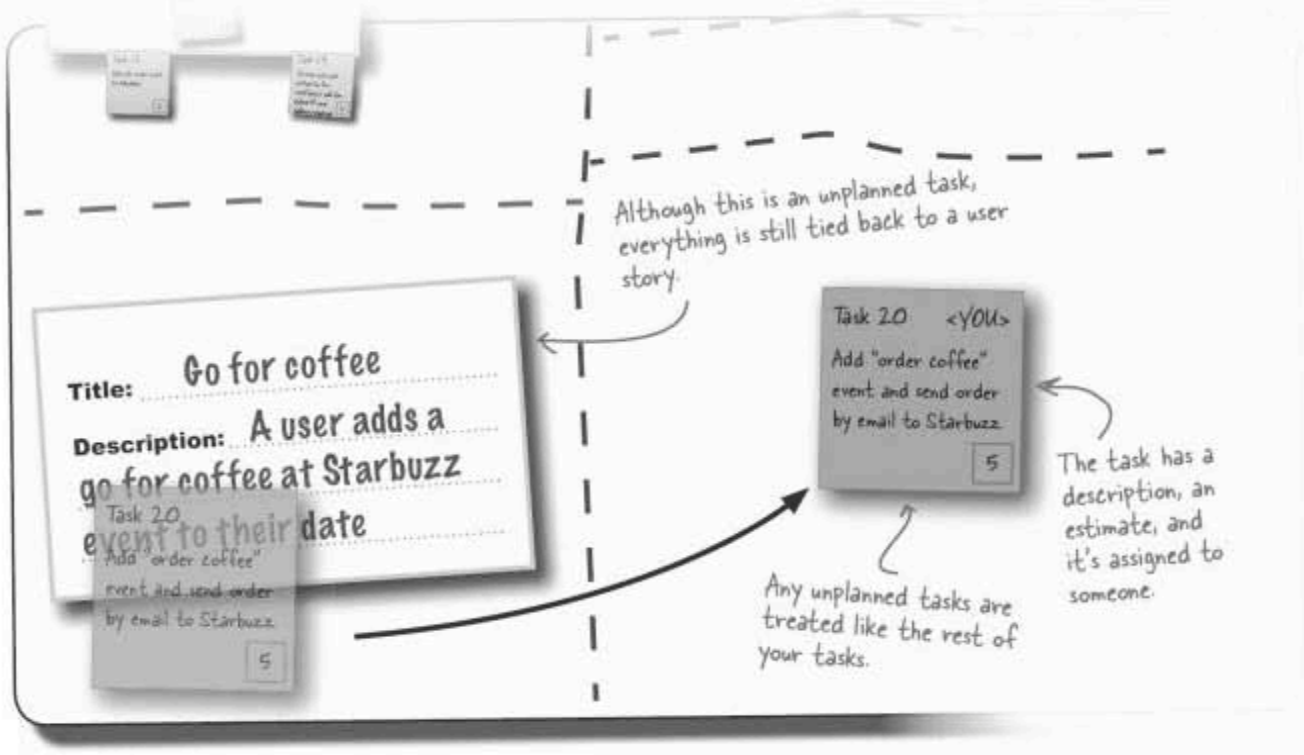




A great design helps you be more PRODUCTIVE as well as making your software more FLEXIBLE.

Unplanned tasks are still just tasks

The Starbuzz CEO's demo is an unplanned task, but you deal with it just like all the other tasks on your board. You estimate it, move it to the In Progress section of your board, and then go to work.



Unplanned tasks *on the board* become planned.

An unplanned task may start out differently, but once it goes on your board, it's treated just like all your planned tasks. In fact, as soon as you assign the task and give it an estimate, it really isn't unplanned anymore. It's just another task that has to be handled, along with everything else in your project.

And that's how you handle a task that starts out unplanned from its inception to completion: just like any other task. You estimate it, move it to the In Progress section of your board, and work it until it's done. Then you move it into the Completed section and move on.

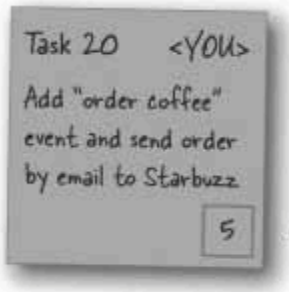
It doesn't matter how a task starts out. Once it's on your board, it's got to be assigned, estimated, and worked on until it's complete.

Part of your task is the demo itself

In addition to the time you'd spend working on the demo, you've got to think about time spent actually ***doing*** the demo. If you and your lead web programmer both spend a day traveling to Starbuzz and showing off iSwoon, that's got to be part of your task estimate.

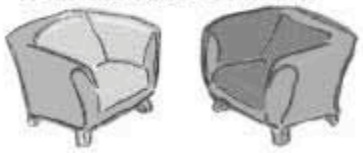
Your estimates should be complete

When you're estimating your tasks, you should come up with the time it takes to complete the task—and sometimes that involves more than just code. If you've got to demo the code or meet with a stakeholder, include time for those activities, too.



Four days to do the development, and another day for the actual demo and to field follow-up questions

Fireside Chats



Tonight's talk: **A sit-down discussion between Perfect Design and "Good Enough" Design.**

"Good Enough" Design

Hi! So you're a Perfect Design? Man, I've always dreamed about meeting you!

Why's that?

Yeah, I suppose so. As long as I help everyone be productive and meet their deadlines, and the customer is getting the software they need, then I'm doing my job.

Huh, I never thought of it like that. I thought when you came along everyone would be all hugs and kisses...

What do you mean? After all that hard work your team might still be able to make you even more, err...perfect?

Perfect Design:

Thanks. Designs like me are pretty rare. In fact I may be the only one you'll ever meet.

Well, the problem is that it's really hard to come up with a design that everyone thinks is perfect. There's always somebody out to get me with their criticisms. And with refactoring, I keep getting changed. But you're pretty valuable yourself, you know...

You see, that's the thing. People spend so much time on me that they never meet their deadlines, they never deliver software, and they never get paid. That can make me pretty unpopular. It kind of sucks, really.

Not at all. Usually by the time I show up, the team is running late and I can't help out anywhere near as much as as they thought. And then there's always the danger that I'm not completely perfect...

Unfortunately, yes. You see, perfection is a bit of a moving target. Sometimes, I just wish I could be like you and actually deliver. Maybe not great, but—

"Good Enough" Design:

Hey, wait a second. That sounded pretty condescending.

Yeah, I suppose everyone is pretty stoked when I help them get great software out of the door. But I always figured that I was second class somehow and that they loved you...

So really what you're saying is that you'd like to be a design for software that actually got delivered?

So, I guess I'm good enough to get the job done, to meet the customer's needs, and to be easy enough to work with that my developers can develop code on time. That's what really matters.

Perfect Design:

Well, sure. Everyone ships you out because you draw a line in the sand and say you're finished when the customer gets what they want. So even though you're not perfect you deliver. And a developer who delivers great software, whether it's designed perfectly or not, is a happy developer.

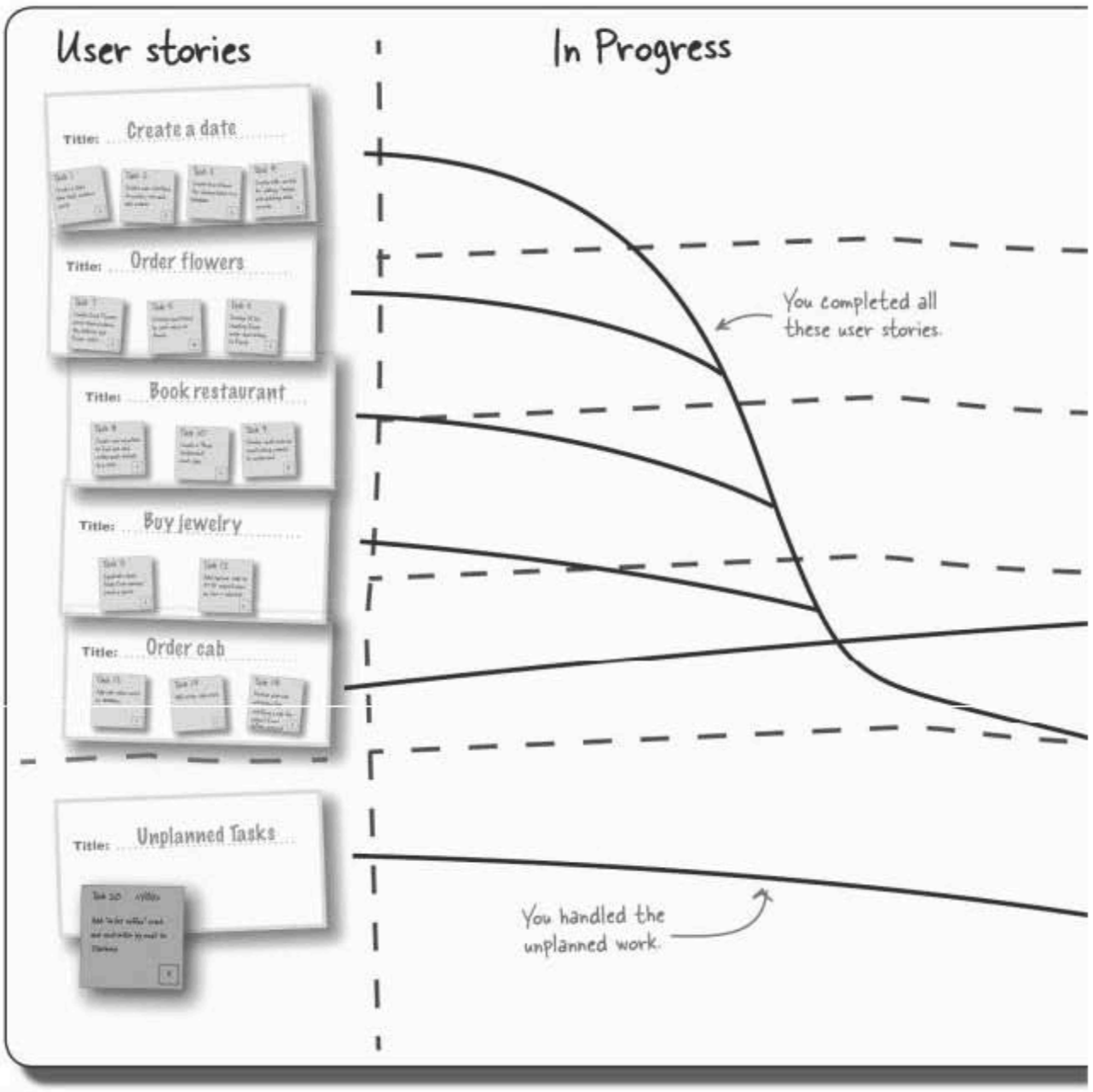
If by love, you mean "never have time for," then you're right.

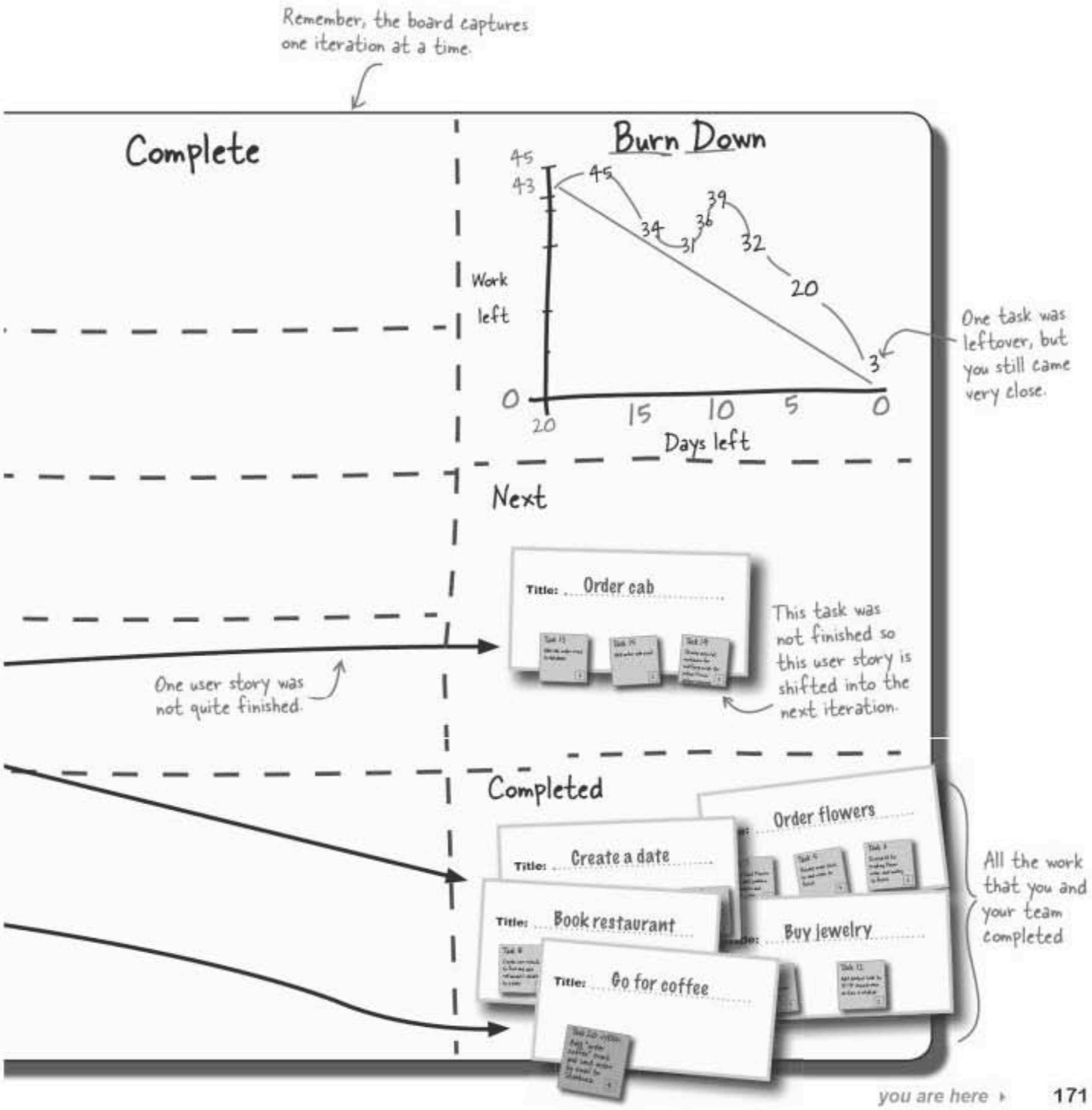
Exactly! I aspire to be you, in many respects. People *want* to meet their deadlines and to ship software that the customer will sign off on. That's not settling; that's just being good developers and getting paid. You know developers, right, those guys that get paid for delivering? Well, I'm not in their good graces when they've come up with me and no software to actually ship...

Yep, don't ever put yourself down. In this world it's nice to be perfect, but it's better to be ready and shipping.

When everything's complete, the iteration's done

Once you finish all your tasks, including any unplanned demos for forward-looking coffee addicts, you should end up with all your user stories, and the tasks that make them up, in your completed area of the board. And when you've got that, you're finished! There's nothing magical about it: when the work is done, so is your iteration.




Remember, the board captures one iteration at a time.
Complete
Burn Down
45
43
45
34
31
36
39
32
20
3
0
Work left
20
15
10
5
0
Days left
One task was leftover, but you still came very close.
Next
Title: Order cab
This task was not finished so this user story is shifted into the next iteration.
One user story was not quite finished.
Completed
Order flowers
Title: Create a date
Title: Book restaurant
Buy jewelry
Title: Go for coffee
All the work that you and your team completed

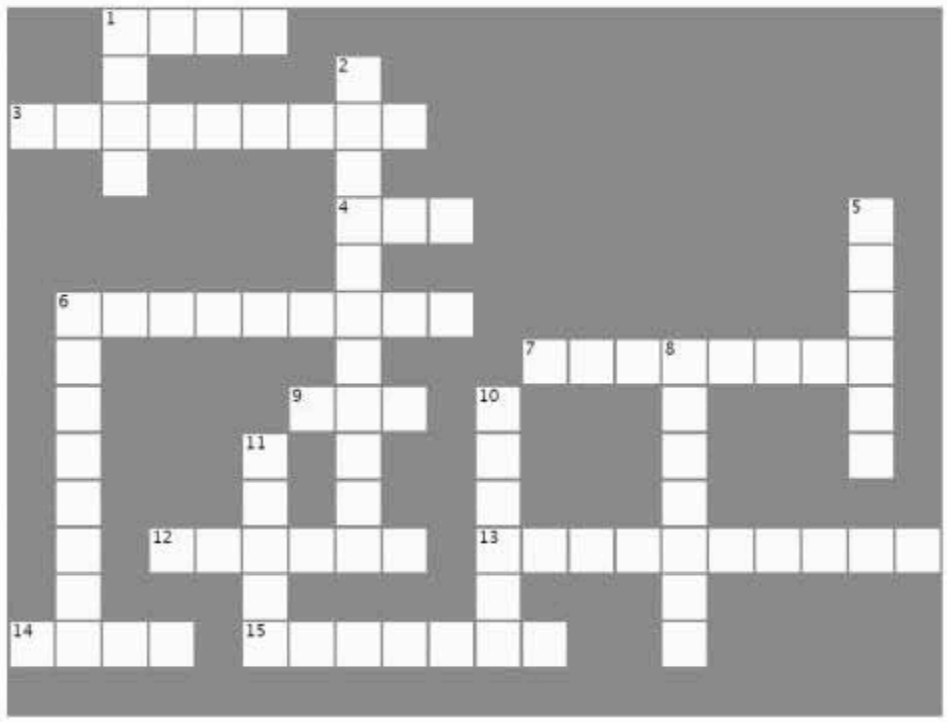
WHAT'S MY PURPOSE?

Take each of the following techniques and artifacts from this chapter and match it to what it does.

Unplanned tasks and user stories	I help you make sure that everything has its place, and that place is only **one** place.
Perfect design	With me, the design gets better with small improvements throughout your code.
SRP	I make sure that the unexpected becomes the expected and managed.
Refactoring	My mantra is, "Perfect is great, but I deliver."
DRY	I make sure that all the parts of your software have one well-defined job.
Good-enough design	I'm what you strive for, but ultimately you might not deliver.

Software Development Design Cross

Let's put what you've learned to use and stretch out your left brain a bit! All of the words below are somewhere in this chapter. Good luck!



Across

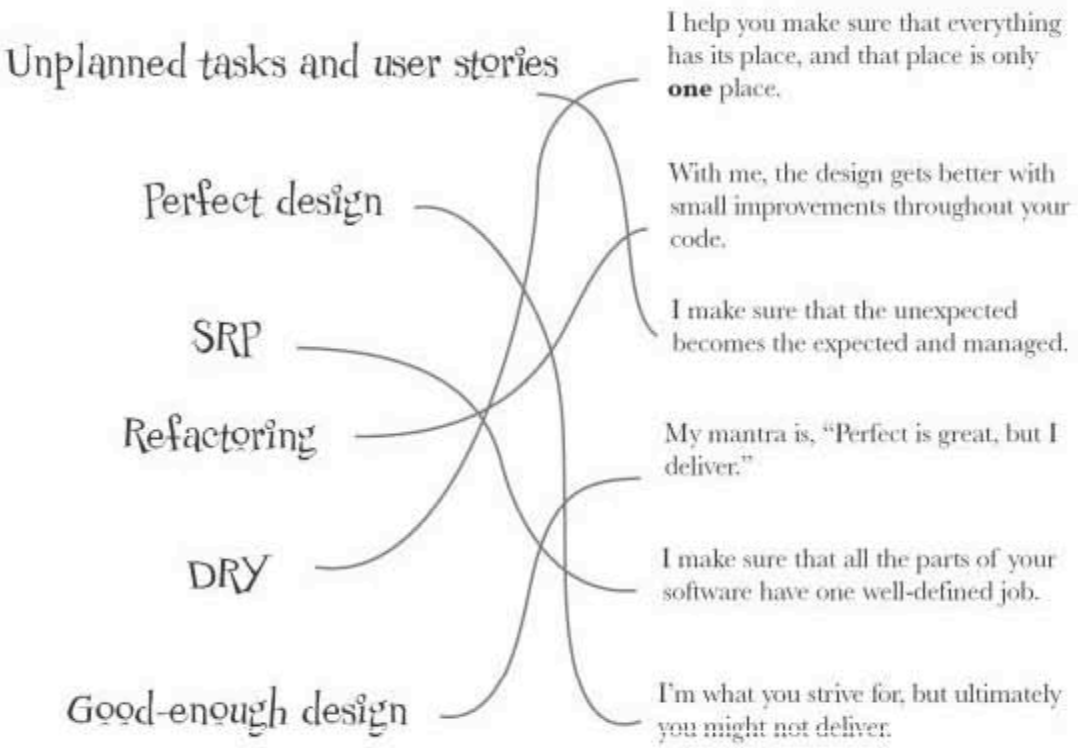
1. Great developers .....
3. When an unplanned task is finished it is moved into the ..... column.
4. Your burn down rate should show ..... the work on your board, including any new unplanned tasks.
6. When a task is finished it goes in the ..... column.
7. An unplanned user story and its tasks are moved into the ..... bin on your project board when they are all finished.
9. If you find you are cutting and pasting large blocks of your design and code then there's a good chance that you're breaking the ..... principle.
12. ..... is the only constant in software development.
13. When a design helps you meet your deadlines, it is said to be a .... design.
14. If a user story is not quite finished at the end of an iteration, it is moved to the ..... bin on your project board.
15. A good enough design helps you .....

Down

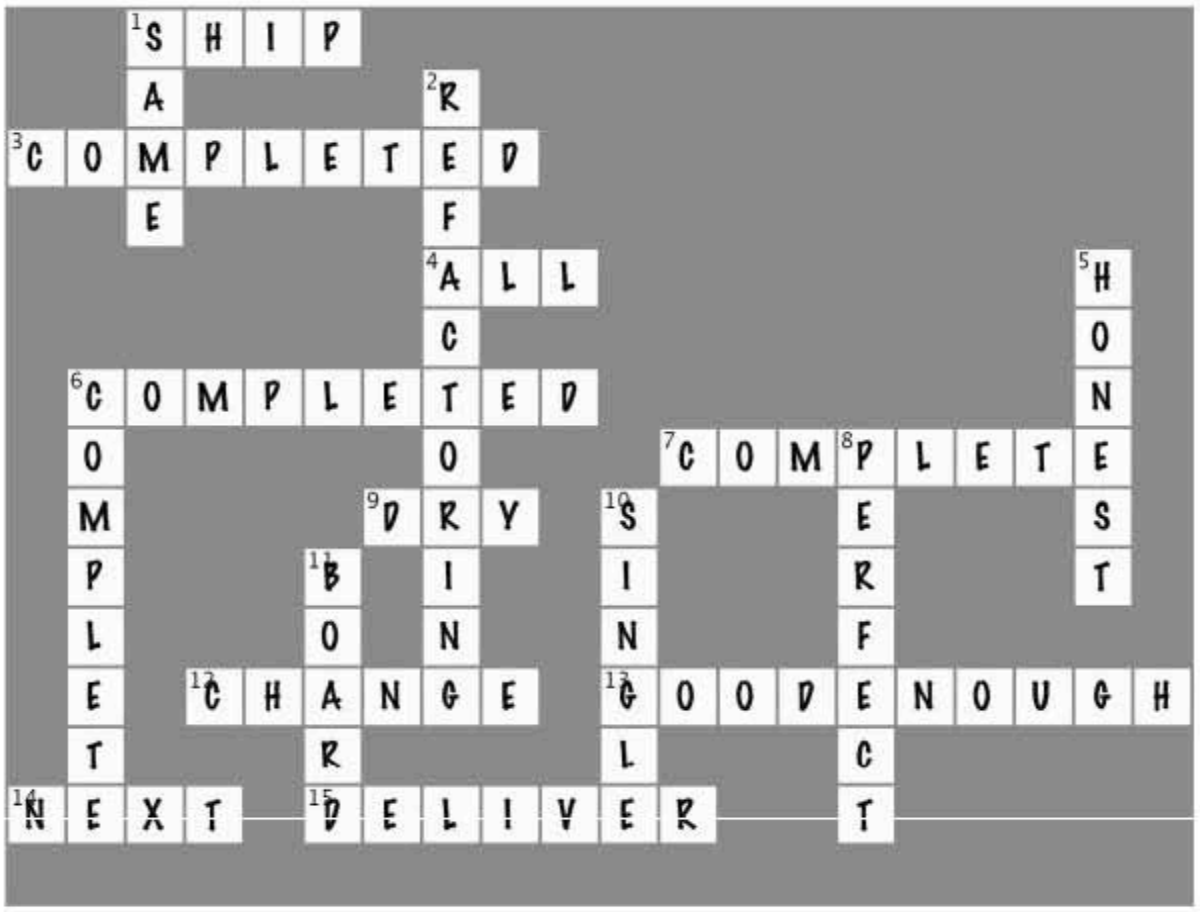
1. Unplanned tasks are treated the ..... as unplanned tasks once they are on your board.
2. When you improve a design to make it more flexible and easier to maintain you are ..... the design.
5. You should always be ..... with your customer.
6. When all the tasks in a user story are finished, the user story is transferred to the ..... bin
8. Striving for a ..... design can mean that you never actually cut any code.
10. When a class does one job and it's the only class that does that job it is said to obey the ..... responsibility principle.
11. An unplanned task is going to happen in your current iteration once you have added it to your .....

WHAT'S MY PURPOSE? SOLUTION

Take each of the following techniques and artifacts from this chapter and match it to what it does.

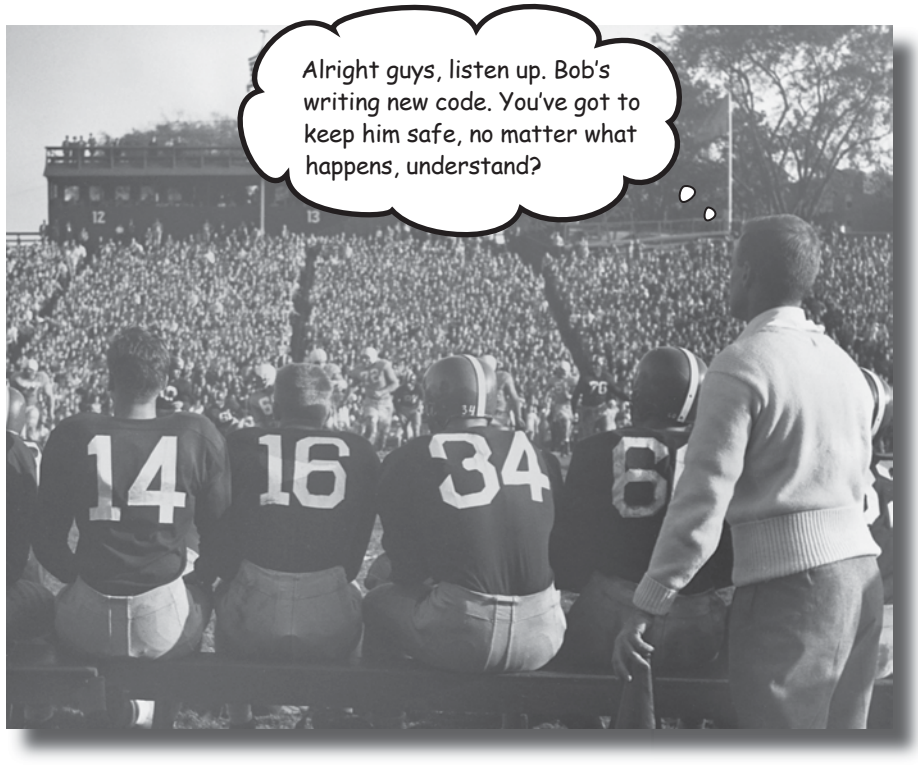


Software Development Design Cross Solution



6 version control

Defensive development



When it comes to writing great software, *Safety First!*

Writing great software isn't easy...especially when you've got to make sure your code works, and **make sure it keeps working**. All it takes is one typo, one bad decision from a co-worker, one crashed hard drive, and suddenly all your work goes down the drain. But with **version control**, you can make sure your **code is always safe** in a code repository, you can **undo mistakes**, and you can make **bug fixes**—to new and old versions of your software.

You've got a new contract—BeatBox Pro

Congratulations—you've been getting rave reviews from iSwoon, and you've landed a new contract. You've been hired to add two new features to the legendary *Head First Java* BeatBox project. BeatBox is a multi-player drum machine that lets you send messages and drum loops to other users over the network.

Like every other software development project out there, the customer wants things done as soon as possible. They even let you bring along Bob, one of your junior developers, to help out. Since the stories aren't big enough to have more than one person work on them at a time, you'll work on one and Bob will work on the other. Here are the user stories for the new features you've got to add:

Title: Send a Poke to other users

Description: Click on the "Send a Poke" button to send an audible and visual alert to the other members in the chat. The alert should be short and not too annoying—you're just trying to get their attention.

Priority: 20 **Estimate:** 3

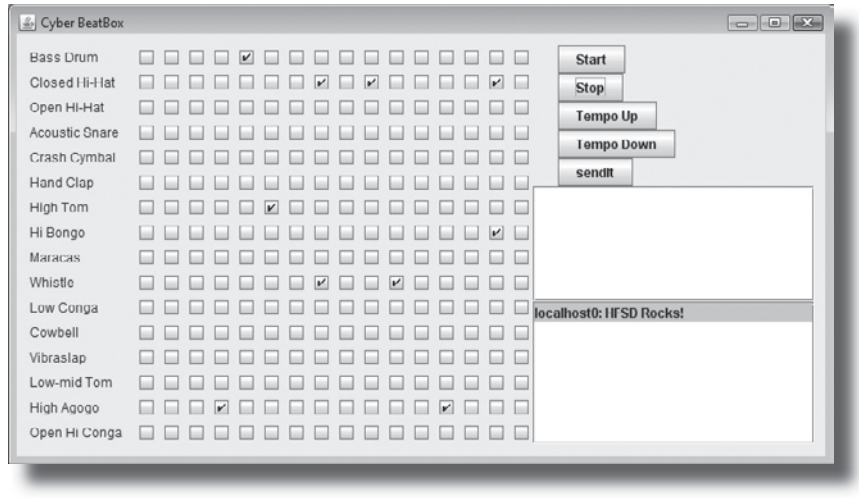
You'll take tasks associated with this story.

Title: Send a picture to other users

Description: Click on the "Send a Picture" button to send a picture (only JPEG needs to be supported) to another user. The other user should have the option to not accept the file. There are no size limits on the file being sent.

Priority: 20 **Estimate:** 4

Bob will pull tasks from this story.



The BeatBox program from Head First Java, our starting point.

*You can download the code that we're starting with from http://www.headfirstlabs.com/books/hfsd/

Task Stickies / Magnets

Let's get right to the new features. Here's a snippet from the BeatBox client code. Your job is to map the task stickies to the code that implements each part of the "Send a Poke..." story. We'll get to the GUI work in a minute.

```
// ... more BeatBox.java code above this

public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
    try {
      while((obj=in.readObject()) != null) {
        System.out.println("got an object from server");
        System.out.println(obj.getClass());
        String nameToShow = (String) obj;
        checkboxState = (boolean[]) in.readObject();

        if (nameToShow.equals(POKE_START_SEQUENCE)) {
          playPoke();
          nameToShow = "Hey! Pay attention.";
        }

        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
      } // close while
    } catch (Exception ex) { ex.printStackTrace(); }
  } // close run

  private void playPoke() {
    Toolkit.getDefaultToolkit().beep();
  }
} // close inner class
```

Task 1 — MDE

Sound an audible alert when receiving a poke message (can't be annoying!)

.5

Task 2 — LUG

Add support for checking for the Poke command and creating a message.

.5

Task 4 — BJD

Merge Poke visual alert into message display system.

.5

Task 3 — MDE

Implement receiver code to read the data off of the network.

1

Stickies
Task ~~Magnets~~ Solution

We're not in *Head First Java* anymore; let's get right to the new features. Here's a snippet from the BeatBox client code. Your job was to map the task magnets to the code that implements each part of the "Send a Poke..." story.

All of this code goes into BeatBox.java.

```
// ... more BeatBox.java code above this

public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
    try {
      while((obj=in.readObject()) != null) {
        System.out.println("got an object from server");
        System.out.println(obj.getClass());
        String nameToShow = (String) obj;
        checkboxState = (boolean[]) in.readObject();

        if (nameToShow.equals(POKE_START_SEQUENCE)) {
          playPoke();
          nameToShow = "Hey! Pay attention.";
        }

        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
      } // close while
    } catch (Exception ex) { ex.printStackTrace(); }
  } // close run

  private void playPoke() {
    Toolkit.getDefaultToolkit().beep();
  }
} // close inner class
```

Here's the code that will run in the new thread context for BeatBox.

This is the inner class that receives data from the server.

This is original code—it reads messages sent from the server.

If we get the POKE_START_SEQUENCE, we play the poke sound and replace the message with our alert text.

Here's our new playPoke() method that just beeps for now. If you want a real challenge, add MP3 poke-sound support.

Task 1 MDE
Sound an audible alert when receiving a poke message (can't be annoying!) .5

Task 2 LUG
Add support for checking for the Poke command and creating a message. .5

Task 3 MDE
Implement receiver code to read the data off of the network. 1

Task 4 BJD
Merge Poke visual alert into message display system. .5

there are no Dumb Questions

Q: This isn't a Java programming book. Why are we wasting time looking through all this code?

A: Software development techniques cover everything related to a project, from organization and estimation down through code. Earlier, we talked about the planning and execution parts of a project, and then we got a little closer to code and talked about design. Now, we need to dive all the way down and talk about some tools and techniques you can use ***on your code itself***. Software development isn't just about prioritization and estimation; you've still got to write good, working, reliable code.

Q: I don't develop in Java. I'm not sure what some of the code in there does. What do I do?

A: That's OK. Do your best to understand what the code is doing, and don't worry about all the Java-specific details. The main thing is to get an idea of how to handle and think about code in a solid software development process. The tools and techniques we'll talk about should make sense whether you know what a Java thread is or not.

Q: I think I must have...misplaced... my copy of *Head First Java*. What's this whole BeatBox thing about?

A: BeatBox is a program first discussed in Head First Java. It has a backend `MusicServer` and a Java Swing–based client piece (that's Java's graphical toolkit API). The client piece uses the Java Sound API to generate sound sequences that you can control with the checkboxes on the form's main page. When you enter a message and click "sendit," your message and your BeatBox settings are sent to any other copies of BeatBox connected to your `MusicServer`. If you click on the received message, then you can hear the new sequence that was just sent.

Q: So what's the deal with that POKE_START_SEQUENCE thing?

A: Our story requires us to send a poke message to the other BeatBoxes connected to the `MusicServer`. Normally when a message gets sent it's just a string that is displayed to the user. We added the Poke functionality on top of the original BeatBox by coming up with a unique string of characters that no one should ever type on purpose. We can use that to notify the other BeatBoxes that a "poke" was sent. This sequence is stored in the `POKE_START_SEQUENCE` constant (the actual string value is in the `BeatBox.java` file in the code you can download from *http://www.headfirstlabs.com/books/hfsd/*).

When other BeatBox instances see the `POKE_START_SEQUENCE` come through, they replace it with our visual alert message, and the receiving user never actually sees that code sequence.

Q: What's all this threading and Runnable stuff about?

A: BeatBox is always trying to grab data from the network so it can display incoming messages. However, if there's nothing available on the network, it could get stuck waiting for data. This means the screen wouldn't redraw and users couldn't type in a new message to send. In order to split those two things apart, BeatBox uses threads. It creates a thread to handle the network access, and then uses the main thread to handle the GUI work. The `Runnable` interface is Java's way of wrapping up some code that should be run in another thread. The code you just looked at, in the last exercise, is the network code.

Bob's making good progress on his end, too. Can you think of anything else you should be worrying about at this point?

And now the GUI work...

We need one more piece of code to get this story together. We need to add a button to the GUI that lets the user actually send the Poke. Here's the code to take care of that task:

Task 5 <YOU>

Add button to GUI to send Poke sequence to other BeatBox instances. .5

```
// The code below goes in BeatBox.java,
//   in the buildGUI() method
  JButton sendIt = new JButton("sendIt");
  sendIt.addActionListener(new MySendListener());
  buttonBox.add(sendIt);

  JButton sendPoke = new JButton("Send Poke");
  sendPoke.addActionListener(new MyPokeListener());
  buttonBox.add(sendPoke);

  userMessage = new JTextField();
  buttonBox.add(userMessage);

// Below is new code we need to add, also to BeatBox.java
public class MyPokeListener implements ActionListener {

  public void actionPerformed(ActionEvent a) {
    // We'll create an empty state array here
    boolean[] checkboxState = new boolean[255];

    try {
      out.writeObject(POKE_START_SEQUENCE);
      out.writeObject(checkboxState);
    } catch (Exception ex) {
      System.out.println("Failed to poke!");
    }
  }
}
```

First we need to create a new button for our Poke feature.

Then we set up a listener so we can react when it's clicked.

Finally, add the button to the box holding the other buttons.

Here we create an array of booleans for our state. We can leave them all false because the receiving side ignores them when it gets the POKE command.

Here's the magic: to send a poke we send the magic POKE_START_SEQUENCE and our array of booleans to the server. The server will relay our magic sequence to the other clients, and they'll beep at the user because of the earlier code we wrote (back on page 180).

And a quick test...

Now that both the client and server are implemented it's time to make sure things work. No software can go out without testing so...

1. First compile and start up the MusicServer.

The MusicServer will listen for connections and print out a line each time it gets one.

```
File Edit Window Help Buildin'
hfsd> mkdir bin
hfsd> javac -d bin src\headfirst\sd\chapter6\*.java
hfsd> java -cp bin headfirst.sd.chapter6.MusicServer
```

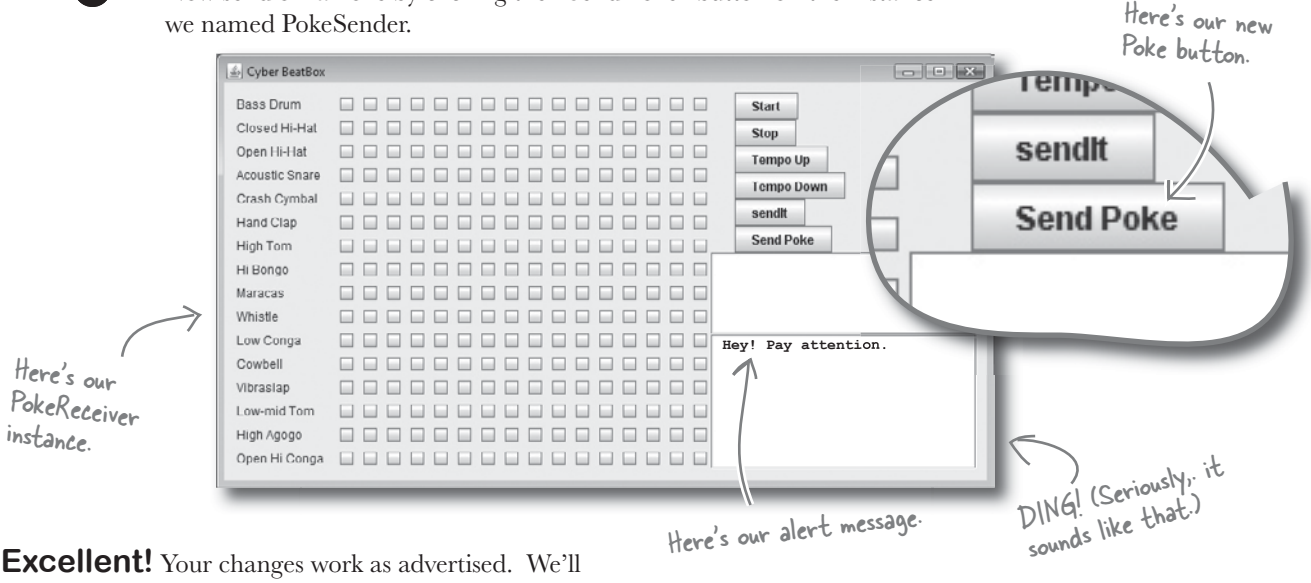
The "-d" tells the java compiler to put the classes in the bin directory.

2. Then start the new BeatBox—we'll need two instances running so we can test the Poke.

We use different names here so we know which is which.

```
File Edit Window Help Ouch
hfsd> java -cp bin headfirst.sd.chapter6.BeatBox PokeReceiver
File Edit Window Help Hah
hfsd> java -cp bin headfirst.sd.chapter6.BeatBox PokeSender
```

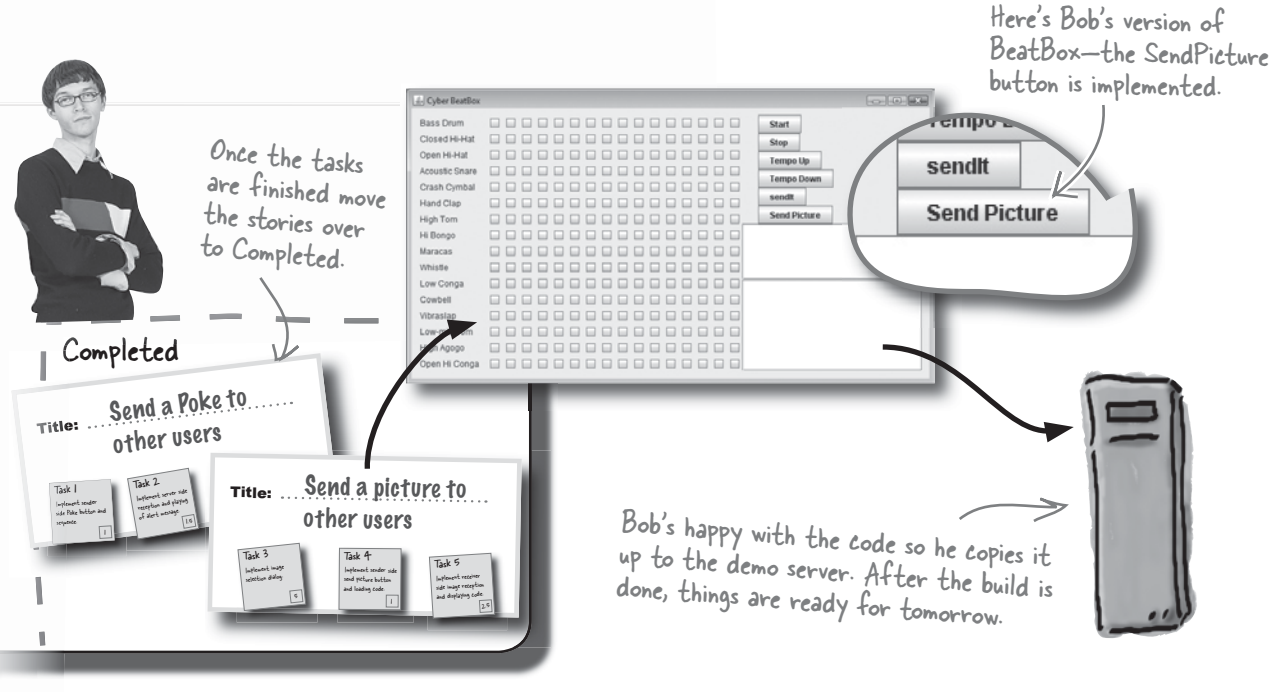
3. Now send off a Poke by clicking the "Send Poke" button on the instance we named PokeSender.



Excellent! Your changes work as advertised. We'll copy the code up to the demo server, and all that's left is for Bob to merge his stuff in. Time to call it a night.

And Bob does the same...

Bob finished up the tasks related to his story and ran a quick test on his end. His task is working, so he copies his code up to the server. In order to do the final build he merges his code in with ours, gets everything to compile, and retests sending a picture. Everything looks good. Tomorrow's demo is going to rock...



there are no Dumb Questions

Q: I'm not familiar with networking code. What's happening in that code we just added?

A: On the sending side we represent the sequence settings as an array of checkboxes. We don't really care what they're set to, since we won't use them on the receiving side. We still need to send something, though, so the existing code works. We use Java's object serialization to stream the array of checkboxes and our secret message that triggers the alert on the other side.

On the receiving side we pull off the secret sequence and the array of checkboxes. All of the serialization and deserialization is handled by Java.

Q: Why did we make the bin directory before we compiled the code?

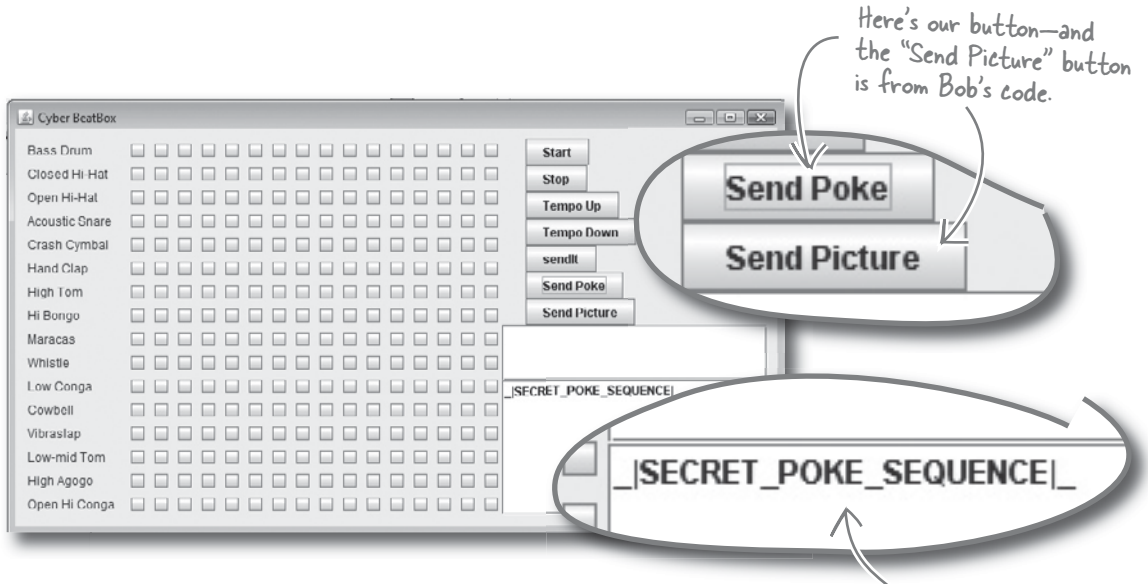
A: We'll talk more about this in the next chapter, but in general it's a good idea to keep your compiled code separate from the source. It makes it a lot simpler to clean up and rebuild when you make changes. There's nothing special about the name "bin"; it's just convention and is short for "binaries"—i.e., compiled code.

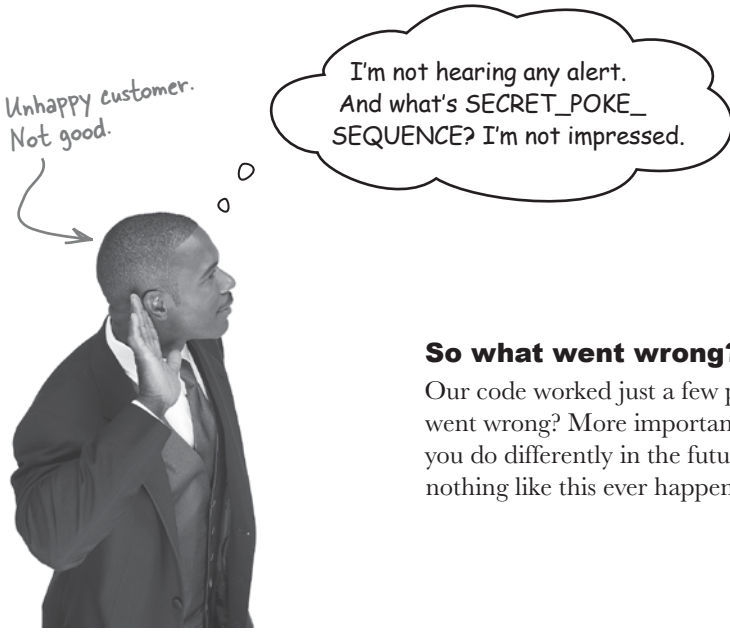
Q: Wait, did Bob just merge code on the demo server?

A: Yup...

Demo the new BeatBox for the customer

We're all set to go. Your code is written, tested, and copied up to the demo server. Bob did the final build, so we call the customer and prepare to amaze the crowds.





So what went wrong?

Our code worked just a few pages ago. So what went wrong? More importantly, what would you do differently in the future to make sure nothing like this ever happens again?

Think beyond, "Do more testing." How can you prevent this problem from occurring in the first place?

Something's clearly gone wrong. Below is some code we compiled on our machine and the same section of code from the demo machine. See if you can figure out what happened.

Here's the code from our machine—it worked fine when we ran it.

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
    try {
      while((obj=in.readObject()) != null) {
        System.out.println("got an object from server");
        System.out.println(obj.getClass());
        String nameToShow = (String) obj;
        checkboxState = (boolean[]) in.readObject();
        if (nameToShow.equals(POKE_START_SEQUENCE)) {
          playPoke();
          nameToShow = "Hey! Pay attention.";
        }
        otherSeqsMap.put(nameToShow, c
        listVector.add(nameToShow);
        incomingList.setListData(listV
      } // close while
    } catch (Exception ex) { ex.print
  } // close run
```

And here's the code on the demo server—the code that tanked.

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
  try {
    while ((obj = in.readObject()) != null) {
      System.out.println("got an object from server");
      System.out.println(obj.getClass());
      String nameToShow = (String) obj;
      checkboxState = (boolean[]) in.readObject();
      if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
         receiveJPEG();
       }
      else {
        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
      }
    } // close while
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} // close run
```

What went wrong? .....

.....

.....

How did this happen? .....

.....

.....

What would you do? .....

.....

.....

Standup meeting

Your team, after the big flop at the customer demo

Mark: Wow. Bob really blew it with that demo.

Bob: What are you talking about? My code worked!

Laura: But you broke the other story we were trying to demo! It worked fine before you got to it.

Bob: Wait a minute—why am I getting blamed for this? You asked me to copy my code up to the demo server so we could build it. When I did that, I saw you guys had changed a lot of the same stuff. It was a mess.

Mark: So you just overwrote it??

Bob: No way—I spent a bunch of time comparing the files trying to figure out what you had changed and what I had changed. To make things worse, you guys had some variables renamed in your code so I had to sort that out, too. I got the button stuff right, but I guess I missed something in the receiver code.

Laura: So do we still have the working Poke code on there?

Bob: I doubt it. I copied my stuff up with a new name and merged them into the files you had up there. I didn't think to snag a copy of your stuff.

Mark: Not good. I probably have a copy on my machine, but I don't know if it's the latest. Laura, do you have it?

Laura: I might, but I've started working on new stuff, so I'll have to try and back all my changes out. We really need to find a better way to handle this stuff. This is costing us a ton of time to sort out and we're probably adding bugs left and right...

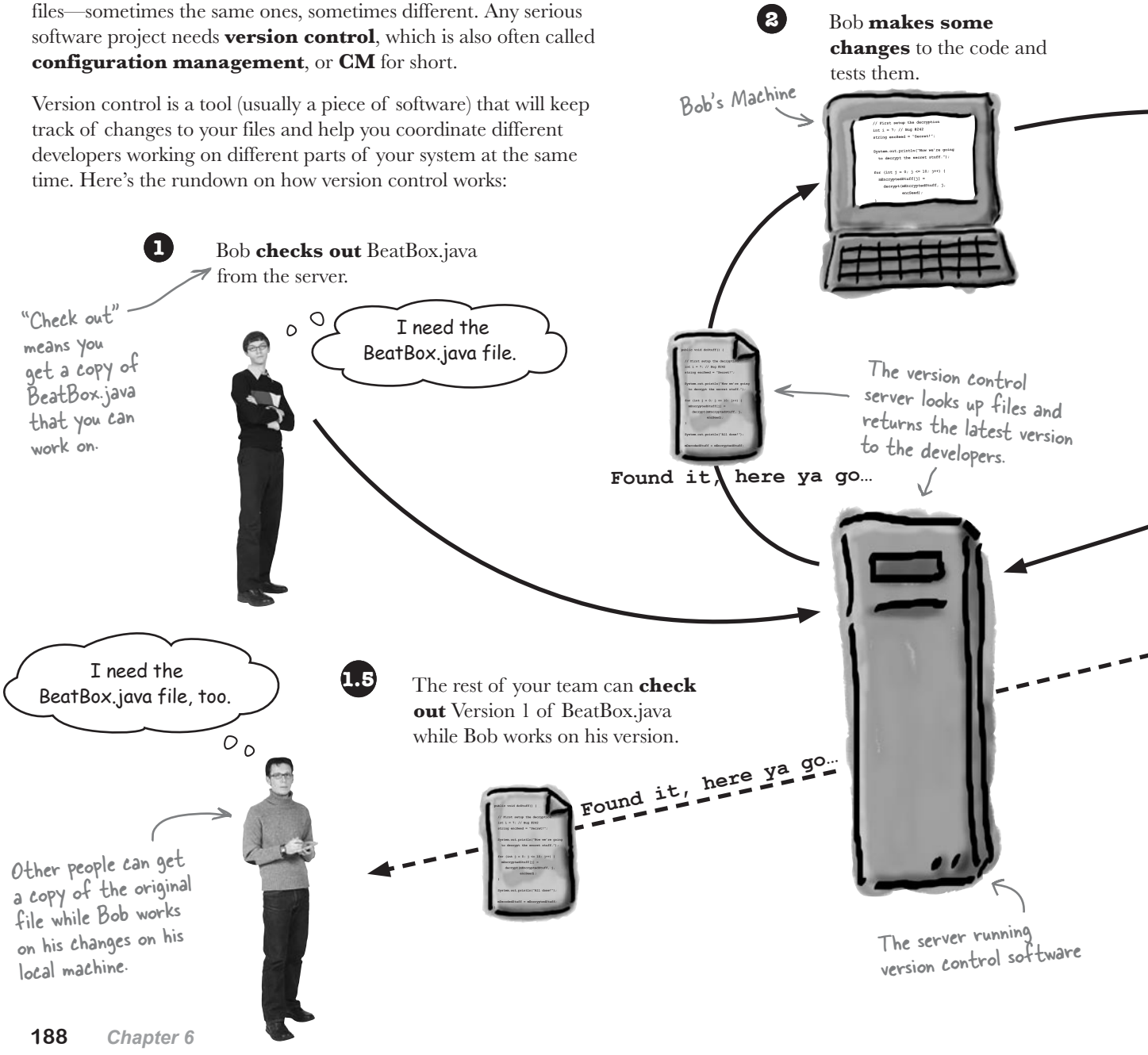
Not to mention we're going the wrong way on our burn-down rate again.

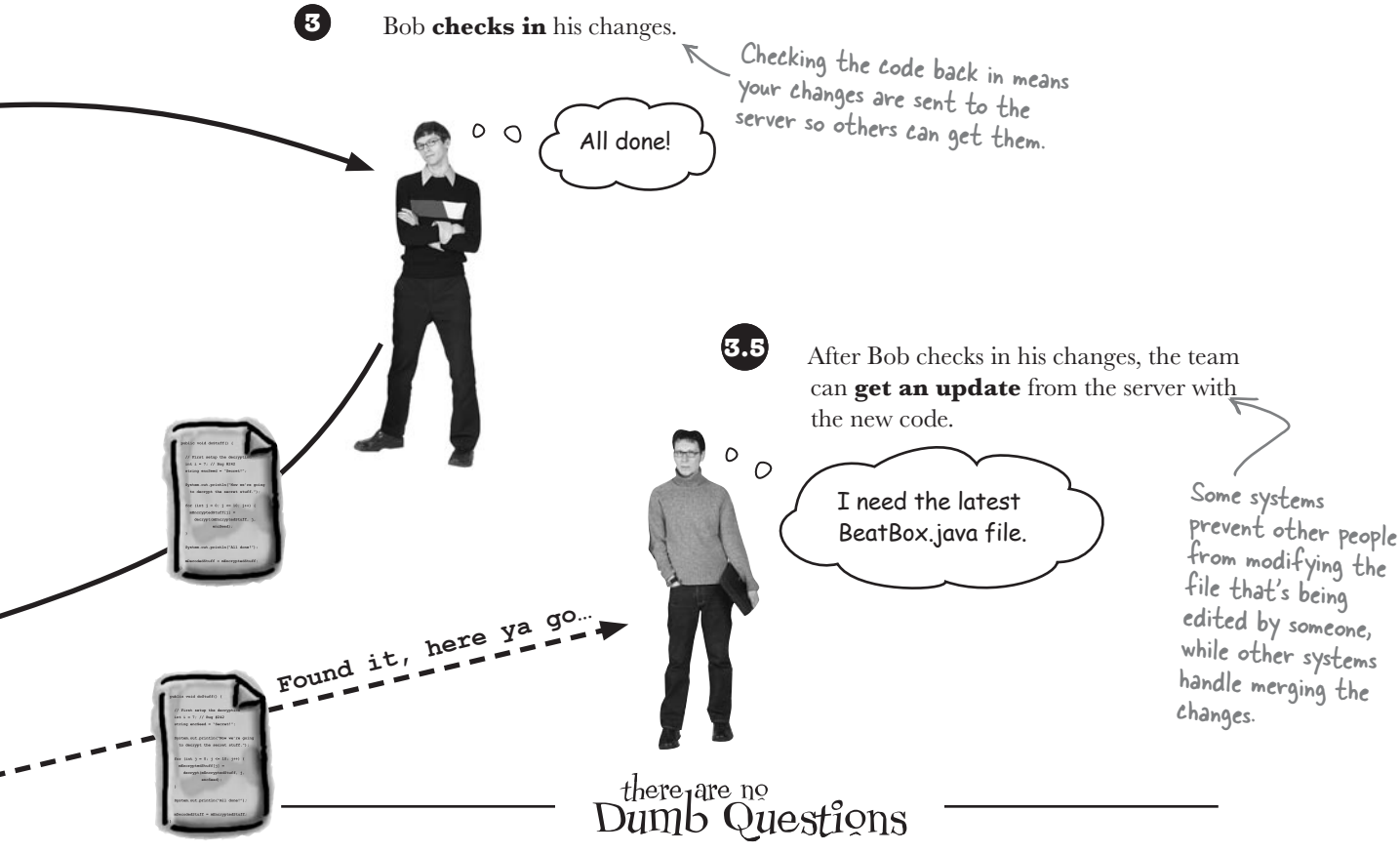
Let's start with VERSION CONTROL

You'll also see this referred to as configuration management, which is a little more formal term for the same thing.

Keeping track of source code (or any kind of files for that matter) across a project is tricky. You have lots of people working on files—sometimes the same ones, sometimes different. Any serious software project needs **version control**, which is also often called **configuration management**, or **CM** for short.

Version control is a tool (usually a piece of software) that will keep track of changes to your files and help you coordinate different developers working on different parts of your system at the same time. Here's the rundown on how version control works:





there are no Dumb Questions

Q: So if version control is a piece of software, which version control product should I use?

A: There are lots of choices out there for version control tools, both commercial and open source. One of the most popular open source ones is called Subversion, and that's the one we'll use in this chapter. Microsoft tools such as Visual Studio like to work with Microsoft's version control tool, called Visual SourceSafe, or Microsoft's new Team Foundation product.

Version control tools all do pretty much the same thing, but some offer different ways to do it. For example, some commercial systems have strict access control on where you can commit code so that your organization can control what goes into what build. Other tools show you the different versions of files as virtual directories.

Q: You're only showing one file and two developers. I'm guessing it can do more than that, right?

A: You bet. In fact, a good version control tool is really the only way you can ***scale a team***. We'll need some of those more sophisticated features (like merging changes, tagging versions, etc.) in just a minute...

First set up your project...

We're assuming you've got your version control software installed. If not, you can download it from the Subversion web site.

The first step in using a version control tool is to put your code in the **repository**; that's where your code is stored. There's nothing tricky about putting your code in the repository, just get the original files organized on your machine and create the project in the repository:

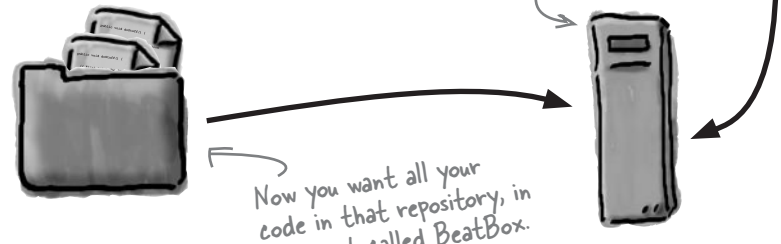
1 First create the repository—you only need to do this once for each version control install. After that you just add projects to the same repository.

```
File  Edit  Window  Help  TakeBacks
hfsd> svnadmin create c:\Users\Developer\Desktop\SVNRepo
hfsd>
```

This tells Subversion to create a new repository...

...in this directory.

After that runs, we have our repository.



2 Next you need to import your code into the repository. Just go to the directory above your code and tell your version control server to import it. So, for your BeatBox project, you'd go to the directory that contains your beat box code. If you're using the downloaded files, that directory is called `Chapter6`:

Now you want all your code in that repository, in a project called BeatBox.

Here you tell Subversion to import your code.

This is the repository you created in step 1. On Windows you'll need to use forward slash notation.

```
File  Edit  Window  Help  Tariffs
hfsd> svn import Chapter6 file:///c:/Users/Developer/Desktop/
SVNRepo/BeatBox/trunk -m "Initial Import"

Adding          Chapter6\src
Adding          Chapter6\src\headfirst
Adding          Chapter6\src\headfirst\sd
Adding          Chapter6\src\headfirst\sd\chapter6
Adding          Chapter6\src\headfirst\sd\chapter6\BeatBox.java
Adding          Chapter6\src\headfirst\sd\chapter6\MusicServer.java

Committed revision 1.

hfsd>
```

This is just a comment describing what we're doing; we'll talk more about this later, too.

Here's what we want our project to be called—ignore the "trunk" thing for right now.

Subversion adds each file it finds into your repository for the BeatBox project.

* You can get the full Subversion documentation here: http://svnbook.red-bean.com/

...then you can check code in and out.

Now that your code is in the repository, you can check it out, make your changes, and check your updated code back in. A version control system will keep track of your original code, all of the changes you make, and also handle sharing your changes with the rest of your team. First, check out your code (normally your repository wouldn't be on your local machine):

1 To check out your code, you just tell your version control software what project you want to check out, and where to put the files you requested.

This tells Subversion to check out a copy of the code.

This pulls code from the BeatBox project in the repository and puts it in a local directory called BeatBox.

```
File Edit Window Help Girvine
hfsd> svn checkout file:///c:/Users/Developer/Desktop/SVNRepo/
BeatBox/trunk BeatBox

A    BeatBox\src
A    BeatBox\src\headfirst
A    BeatBox\src\headfirst\sd
A    BeatBox\src\headfirst\sd\chapter6
A    BeatBox\src\headfirst\sd\chapter6\BeatBox.java
A    BeatBox\src\headfirst\sd\chapter6\MusicServer.java

Checked out revision 1.

hfsd>
```

Subversion pulls your files back out of the repository and copies them into a new BeatBox directory (or an existing one if you've already got a BeatBox directory).

2 Now you can make changes to the code just like you normally would. You just work directly on the files you checked out from your version control system, compile, and save.

You can re-implement the Poke story, since Bob broke that feature when he wrote code for the Send Picture story.

This is a normal .java file. Subversion doesn't change it in any way...it's still just code.

3 Then you commit your changes back into the repository with a message describing what changes you've made.

This tells Subversion to commit your changes; it will figure out what files you've changed.

This is a log message, indicating what you did.

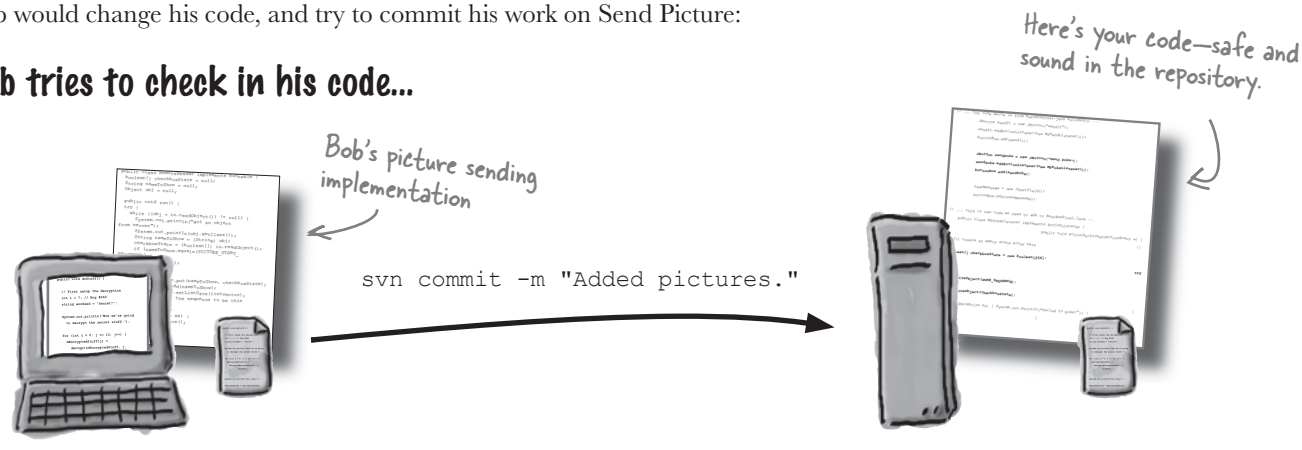
```
File Look What IDid
hfsd> svn commit -m "Added POKE support."

Sending        src\headfirst\sd\chapter6\BeatBox.java
Transmitting file data .
Committed revision 2.

hfsd>
```

Since you only changed one file, that's all that subversion sent to the repository—and notice that now you have a new revision number.

Most version control tools will try and solve problems for you

Suppose you had a version control system in place before the great BeatBox debacle of '08. You'd check in your code (with `commit`) to implement Send Poke, and then Bob would change his code, and try to commit his work on Send Picture:

Bob tries to check in his code...



...but quickly runs into a problem.

Bob's code

You and Bob both made changes to the same file; you just got yours into the repository first.

The code on the server, with your changes

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
  try {
    while ((obj = in.readObject()) != null) {
      System.out.println("got an object from server");
      System.out.println(obj.getClass());
      String nameToShow = (String) obj;
      checkboxState = (boolean[]) in.readObject();
      if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
        receiveJPEG();
      }
      else {
        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
        // now reset the sequence to be this
      }
    } // close while
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} // close run
} // close inner class
```

Bob's BeatBox.java

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
   try {
     while((obj=in.readObject()) != null) {
       System.out.println("got an object from server");
       System.out.println(obj.getClass());
       String nameToShow = (String) obj;
       checkboxState = (boolean[]) in.readObject();
       if (nameToShow.equals(POKE_START_SEQUENCE)) {
         playPoke();
         nameToShow = "Hey! Pay attention.";
       }
       otherSeqsMap.put(nameToShow, checkboxState);
       listVector.add(nameToShow);
       incomingList.setListData(listVector);
   } // close while
  } catch(Exception ex) {ex.printStackTrace();}
  } // close run

  private void playPoke() {
    Toolkit.getDefaultToolkit().beep();
  }
} // close inner class
```

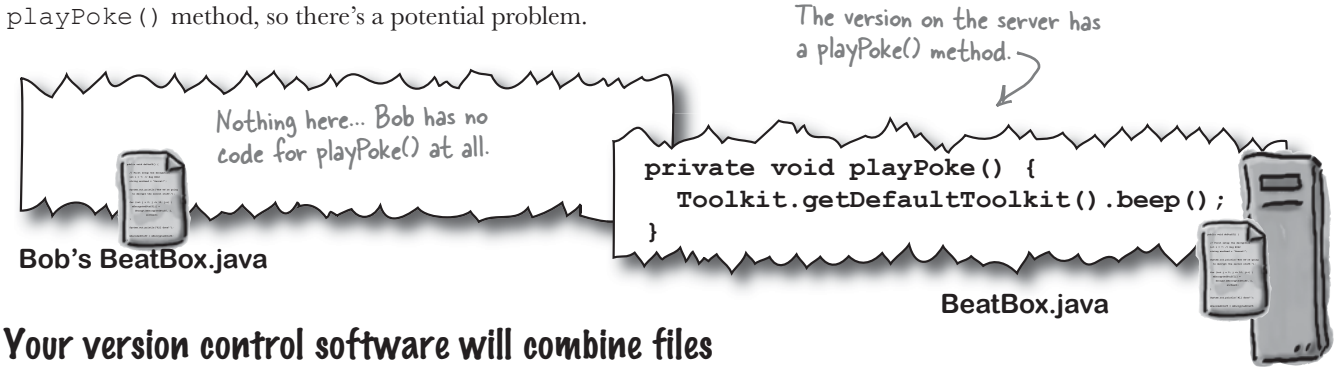
BeatBox.java

The server tries to MERGE your changes

If two people make changes to the same file but in different places, most version control systems try to merge the changes together. This isn't *always* what you want, but most of the time it works great.

Nonconflicting code and methods are easy

In `BeatBox.java`, you added a `playPoke()` method, so the code on the version control server has that method. But Bob's code has no `playPoke()` method, so there's a potential problem.

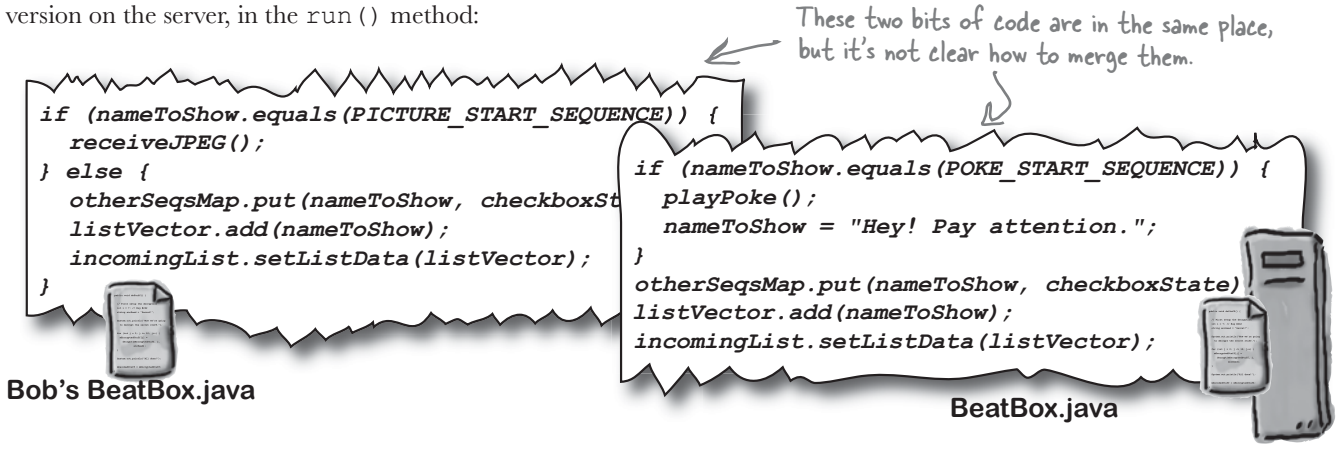


Your version control software will combine files

In a case like this, your version control server can simply combine the two files. In other words, the `playPoke()` method gets combined with nothing in Bob's file, and you end up with a `BeatBox.java` on the server that still retains the `playPoke()` method. So no problems yet...

But conflicting code IS a problem

But what if you have code in the same method that is different? That's exactly the case with Bob's version of `BeatBox.java`, and the version on the server, in the `run()` method:



If your software can't merge the changes, it issues a conflict

If two people made changes to the same set of lines, there's no way for a version control system to know what to put in the final server copy. When this happens, most systems just punt. They'll kick the file back to the person trying to commit the code and ask them to sort out the problems.

Subversion rejects your commit. You can use the update command to pull the changes into your code, and Subversion will mark the lines where there are conflicts in your files... after you sort out the conflicts, you can recommit.

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
  try {
    while ((obj = in.readObject()) != null) {
      System.out.println("got an object from server");
      System.out.println(obj.getClass());
      String nameToShow = (String) obj;
      checkboxState = (boolean[]) in.readObject();
      if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
        receiveJPEG();
      }
      else {
        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
        // now reset the sequence to be this
      }
    } // close while
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} // close run
} // close inner class
```

Bob's BeatBox.java

Your version control software doesn't know what to do with this conflicting code, so to protect everyone, it refuses to commit the new code, and marks up where problems might be.

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
   try {
     while((obj=in.readObject()) != null) {
       System.out.println("got an object from server");
       System.out.println(obj.getClass());
       String nameToShow = (String) obj;
       checkboxState = (boolean[]) in.readObject();
       if (nameToShow.equals(POKE_START_SEQUENCE)) {
         playPoke();
         nameToShow = "Hey! Pay attention.";
       }
       otherSeqsMap.put(nameToShow, checkboxState);
       listVector.add(nameToShow);
       incomingList.setListData(listVector);
   } // close while
  } catch(Exception ex) {ex.printStackTrace();}
  } // close run

  private void playPoke() {
    Toolkit.getDefaultToolkit().beep();
  }
} // close inner class
```

BeatBox.java

Conflict Resolution: Here's the file the version control software kicked back to Bob, with all the conflicts marked. What should the final code look like that Bob commits back in?

```
public class RemoteReader implements Runnable {
  // variable declarations
  public void run() {
  try {
    // code without problems

<<<<<<< .mine
      if (nameToShow.equals(
                    PICTURE_START_SEQUENCE)) {
        receiveJPEG();
      } else {
        otherSeqsMap.put(
                  nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
        // now reset the sequence to be this
      }
=======
       if (nameToShow.equals(
                    POKE_START_SEQUENCE)) {
         playPoke();
         nameToShow = "Hey! Pay attention.";
       }
       otherSeqsMap.put(
                 nameToShow, checkboxState);
       listVector.add(nameToShow);
       incomingList.setListData(listVector);
>>>>>>> .r2
    } // close while
  // more code without problems
} // close run
} // close inner class
```

Files with conflicts get both the local changes (Bob's changes) and the changes from the server. The ones between "<<<<<<< .mine" and the ==='s are Bob's— the ones after that up to the ">>>>>>> .r2" are the ones from the server.

```
public class RemoteReader implements
Runnable {
  // variable declarations
  public void run() {
  try {
    // code without problems

  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
    } // close while
  // more code without problems
} // close run
} // close inner class
```

Conflict Resolution: Here's the file version control kicked back to Bob with both changes in it. What should the final section look like that Bob commits back in?

```
public class RemoteReader implements Runnable {
  // variable declarations
  public void run() {
  try {
    while ((obj = in.readObject()) != null) {
      System.out.println("got an object from server");
      System.out.println(obj.getClass());
      String nameToShow = (String) obj;
      checkboxState = (boolean[]) in.readObject();
      if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
        receiveJPEG();
      }
      else {
       if (nameToShow.equals(POKE_START_SEQUENCE)) {
         playPoke();
         nameToShow = "Hey! Pay attention.";
       }

        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
        // now reset the sequence to be this
      }
    } // close while
  } catch (Exception ex) {
    ex.printStackTrace();
  }
} // close run
} // close inner class
```

We need to support both the picture sequence and the poke sequence so we need to merge the conditionals.

Make sure you delete the conflict characters (<<<<<<<, =======, and >>>>>>>).

Make these changes to your own copy of `BeatBox.java`, and commit them to your code repository:

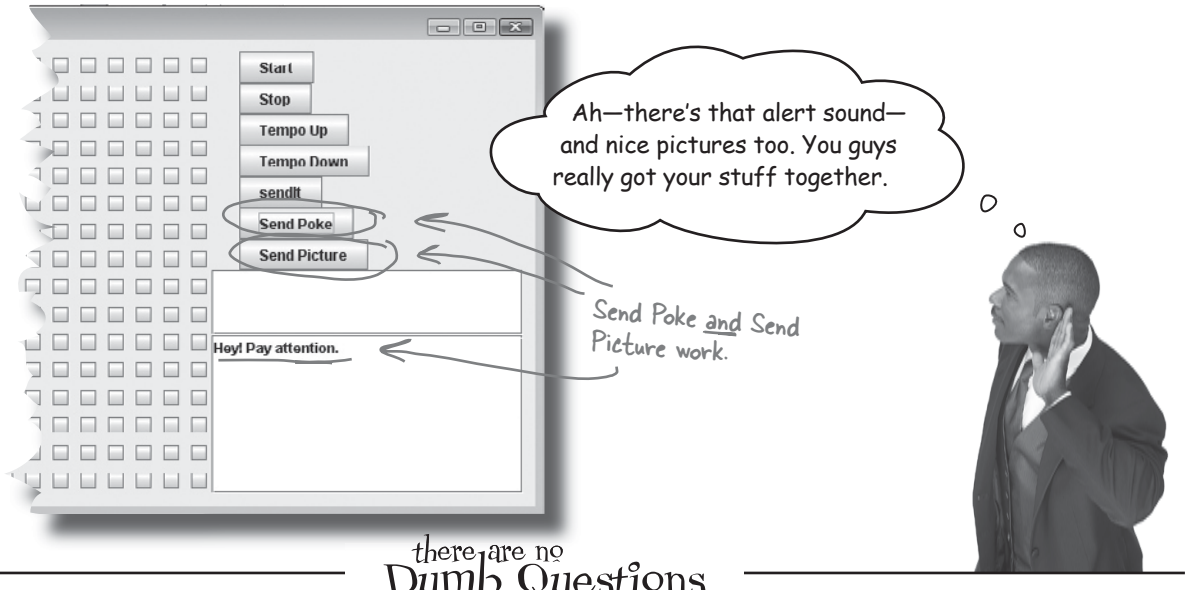
First, tell Subversion you resolved the conflict in the file using the "resolved" command and the path to the file.

You can skip this step if you didn't really get a conflict from Subversion.

Now, commit the file to your server, adding a comment indicating what you did.

```
File  Edit  Window  Help  Tranquility
hfsd> svn resolved src/headfirst/sd/chapter6/BeatBox.java
Resolved conflicted state of 'BeatBox.java'

hfsd> svn commit -m "Merged picture support with Poke stuff."
Sending        src\headfirst\sd\chapter6\BeatBox.java
Transmitting file data .
Committed revision 3.

hfsd>
```

Now show the customer...



there are no Dumb Questions

Q: I see how checking out and committing works, but how do other people on the team get my changes?

A: Once you've got your project checked out, you can run `svn update`. That tells the version control server to give you the latest versions of all files in the project. Lots of teams run an update every morning, to make sure they're current with everyone else's work.

Q: This whole conflict thing seems pretty hairy. Can't my version control software do anything besides erroring out?

A: Some can. Certain version control tools work in a ***file locking mode***, which means when you check out files, the system locks those files so no one else can check them out. Once you make your changes and check the files back in, the system unlocks the files. This prevents conflicts, since only one person can edit a file at a time. But, it also means you might not be able to make changes to a file when you want to; you might need to wait for someone else to finish up first. To get around that, some locking version control systems allow you to check out a file in read-only mode while it's locked. But that's a bit heavy-handed, so other tools like Subversion allow multiple people to work on the same file at once. Good design, good division of labor, frequent commits, and good communication help reduce the number of manual merges you actually have to do.

Q: What is all this trunk business you keep saying to ignore?

A: The Subversion authors recommend putting your code into a directory called `trunk`. Then, other versions would go into a directory called `branches`. Once you've imported your code, the trunk thing doesn't really show up again, except during an initial checkout. We'll talk more about branches later in the chapter, but for now, stick with the `trunk`.

Q: Where are all of my messages going when I do a commit?

A: Subversion keeps track of each time you commit changes into the repository and associates your message with those changes. This lets you look at why people made a certain change—for instance, if you need to go back and figure out why something was done. That's why you should always use a sensible, explanatory message when you do a commit. The first time you go back through old commits and find "I changed stuff" as the log message, you'll be pretty cranky.

Q: Do I have to commit all of my changes at the same time?

A: Nope! Just put the path to the filename on the `commit` command like you did for the `resolved` command. Subversion will commit just the file(s) you specify.

More iterations, more stories...

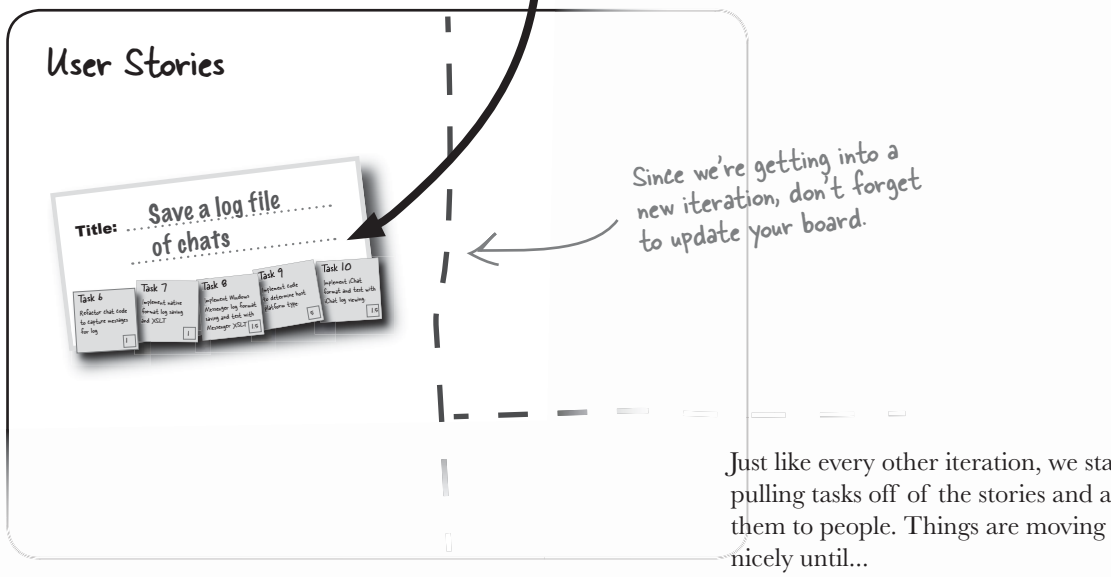
Things are going well. The customer was happy with our Poke and Picture support, and after one more iteration, felt we had enough for Version 1.0. A few iterations later and everyone's looking forward to Version 2.0. Just a few more stories to implement...

The customer gave us this new user story (which we'll have to break into tasks).

Title: Save a log file of chats

Description: The user should be able to save a history of their chat messages to a file. The file should be compatible with Windows Live Messenger on Windows, and iChat on a Mac.

Priority: 40 **Estimate:** 6



Since we're getting into a new iteration, don't forget to update your board.

Just like every other iteration, we start pulling tasks off of the stories and assigning them to people. Things are moving along nicely until...

Standup meeting

Bob: Hey guys. Good news: I'm just about done with the Windows Messenger version, and it's working well. But there's bad news, too. I just found a bug in the way images are handled in our Send Picture feature from way back in the first iteration.

Laura: That's not good. Can we wait on fixing it?

Bob: I don't think so—it's a potential security hole if people figure out how to send a malicious picture. The users will be pretty annoyed over this.

Mark: Which means the customer is going to be *really* annoyed over this. Can you fix it?

Bob: I can fix it—but I've got a ton of code changes in there for the new story, the log files, that aren't ready to go out yet.

Laura: So we're going to have to roll your changes back and send out a patched 1.0 version.

Mark: What do we roll it back to? We have lots of little changes to lots of files. How do we know where version 1.0 was?

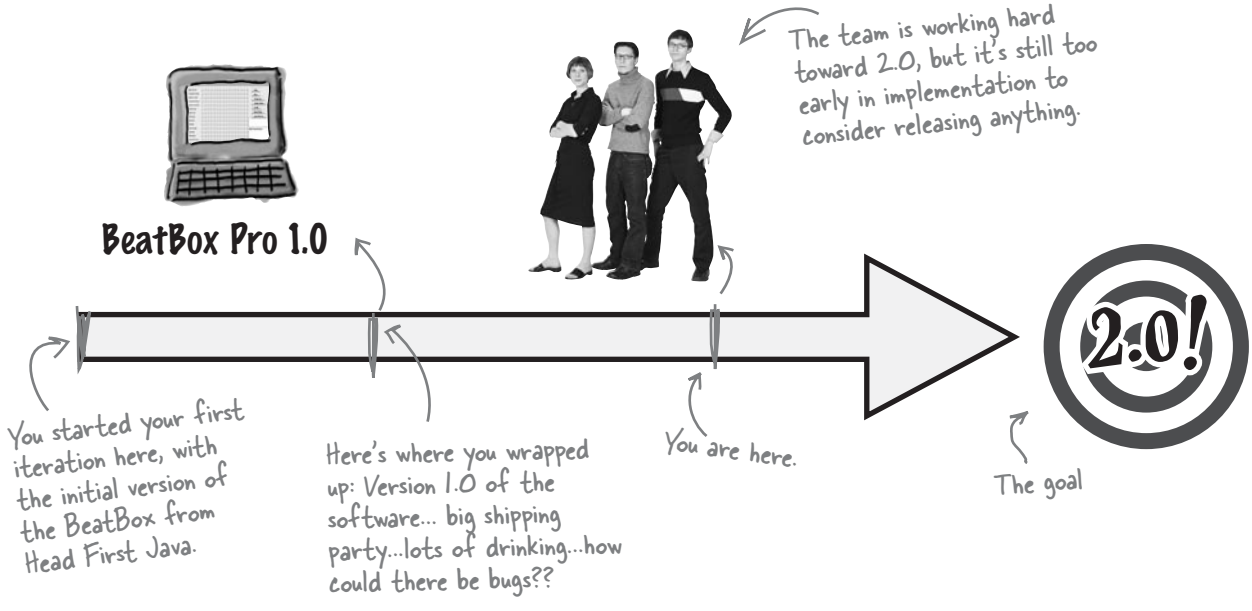
Bob: Forget version 1.0, what about all of my work?? If you roll back, you're going to drop everything I did.

The team's in a tough spot—there's a pretty serious bug in the released version, but there's a lot effort invested in the new version. The new version isn't ready to go out the way it is. What would you do?

We have more than one version of our software...

The real problem here is that we have more than one version of our software—or more accurately, more than one version of our source code—that we need to make changes to. We have version 1.0 of the code built and out there, but Bob found a pretty serious bug. On top of that, we've got version 2.0 in the works, but it's full of untested, unworking features.

We need to separate them somehow...



BULLET POINTS

- Bugs to released versions are usually a higher priority to the customer than implementing new features.
- Your bug fixes should affect released software and still be implemented in in-progress versions of your software.
- Effective bug fixing depends on being able to locate specific versions of your software and make changes to those versions without affecting current development.

You'll always have tension between bugs cropping up in released versions, and new features in upcoming versions. It's up to you to work with the customer to BALANCE those tensions.

You keep saying "Version 1.0," but what does that mean? We've committed tons of changes since then into the repository....

By default, your version control software gives you code from the trunk.

Some systems call this the HEAD or the main line.

You're right. When you check out the code from your version control system, you're checking it out from the **trunk**. That's the latest code by default and (assuming people are committing their changes on a regular basis) has all of the latest bugs features.

Remember the trunk thing that keeps coming up? That's the place where all the latest and greatest code is stored.

```
File Edit Window Help
hfsd> svn checkout file:///c:/Users/Developer/Desktop/SVNRepo/
BeatBox/trunk BeatBox

A    BeatBox\src
A    BeatBox\src\headfirst
A    BeatBox\src\headfirst\sd
A    BeatBox\src\headfirst\sd\chapter6
A    BeatBox\src\headfirst\sd\chapter6\BeatBox.java
A    BeatBox\src\headfirst\sd\chapter6\MusicServer.java

Checked out revision 1.

hfsd>
```

But we **do** have the 1.0 code **somewhere**, even if it's not labeled, right? We just have to find it on our server somehow...

Version control software stores ALL your code.

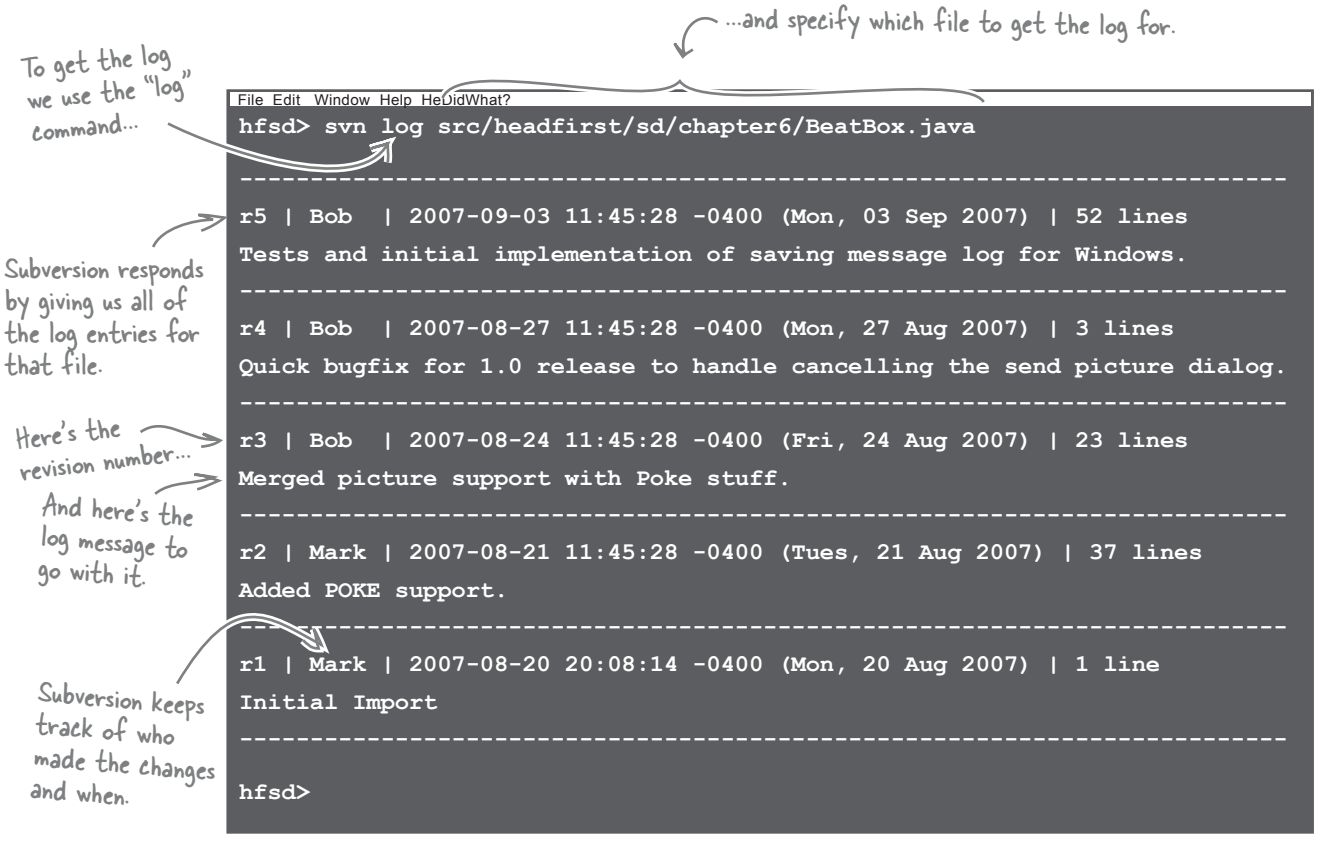
Every time you commit code into your version control system, a revision number was attached to the software at that point. So, if you can figure out which revision of your software was released as Version 1.0, you're good to go.

Here's the revision number for this set of changes; it increases with each commit.

```
File Edit Window Help
hfsd> svn commit -m "Added POKE support."

Sending        src\headfirst\sd\chapter6\BeatBox.java
Transmitting file data .
Committed revision 18.

hfsd>
```

Good commit messages make finding older software easier

You've been putting nice descriptive messages each time you committed code into your version control system, right? Here's where they matter. Just as each commit gets a revision number, your version control software also keeps your commit messages associated with that revision number, and you can view them in the log:



Play "Find the features" with the log messages

You've got to figure out which features were in the software—in this case, for Version 1.0. Then, figure out which revision that matches up with.

Using the log messages above, which revision do you think matches up with Version 1.0 of BeatBox Pro?

..... ← Write down the revision number you want to check out to get Version 1.0.

Now you can check out Version 1.0

In Subversion, -r indicates you want a specific revision of code. We're grabbing revision 4.

1. Once you know which revision to check out, your version control server can give you the code you need:

This puts the code in a new directory, for Version 1.0.

```
File  Edit  Window  Help  ThatOne
hfsd> svn checkout -r 4 file:///c:/Users/Developer/Desktop/
SVNRepo/BeatBox/trunk BeatBoxV1.0

A    BeatBoxV1.0\src
A    BeatBoxV1.0\src\headfirst
A    BeatBoxV1.0\src\headfirst\sd
A    BeatBoxV1.0\src\headfirst\sd\chapter6
A    BeatBoxV1.0\src\headfirst\sd\chapter6\BeatBox.java
A    BeatBoxV1.0\src\headfirst\sd\chapter6\MusicServer.java

Checked out revision 4.

hfsd>
```

2. Now you can fix the bug Bob found...

BeatBox.java

Once again, the version control server gives you normal Java code you can work on.

3. With the changes in place, commit the code back to your server...

Uh oh, looks like the server isn't happy with your updated code.

```
File  Edit  Window  Help  Trouble
hfsd> svn commit src/headfirst/sd/chapter6/BeatBox.java -m
"Fixed the critical security bug in 1.0 release."

Sending        src\headfirst\sd\chapter6\BeatBox.java
svn: Commit failed (details follow):

svn: Out of date: '/BeatBox/trunk/src/headfirst/sd/chapter6/
BeatBox.java' in transaction '6-1'

hfsd>
```

Sharpen your pencil

What happened? .....

.....

Why? .....

.....

So now what do we do? .....

.....

(Emergency) standup meeting

If you're having a problem, don't wait for the next day. Just grab everyone and have an impromptu standup meeting.

Laura: We could check out the version 1.0 code just fine, but now the version control server won't let us commit our changes back in. It says our file is out of date.

Mark: Oh—ya know, that's probably a good thing. If we could commit it, wouldn't that become revision 6, meaning the latest version of the code wouldn't have Bob's changes?

Bob: Hey that's right—you'd leapfrog my code with old version 1.0 code. I don't want to lose all of my work!

Laura: You still have your work saved locally, right? Just merge it in with the new changes and recommit it. You'll be fine.

Bob: Uggh, all that merging stuff sucks; it's a pain. And what about the next time we find a bug we need to patch in Version 1.0?

Mark: We'll have to remember what the new 1.0 revision is. Once we figure out how to commit this code, we'll write down the revision number and use that as our base for any other 1.0 changes.

Laura: New 1.0 changes? Wouldn't we be at Version 1.1 now?

Bob: Yeah, that's right. But this is still a mess...

Sharpen your pencil

Write down three problems with the approach outlined above for handling future changes to Version 1.0 (or is it 1.1?).

1. .....
.....
2. .....
.....
3. .....
.....

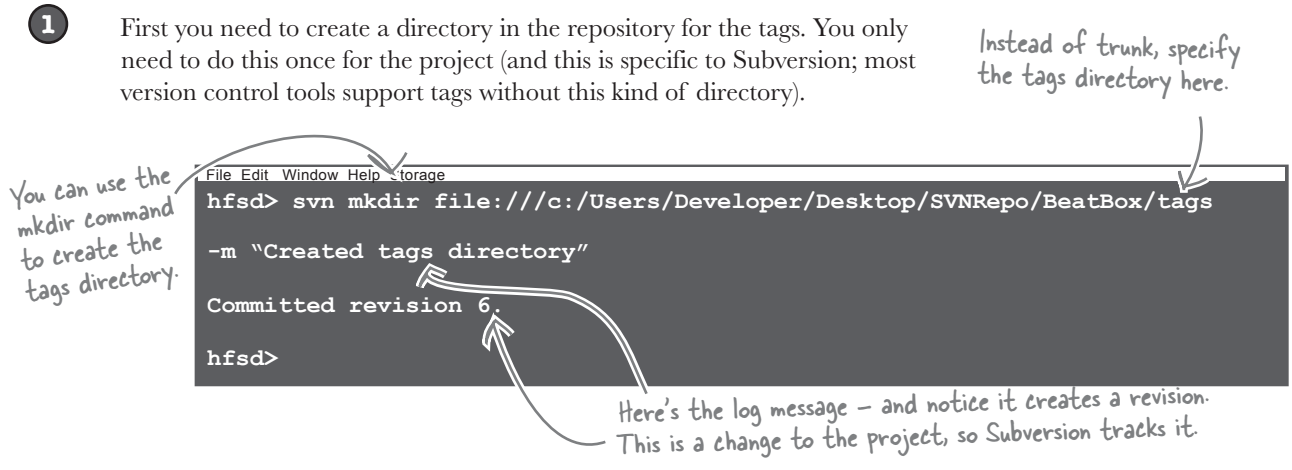
Answers on page 217.

Tag your versions

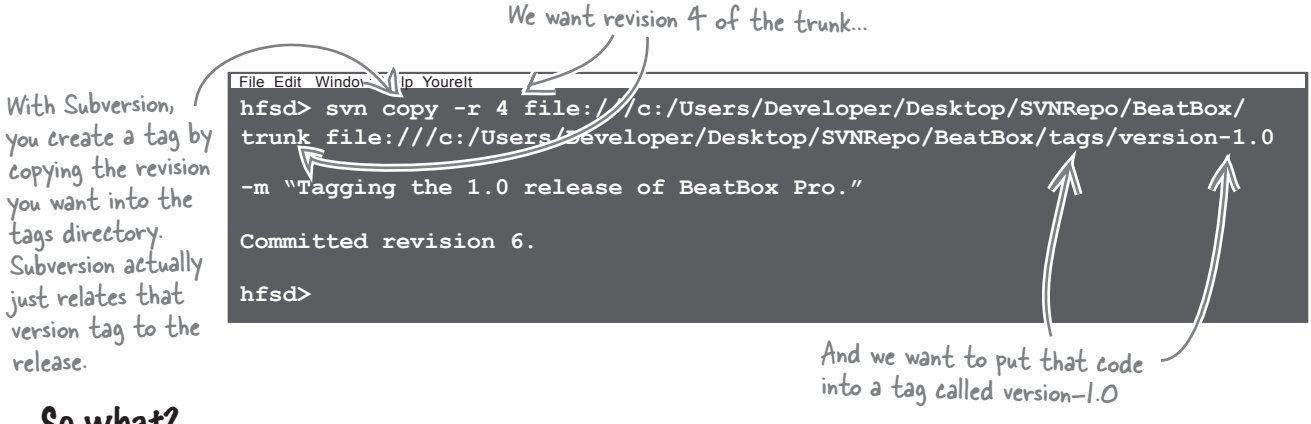
The revision system worked great to let us get back to the version of the code we were looking for, and we got lucky that the log messages were enough for us to figure out what revision we needed. Most version control tools provide a better way of tracking which version corresponds to a meaningful event like a release or the end of an iteration. They're called **tags**.

Let's tag the code for BeatBox Pro we just located as Version 1.0:

1. First you need to create a directory in the repository for the tags. You only need to do this once for the project (and this is specific to Subversion; most version control tools support tags without this kind of directory).



2. Now tag the initial 1.0 release, which is revision 4 from the repository.



So what?

So what did that get us? Well, instead of needing to know the revision number for version 1.0 and saying `svn checkout -r 4 ...`, you can check out Version 1.0 of the code like this:

```
svn checkout file:///c:/Users/Developer/Desktop/SVNRepo/BeatBox/tags/version-1.0
```

And let Subversion remember which revision of the repository that tag relates to.

So now I know where Version 1.0 is, great. But we still only have the 1.0 code, and need to commit those changes. Do we just commit our updated code into the Version 1.0 tag?

No! The tag is just that; it's a snapshot of the code at the point you made the tag. You don't want to commit any changes into that tag, or else the whole "version-1.0" thing becomes meaningless. Some version control tools treat tags so differently that it's impossible to commit changes into tags at all (Subversion doesn't. It's possible to commit into a tag, but it's a very, very bad idea).

BUT we can use the same idea and make a copy of revision 4 that we will commit changes into; this is called a branch. So a **tag** is a snapshot of your code at a certain time, and a **branch** is a place where you're working on code that isn't in the main development tree of the code.

1 Just like with tags, we need to create a directory for branches in our project.

Instead of trunk, we specify the branches directory here.

Use the mkdir command again to create the branches directory.

```
File  Edit  Window  Help  Expanding
hfsd> svn mkdir file:///c:/Users/Developer/Desktop/SVNRepo/BeatBox/branches
-m "Created branches directory"

Committed revision 8.

hfsd>
```

2 Now create a `version-1` branch from revision 4 in our repository.

We want revision 4 of the trunk...

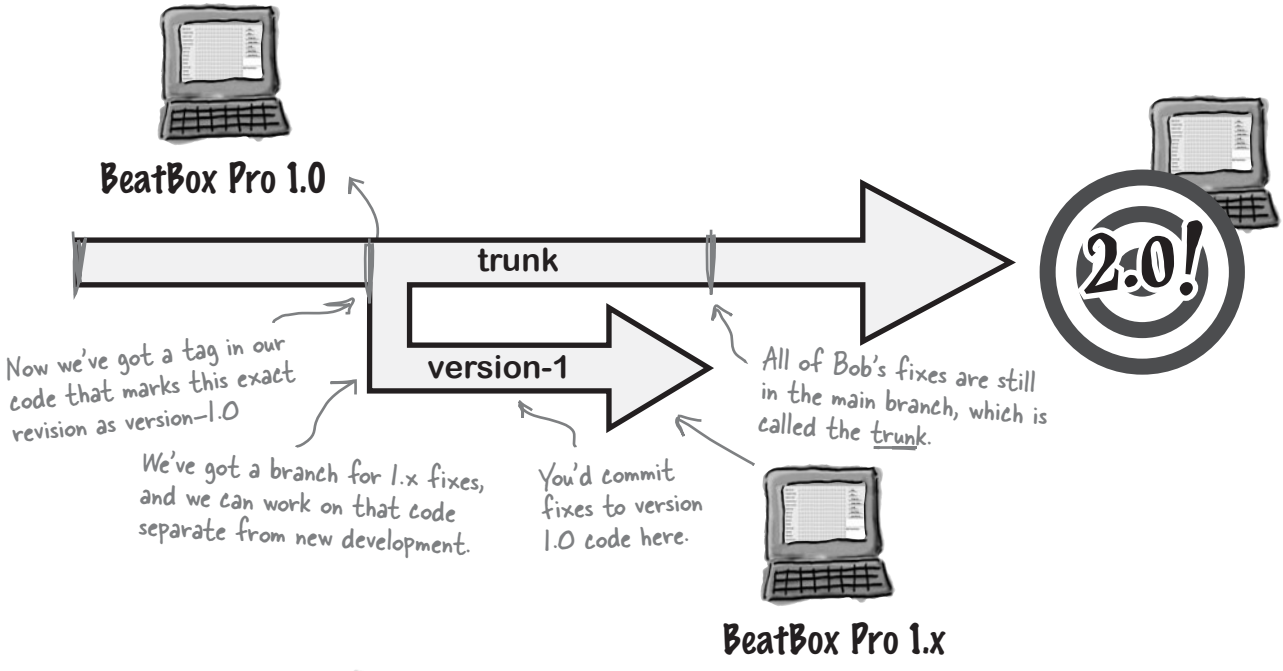
```
File  Edit  Window  Help  Dup
hfsd> svn copy -r 4 file:///c:/Users/Developer/Desktop/SVNRepo/BeatBox/trunk
file:///c:/Users/Developer/Desktop/SVNRepo/BeatBox/branches/version-1

-m "Branched the 1.0 release of BeatBox Pro."

Committed revision 9.

hfsd>
```

With Subversion you create a branch just like a tag; you copy the revision you want into the branches directory. It won't actually copy anything; it just stores the revision number you supplied.

And we want to put it into a branch called version-1 (not Version 1.0, because we'll use this for Version 1.1, 1.2, etc.).

Tags, branches, and trunks, oh my!

Your version control system has got a lot going on now, but most of the complexity is managed by the server and isn't something you have to worry about. We've tagged the 1.0 code, made fixes in a new branch, and still have current development happening in the trunk. Here's what the repository looks like now:



Tags are snapshots of your code. You should always commit to a branch, and never to a tag.

BULLET POINTS

- The **trunk** is where your active development should go; it should always represent the latest version of your software.
- A **tag** is a name attached to a specific revision of the items in your repository so that you can easily retrieve that revision later.
- Sometimes you might need to **commit the same changes to a branch and the trunk** if the change applies to both.
- **Branches** are copies of your code that you can make changes to without affecting code in the trunk. Branches often start from a tagged version of the code.
- **Tags are static**—you don't commit changes into them. **Branches** are for **changes that you don't want in the trunk** (or to keep code away from changes being made in the trunk).

Fixing Version 1.0...for real this time.

When we had everything in the trunk, we got an error trying to commit old patched code on top of our new code. Now, though, we've got a tag for version 1.0 and a branch to work in. Let's fix Version 1.0 in that branch:

1. First, check out the `version-1` branch of the BeatBox code:

Notice we didn't need to specify a revision here. The branch is a copy of the version 1.0 code.

We'll put this in the BeatBoxV1 directory this time.

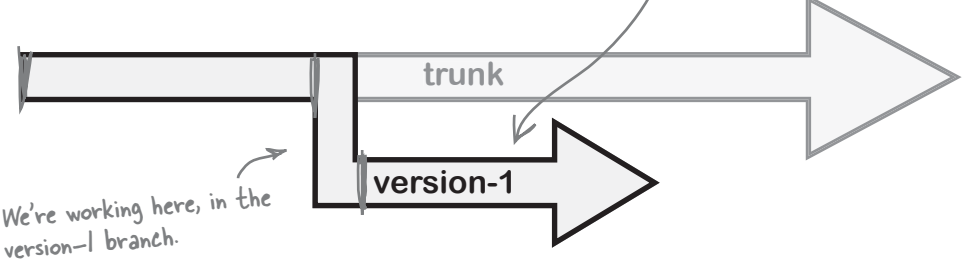
```
File  Edit  Window  Help  History
hfsd> svn checkout file:///c:/Users/Developer/Desktop/SVNRepo/BeatBox/
branches/version-1 BeatBoxV1

A    BeatBoxV1\src
A    BeatBoxV1\src\headfirst
A    BeatBoxV1\src\headfirst\sd
A    BeatBoxV1\src\headfirst\sd\chapter6
A    BeatBoxV1\src\headfirst\sd\chapter6\BeatBox.java
A    BeatBoxV1\src\headfirst\sd\chapter6\MusicServer.java

Checked out revision 9.

hfsd>
```

These revisions numbers stop meaning as much, because we're using tags to reference revisions instead of revision numbers.

2. Now you can fix the bug Bob found...

BeatBox.java

This time, we're working on code from the version–1 branch.



We're working here, in the version–1 branch.

3 ...and commit our changes back in. This time, though, no conflicts:

The fix is in the branch.

```
File Edit Window Help Sweet
hfsd> svn commit src/headfirst/sd/chapter6/BeatBox.java -m "Fixed the
critical security bug in 1.0 release."

Sending        src\headfirst\sd\chapter6\BeatBox.java

Committed revision 10.

hfsd>
```

We have TWO code bases now

With all these changes, we've actually got two different sets of code: the 1.x branch, where fixes are made to Version 1.0, and the trunk, which has all the new development.

- ☑ Our `trunk` directory in the repository has the latest and greatest code that's still in development (and Bob applied the security fix there, too).
- ☑ We have a `version-1.0` tag in our `tags` directory so we can pull out Version 1.0 whenever we want.
- ☑ We have a `version-1` branch in our `branches` directory that has all of our critical patches that have to go out as a 1.x version without any of the new development work.

Don't forget: when you actually do release v1.1 with these patches, create a version-1.1 tag in the tags directory so you can get back to that version later if you have to.

there are no Dumb Questions

Q: I've heard branches are a bad idea and should be avoided. Why are we talking about them?

A: Branches aren't always a bad thing; they have an important place in software development. But, they do come with a price. We'll talk about that over the next few pages.

Q: What else can tags be used for?

A: Tags are great for tracking released versions of software, but you can also use them for keeping track of versions as software goes through testing or QA—think `alpha1`, `alpha2`, `beta1`, `ReleaseCandidate1`, `ReleaseCandidate2`, `ExternalTesting`, etc. It's also a good practice to tag the project at the end of each iteration.

Q: Earlier, you said not to commit changes to a tag. What's that supposed to mean? And how can you prevent people from doing it?

A: The issue with commiting changes to a tag is really a Subversion peculiarity; other tools explicitly prohibit commiting to a tag. Since Subversion uses the copy command to create a tag, exactly like it does a branch, you technically can commit into a tag just like any other place in the repository. However, this is almost always a bad idea. The reason you tagged something was to be able to get back to the code ***just as it was when you tagged it***. If you commit changes into the tag, it's not the same code you originally tagged.

Subversion does have ways of putting permission controls on the tags directory so that you can prevent people from committing into it. However, once people get used to Subversion, it's usually not a major problem, and you can always revert changes to a tag in the odd case where it happens.

Q: We've been using file:///c:/... for our repository. How is that supposed to work with multiple developers?

A: Great question—there are a couple things you can do here. First, Subversion has full support for integration with a web server, which lets you specify your repository location as http:// or https://. That's when things get really interesting. For example, with https you get encrypted connections to your repository. With either web approach, you can share your repository over a much larger network without worrying about mapping shared drives. It's a little more work to configure, but it's great from the developer perspective. If you can't use http access for your repository, Subversion also supports tunneling repository access through SSH. Check out the Subversion documentation (*http://svnbook.red-bean.com/*) for more information on how to set these up.

Q: When I run the log command, I see the same revision number all over the place. What's that about?

A: Different tools do versioning (or revisioning) differently. What you're seeing is how Subversion does its revision tracking. Whenever you commit a file, Subversion applies a revision number ***across the whole project***. Basically, that revision says that "The entire project looked like this at revision 9." This means that if you want to grab the project at a certain point you only need to know *one* revision number. Other tools version each file separately (most notably the version control tool called CVS which was a predecessor to Subversion). That means that to get a copy of a project at a certain state, you need to know the version numbers *of each file*. This really isn't practical, so tags become even more critical.

Q: Why did we branch the Version 1.0 code instead of leaving Version 1.0 in the trunk, and branch the new work?

A: That would work, but the problem with that approach is you end up buried in branches as development goes on. The trunk ends up being ancient code, and all the new work happens several branches deep. So you'd have a branch for the next version, and another branch for the next...

With branches for older software, you'll eventually stop working with some of those branches. (Do you think Microsoft is still making fixes to Word 95?)

Q: To create tags and branches with Subversion, we used the copy command. Is that normal?

A: Well, it's normal for Subversion. That's because Subversion was designed for very "cheap" copies, which just means a copy doesn't create lots of overhead. When you create a copy, Subversion actually just marks the revision you copied from, and then stores changes relative to that. Other version control tools do things differently. For example, CVS has an explicit tag command, and branches result in "real" copies of files, meaning they take a lot of time and resources.

Sharpen your pencil

WIth the security fix to Version 1.0 taken care of, we're back to our original user story. Bob needs to implement two different saving mechanisms for the BeatBox application: one for when the user is on a Mac, and one for when a user is on a Windows PC. Since these are two completely different platforms, what should Bob do here?

What should Bob do? .....

.....

.....

.....

Why? .....

.....

.....

.....

When NOT to branch...

Did you say that Bob should branch his code to support the two different features? Modern version control tools do make branching cheap from a **technical perspective.** The problem is there's a lot of hidden cost from the **people perspective**. Each branch is a separate code base that needs to be maintained, tested, documented, etc.

For example, remember that critical security fix we made to Version 1.0 of BeatBox? Did that fix get applied to the trunk so that it stays fixed in Version 2.0 of the software? Has the trunk code changed enough that the fix isn't a straightforward copy, and we need to so something differently to fix it?

The same would apply with branching to support two different platforms. New features would have to be implemented to **both** branches. And then, when you get to a new version, what do you do? Tag both branches? Branch both branches? It gets confusing, fast. Here are some rules of thumb for helping you know when ***not*** to branch:

Branch when...

- ☐ You have released a **version of the software** that you need to maintain **outside of the main development cycle**.
- ☐ You want to try some **radical changes to code** that you might need to throw away, and you **don't want to impact the rest of the team** while you work on it.

Do not branch when...

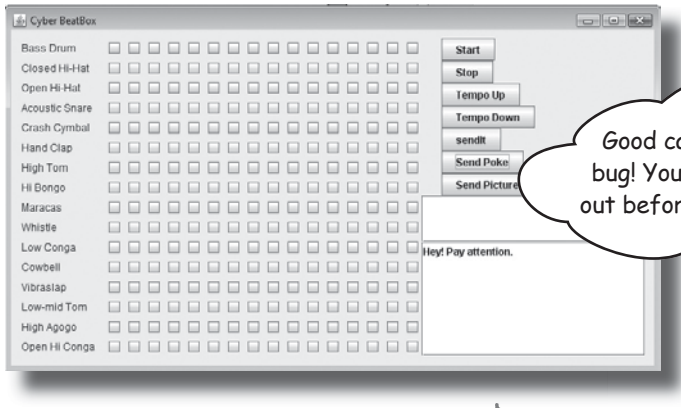
- ☐ You can accomplish your goal by **splitting code into different files** or libraries that can be built as appropriate on different platforms.
- ☐ You have a bunch of developers that can't keep their code compiling in the trunk so you try to **give them their own sandbox** to work in.

There are other ways to keep people from breaking other people's builds. We'll talk about those in a later chapter.

The Zen of good branching

Branch only when you absolutely have to. Each branch is a potentially large piece of software you have to maintain, test, release, and keep up with. If you view branching as a major decision that doesn't happen often, you're ahead of the game.

We fixed Version 1...



Version 1.1 is released, and the security bug is no more.

... and Bob finished Version 2.0 (so he says)

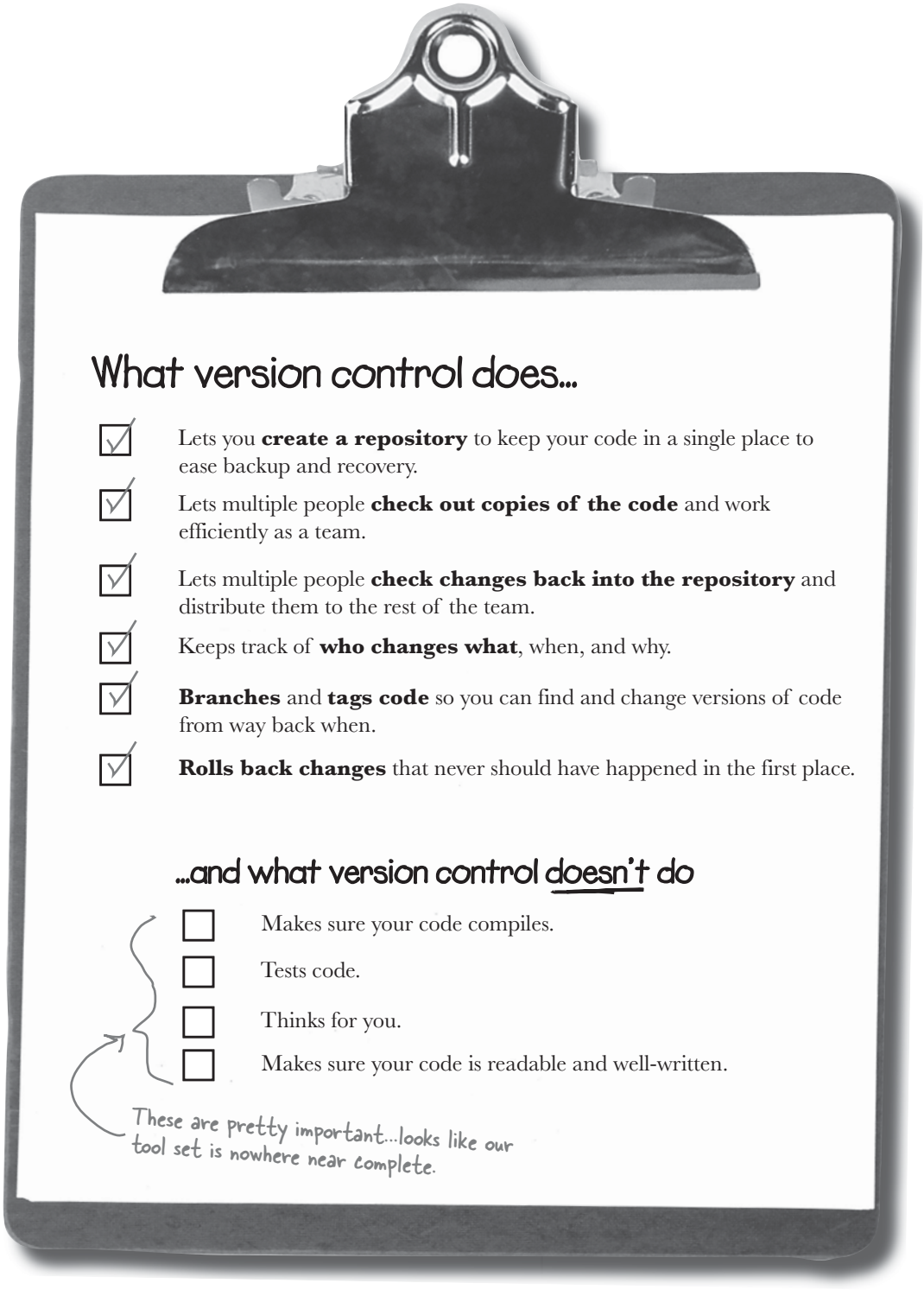
We've come a long way in this chapter, but there are ~~people~~ things that version control alone just can't fix...Can you list some troubles that Bob can still get into, even if he uses version control to manage his code?

.....

.....

.....

.....



What version control does...

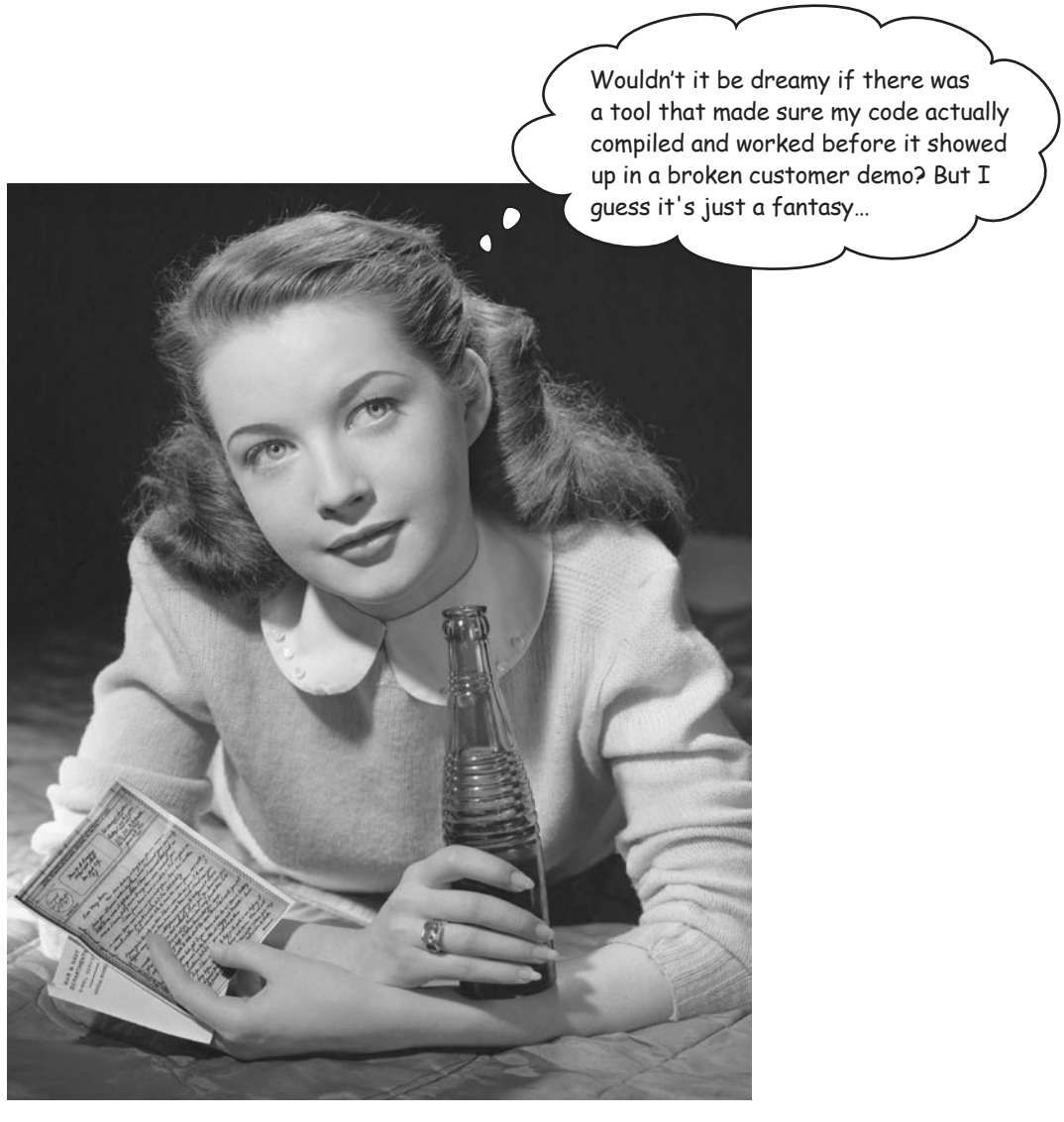
- ☑ Lets you **create a repository** to keep your code in a single place to ease backup and recovery.
- ☑ Lets multiple people **check out copies of the code** and work efficiently as a team.
- ☑ Lets multiple people **check changes back into the repository** and distribute them to the rest of the team.
- ☑ Keeps track of **who changes what**, when, and why.
- ☑ **Branches** and **tags code** so you can find and change versions of code from way back when.
- ☑ **Rolls back changes** that never should have happened in the first place.

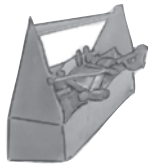
...and what version control doesn't do

- ☐ Makes sure your code compiles.
- ☐ Tests code.
- ☐ Thinks for you.
- ☐ Makes sure your code is readable and well-written.

These are pretty important...looks like our tool set is nowhere near complete.

Version control can't make sure your code actually works...





Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you learned about several techniques to keep you on track. For a complete list of tools in the book, see Appendix ii.

Development Techniques

Use a version control tool to track and distribute changes in your software to your team

Use tags to keep track of major milestones in your project (ends of iterations, releases, bug fixes, etc.)

Use branches to maintain a separate copy of your code, but only branch if absolutely necessary

Here are some of the key techniques you learned in this chapter...

... and some of the principles behind those techniques.

Development Principles

Always know where changes should (and shouldn't) go

Know what code went into a given release – and be able to get to it again

Control code change and distribution

BULLET POINTS

- **Back up** your version control repository! It should have all of your code and a history of changes in it.
- Always use a **good commit message** when you commit your code—you and your team will appreciate it later.
- **Use tags liberally**. If there's any question about needing to know what the code looked like before a change, tag that version of your code.
- **Commit frequently** into the repository, but be careful about breaking other people's code. The longer you go between commits, the harder merges will be.
- There are lots of **GUI tools** for version control systems. They help a lot with merges and dealing with conflicts.

Sharpen your pencil Solution

Write down three problems with the approach outlined above for handling future changes to version 1.0 (or is it 1.1?).

1. You need to keep track of what revisions go with what version of the software.

2. It's going to be very difficult to keep 2.0 code changes from slipping into v1.x patches.

3. Changes for Version 2.0 could mean you need to delete a file or change a class so much that it would be very difficult to keep a v1.x patch without conflicting.

Insert tab a into slot b...

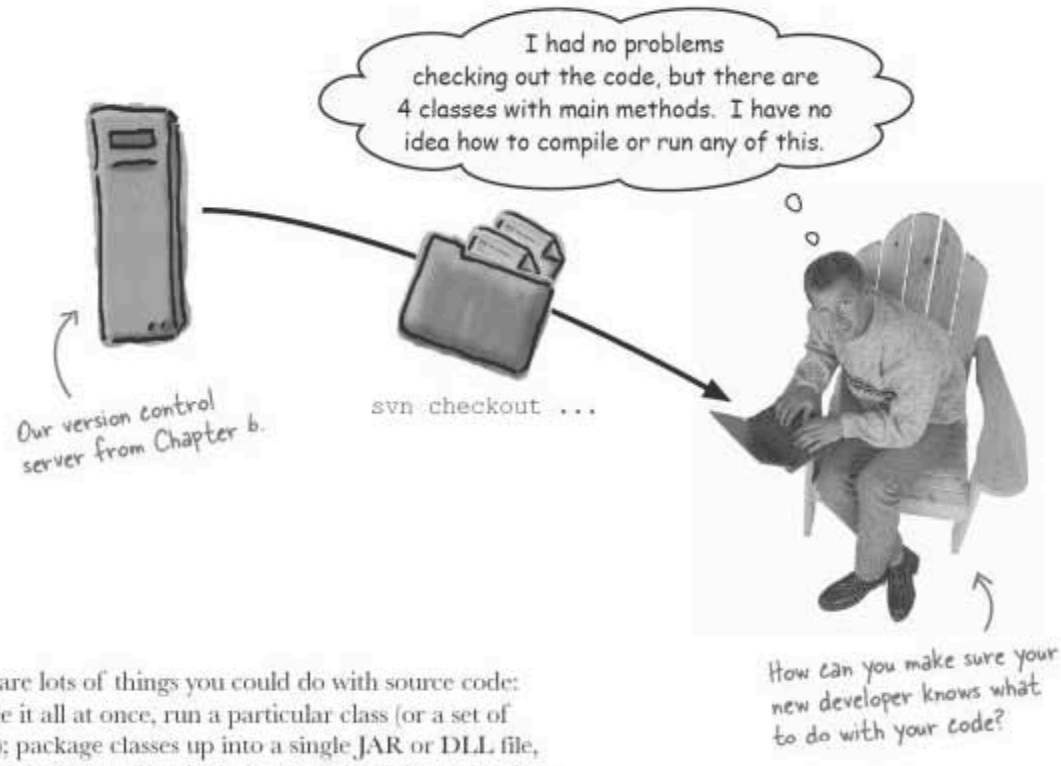
It pays to follow the instructions...

...especially when you write them yourself.

It's not enough to use version control to ensure your code stays safe. You've also got to worry about **compiling your code** and packaging it into a deployable unit. On top of all that, which class should be the main class of your application? How should that class be run? In this chapter, you'll learn how a **build tool** allows you to **write your own instructions** for dealing with your source code.

Developers aren't mind readers

Suppose you've got a new developer on your team. He can check out code from your version control server, and you're protected from his overwriting your code, too. But how does your new team member know which dependencies he's got to worry about? Or which class he should run to test things out?



There are lots of things you could do with source code: compile it all at once, run a particular class (or a set of classes); package classes up into a single JAR or DLL file, or multiple library files; include a bunch of dependencies... and these details change for every project you'll work on.

Software must be usable

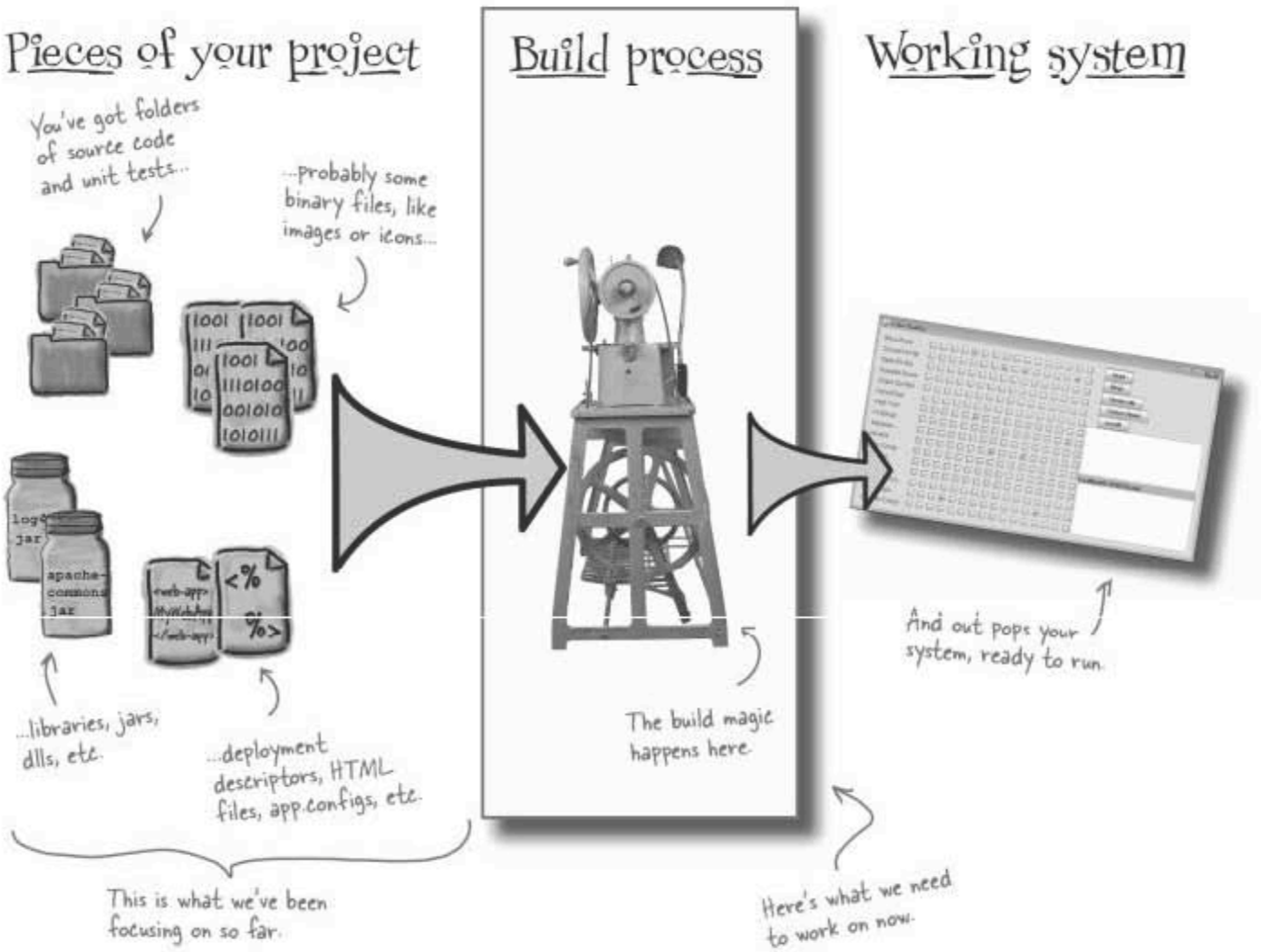
It doesn't do you much good to put in a version control server if you can't also be sure your code is used properly once it's checked out. And that's where build scripts come in.

Good code is easy to USE, as well as easy to GET.

Building your project in one step

When someone wants to run your project, they need to do more than just compile source code—they need to **build** the project. Compiling source code into binary files is important, but building a project usually involves ***finding dependencies***, ***packaging up your project*** into a usable form, and more.

And since tasks like these are the same each time they're run, building a project is a perfect candidate for **automation**: using a tool to handle the repetitive work for you. If you're using an IDE to write your code, a lot of this is handled for you when you click "Build." But there's a lot of work going on when you press that "Build" button:



Ant: a build tool for Java projects

Ant is a build tool for Java that can compile code, create and delete directories, and even package up files for you. It all centers around a **build script**. That's a file you write, in XML for Ant, that tells the tool what to do when you need to build your program.

You can download Ant from http://ant.apache.org/

The steps to build your project are stored in an XML file, usually named build.xml.

This whole file is called a build script.

Each build file represents a single project.

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

  <property name="src" location="src"/>
  <property name="bin" location="bin"/>
  <property name="bin-instrumented" location="bin-
instrumented"/>
  <property name="reportdir" location="reportdir"/>

  <target name="clean">
        <delete dir="${bin}"/>
        <delete dir="${reportdir}"/>
        <delete dir="${bin-instrumented}"/>
        <delete file="cobertura.ser"/>
  </target>

  <target name="init">
        <mkdir dir="${bin}"/>
        <mkdir dir="${reportdir}"/>
        <mkdir dir="${bin-instrumented}"/>
  </target>
```

What's needed to build your project is broken up into steps called targets. Each target can have more than one task.

You just kick off a build with a single command. Ant runs the default target in build.xml, and follows your instructions.

```
File Edit Window Help Dolt
hfsd> ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\Users\Developer\workspaces\HFSD\BeatBox\bin
    [mkdir] Created dir: C:\Users\Developer\workspaces\HFSD\BeatBox\dist

compile:
    [javac] Compiling 4 source files to C:\Users\Developer\workspaces\HFSD\BeatBox\bin

dist:
      [jar] Building jar: C:\Users\Developer\workspaces\HFSD\BeatBox\dist\BeatBox.jar

BUILD SUCCESSFUL
Total time: 16 seconds

hfsd>
```

In this case, Ant creates a directory structure...

...compiles code into that structure...

...and builds a JAR file.

Projects, properties, targets, tasks

An Ant build file is broken into four basic chunks:

1 Projects

Everything in your build file is part of a single project:

In this case, Ant will run the dist target when the script is run.

Everything in Ant is represented by an XML element tag.

```
<project name="BeatBox" default="dist">
```

Everything else in the build file is nested inside the project tag.

Your project should have a name and a default target to run when the script is run.

2 Properties

Ant properties are a lot like constants. They let you refer to values in the script, but you can change those values in a single place:

A property has a name and a value.

```
<property name="version" value="1.1" />
<property name="src" location="src" />
<property name="xerces-src" location="${src}/xerces" />
```

You can use properties with ${property-name}, like this.

You can use location instead of value if you're dealing with paths.

3 Targets

You can group different actions into a target, which is just a set of work. For example, you might have a `compile` target for compilation, and an `init` target for setting up your project's directory structure.

A target has a bunch of tasks nested within it

```
<target name="compile" depends="init">
```

A target has a name, and optionally a list of targets that must be run before it.

4 Tasks

Tasks are the work horses of your build script. A task in Ant usually maps to a specific command, like `javac`, `mkdir`, or even `javadoc`:

This makes a new directory, using the value of the src property.

```
<mkdir dir="${src}">
<javac srcdir="${src}" destdir="${bin}" />
```

Each Ant task has different parameters, depending on what the task does and is used for.

The syntax here is particular to Ant, but the principles work with all build tools, in any language.

Ant is great for Java, but not everyone uses Java. For now, though, focus on what a good build tool gives you: a way to manage projects, constants, and specific tasks. In a few pages, we'll talk about build tools that work with other languages, like PHP, Ruby, and C#.

No... you're supposed to learn a new tool so *someone (or something) ELSE* can *build* your project.

It's easy to see a build tool as just one more thing to learn and keep up with. But most build tools, like Ant, are really easy to learn. In fact, you're just about to put together your first build script, and you already know more than you think!

On top of that, your build tool is just that: ***a tool***. It helps you get things done faster, especially over a lot of projects. You'll learn a little bit of syntax for your build tool, and hardly need to learn anything else about it.

Oh, and remember: ***the build tool is for your team, not just you***. While you may know how to compile your project, and keep up with its dependencies, everyone else might not. A build tool and build script lets everyone on your team use the same process to turn source code into a running application. With a good build script all it takes is one command to build the software; it's impossible for a developer to accidentally leave a step out—even after working on two other projects for six months.

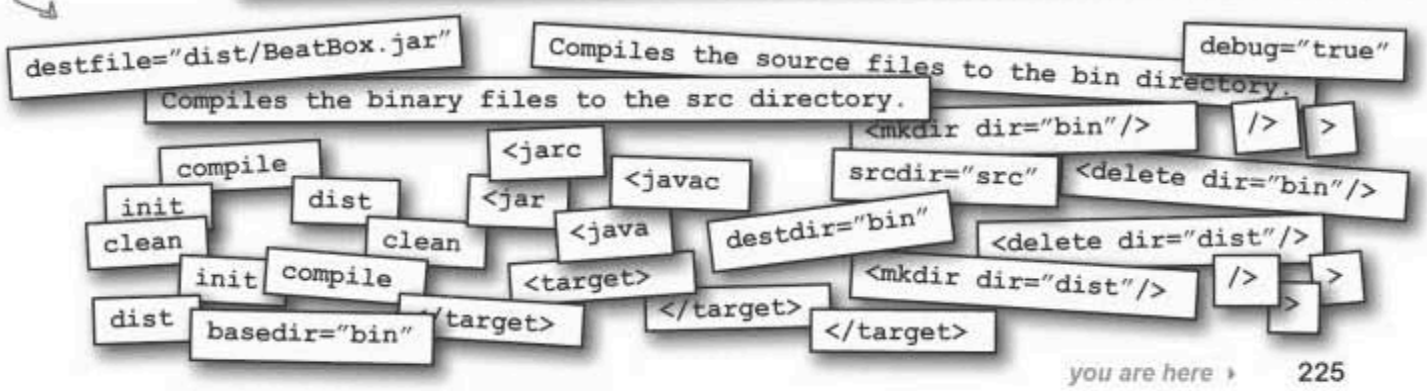
Ant Build Magnets

Ant files are easier to use—and write—than you think. Below is part of a build script, but lots of pieces are missing. It's up to you to use the build magnets at the bottom of the page to complete the build script.

Put the magnets between the target elements to complete the build.xml file.

```
<project name="BeatBox" default="dist">
  <target name="init"
          description="Creates the needed directories.">
      _____
      _____
  </target>

  <target name="clean"
          description="Cleans up the build and dist directories.">
      _____
      _____
  </target>

  <target name="compile" depends="init"
          description="_____">
      _____
      _____
  </target>

  <target name="dist" depends="_____"
          description="Packages up BeatBox into BeatBox.jar">
      _____
    _____

</project>
```

Seems like there are a couple of extra magnets, so be careful.



Ant Build Magnet Solutions

Your task was to reassemble a working build file for building the BeatBox application.

```
<project name="BeatBox" default="dist">
  <target name="init"
          description="Creates the needed directories.">
    <mkdir dir="bin"/>
    <mkdir dir="dist"/>
  </target>

  <target name="clean"
          description="Cleans up the build and dist directories.">
    <delete dir="bin"/>
    <delete dir="dist"/>
  </target>

  <target name="compile" depends="init"
          description="Compiles the source files to the bin directory.">
    <javac srcdir="src" destdir="bin" />
  </target>

  <target name="dist" depends="compile"
          description="Packages up BeatBox into BeatBox.jar">
    <jar destfile="dist/BeatBox.jar" basedir="bin" />
  </target>
</project>
```

You specify the default target to call (in this case, dist) if the person running Ant doesn't specify one. In general this should do everything it needs to do to get your project from zero to running.

The mkdir task creates the directory specified by the "dir" attribute.

The delete task can delete directories or files by specifying a dir or file attribute.

The javac task compiles java code in the srcdir and puts classes in the destdir.

Each target can have a description that is printed if you ask Ant to display project information.

The dist target depends on compile, which in turn depends on the init target.

Here's the default target

The jar task creates a JAR from the files it finds in the basedir. You can also specify manifest information, files to exclude, etc.

Be sure to close these elements with "/>", which is like a closing tag.

there are no Dumb Questions

Q: My project isn't in Java—do I still need a build tool?

A: Probably, and depending on what environment you're working in, you might already be using one. If you're developing in Microsoft Visual Studio, you're almost certainly ***already*** using their build system, called MSBuild (open your csproj file in Notepad...seriously). It uses an XML description of the build process similar to the way Ant does. Visual Studio started that file for you, but there's a whole lot more MSBuild can do for you that the IDE doesn't expose. If you're not in Visual Studio but are doing .NET development, you might want to check out NAnt. It's basically a port of Ant for the .NET world. Ruby uses a tool called rake to kick off tests, package up the application, clean up after itself, etc.

But there are some technologies, like Perl or PHP, where build scripts aren't quite as valuable, because those languages don't compile or package code. However, you can still use a build tool to package, test, and deploy your applications, even if you don't need everything a build tool brings to the table.

Q: I'm using an IDE that builds everything for me. Isn't that enough?

A: It might be enough for *you*, but what about everyone else on your team? Does everyone on your team have to use that IDE? This can be a problem on larger projects where there's an entirely separate group responsible for building and packaging your project for other teams like testers or QA.

Then there are tasks that your IDE *doesn't* do... (If you can't think of anything like that, we'll talk about some great ones in the next few chapters). In general, if your project is more than a one-person show (or you want to use any of the best practices we're going to talk about in the next few chapters) you need to think about a build tool.

Q: Where did you come up with those bin, dist, and src directory names?

A: Those directories are an unofficial standard for Java projects. A few others you're likely to see are `docs` for generated documenation, `generated` for things like web-service-generated clients and stubs, and `lib` for library dependencies you might need.

There's nothing about these directory names that's set in stone, and you can adjust your build file to deal with whatever you use on your project. However, if you stick with common conventions, it makes it easier for new team members to get their heads around your project.

Q: Why are you even talking about Ant? Don't you know about Maven?

A: Maven is a Java-oriented "software project management and comprehension tool." Basically, it goes beyond the smaller-scale Ant tasks we've been talking about and adds support for automatically fetching library dependencies, publishing libraries you build to well-known places, test automation, etc. It's a great tool, but it masks a lot of what's going on behind the scenes. To get the most out of Maven you need to structure your project in a particular way.

For most small- to medium-sized projects, Ant can do everything you'll need. That's not to discourage you from checking out Maven, but it's important to understand the underlying concepts that Maven does such a great job of hiding. You can find out more about Maven at *http://maven.apache.org/*.

Q: What should my default target be? Should it compile my code, package it, generate documentation, all of the above?

A: That really depends on your project. If someone new was to check out your code, what are they most likely looking to do with it? Would they want to be able to check out your project and expect to be able to run it in one step? If so, you probably want your default target to do everything. But if "everything" means signing things with encryption keys and generating an installer with InstallShield and so on, you probably don't want that by default. A lot of projects actually set up the default target to output the project help information so that new people can see what their options are and pick appropriately.

Q: The build.xml file has directory names repeated all over the place. Is that a good idea?

A: Great catch! For a build script the size of the one we're using here, it's OK. But if you're writing a more complex build file, it's generally a good idea to use properties to let you define the directories once, and refer to them by aliases throughout the rest of the file. In Ant, you'd use the `property` tag for this, like on page 223.

Q: Couldn't I just do all of this with a batch file or shell script?

A: Technically, yes. But a build system comes with lots of software-development-oriented tasks that you'd either have to write yourself or lean on external tools to handle. Build tools also integrate into continuous integration systems, which we'll talk about in the next chapter.

Good build scripts...

A build script captures the details that developers probably don't need to know right from the start about how to compile and package an application, like BeatBox. The information isn't trapped in one person's head; it's captured in a version-controlled, repeatable process. But what exactly should a standard build script do?

You'll probably add tasks to your own build scripts, but all build scripts should do a few common things...

...generate documentation

Remember those description tags in the build file? Just type `ant -projecthelp` and you'll get a nice printout of what targets are available, a description of each, and what the default target is (which is usually what you want to use).

Your build tool probably has a way to generate documentation about itself and your project, even if you're not using Ant and Java.

```
File Edit Window Help Huh?
hfsd> ant -projecthelp
Buildfile: build.xml

Main targets:

 clean    Cleans up the build and dist directories.
 compile  Compiles the source files to the bin directory.
 dist     Packages up BeatBox into BeatBox.jar
 init     Creates the needed directories.
Default target: dist

hfsd>
```

...compile your project

Most importantly, your build scripts compile the code in your project. And in most scripts, you want a single command that you can run to handle everything, from setup to compilation to packaging.

Here you can see the target dependencies in action: our build script tells Ant to run the dist target by default, but in order to do that, it has to run compile, and in order to do that, it has to run init

```
File Edit Window Help Build
hfsd> ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\Users\Developer\workspaces\HFSD\BeatBox\bin
    [mkdir] Created dir: C:\Users\Developer\workspaces\HFSD\BeatBox\dist

compile:
    [javac] Compiling 4 source files to C:\Users\Developer\workspaces\HFSD\BeatBox\bin

dist:
      [jar] Building jar: C:\Users\Developer\workspaces\HFSD\BeatBox\dist\BeatBox.jar

BUILD SUCCESSFUL
Total time: 16 seconds

hfsd>
```

This is cool. But I saw there's a clean target in the build file we haven't talked about yet...

Good catch—a clean target is there to clean up the scraps of things that compiling leaves laying around. It's important to have a target that will get the project back to what it would look like if you checked the project out from the repository. That way, you can test things from a new developer's perspective.

Your tool may call this something else, but the idea is the same—clean up the mess made by building your project.

...clean up the mess they make

The final target we'll discuss in the BeatBox build script deletes the directories created during the build process: the `bin` directory for compiled classes and the `dist` directory for the final JAR file.

Since dist is the default target, you have to explicitly tell Ant to run the clean target.

```
File Edit Window Help Scrub
hfsd> ant clean
Buildfile: build.xml

clean:
   [delete] Deleting directory C:\Users\Developer\workspaces\HFSD\BeatBox\bin
   [delete] Deleting directory C:\Users\Developer\workspaces\HFSD\BeatBox\dist

BUILD SUCCESSFUL
Total time: 3 seconds

hfsd>
```

Ant runs the delete tasks to clean up the bin and dist directories and remove all of their contents

Good build scripts go BEYOND the basics

Even though there are some standard things your scripts should do, you'll find plenty of places a good build tool will let your script go beyond the basics:

1. **Reference libraries your project needs**

You can add libraries to your build path in Ant by using the `classpath` element in the `javac` task:

```
<javac srcdir="src" destdir="bin">
  <classpath>
    <pathelement location="libs/junit.jar"/>
    <pathelement location="libs/log4j.jar"/>
  </classpath>
</javac>
```

Each pathelement points to a single JAR to add to the classpath. You can also point to a directory if you need to.

If your project depends on libraries you don't want to include in your `libs` directory, you can also have Ant download libraries using FTP, HTTP, SCP, etc., using additional Ant tasks (check out the Ant task documentation for details).

2. **Run your application**

Sometimes it's not just compiling your application that requires some background knowledge; running it can be tricky, too. Suppose your app requires the setting of a complex library path or a long string of command-line options. You can wrap all of that up in your build script using the `exec` task:

```
<exec executable="cmd">
  <arg value="/c"/>
  <arg value="iexplorer.exe"/>
  <arg value="http://www.headfirstlabs.com/"/>
</exec>
```

Executing something on the system directly is obviously going to be platform-dependent. Don't try to run iexplorer.exe on Linux.

(but do go to Head First Labs)

or the `java` task:

```
<java classname="headfirst.sd.chapter6.BeatBox">
  <arg value="HFBuildWizard"/>
  <classpath>
    <pathelement location="dist/BeatBox.jar"/>
  </classpath>
</java>
```

If you wrap this in a target then you won't ever have to type "java -cp blahblah..." again to launch BeatBox.

3 **Generate documentation**

You've already seen how Ant can display documentation for the build file, but it can also generate JavaDoc from your source code:

```
<javadoc packagenames="headfirst.sd.*"
  sourcepath="src"
  destdir="docs"
  windowtitle="BeatBox Documentation"/>
```

Note that Ant can generate your HTML files for you—but it can't write the documentation you've been putting off.

There are other elements you can include in the JavaDoc task to generate headers and footers for each page if you need to.

4 **Check out code, run tests, copy builds to archival directories, encrypt files, email you when the build finishes, execute SQL...**

There are lots more tasks you can use depending on what you need your build file to do. Now that you know the basics, all of the other tasks look pretty much the same. To get a look at the tasks Ant offers go to: `http://ant.apache.org/manual/index.html`.

Automation lets you focus on code, not repetitive tasks.

With a good build script, you can automate a pretty sophisticated build process. It's not uncommon to see multiple build files on a single project, one for each library or component. In cases like that, you might want to think about a master build file (sometimes called a **bootstrap** script) that ties everything together.

Your build script is code, too

You've put a lot of work into your build script. In fact, ***it's really code***, just like your source files and deployment descriptors. When you look at your build script as code, you'll realize there are lots of clever things you can do with it, like deal with platform differences between Windows and Unix, use timestamps to track builds or figure out what needs to be recompiled—all completely hidden from the person trying to do the build. But, like all other code, it belongs in a repository...

You should always check your build script into your code repository:

Since we're adding a new file to our checked-out code, we use the subversion add command and tell it which file we want added.

When you add a file to the repository, you're telling Subversion that it should care about that file. But you still need to commit that file, even after it's been added.

```
File  Edit  Window  Help  Safe
hfsd> svn add build.xml

A         build2.xml

hfsd> svn commit -m "Added Ant build.xml file."

Sending        build.xml
Transmitting file data .

Committed revision 11.

hfsd>
```

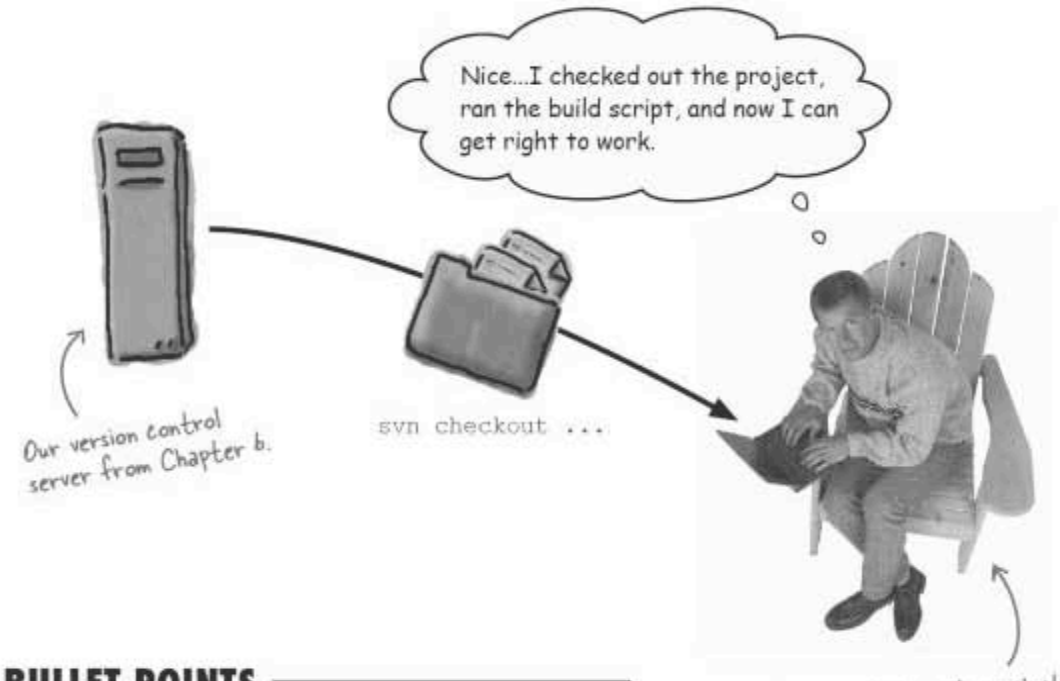
Don't forget your check in comments.

With your build script in the repository, it's available to everyone else when they do an update. Your version control software will track any changes to the script, and the script gets tagged with everything else whenever you do a release. This means that you won't have to remember all the magic commands you needed to build the nostalgic Version 1.0 in a few years at your IPO party!

Your build script is code...ACT LIKE IT! Code belongs in a version control system, where it's versioned, tagged, and saved for later use.

New developer, take two

We haven't written any new classes, talked to the customer, broken tasks up into stories, or demoed software for the customer...but things are still looking a lot better. With a build tool in place, let's see what bringing on the new developer looks like:



Your new developer's productive within minutes, instead of hours (or worse, days) and won't spend that time bugging you for help on how to build the system.

BULLET POINTS

- A build tool is simply a **tool**. It should make building your project **easier**, not harder.
- Most build tools use a **build script**, where you can specify what to build, several different instruction sets, and locations of external files and resources.
- Be sure you create a way to **clean up** any files your script creates.
- Your build script is **code** and should be versioned and checked into your code repository.
- **Build tools are for your team**, not just you. Choose a build tool that works for everyone on your team.

Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you learned about several techniques to keep you on track. For a complete list of tools in the book, see Appendix ii.

Development Techniques

Use a build tool to script building, packaging, testing, and deploying your system

Most IDEs are already using a build tool underneath. Get familiar with that tool, and you can build on what the IDE already does

Treat your build script like code and check it into version control

Here are some of the key techniques you learned in this chapter...

... and some of the principles behind those techniques.

Development Principles

Building a project should be repeatable and automated

Build scripts set the stage for other automation tools

Build scripts go beyond just step-by-step automation and can capture compilation and deployment logic decisions

BULLET POINTS

- All but the smallest projects have a **nontrivial build process**.
- You want to **capture** and **automate** the knowledge of **how to build your system**—ideally in a single command.
- **Ant** is a build tool for **Java projects** and captures build information in an XML file named build.xml.
- The more you take advantage of **common conventions**, the more **familiar** your project will look to someone else, and the easier the project will be to integrate with external tools.
- Your **build script** is just as much a part of your project as **any other piece of code**. It should be checked into version control with everything else.