Xcode Tools Sensei Changes

This document lists the changes I've made from the original version of *Xcode Tools Sensei* to the latest version. Most of the changes are due to updates in the Xcode Tools, but I also added material that needed to be in the book. People who bought the original version of the book will benefit the most from this document, but there is some useful information for other Mac OS X developers.

I've organized the changes by chapter and by section in each chapter. The section order reflects the order the sections appear in the book.

Chapter 1

New Project Templates

Xcode 2.2 added the following project types:

- The Definition Bundle project allows you to define new data types for Automator actions.
- An Automator action project that uses shell scripting.
- Audio unit effect projects let you create Core Audio audio units. Audio units are software components that work with audio data. Audio units let you create plug-ins for audio applications and create your own sound effects.
- The Java signed applet project lets you create applets with a digital signature.
- Java Web Start Application project. Web Start is a technology that lets users easily download and launch Java applications from the Internet.

Groups and Files List

Control-clicking an item in the Groups and Files list opens a contextual menu. The contents of the contextual menu depend on the item you click, but the tasks you can perform from the contextual menu include adding files to a project, adding targets to a project, adding build phases to a target, compiling a single file, and building a target.

Adding Files You've Already Created

There can be a fifth Reference Type option for files you add to a project: by source tree. A source tree is a root path to place files that you want multiple people to work on. Source trees allow your project's files to remain independent of the project folder. The point of using source trees is to allow multiple people to work on a project. You can transfer the project to other people's computers without messing up the project's file references. To create a source tree:

1) Choose Xcode > Preferences to open Xcode's preferences panel.
2) Select Source Trees from the preferences panel.
3) Click the + button to add a source tree.
4) Click the OK button.

There are three pieces of information you must supply to add a source tree.

- Setting Name is the name of the tree. Everyone who wants to use a source tree must give it the same setting name on their Macs.
- Display Name is the name Xcode shows for the source tree.
- Path is the location of the source tree on your hard disk.

Each person who wants to use a source tree can store it wherever they want on their hard disk. As long as everyone uses the same name for the source tree, everyone can use it. If you want other people to work on your projects, copy the files from the location of the source tree on your Mac to the location of the source tree on their Macs.

Code Completion

The automatically suggest checkbox in the Code Sense preferences panel is now a pop-up menu. Xcode can automatically open the completion list in two ways: open the list all the time or open the list when accessing your classes' members.

Adding Targets

Xcode 2.2 added object file and resource file targets for Carbon programs and object file targets for Cocoa programs.

Target Rules Panel

`gcc` 4.0.1 fixed a bug with the C and Objective C compilers. C and Objective C programs compiled with `gcc` 4.0.1 can run on Mac OS X 10.2 and earlier versions of Mac OS X.

Linking Collection

Xcode 2.2 added several interesting linker build settings. Activating the Only Link In Essential Symbols build setting tells Xcode to copy only enough information for the debugger to do the work of the linker during debugging. This option speeds linking, but your program needs all of the object files to be available to be debuggable.

The Display Mangled Names build setting tells Xcode to display mangled names for C++ symbols. Because multiple C++ functions can have the same name, the compiler mangles the function names to give each function a unique name. If you get link errors in your C++ program, activating the Display Mangled Names build setting can help you locate the source of the error.

Activating the Verbose Undefined Symbols setting tells Xcode to display additional information when an undefined symbol link error occurs. The additional information includes the file where the compiler found the symbol and tells you if the symbol is defined or referenced in the file.

Code Generation Build Settings Collection

Xcode 2.2 added one important build setting to the Code Generation collection. The Separate PCH Symbols build setting tells Xcode to create a separate file to store the debug symbols of a precompiled header. If you have this setting turned off, the debug symbols wind up in the binary file. The Separate PCH Symbols build setting gives you smaller binary files and faster link times.

To use the Separate PCH Symbols build setting, ZeroLink must be disabled. The Level of Debug Symbols build setting, which is also in the Code Generation collection, must be set to All Symbols.

ZeroLink

Xcode 2.2 lets you turn off ZeroLink globally so you don't have to turn it off for every project you create. Choose Build > Allow ZeroLink to turn it off, and choose Build > Allow ZeroLink to allow it in your projects. If you allow ZeroLink, the ZeroLink build setting determines whether or not Xcode builds your project with ZeroLink.

Distributed Builds

When you select the Distribute builds to checkbox, the Bonjour build set checkbox gets selected too. By selecting the Bonjour checkbox, Xcode automatically lists the computers you're connected to. For each connected computer, Xcode tells you the computer's name, the operating system it's running, the compilers installed on that computer, and its status. You can distribute builds only to computers with Sharing status. Click the Apply button to have your builds distributed to every computer with Sharing status.

Having your builds distributed to every available computer on the network may not be what you want. You can create custom build sets that contain the computers you want to distribute builds to. Click the Duplicate button to duplicate an existing build set or click the + button to add a build set. Use the + and minus buttons under the host list to customize your build set. Select your build set from the build set list to distribute builds to the computers you want.

Xcode 2.2 forces you to click the lock at the bottom of the distributed builds preferences panel to share your computer.

Building Universal Binaries for Older Versions of Mac OS X

Mac OS X 10.4 is the earliest version of Mac OS X that will run on Intel Macs, but you would like your universal binary to run on PowerPC Macs running older versions of OS X. To support earlier versions of Mac OS X on PowerPC, you must add the build setting `MACOSX_DEPLOYMENT_TARGET_ppc`. This setting specifies the earliest version of Mac OS X that can run the PowerPC version of your universal binary.

1) Open the build setting information panel.
2) Choose Customized Settings from the Collection pop-up menu.
3) Click the + button to create a new build setting.
4) Give the setting the name `MACOSX_DEPLOYMENT_TARGET_ppc`.
5) Give the setting the value of the operating system version you want as your deployment target. To run on Mac OS X 10.2 and later, use the value 10.2.

C++ programs require a little more work. Programs for Intel Macs must be built with `gcc` 4, but C++ programs built with `gcc` 4 run only on Mac OS X 10.3.9 and later. To support earlier versions of Mac OS X, you must build the PowerPC version of your program with `gcc` 3.3.

1) Open the build setting information panel.
2) Choose Customized Settings from the Collection pop-up menu.
3) Click the + button to create a new build setting.
4) Give the setting the name `GCC_VERSION_ppc`.
5) Give the setting the value 3.3.

Chapter 2

Setting Environment Variables for Debugging

Xcode 2.2 added three environment variables to the malloc library. Setting the `MallocCheckHeapSleep` environment variable to 1 tells your program to go to sleep when a heap corruption occurs. Setting the `MallocCheckHeapAbort` environment variable to 1 tells your program to abort when a heap corruption occurs. Setting the `MallocBadFreeAbort` environment variable to 1 tells your program when it illegally frees memory.

Launching the Debugger

Xcode 2.2 lets you attach running programs to the debugger. Choose Debug > Attach > Program Name to start debugging a program that is already running.

Variable Viewer

To show variables in a separate window, select the variables in the debugger window and choose Debug > Variables View > View Variable in Window.

Custom Data Formatters

This section is new material. Custom data formatters let you customize what appears in the Value and Summary columns for each variable. Data formatters are especially useful for displaying the data structures you create for your programs. By using a data formatters, you can display the most important data structure information in the Summary column. For data formatters to work, they must be enabled in Xcode. Choose Debug > Variables View and make sure the Enable Data Formatters menu item has a check mark next to it.

To create a custom data formatter, double-click the Value or Summary column for a variable and enter a format string. Any literal text you enter will appear in the debugger window exactly as you type it, which makes the literal text good for labels. To refer to the variable itself, use the value `$VAR`. If the variable is a data structure, you can refer to the structure's individual members by placing the character % before and after the member name.

```
%MemberName%
```

If one of your data structure's members is another data structure, use the dot operator. Suppose you have a data structure with a member named area, which is of type `Rect`. You want to show the left edge in the format string. You would type the following format string:

```
%area.left%
```

Let's walk through a simple example of using data formatters. Suppose you have a data structure for 3D vectors named `Vector3D` with members `x`, `y`, and `z`. You would like the Summary column to show the `x`, `y`, and `z` components of the vector. Select the `Vector3D` variable in the debugger window and double-click the Summary column. Enter the following text in the Summary column:

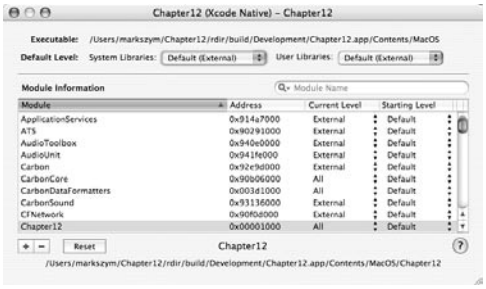
```
x=%x%, y=%y%, z=%z%
```

Now every `Vector3D` variable in your program will show the `x`, `y`, and `z` components in the Summary column. If `x` has a value of 1, `y` has a value of 3, and `z` has a value of 5, the Summary column shows the following output:

```
x=1, y=3, z=5
```

Viewing Shared Libraries

Xcode 2.2 lets you see which libraries have been loaded by your program. Choose Debug > Tools > Shared Libraries to open the Shared Libraries window, which you can see in the following figure:



At the top of the window are two pop-up menus that let you specify the default level of debugging symbols to show for system and user libraries. Most of the shared libraries in your programs are going to be system libraries. Only libraries that your project's targets produce are user libraries.

There are three levels of symbol showing. You can tell the debugger to show all symbols, no symbols, or external symbols, symbols declared external in the library's code. Showing external symbols is the default. The debugger loads all external symbols and loads other symbols when necessary.

Why so much concern over symbol levels? Can't you show all debugging symbols and be done with it? Showing debugging symbols is a balance between the ability to debug and speed of debugging. Mac OS X applications use a lot of shared libraries, and these libraries have lots of symbols. Loading every one of these symbols takes time and make debugging slower. Loading external symbols balances your need for information with your need for speed.

Below the pop-up menus is a list of libraries. The Shared Libraries window displays the following information for each library:

- The name of the library.
- The library's memory address. If there is no memory address, the library has not been loaded.
- The current level of debugging symbols to display for the library.
- The starting level of debugging symbols to display for the library.

Each library has a menu to set the current and starting level of debugging symbols. If you have some libraries where you want to see more debugging symbols than normal, you can set the starting level for those libraries. Setting the current level is useful if you want to see more debugging symbols at certain times when you're debugging.

Clicking the Reset button sets the starting level of debugging symbols back to the default value you set in the pop-up menus. Select a library and click the minus button to remove a library from the Shared Libraries window. Click the + button to add a library to the window.

Viewing Global Variables

In Xcode 2.2, clicking the disclosure triangle next to Globals in the variable viewer opens the global variables browser.

Viewing Registers

To view the contents of registers with Xcode 2.2, you must be viewing disassembled code. Choose Debug > Toggle Disassembly Display to see disassembled code in the debugger window.

Intel Macs have been released since the release of *Xcode Tools Sensei*. If you have an Intel Mac, you will see Intel registers instead of PowerPC registers. The table below contains a list of the Intel registers. The *IA-32 Intel Architecture Software Developer's Manual* contains detailed information about the Intel processor's registers. You can download the book from Intel's website.

Breakpoints

Clicking the gray arrow in the gutter disables the breakpoint you set. Drag the arrow off the gutter to remove the breakpoint permanently.

Viewing Memory

To view a variable's memory contents, select the variable in the debugger window and choose Debug > Variables View > View As Memory.

In Xcode 2.2 if you select the Auto Update checkbox, the memory browser updates its contents when stepping through code.

Intel Registers

Register	Description
EAX	Accumulator for arithmetic operations.
EBX	Pointer to data in the DS segment.
ECX	Counter for loops and string operations.
EDX	I/O pointer.
ESI	Source pointer for string operations. It can also be used as a pointer to data the DS register points to.
EDI	Destination pointer for string operations. It can also be used as a pointer to data the ES register points to.
EBP	Pointer to data on the stack.
ESP	Stack pointer.
CS	Code segment register. The segment registers help the CPU translate logical memory addresses to linear addresses. The CPU then translates the linear addresses to physical addresses.
DS	Data segment register.
SS	Stack segment register.
ES	Data segment register.
FS	Data segment register.
GS	Data segment register.
EFLAGS	Contains a group of status flags.
EIP	Instruction pointer. It contains the location of the next instruction to be executed.
MM0-MM7	Registers used with Intel's MMX technology. MMX registers are 64 bits.
XMM0-XMM7	Registers used with Streaming SIMD Extensions (SSE). SSE is similar to AltiVec on PowerPC processors. SSE registers are 128 bits.
MXSCR	Status and control register for SSE operations.

Chapter 3

Creating Matrices of Controls

Interface Builder lets you create matrices of controls for Cocoa programs.

1) Add the control to the window.
2) Select the control.
3) Option-drag to create the matrix.

To change the spacing between cells in the matrix, select the matrix and command-drag.

If a control cannot be part of a matrix, option-dragging creates a copy of the control instead of a matrix of that control.

Chapter 6

Shark

Shark 4.3 added data cache miss profiles for the Pentium M architecture.

If you're running Shark on an Intel Mac, you have two architecture choices: Pentium 4 and Pentium M. Use the Pentium M architecture. The Core Duo and Core Solo processors that power Intel Macs inherit from the Pentium M architecture. The downside of the Pentium M for Shark is that it has only two performance monitor counters (PMC). Having two PMCs means you can count only two CPU, memory, and operating system events.

Shark 4.3 moved the Show Advanced Settings menu item from the View menu to the File menu. To see the performance events in Shark's output, you must open the advanced settings drawer by choosing File > Show Advanced Settings. In the Performance Count Data Mining section, select the checkbox in the left column (the eye column) next to an event to see that event in Shark's output.

Chapter 9

fs_usage

The latest version of `fs_usage` does not report cache hits by default. To see cache hits, you must run `fs_usage` with the `–f` option and the cachehit mode.

```
fs_usage –f cachehit AppTitle
```

vmmap

`vmmap` has undergone a lot of changes. It has several new options. Submap information does not appear in the default `vmmap` report. To view information about submaps, you must run `vmmap` with the `–submap` option.

Non-Writable Memory Regions

The latest version of `vmmap` has two additional memory region purposes: shared memory and mapped file. Shared memory regions are shared by multiple applications. System libraries like Cocoa and Carbon are the major source of shared memory regions.

Mapped file regions are memory mapped files, where the operating system maps part of a file into a program's virtual address space. Memory mapped files let multiple programs read from and write to the same file.

Writable Memory Regions

The latest version of `vmmap` has the following new region purposes:

- `MALLOC_TINY` regions consist of small memory allocations, allocations 512 bytes and smaller.
- `MALLOC_LARGE` regions consist of large memory allocations. Apple's documentation says large memory allocations are at least 4 pages in size, or at least 16KB. But I saw `MALLOC_LARGE` regions that were 4KB when I ran `vmmap`.
- `VM_ALLOCATE` regions are regions that were allocated with the `vm_allocate()` function. This function allocates virtual memory regions from which `malloc()` allocates memory.

`MALLOC_USED` regions are now called `MALLOC` regions. The `MALLOC_OTHER` and `VALLOC_USED` regions are not as common in the latest version of `vmmap`.

Summary Report

The summary report now breaks down the memory map by region type and tells you the total amount of virtual memory for each region type.

-w Option

The `–w` option displays wide output for each memory region. Wide output allows you to see a greater amount of additional data for each region.

-resident Option

When you run `vmmap` with the `–resident` option, it lists the virtual and resident size of each memory region. The resident size tells you how much of the region is in physical memory.

-pages Option

When you run `vmmap` with the `–pages` option, it lists the size of each memory region in pages instead of bytes. Memory pages on PowerPC Macs are 4KB. The default page size is also 4KB on Intel Macs, but pages can also be 2MB and 4 MB.

-interleaved Option

When you run `vmmap` with the `–interleaved` option, the report does not separate the memory regions into writable and non-writable regions. It lists the memory regions in order of their starting address, with the lowest memory regions appearing first.

-submap Option

The `—submap` option tells `vmmap` to print information about memory submaps.

-allSplitLibs Option

The `—allSplitLibs` option tells `vmmap` to print information about all shared system split libraries. The default behavior is to print information about only the libraries your program loads. Shared system split libraries are dynamic libraries that multiple programs share. Examples of these libraries are the Cocoa and Carbon libraries. The `—allSplitLibs` option generates a lot of unused split lib listings for non-writable regions.

heap

`heap` has a new option for Objective C programs. The `—guessNonObjects` option tells heap to search the memory of each object for pointers to non-objects, blocks of memory allocated by malloc. The `heap` report has one listing for each of these memory blocks. The block listings list the Objective C object, with the offset from the start of the object in brackets. The following listing:

```
NSCFArray[36]
```

References a block of memory that is located 36 entries from the start of the array.

Chapter 10

The latest version of `gcov` for `gcc` 4 requires you to build your program with the linker flag `—lgcov`. In Xcode add the `—lgcov` flag to the Other Linker Flags build setting, which is in the Linking collection.

The latest version of `gcov` creates two files for each source code file in your project. One file has the extension `.gcno`, and it replaces the `.bb` and `.bbg` files. The second file has the extension `.gcda`, and it replaces the `.da` file.

-a Option

The `—a` option tells `gcov` to write execution counts for each basic block in a line of code. A basic block is a sequence of instructions that executes in order with no chance of branching. One statement in C, C++, or Objective C consists of multiple assembly language instructions so one statement can have multiple basic blocks. The default `gcov` behavior is to report the execution count for the main block in a line of code.

-b Option

When you run `gcov` with the `—b` option, a listing precedes each function in `gcov's` report. The listing tells you the number of times your program called the function, the percentage the function returned, and the percentage of basic blocks that were executed.

-u Option

The –u option has been added to the latest version of `gcov`. It works with the –b option. The –u option tells `gcov` to include the branch probabilities of unconditional branches.

Chapter 12

I added a new section on the Applet Launcher program.

Applet Launcher

As its title suggests, Applet Launcher is a tool to launch Java applets. It serves two purposes. The first purpose is to test your applets without having to open an Internet browser. The second purpose is to test newer applets on Java virtual machine 1.4.2.

Launching an Applet

When you launch Applet Launcher, you see the launch window, shown in the figure below. If the applet is on your Mac, click the Open button to find the applet you want to launch. Click the Launch button to launch the applet. When you click the Launch button, Applet Launcher adds the applet to the History menu so you can easily relaunch it at a later date.

Keep in mind that applets require an HTML file to launch. If you supply your applet's jar file, Applet Launcher won't be able to launch the applet. Xcode Java applet projects include a file named `example1.html`. This file is what you must supply to Applet Launcher.

If the applet you want to launch is located on a website instead of your Mac, clicking the Open button won't work. You must type the applet's URL.

```
http://www.example.com/example1.html
```

After launching the applet, the File menu becomes the Applet menu. Use the Applet menu to stop, restart, reload, clone, and quit (close) the applet.

Getting Applet Information

The Applet menu also lets you retrieve information about the running applet. Choose Applet > Tag to see the applet's HTML tag. Choose Applet > Info to view information about the applet and its parameters. Choose Applet > Properties to set a HTTP proxy server for the applet. You shouldn't need to set a proxy server for applets that reside on your Mac.

Serializing Applets

Choose Applet > Save to serialize the applet. Serialization takes objects in a program and converts them into streams of data. These streams can be saved to a file and restored later.

Chapter 13

OpenGL Profiler

The latest version of OpenGL Profiler added two new windows to open from the View menu. The pixel format window lets you view the pixel format information for each OpenGL context your application uses. The messages window displays the log messages OpenGL Profiler generates as your program runs.