Chapter

33

Managing Source Code

In the description that follows, I am assuming that you are not using a product such as ENVY to manage your source code and configurations. Instead, you are using some combination of filing out your changes, saving your image, and filing in your changes. We'll talk more about ENVY in Chapter 34, ENVY.

As you work, you are probably saving your changes in some way, whether it's by filing out categories, classes, or methods, or by periodically saving your image. Certainly you'll need to file out your changes if other developers are to incorporate them. Whatever technique you use to protect your work from crashes and to distribute your changes to others, it's important to be able to rebuild your image with all the correct code.

The technique that I've usually used is to rely on developers filing out changes, usually at the class category level, then every day starting up a clean image — one containing no source code — and filing in all the latest code. If you choose to save your image periodically instead of filing out your changes, then I recommend that you also have available a clean image you can start from, again filing in the latest code. In what follows, we'll look closely at some of the aspects of filing out code and filing in code.

What we describe is certainly not the only way to manage your source code and may not even the best way. If you have a very large application, it may take too long to do what is shown here more than perhaps once a week. It may be a better option to maintain an *application base image* that contains all the needed system class additions and modifications, using a fileIn such as shown below to build the application base. Derived from the application base might be a *developer base image* that also contains development environment changes that are expected to be in the images of all developers. Again, this could be maintained using some of the code shown below. Derived from the developer base might be *individual base images* that contain the preferences of individual developers. Again, the code below can help track those changes. Periodically the developers would file in all the latest application code, then would save their new *working image*. Also periodically, the latest application code would be filed into the application base image and saved as a *testing image*.

Treat the code shown as possibility code. Take what you like, discard what you don't like, and above all, use it to generate ideas that work for your development environment.

Change Set

The Change Set is a wonderful tool for keeping track of what changes you are making. Every change you make gets recorded in the Change Set so rather than writing down your changes on paper (as I did when I first started programming in Smalltalk), you can simply look at the Change Set. By doing two small things, you can make it truly useful. We show how it might be done later in this chapter, but basically, when you have filed in your changes, you do `ChangeSet noChanges` to empty the change set. After all, you know that there have been a lot of changes due to filing in all those classes and methods; what you are interested in is changes made since then.

The second change is `Browser removeChangesOnFileOut: true`. This tells the Browser that when you file out changes, those changes should be removed from the Change Set. So, by looking at the Change Set, you can tell what changes you have made but not yet saved. As you file out the relevant categories, classes, or methods, the Change Set gets smaller. When you get to an empty Change Set, you have filed out all the changed code. (Unfortunately there is one small gotcha: if you rename classes using the *rename* menu option, that change is not recorded in the Change Set, so you'll have to remember it.)

Filing in code

It's important to periodically file in the latest code: application code, necessary changes to system classes, and development environment code. In the subsections that follow, we'll look at a mechanism for filing in all three types of code. Whether you do this daily, more frequently, or less frequently, it's something that you'll need to do at times in order to keep the images of all developers synchronized. In the fileIn examples that follow, I'll intersperse text with the fileIn code to explain what each section is doing. The code can be found in the `manage` directory in the file `install.st`.

Filing in the application

```
"install.st - Install the Application software
Created: When. Who. Why
Changed: When. Who. Why
"

| sourceDir sysAddDir sysModDir localFile
beforeClassCount beforeMethodCount afterClassCount afterMethodCount |
```

The next section sets up the various directories and files that we will need later on. In this example, we are assuming that development is done on both UNIX and MS-Windows platforms, so the code determines which platform we are currently on and sets the names for that platform.

```
"Figure out the various directories. We allow development to be done
on different platform"
Window platformName = 'X11'
   ifTrue:
      [sourceDir := '/product/src' asFilename.
      sysAddDir := '/visual/sys/additions' asFilename.
      sysModDir := '/visual/sys/modifications'.
      localFile := '/home/', (CEnvironment getenv: 'LOGNAME'),
'/.vwlocal' asFilename]
   ifFalse:
```

```
[sourceDir := 'c:\alec\st\source' asFilename.
sysAddDir := 'c:\alec\st\sysadds' asFilename.
sysModDir := 'c:\alec\st\sysmods' asFilename.
localFile := 'c:\visual\local\local.st' asFilename].
```

If the directories are subdirectories of a base directory, we could write code without hard-coding the base directory. In the example that follows, we assume that `install.st` is being filed in from the base directory. The idea is that we trace the context back until we find a stream, which we assume is the stream filing in this file. Once we have the base directory, we can build up the subdirectories. An example of this code can be found in the `manage` directory in the file `install2.st`.

```
context := thisContext.
[context receiver isKindOf: Stream]
   whileFalse:
      [context := context sender].
baseDir := context receiver name asFilename head asFilename.
sourceDir := baseDir construct: 'source'.
```

The next section files in any system additions and system modifications that are required for the application. It makes the assumption that it should file in all the `.st` files in these directories. In other words, if you don't want something filed in, don't put it in one of these directories.

```
"File in any required system additions and modifications that we
find"
(sysAddDir filesMatching: '*.st') do: [ :each | each asFilename
fileIn].
(sysModDir filesMatching: '*.st') do: [ :each | each asFilename
fileIn].
```

We now file in any development environment changes that are specific to the programmer. If there is a named file and the file is readable we go ahead and read it. The intention is that each developer will have a file with a known name in a known, but unique to them, directory, and that if this file exists, it will be filed in. The names of the directory and file will depend on what type of platform you are working on and how your computers are networked. This scheme will work also for building the deployed application image because you don't need to set up a local file for the user that creates the application image.

```
"File in any programmer-specific changes that we find"
(localFile notNil and: [localFile isReadable])
   ifTrue: [localFile fileIn]
   ifFalse: [Transcript cr; show: 'No local file'].
```

Before we file in the application code we want to count the number of classes and methods in the image. We'll do the same after filing in the code, then we'll be able to report how many classes and methods our application contains.

```
"Count the number of classes and methods in the image before filing
in the application code"
beforeClassCount := Smalltalk classNames size.
beforeMethodCount := Smalltalk classNames
   inject: 0
   into:    [:subtotal :each | | class|
         class := Smalltalk at: each.
```

```
            subtotal + class selectors size + class class selectors
size].
```

Now we create any necessary pool dictionaries and add all the pool dictionary variables. We need to do this in a fileIn otherwise there will be problems: if the pool dictionary doesn't exist, the fileIn will raise an exception; if the pool dictionary variables don't exist, then references to them will fail when you try to accept a method or run the code.

```
"Create any necessary pool dictionaries and add all the pool
dictionary variables"
Smalltalk at: #MyPoolDictionary put: Dictionary new.
MyPoolDictionary at: #PoolVariableOne put: nil.
MyPoolDictionary at: #PoolVariableTwo put: nil.
```

The next section files in all the application code. In this example, we specify exactly what application code we want in the image, unlike the system additions where we took all the files in the directory. We also assume that you filed out by class category. If you file out by class or by method, it would probably be too painful to specify all the files, and would be easier to file in every file in the directory.

```
"File in all the named categories"
#( 'CatgOne.st'
   'CatgTwo.st'
   'CatgThree.st'
) do: [ :each | (sourceDir construct: each) fileIn ].
```

Now we count the number of classes and methods in the image after filing in the application code, and report on the counts.

```
"Count the number of classes and methods in the image after filing in
the application code,
then report on the counts."
afterClassCount := Smalltalk classNames size.
afterMethodCount := Smalltalk classNames
   inject: 0
   into:    [:subtotal :each | | class|
         class := Smalltalk at: each.
         subtotal + class selectors size + class class selectors
size].

Transcript
   cr; nextPutAll: 'Before loading in:';
   crtab; nextPutAll: 'Class count = '; print: beforeClassCount;
   crtab; nextPutAll: 'Method count = '; print: beforeMethodCount;
   cr; nextPutAll: 'After loading in:';
   crtab; nextPutAll: 'Class count = '; print: afterClassCount;
   crtab; nextPutAll: 'Method count = '; print: afterMethodCount;
   cr; nextPutAll: 'New:';
   crtab; nextPutAll: 'Classes = '; print: (afterClassCount -
beforeClassCount);
   crtab; nextPutAll: 'Methods = '; print: (afterMethodCount -
beforeMethodCount);
   endEntry.
```

The last section does any final system changes that need to be done. In particular, we empty the change set so that the Change Set just reflects changes we make after filing in our application code, and make sure that the delete key deletes in a forward direction.

```
"Do any final system changes"
LookPreferences deleteForward: true"Make the delete key delete
forwards"
ChangeSet noChanges.                "Empty the change set"
```

Filing in development environment changes

Let's now look at an example of a local file that files in development environment changes. Again, explanatory text will be interspersed with the code. The code can be found in the `manage` directory in the file `local.st`.

```
"local.st - Dave's specific environment changes
When. Who.     Why.
"

| genericDir localDir |

Window platformName = 'X11'
   ifTrue:
      [genericDir := '/visual/generic'.
      localDir := '/home/alec/st/fileIns']
   ifFalse:
      [genericDir := 'c:\visual\generic'.
      localDir := 'c:\alec\st\fileIns'].
```

In this section we file in everything we find in two directories. The first one is our own personal fileIn directory. This allows us to include any other enhancements we need. After we've done our personal changes to the image, we then fileIn everything in the generic directory. These are changes that everyone should be picking up. The reason for having this is that there may be modifications to the development environment that are standard across the organization and which a programmer can expect to see in another programmer's environment.

```
"Do any personal and generic development environment fileIns"
 (localDir asFilename filesMatching: '*.st')   do: [ :each | each
asFilename fileIn].
(genericDir asFilename filesMatching: '*.st') do: [ :each | each
asFilename fileIn].
```

The code that follows could easily be in one of our personal fileIns, but instead, we'll show it here. The code shown is described in detail in Chapter 31, Customizing your Environment, so we won't explain it in depth. The next section remaps the keyboard, assuming that you have defined a `keyboard` method on ParagraphEditor, remaps the interrupt key, then changes the window colors.

```
"Remap the keyboard"
ParagraphEditor keyboard
   bindValue: #cutKey: to: (TextConstants at: #Ctrlx);
   bindValue: #copyKey: to: (TextConstants at: #Ctrlc);
   bindValue: #pasteKey: to: (TextConstants at: #Ctrlv);
   bindValue: #acceptKey: to: (TextConstants at: #Ctrla).

"If this is not VW2.0, initialize the TextEditorController so it gets
the new keyboard mappings"
(Smalltalk version findString: '2.0' startingAt: 1) == 0
   ifTrue: [TextEditorController initialize].
```

```
"Change the Interrupt key from Ctrl-C to Ctrl-Q"
InputState interruptKeyValue: (TextConstants at: #Ctrlq).

"Change my window colors"
Win3WidgetPolicy defaultColorWidgetColors
   matchAt: SymbolicPaint background put: ColorValue lightCyan;
   matchAt: SymbolicPaint selectionBackground put: ColorValue pink;
   matchAt: SymbolicPaint selectionForeground put: ColorValue black.
Screen default updatePaintPreferences.
ScheduledControllers restore.
```

Windows created before we remapped the keyboard still use the old keyboard bindings so we create some new windows that use the new key bindings. We want the windows to be created automatically rather than waiting for us to position them, so we tell the ScheduledWindow class not to prompt us when opening the window. We also create a SystemWorkspace-like window which contains useful statements that can be highlighted and executed.

```
"Create some new windows with my new key bindings"
prompt := ScheduledWindow promptForOpen.
ScheduledWindow promptForOpen: false.
FullBrowser open.
ComposedTextView
   open: (ValueHolder with: ('c:\alec\st\useful' asFilename
contentsOfEntireFile))
   label: 'Useful Things'
   icon: nil
   extent: 500@200.
ComposedTextView open.
ScheduledWindow promptForOpen: prompt.
```

Generic Development Environment changes

In the directory containing changes to the generic development environment, we have at least one file. This file tells the Browser that when developers file out code, the changes should be removed from the Change Set.

```
"general.st - General Development Environment changes
Created: When. Who. Why
"

"Remove changes from the change set after doing a file out"
Browser removeChangesOnFileOut: true.
```

Using a class to file in the code

Besides filing in from a text file, we could use a class to control what gets filed in. For my personal customization I use a class rather than a file because I find it easier to make changes to a class. Here's an example of the main flow of my *Customizer* class. The code for this class can be found in the file `custom.st`.

```
Customizer>>customize
   "self new customize"
   self myFileInSystemAdditions.
   self myFileInSystemModifications.
   self myFileInNewClasses.
   self myKeyboardChanges.
   self myLauncherMenu.
```

```
self myResetChangeSet.
self myOpenWindows.
```

Filing out

Depending on how you manage your fileOuts, you may want to run programs like *diff* to find the differences between two versions. Unfortunately methods are filed out in a seemingly random order, which makes it difficult to discover the real differences. By changing `self selectors` to `self selectors asSortedCollection` in `ClassDescription>> fileOutMethodsOn:` you can have the methods file out in sorted order.

Creating a production image

When you rely on people to do things there is always a chance of error. People are creative, but also prone to making errors, which is why *make* schemes are so popular[1]. They apply rules to determine what to build, and then automatically compile and link any files that are out of date. The process of building and saving a Smalltalk image is usually somewhat manual. We may have a fileIn that gets the right code in, but a person usually has to decide that an image needs building and then do the manual steps of invoking the fileIn and saving the image. Here is a scheme that allows the building of a Smalltalk image to be done automatically, possibly as part of a *make* scheme. In VisualWorks 2.0 it was difficult to create a production image and this process makes it a lot easier. VisualWorks 2.5 has a new facility, the ImageMaker, that makes it much easier to create a production image. Below are two schemes to automate the image creation, one for VisualWorks 2.0 and one for 2.5. The code can be found in the files `imcr20.st` and `imcr25.st`.

Common to VisualWorks 2.0 and 2.5

The basic idea is that we start with a clean image that has either the *Stripper* or the *ImageMaker* code loaded and has a new *ImageCreator* class loaded. It does *not* have the application code loaded. The *Undeclared* dictionary should be cleaned up apart from the reference to your main application class (the ImageCreator will start your application, but since the application code is not yet loaded, the ImageCreator will be referring to a class that doesn't yet exist in the image). We then save the image under a descriptive name such as `makeIm`. When we run `makeIm`, it files in the application code, removes unnecessary classes, opens the application, closes the VisualLauncher, sets up a new emergency handler, and then saves the application image. We start by creating *ImageCreator*, a new subclass of Object with no instance or class variables.

In the ImageCreator class initialization, we register ourself as a dependent of *ObjectMemory*. This means that when ObjectMemory sends `changed:` messages, the ImageCreator class will receive them. Since it is the class that is registered as a dependent, we write `update:with:from:` on the class side. In this method, the only thing we want to hear is that we have just returned from a snapshot — that the image has just come up. When this is the case, we create a new instance of ImageCreator and tell it to make a new application image by sending the `createImage` message.

```
ImageCreator class>>initialize
```

[1] Make is a standard UNIX utility for building programs from source, header, and object files. It checks dates on the various files to determine which object files need to be rebuilt and which executable files need to be relinked.

```
   "Register ourselves to receive an update message when the image
comes up."
   "self initialize"
   (ObjectMemory dependents includes: self)
      ifFalse: [ObjectMemory addDependent: self]

ImageCreator class>>update: anAspect with: arguments from: anObject
   (anObject == ObjectMemory and: [anAspect == #returnFromSnapshot])
      ifTrue: [self new createImage]
```

VisualWorks 2.0

In VisualWorks 2.0 we need to first load in the Stripper code. We will create a subclass of Stripper called MyStripper where we will override a few methods to prevent dialogs coming up and to initialize variables. However, let's first look at `createImage` and its support methods. Most of the methods are short so it would be possible to put all the code in one method. However, by having several methods it makes it easier to modify and tailor. In this section we show the methods that are specific to VisualWorks 2.0; methods common to both 2.0 and 2.5 are shown later. Two things to note are that we run MyStripper after filing in the application since we need the compiler when filing in, and we install the application emergency handler at the end in case something goes wrong and we want a regular Notifier raised.

```
ImageCreator>>createImage
   ObjectMemory globalGarbageCollect.
   self myRemoveSelfAsDependent.
   self myFileInApplication.
   self myRunStripper.
   self myOpenApplication.
   self myCloseLauncher.
   self myInstallEmergencyHandler.
   self mySaveImageAndQuit

ImageCreator>>myRunStripper
   "Run the stripper to remove any unnecessary system classes."
   MyStripper new
      doRemoveOfSubsystems;
      stripSystem.

ImageCreator>>myCloseLauncher
   "Close any open Launchers "
   self myLauncherClass allInstances do:
      [:each | each builder window controller closeAndUnschedule]

ImageCreator>>mySaveImageAndQuit
   "Do a final garbage collect then save the image and quit.
    Saving as a snapshot means that no change file is created.
   The loadPolicy parameter specifies that objects in old space are
loaded in perm space."
   ObjectMemory globalGarbageCollect.
   ObjectMemory
      snapshotAs: self myImageName
      thenQuit: true
      withLoadPolicy: 1
```

Besides the methods that do the real work, we have some methods in the `names` protocol. These are the methods that should be modified for different applications because they provide the image name, the main

application class, etc. As mentioned above, only the methods that are specific to VisualWorks 2.0 are shown here; the common methods are shown later.

```
ImageCreator>>myImageName
   "Return the name of the image to save"
   ^'myApp' copy

ImageCreator>>myLauncherClass
   "We may be using a subclass of VisualLauncher as our Launcher"
   ^VisualLauncher
```

There are several changes you must make to the stripper code for it work in this environment. So that the stripping can run automatically, without human intervention, you'll need to override `doRemoveOfSubsystems` and `stripSystem` in MyStripper to remove the dialogs. Depending on which subsystems you want removed, you should set the appropriate options to have a value of *true* in `initialize`. For example, we want to also remove the database tools, the printing facility, and extra classes that we specified.

```
MyStripper>>initialize
   super initialize.
   doDbtools := true asValue.
   doPrinting := true asValue.
   doExtras := true asValue.
```

If there are classes that you know you want to strip, add them to `MyStripper class>>initExtras`, making sure you reinitialize the array. Alternatively, you could strip the basic image down before creating the `makeIm` image, keeping just enough to be able to file in the application code and save the image. You'll still need to strip the compiler later, but creating the application image would be faster if you start from a basic image that has already been stripped of a lot of classes.

If you want more than a VisualLauncher open in the builder image, you can add a message send of `myCloseOtherWindows` before filing in the application code.

```
ImageCreator>>myCloseOtherWindows
   "Close any open windows that aren't Launchers"
   ScheduledControllers scheduledControllers
      do: [:each | each model class ~~ self myLauncherClass
         ifTrue: [each closeAndUnschedule]]
```

VisualWorks 2.5

In VisualWorks 2.5 you need to first load in the ImageMaker code. You then run the ImageMaker (`ImageMaker open`) and select the removal options you want. To remove other classes in addition to the subsystems specified, select Set Additional Removals from the Classes menu, and select the Remove Additional Classes button. Once you have made all your choices, select File Out Choices from the File menu. We'll use the choices file later in one of the ImageCreator methods.

As before, we'll start by looking at `createImage` and its support methods, showing the methods specific to VisualWorks 2.5 here, with methods common to both 2.0 and 2.5 shown later.

```
ImageCreator>>createImage
   self myRemoveSelfAsDependent.
   self myInitializeImageMaker.
   self myFileInChoices.
```

```
   self myFileInApplication.
   self myOpenApplication.
   self myInstallEmergencyHandler.
   self myRunImageMaker

ImageCreator>>myInitializeImageMaker
   "Create a new ImageMaker and make it the current one"
   MyImageMaker current: MyImageMaker new.

ImageCreator>>myFileInChoices
   "File in the ImageMaker choices that we previously saved"
   self myChoicesFile fileIn

ImageCreator>>myRunImageMaker
   "Remove unneeded classes and save the new image"
   MyImageMaker current makeDeploymentImage

ImageCreator>>myChoicesFile
   "Return the filename that contains the ImageMaker choices"
   ^'choices.txt' asFilename
```

We've seen references to MyImageMaker above, so we now need to create it as a subclass of ImageMaker. In MyImageMaker we override a few methods to prevent dialogs coming up and to specify file paths. Override `getZoneDirectory` to remove the Dialogs and to set the directory to `self myBaseDirectory`. Override `makeImageFile` to remove the Dialogs and to set the imagePrefix to `self myImageName`. Override `refreshView` to be an empty method, and `warnUser` to return *true*.

```
MyImagerMaker>>myBaseDirectory
   "The directory where the utils subdirectory resides"
   ^'c:\visual' copy

MyImagerMaker>>myImageName
   ^'myApp' copy
```

In VisualWorks 2.5 there is one additional step that needs to be taken. The first time your new application image is loaded, VisualWorks does some operations to optimize memory use and increase performance, then saves the image again. So if you are creating an image as part of an automatic process, your scripts should create the new application image, *and then start up the new application image*. The code above will automatically save the optimized application image then exit.

Common to VisualWorks 2.0 and 2.5

The following methods are common to both VisualWorks 2.0 and 2.5.

```
ImageCreator>>myRemoveSelfAsDependent
   "Remove ourself as a dependent of ObjectMemory.
   We don't want this code to run every time the application is
started."
   (ObjectMemory dependents includes: self class)
      ifTrue: [ObjectMemory removeDependent: self class]
```

In `myOpenApplication` we assume we can send `open` to the application class. If you need another message sent, you could define a new method `myStartMessage` which returned a symbol. You would then start the application by `self myApplication perform: self myStartMessage`.

```
ImageCreator>>myOpenApplication
   "Open the application. We need to make sure that the user is not
prompted for
   the window position. When you accept this method, it will ask you
to declare your
   application class. Make it Undeclared. "
   | prompt |
   prompt := ScheduledWindow promptForOpen.
   ScheduledWindow promptForOpen: false.
   self myApplication open.
   ScheduledWindow promptForOpen: prompt

ImageCreator>>myFileInApplication
   "File in the application code."
   self myInstallFile fileIn.

ImageCreator>>myInstallEmergencyHandler
   "Install the application Emergency Handler."
   Exception emergencyHandler:
      [:ex :context | self handleException: ex context: context]

ImageCreator>>myInstallFile
   "Return the filename that files in the application code"
   ^'install.st' asFilename

ImageCreator>>myApplication
   "Return the class that starts up the application"
   ^MyApplication
```

In the method that sets up the new emergency handler, you will want to put something more sensible than the code shown here, but what you put will depend on how your application handles errors.

```
ImageCreator>> handleException: ex context: aContext
   "Do some error handling then quit"
   Dialog warn: 'Error: ' , ex errorString , ' in ' , aContext
printString , ' Quitting application'.
   ObjectMemory quit
```

We now have a mechanism for creating new images automatically, with little or no human intervention.

Saving the mkIm image

After filing in either the Stripper or ImageMaker followed by the ImageCreator, we need to save the image under a name such as `mkIm`. This is the image that will create our deployment image using all the code we wrote above. The obvious way to save `mkIm` is to use the File menu to either save the image or exit VisualWorks and save. However, this will create a change file, which will grow every time `mkIm` files in the application code. By adding two methods to the class side of ImageCreator we can save the image without a change file.

```
ImageCreator class>>saveImage
   "Save the image that will be used to create the production image."
   "self saveImage"
   ObjectMemory globalGarbageCollect.
   ObjectMemory
      snapshotAs: self myImageName
      thenQuit: true
      withLoadPolicy: 1
```

```
ImageCreator class>>myImageName
   "Return the name of the image that will be used to create the
deployment image"
   ^'mkIm' copy
```

Rather than saving the image from the file menu, you can now evaluate `ImageCreator saveImage`, which will save the image with no change file, then quit. An alternative mechanism is to save the image with the name *snapshot*. VisualWorks treats this name specially and does not create a change file. After creating the image, you would then rename *snapshot.im* to *mkIm.im*.

Runtime Packager

In Smalltalk method parameters are not typed, and polymorphism allows many classes to define the same method name. It is therefore very difficult to statically tell if classes are referenced and methods invoked. The consequence is that the stripping process can be rather hit and miss as you try to figure out what you can strip from the image. Runtime Packager™, from Advanced Boolean Concepts, provides additional capabilities that make it easier to determine what you can strip and what your application needs. It allows you to specify which classes and methods you want included, which ones you want removed, and which you are not sure about. It will then use both static and dynamic analysis to check your application to determine which classes and methods are safe to remove.

Runtime Packager will statically check all the message sends, starting at the message send that starts your application and following the message send tree until there are no further references. It also allows you to run the application in a special test mode where it records all the classes and methods that were previously marked for deletion but which are referenced in the running application. It automatically returns them to the image and continues running the tests. Once all the static and dynamic analysis is done, it creates a stripped image.

At the time of writing, Runtime Packager is available for $495 for a single user, and $1,200 for a site license. For more information, contact:

Advanced Boolean Concepts, Ltd.
3704 Cameron Mills Road
Alexandria, VA 22305, USA
Phone: (703) 548-5073
Fax: (703) 548-6053
Email: advbool@advbool.com
Web page: http://www.advbool.com/advbool/