1 Generating: Programming in Groups

Your code for exercises might be growing large enough that you may want to work with someone else. How do you work together? How do you merge code? How do you keep track of your changes as opposed to someone else's? The issue gets even more complicated when your changes appear in multiple classes. How do you file out all of your changes in multiple classes to give to someone else?

Smalltalk programmers have always worked together in collaborative groups, so the support for programming in groups in Squeak has its roots in the earliest Smalltalk-80. The basic two ideas are *projects* and *change sets*.

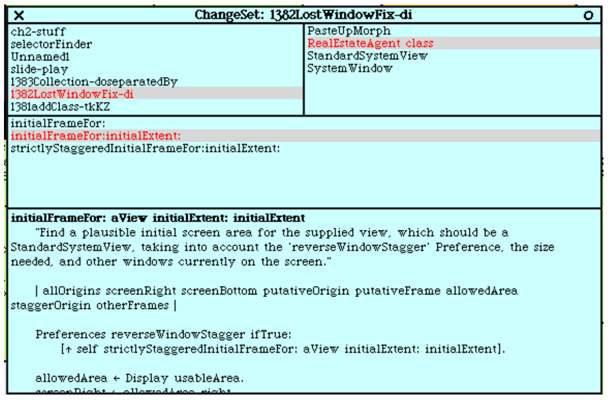
- A project stores the state of a complete Squeak desktop, including all the windows (and in Morphic, all morphic objects), as well as the currently active change set. When you change projects, whether by entering or exiting, all the global state is saved into the project being exited, and loaded from the one being entered. We have already seen a project, including entering and leaving it. Whatever changes you make to classes while within a project gets associated with the change set for that project.
- A change set is a collection of changes to the system. Changes include class changes, method changes, class removals, and method removals. All of these changes impact the *whole* system, e.g., you can't have one version of a method in one change set and another version of the same method in another change set. But you can fileOut all of the changes of one change set at once. You can also copy changes between change sets, compare change sets, and generally manage change sets.

1.1 Creating Projects and Change Sets

You have already seen how to create a project, enter it, and exit it. You can do this from any project. Projects can be nested within projects, and you can jump from any project to another.

Change sets go along with projects, but they don't have to. You can create change sets, and declare that new changes go to the new change set, from any project whatsoever.

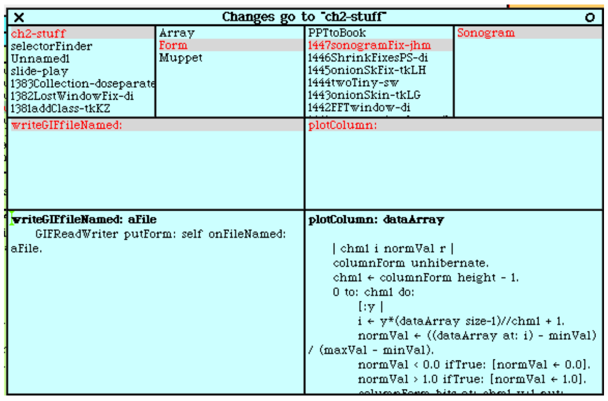
There are two tools available in the *Open…* menu for managing changes: *simple change sorter* and *dual change sorter*. The simple change sorter (Figure 1) lets you see the list of change sets in the current system (upper left hand corner), the list of classes in the selected change set (upper right), the list of methods changed in the selected class (center pane), and the actual code in the lower pane. From any change sorter, you can use the Yellow Button Menu on the change set pane to create a new change set, fileout all the changes of a change set (so that someone else can load in all the things you’ve changed in a given project), or even choose to make changes go to a different change set. Even within a single project, you can choose to make changes go to a different change set.



```
initialFrameFor: aView initialExtent: initialExtent
    "Find a plausible initial screen area for the supplied view, which should be a
StandardSystemView, taking into account the 'reverseWindowStagger' Preference, the size
needed, and other windows currently on the screen."

    | allOrigins screenRight screenBottom putativeOrigin putativeFrame allowedArea
staggerOrigin otherFrames |

    Preferences reverseWindowStagger ifTrue:
        [↑ self strictlyStaggeredInitialFrameFor: aView initialExtent: initialExtent].

    allowedArea ← Display usableArea.
```

Figure 1: Simple Change Sorter

The dual change sorter (Figure 2) allows you to look at two change sets at once. With two change sets open at once, you can copy changes from one to the other, submerge one change set into another, or even subtract all the changes from one change set that are also in the other change set. By manipulating change sets, you can find any particular changes that you need to include in what you give to your collaborators.



```
writeGIFfileNamed: aFile
    GIFReadWriter putForm: self onFileNamed:
aFile.
```

```
plotColumn: dataArray

    | chm1 i normVal r |
    columnForm unhibernate.
    chm1 ← columnForm height - 1.
    0 to: chm1 do:
        [:y |
        i ← y*(dataArray size-1)//chm1 + 1.
        normVal ← ((dataArray at: i) - minVal)
/ (maxVal - minVal).
        normVal < 0.0 ifTrue: [normVal ← 0.0].
        normVal > 1.0 ifTrue: [normVal ← 1.0].
```

Figure 2: Dual Change Sorter

Change set files are a little different than the “.st” files from filing out classes or categories. When a change set is filed out, the filename ends with “.cs” rather than the “.st” from filing out the class or method. A changeset fileOut contains the name of the change set and the date and time of the file out. The time stamp makes it easier to track versions of changes.

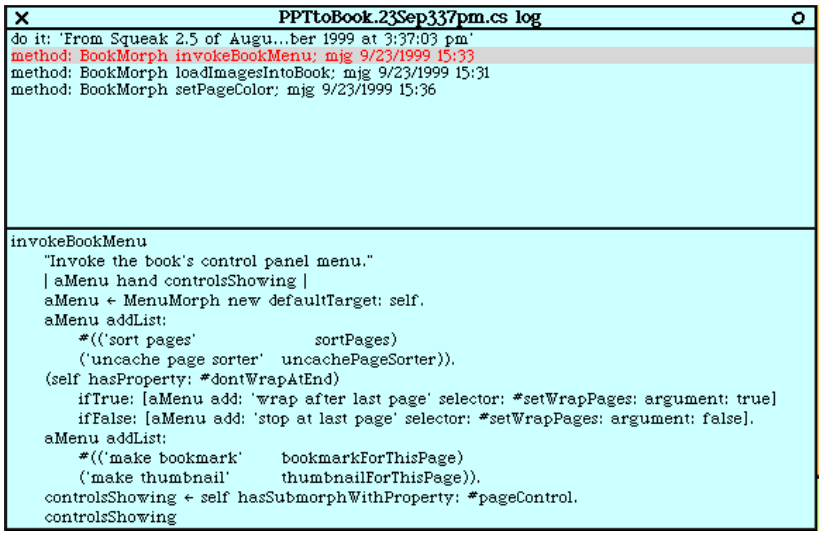
It’s also possible to attach a preamble or postscript to a change set. The preamble appears at the top of a change set, and a postscript appears at the end. In a preamble, you can put in a comment describing the change set, or even provide some set-up code, e.g., check that a prerequisite class is defined in the image before the change set is loaded in. In a postscript, you can clean up things, or perhaps start some of the newly loaded code.

1.2 *Working with Someone Else's Changes*

When someone else gives you a change set, you could simply file it in via the file list. But you might want to check it first, to see how it differs from what you already have (e.g., beware of someone overwriting your methods!) and even just to see what it includes.

From the file list, you have several options from the Yellow Button Menu when a code file (.cs or .st) is selected:

- You can *File into a new change set*. This option puts all of the new changes into one change set that you can easily inspect using a change sorter.
- You can *Browse* changes (Figure 3). This option puts up a window where you can inspect each separate change that the change set is going to do to your system if you file it in. You can choose to file in any one or any set of the changes in the change browser. In addition, you can use the Yellow Button Menu to select just those that are already in your system (conflicts), or those that are not. (Notice that you can see the initials of the author, as well as the time and date, for each change.)

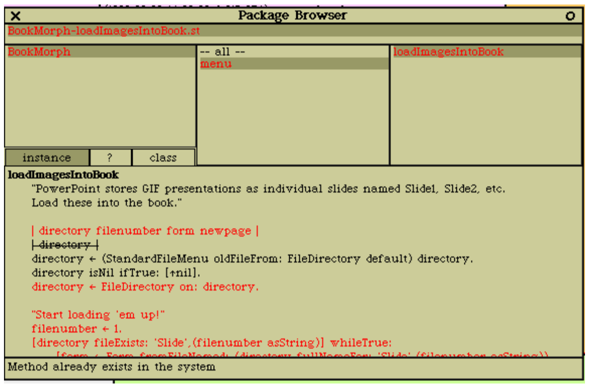


```
do it: 'From Squeak 2.5 of Augu...ber 1999 at 3:37:03 pm'
method: BookMorph invokeBookMenu; mjg 9/23/1999 15:33
method: BookMorph loadImagesIntoBook; mjg 9/23/1999 15:31
method: BookMorph setPageColor; mjg 9/23/1999 15:36
```

```
invokeBookMenu
    "Invoke the book's control panel menu."
    | aMenu hand controlsShowing |
    aMenu ← MenuMorph new defaultTarget: self.
    aMenu addList:
        #(('sort pages'              sortPages)
        ('uncache page sorter'   uncachePageSorter)).
    (self hasProperty: #dontWrapAtEnd)
        ifTrue: [aMenu add: 'wrap after last page' selector: #setWrapPages: argument: true]
        ifFalse: [aMenu add: 'stop at last page' selector: #setWrapPages: argument: false].
    aMenu addList:
        #(('make bookmark'      bookmarkForThisPage)
        ('make thumbnail'       thumbnailForThisPage)).
    controlsShowing ← self hasSubmorphWithProperty: #pageControl.
    controlsShowing
```

Figure 3: Browsing changes from a change set file

- Perhaps the most powerful option is the ability to *Browse code* (Figure 4). The Package Browser that opens allows you to walk through the code that's in the selected file just as if it were already in your System Browser. As you select each method, it tells you if the method is already in your image or not. If it is, strikeouts and color coding shows you how the new method differs from the method that's currently in your system. As from the changes browser, the package browser lets you file in any particular methods or classes that you choose.



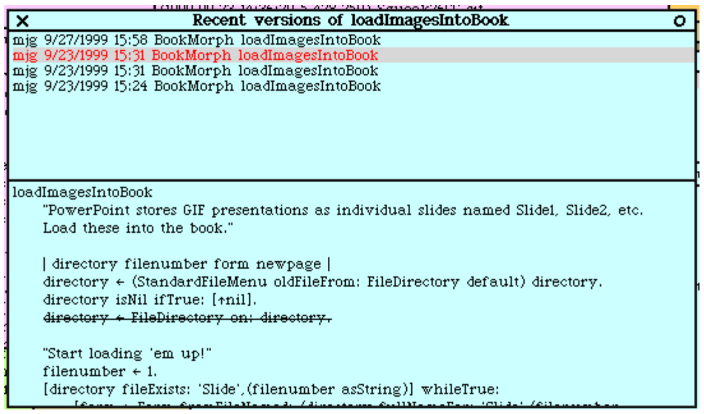
```
loadImagesIntoBook
    "PowerPoint stores GIF presentations as individual slides named Slide1, Slide2, etc.
    Load these into the book."

    | directory filenumber form newpage |
    ~~| directory |~~
    directory ← (StandardFileMenu oldFileFrom: FileDirectory default) directory.
    directory isNil ifTrue: [↑nil].
    directory ← FileDirectory on: directory.

    "Start loading 'em up!"
    filenumber ← 1.
    [directory fileExists: 'Slide',(filenumber asString)] whileTrue:
```

```
Method already exists in the system
```

Figure 4: Browsing Code from a Code File

1.3 *Recovering from Someone Else's Changes*

You have now filed in someone else's code, but missed one critical method, so now all of your code is broken. (Or maybe, you just made a really bad change, and realize that you hadn't filed out the previous, almost-working version.) You really want to turn back the clock to an earlier version of the method.



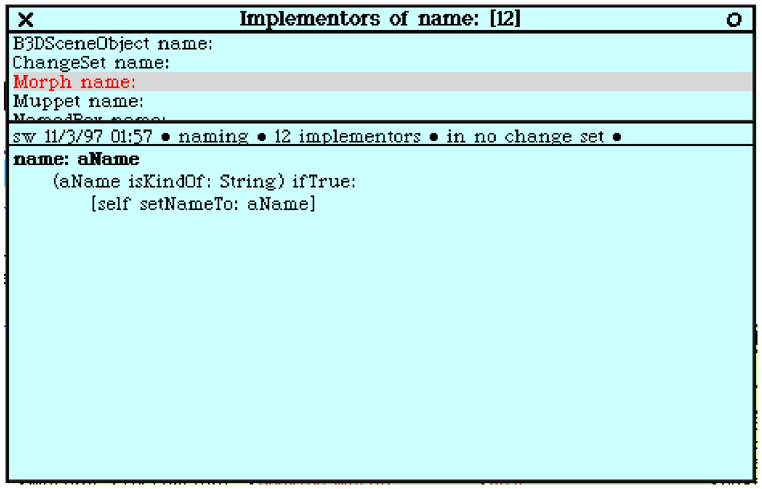
```
mjg 9/27/1999 15:58 BookMorph loadImagesIntoBook
mjg 9/23/1999 15:31 BookMorph loadImagesIntoBook
mjg 9/23/1999 15:31 BookMorph loadImagesIntoBook
mjg 9/23/1999 15:24 BookMorph loadImagesIntoBook
```

```
loadImagesIntoBook
    "PowerPoint stores GIF presentations as individual slides named Slide1, Slide2, etc.
    Load these into the book."

    | directory filenumber form newpage |
    directory ← (StandardFileMenu oldFileFrom: FileDirectory default) directory.
    directory isNil ifTrue: [↑nil].
    ~~directory ← FileDirectory on: directory.~~

    "Start loading 'em up!"
    filenumber ← 1.
    [directory fileExists: 'Slide',(filenumber asString)] whileTrue:
```

Figure 5: Recent versions of a method

Fortunately, the changes file really does capture *all* changes in the system, which includes all those previous versions of a given method. From any method list (e.g., an implementors list, a System Browser, wherever), you have a Yellow Button Menu item *Versions…* The recent versions browser shows you all the versions of the same method in your changes file (Figure 5). Selecting an older version shows you with color and strikeouts how the selected version differs from your current version. You can copy anything you want out of these old versions and paste them into your browser to re-accept them.

If you open up the *Preferences* (under the *Help* menu item in the Desktop or World menus), you can choose to *useAnnotationPanes.* In several of the browsers (especially Senders and Implementors), when annotation panes are selected, you are shown a pane between the list of methods and the code that gives useful information about the selected method. This information typically includes the initials of the author, the timestamp, the number of implementors of this method, the number of senders, and which change set the method came from (Figure 6). What gets displayed in the annotation pane is actually configurable with an easy drag-and-drop interface, which is brought up when you DoIt on **Preferences editAnnotations**,



```
sw 11/3/97 01:57 • naming • 12 implementors • in no change set •
name: aName
    (aName isKindOf: String) ifTrue:
        [self setNameTo: aName]
```

Figure 6: Implementors browser with Annotation Pane