Building a Framework for iOS out of static libraries.

Based on knowledge, techniques, code, and algorithms from:
https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/FrameworkAnatomy.html
http://www.cocoanetics.com/2010/05/making-your-own-iphone-frameworks-in-xcode/
https://github.com/kstenerud/iOS-Universal-Framework
http://code.google.com/p/ios-static-framework/
http://www.gnu.org/software/bash/manual/bashref.html#Conditional-Constructs
http://www.davidpashley.com/articles/writing-robust-shell-scripts.html

use in conjunction with prepare_framework.sh and assemble_fatbin.sh
to create distributable frameworks containing static fat libraries

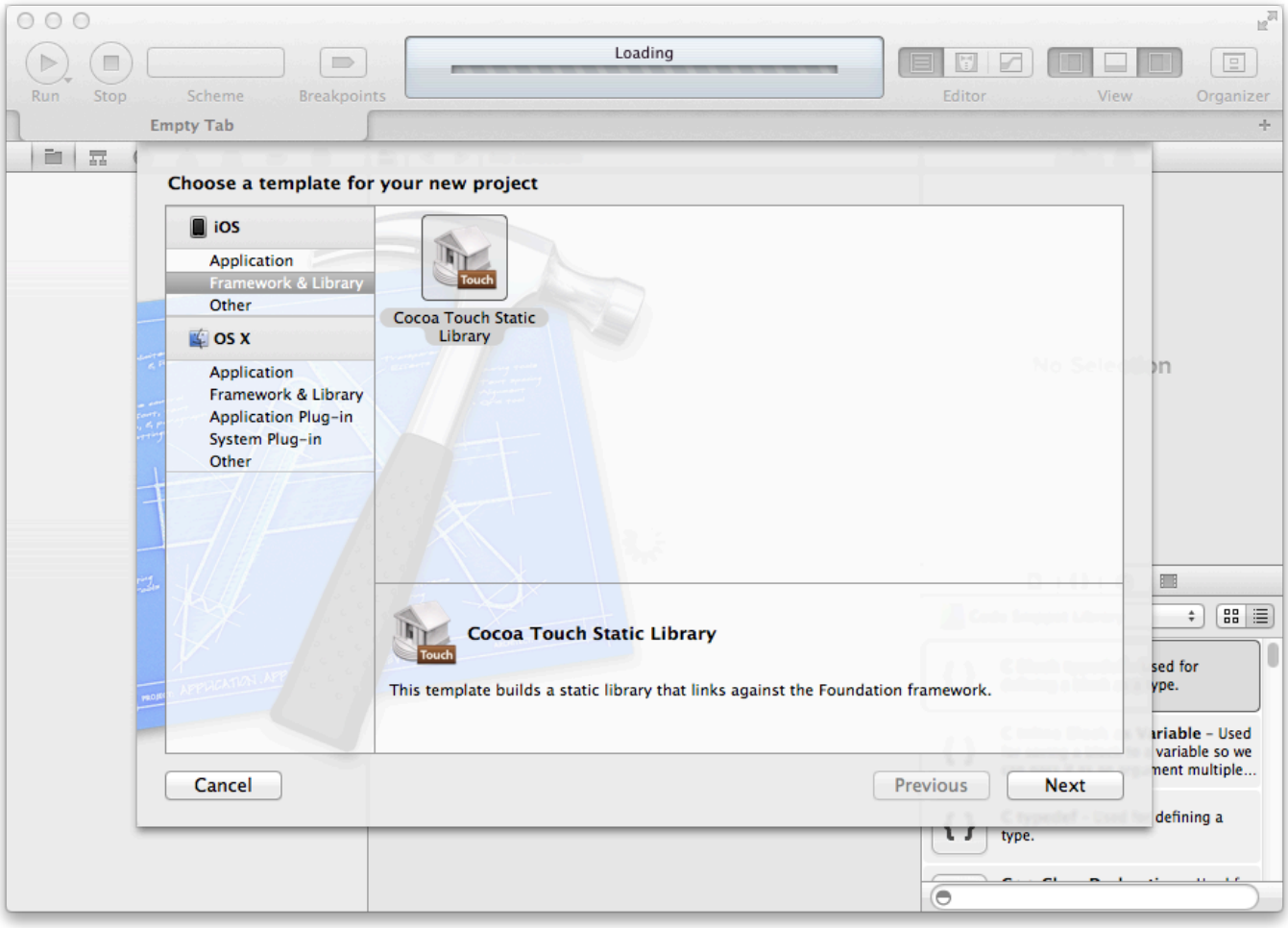
Framework Anatomy
It is common in Unix and linux dev environs to create a layer of indirection when building libraries such that there are directory trees that lead to specific versions of the libs and their headers and sources, and a 'base' or 'current' tree with symbolic links to one of these trees. This is often needed to support dependencies on back-level libs when compilingin-place for production systems.

Apple does a similar thing with Framework bundles using symbolic file links through its internal Versions folder, this supports Darwin's DyLib loading of back-level versions of the libs from an old app that might need it. Tedious details here:
https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/FrameworkAnatomy.html

It's hard to justify all this additional complexity when your framework actually contains libs that will be statically linked into your app bundle, but because XCode's link mechanisms expect to dereference these symbolic file links, all this fussing about with indirection seems to be needed.

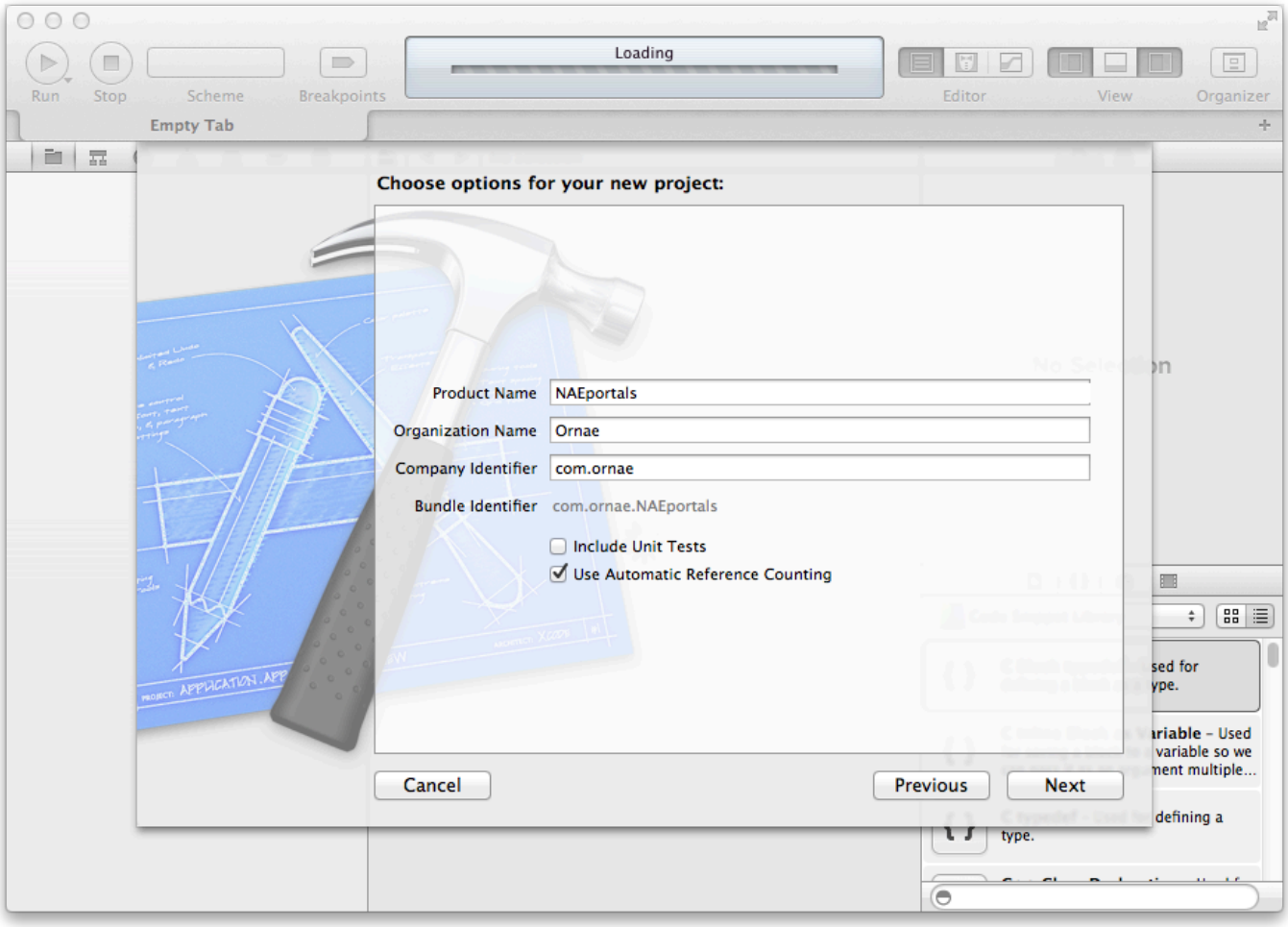
Building up the framework from scratch, assuming XCode 4.5.x, following the accompanying screen images:

1 - Start with Cocoa touch static lib template. This is the closest template to our goal that XCode 4.5.x ships with.



2 - Set product/project name. This will become our framework name, the primary header file included for the framework, our primary instantiated class if this is an ObjC lib, and may contain plain C interfaces as well.

Library names for ObjC applications should respect ObjC namespace conventions, with an understanding of the flat namespace of C, and be prefixed with an organization specific short prefix, like the NS prefix for the NextStep libraries, trying for symbols that are both unique and descriptive.



3 - Settings, Some Unique to Libs and Frameworks:

Project : PROJECT : Build Settings : Search :

a: iOS Deployment Target:

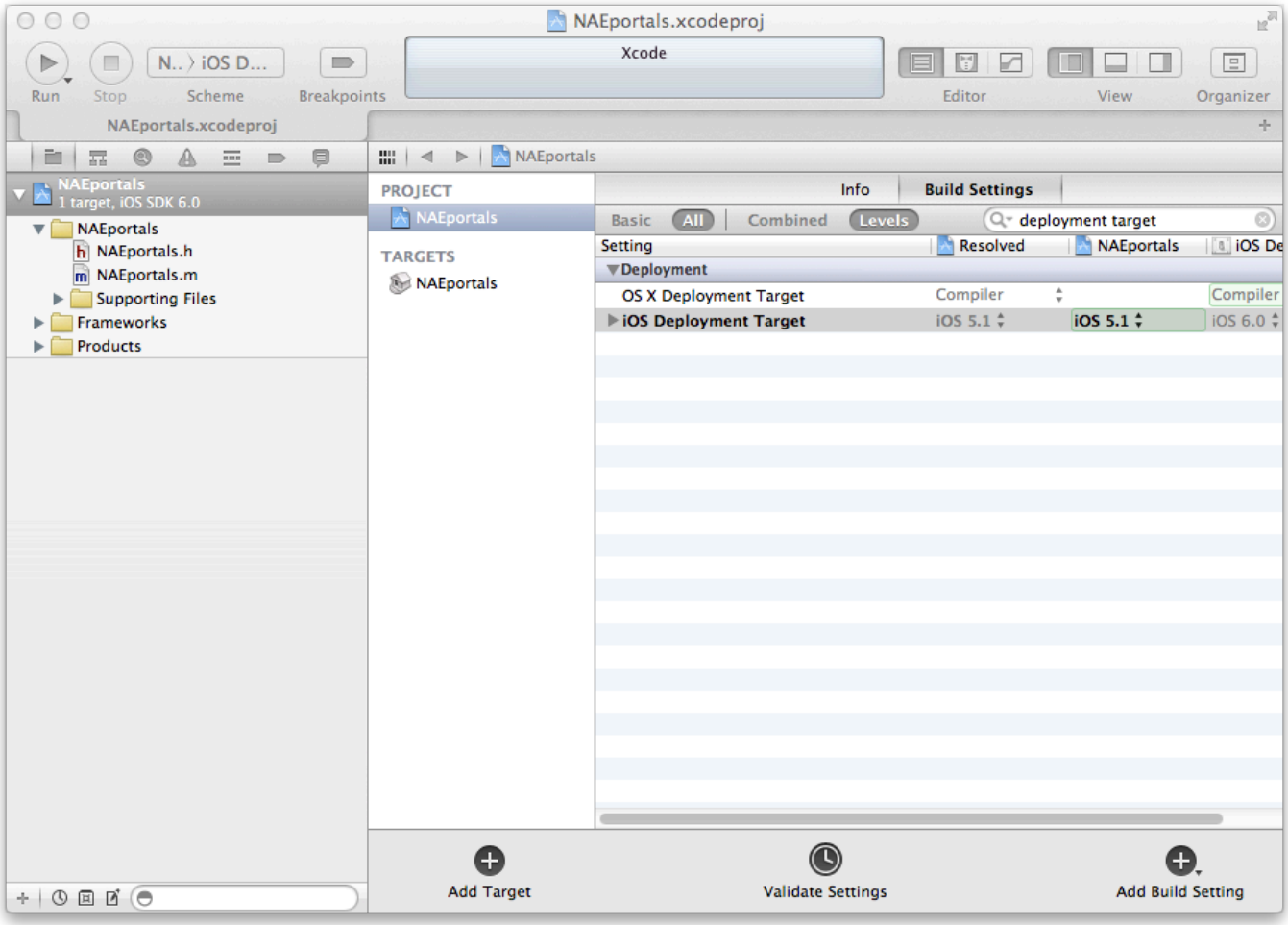
4.2.1 Supports back to 3g iPhones and 2nd gen iPods, but not avail on XCode 4.5

4.3 Pointless to support? Oldest supported on XCode 4.5

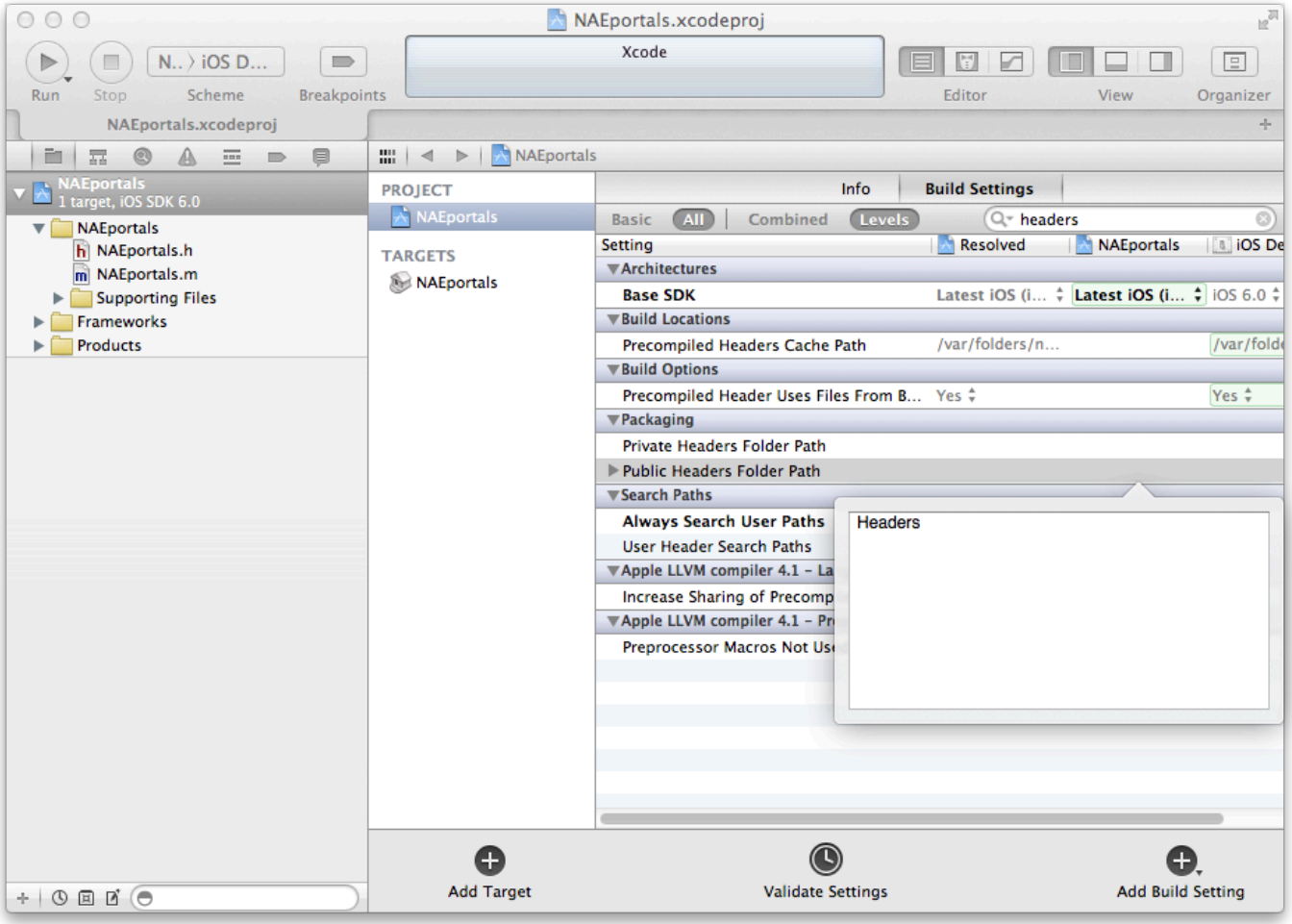
5.1 Supports back to 3rd gen iPods and 1st Gen iPads.

6.x 3GS iPhone, 4th gen iPod, 2nd gen iPad, all iPad mini

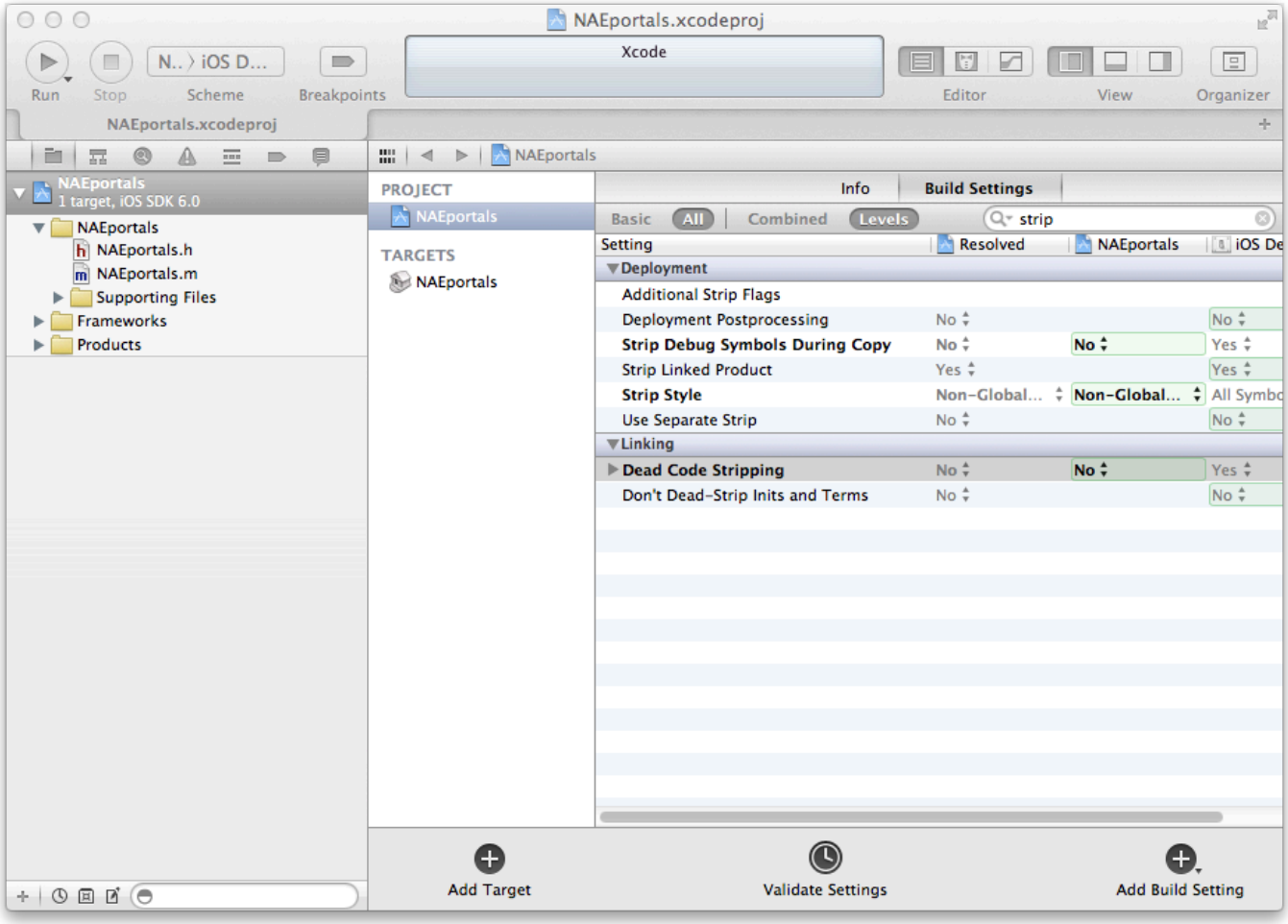
(per https://en.wikipedia.org/wiki/List_of_iOS_devices#Highest_Version_of_iOS_Supported



b: Public Headers Folder Path:
Headers



c: Strip Debug Symbols During Copy:
No
c: Strip Style:
Non-Global Symbols
c: Dead Code Stripping:
No

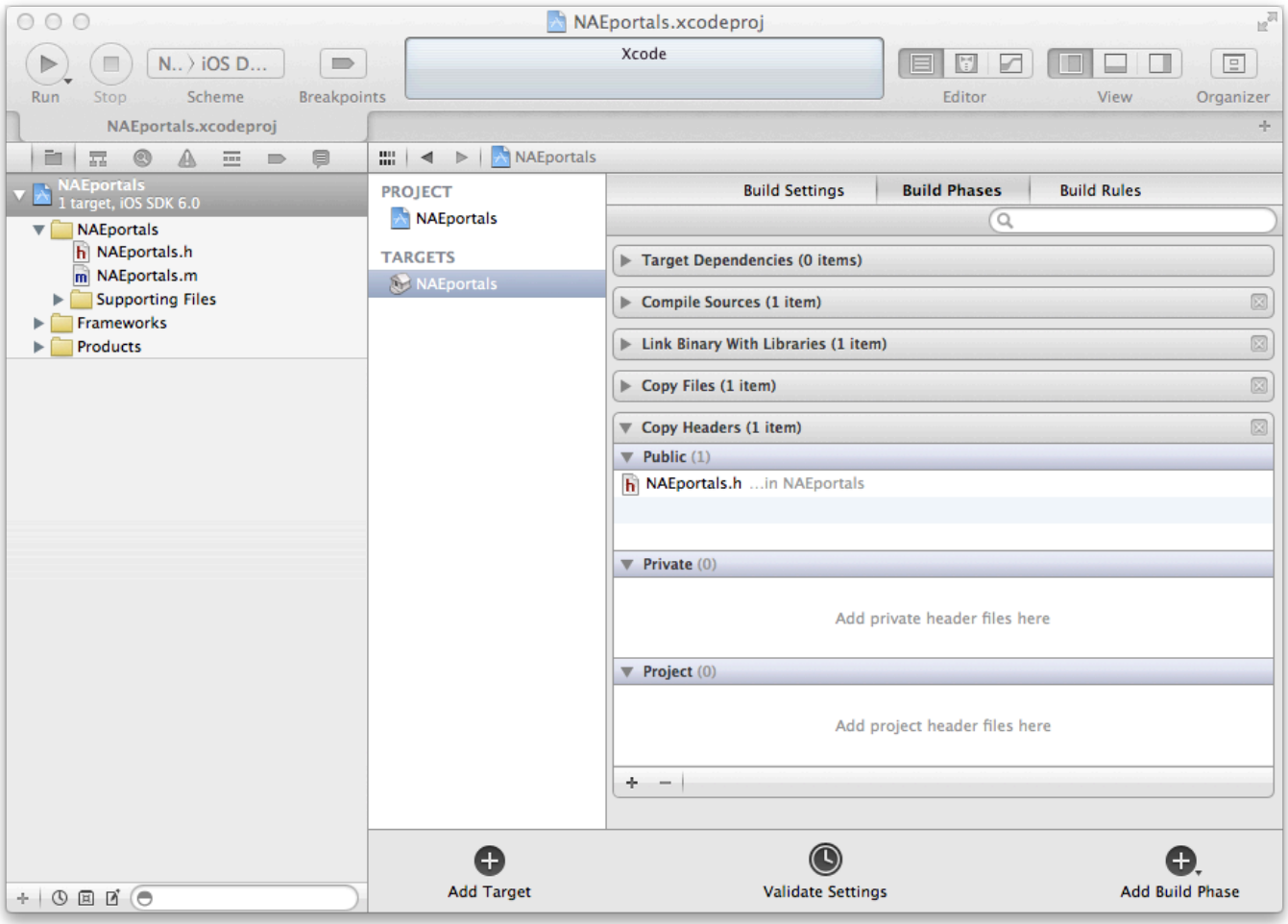


4 - Enable-Add the Copy Files-Headers Build Phase to your target:

Project : TARGETS : Build Phases ! + Add Build Phase : Add Copy Headers

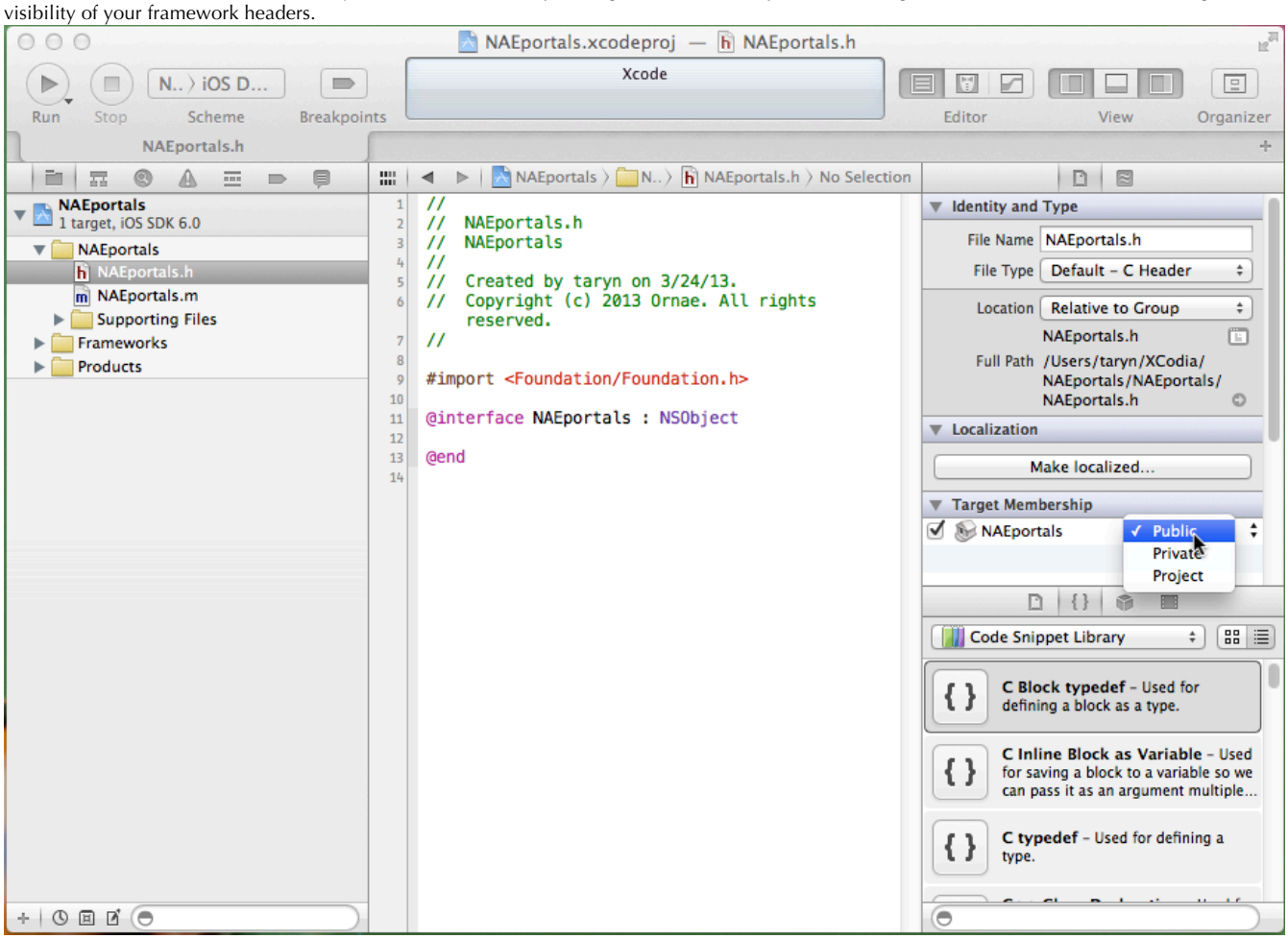
I've built these frameworks a handful of times now, and XCode's behavior remains sketchy at best in this area. The goal is to end up with Copy Headers : Public containing those headers you want exposed as your framework APIs. I think XCode automatically places those same files in the in the Copy Files Group as well, see the slide for details.

In any case, once you've managed to open/create the Copy Headers : Public tab, you can simply drag header files into it from the project browser on the left.



5 - Marking Public headers in XCode:

Sometimes broken, almost certainly won't work before step 4 is right, but the file inspector on the right is an alternate method for setting the visibility of your framework headers.



6 - The Framework Assembly Script:

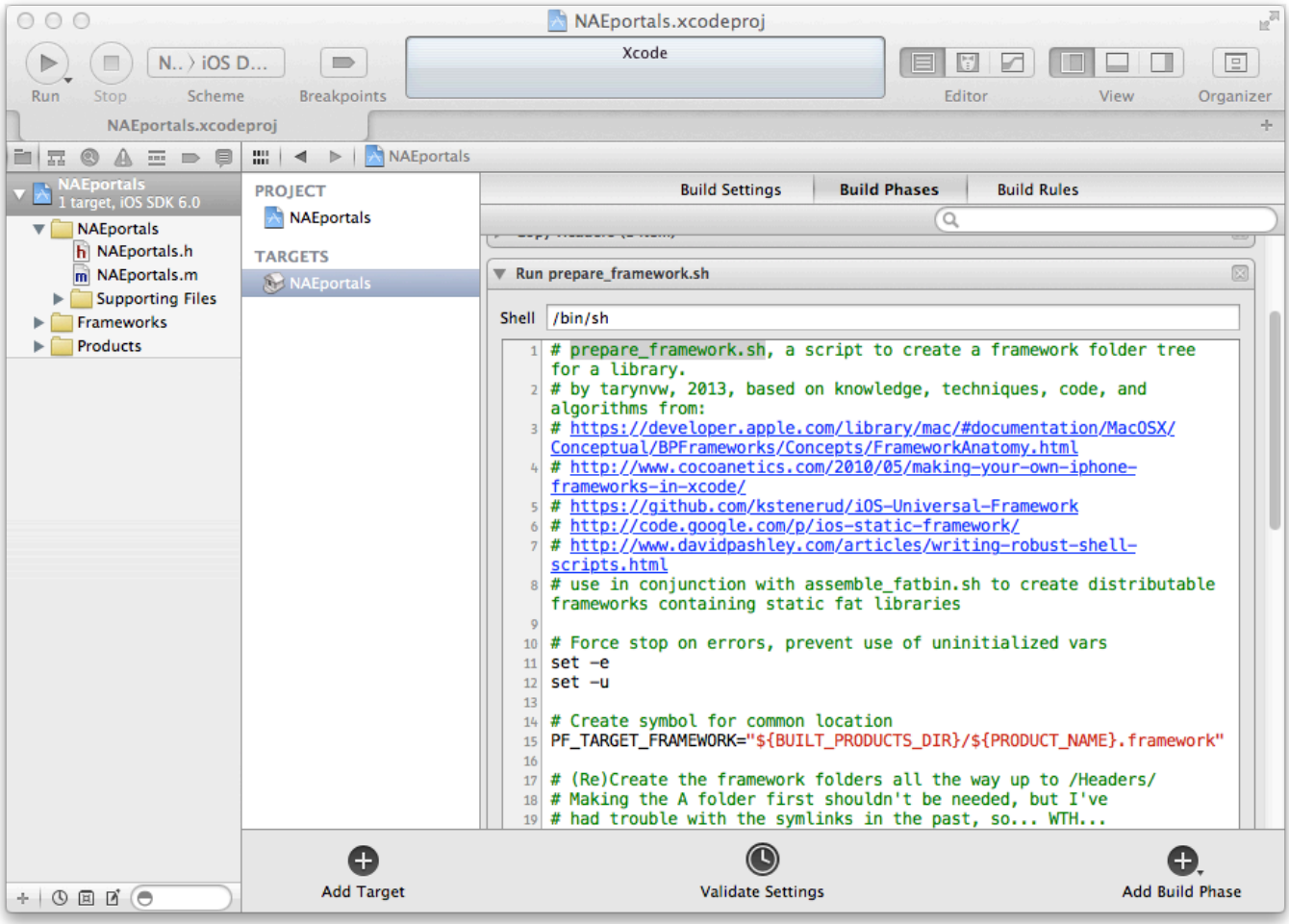
A small shell script, prepare_framework.sh, is run each time the build occurs, (re)creating framework folders and symbolic links, then copying public header files into them. To create the script Start at the project navigator and follow:

Project : TARGETS : Build Phases + Add Build Phase : Add Run Script

Then paste in this script:

```
# ----------------------------------------------------
# prepare_framework.sh, a script to create a framework folder tree for a library.
# by tarynvw, 2013, based on knowledge, techniques, code, and algorithms from:
# https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/FrameworkAnatomy.html
# http://www.cocoanetics.com/2010/05/making-your-own-iphone-frameworks-in-xcode/
# https://github.com/kstenerud/iOS-Universal-Framework
# http://code.google.com/p/ios-static-framework/
# http://www.davidpashley.com/articles/writing-robust-shell-scripts.html
# use in conjunction with assemble_fatbin.sh to create distributable frameworks containing static fat libraries

# Force stop on errors, prevent use of uninitialized vars
set -e
set -u

# Create symbol for common location
PF_TARGET_FRAMEWORK="${BUILT_PRODUCTS_DIR}/${PRODUCT_NAME}.framework"

# (Re)Create the framework folders all the way up to /Headers/
# Making the A folder first shouldn't be needed, but I've
# had trouble with the symlinks in the past, so... WTH...
mkdir -vp "${PF_TARGET_FRAMEWORK}/Versions/A/"
mkdir -vp "${PF_TARGET_FRAMEWORK}/Versions/A/Headers"

# Create symbolic file links as needed for Framework tree
# Reminder to self:
#  ln's first arg is the 'pointed to' filesystem element, which can be relative, and will
#  be stored as is, and only parsed when referenced, so it doesn't matter what directory one
#  is in when creating these links, as long as the second argument, the created 'pointing'
#  file entry is placed correctly.
ln -vsf A "${PF_TARGET_FRAMEWORK}/Versions/Current"
ln -vsf Versions/Current/Headers "${PF_TARGET_FRAMEWORK}/Headers"
ln -vsf "Versions/Current/${PRODUCT_NAME}" "${PF_TARGET_FRAMEWORK}/${PRODUCT_NAME}"

# Copy public headers, preserving date attribute (-a) so build dependencies propagate correctly.
# Force (-f) overwrites existing files without question.
cp -vaf "${BUILT_PRODUCTS_DIR}/${PUBLIC_HEADERS_FOLDER_PATH}/" "${PF_TARGET_FRAMEWORK}/Versions/A/Headers"
# ----------------------------------------------------
```

Double clicking on "Run Script" allows you to edit the name of the step, I suggest renaming it to something like "Run prepare_framework.sh" which will be more helpful in build logs.

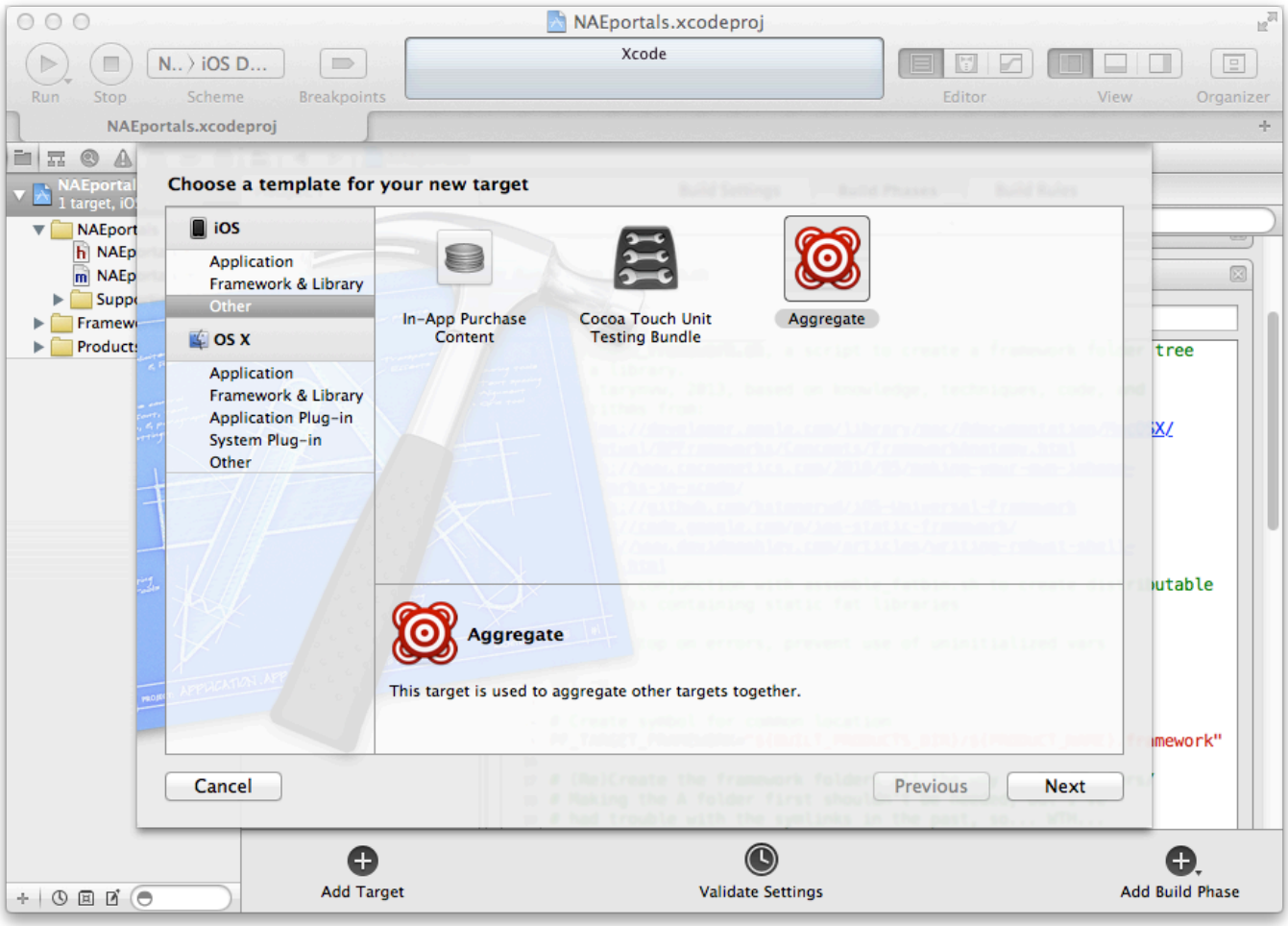

NAEportals.xcodeproj
Xcode
Run
Stop
Scheme
Breakpoints
Editor
View
Organizer
NAEportals.xcodeproj
NAEportals
NAEportals
1 target, iOS SDK 6.0
NAEportals
NAEportals.h
NAEportals.m
Supporting Files
Frameworks
Products
PROJECT
NAEportals
TARGETS
NAEportals
Build Settings
Build Phases
Build Rules
Run prepare_framework.sh
Shell
/bin/sh
prepare_framework.sh, a script to create a framework folder tree for a library.
by tarynvw, 2013, based on knowledge, techniques, code, and algorithms from:
https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/FrameworkAnatomy.html
http://www.cocoanetics.com/2010/05/making-your-own-iphone-frameworks-in-xcode/
https://github.com/kstenerud/iOS-Universal-Framework
http://code.google.com/p/ios-static-framework/
http://www.davidpashley.com/articles/writing-robust-shell-scripts.html
use in conjunction with assemble_fatbin.sh to create distributable frameworks containing static fat libraries
Force stop on errors, prevent use of uninitialized vars
set -e
set -u
# Create symbol for common location
PF_TARGET_FRAMEWORK="${BUILT_PRODUCTS_DIR}/${PRODUCT_NAME}.framework"
# (Re)Create the framework folders all the way up to /Headers/
# Making the A folder first shouldn't be needed, but I've
# had trouble with the symlinks in the past, so... WTH...
Add Target
Validate Settings
Add Build Phase

7 - Who knew Aggregating was fun.

Another shell script will be used to create the 'alternate' lib and package it into a fat binary in the currently building framework. To do this we need another target, an Aggregate. Start at the project navigator and follow:

Project : TARGETS + Add Target : iOS : Other : Aggregate, then Next

I'd suggest naming it something like FatFramework, to identify it as making the double lib.



8 - The Fatbin Assembly Script:
Assemble_fatbin.sh, is run each time aggregation occurs. To create the script Start at the project navigator and follow:

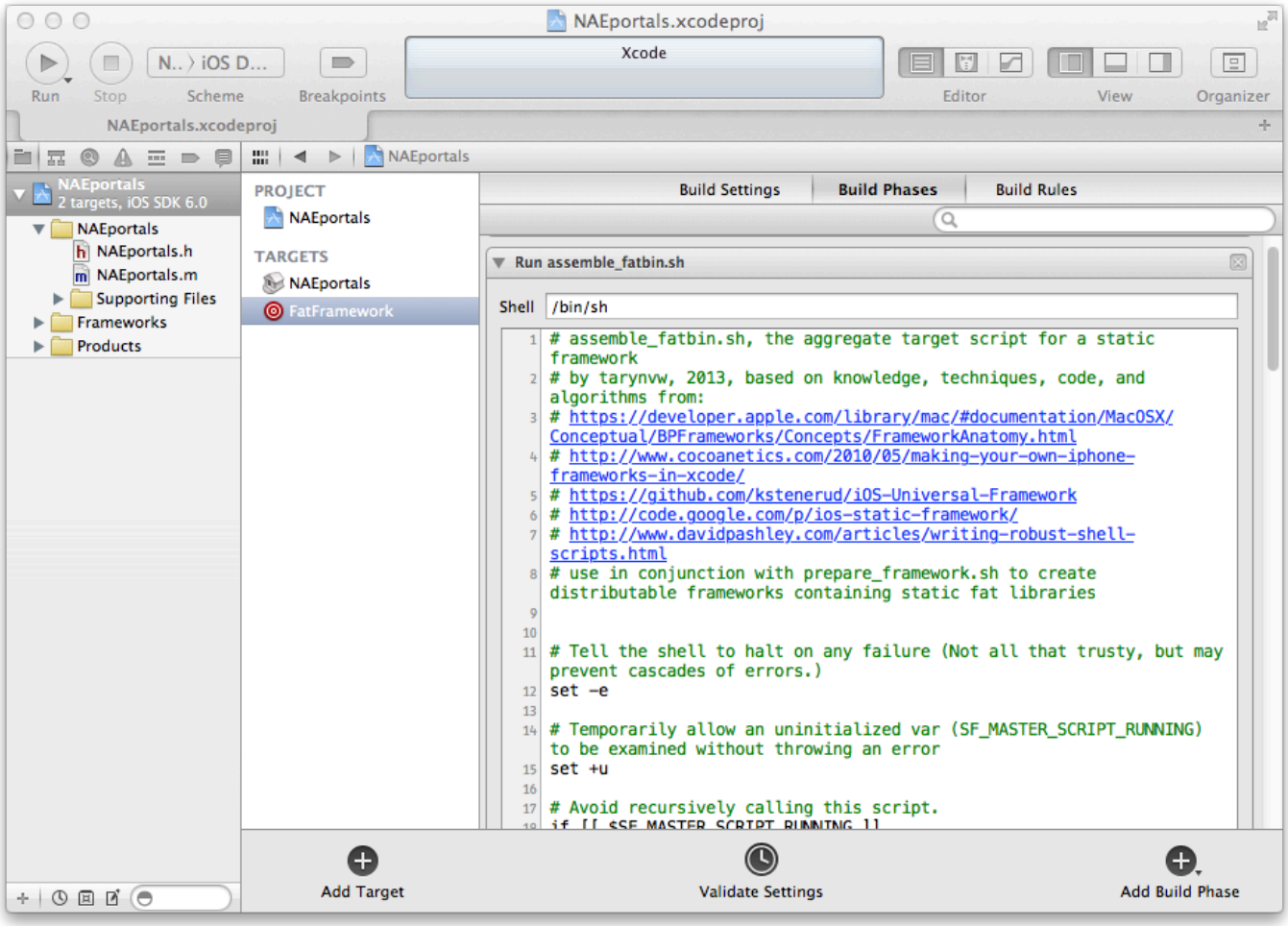
Project : TARGETS : FatFramework : Build Phases + Add Build Phase : Add Run Script
Then paste in this script:

```
# ----------------------------------------------------
# assemble_fatbin.sh, the aggregate target script for a static framework
# by tarynvw, 2013, based on knowledge, techniques, code, and algorithms from:
# https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/FrameworkAnatomy.html
# http://www.cocoanetics.com/2010/05/making-your-own-iphone-frameworks-in-xcode/
# https://github.com/kstenerud/iOS-Universal-Framework
# http://code.google.com/p/ios-static-framework/
# http://www.davidpashley.com/articles/writing-robust-shell-scripts.html
# use in conjunction with prepare_framework.sh to create distributable frameworks containing static fat libraries


# Tell the shell to halt on any failure (Not all that trusty, but may prevent cascades of errors.)
set -e

# Temporarily allow an uninitialized var (SF_MASTER_SCRIPT_RUNNING) to be examined without throwing an error
set +u

# Avoid recursively calling this script.
if [[ $SF_MASTER_SCRIPT_RUNNING ]]
then
    exit 0
fi

# Prevent any further use of uninitialized vars, and block recursion
set -u
export SF_MASTER_SCRIPT_RUNNING=1

SF_TARGET_NAME=${PROJECT_NAME}
SF_EXECUTABLE_PATH="lib${SF_TARGET_NAME}.a"
SF_WRAPPER_NAME="${SF_TARGET_NAME}.framework"


# Extract platform name and version number for later use (surprising these aren't available directly)
if [[ "$SDK_NAME" =~ ([A-Za-z]+) ]]
then
#   Incredibly powerful, the fact that a test like '=~' leaves dangling side effects to
#   be reused just boggles the mind, and makes scripts like this hopelessly unreadable.
#   BASH_REMATCH[1] will contain the char part of SDK_NAME, e.g. "iphoneos" out of "iphoneos6.0"
    SF_SDK_PLATFORM=${BASH_REMATCH[1]}
    echo "Extracted SF_SDK_PLATFORM: $SF_SDK_PLATFORM from SDK_NAME: $SDK_NAME"
else
    echo "Could not find platform name from SDK_NAME: $SDK_NAME"
    exit 1
fi
```

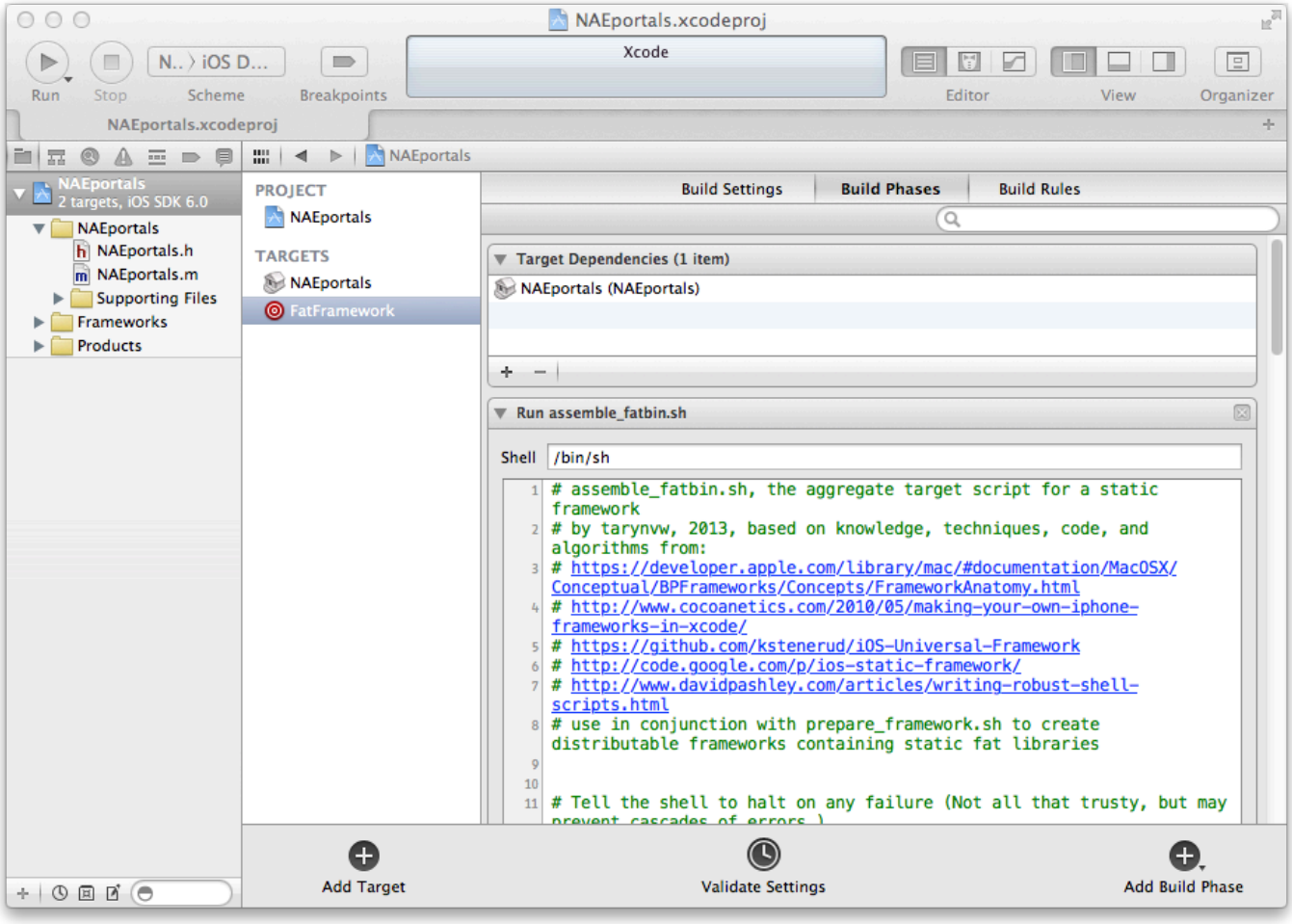
```
if [[ "$SDK_NAME" =~ ([0-9]+.*$) ]]
then
#   BASH_REMATCH[1] now contains numeric part of SDK_NAME, e.g. "6.0" out of "iphoneos6.0"
    SF_SDK_VERSION=${BASH_REMATCH[1]}
    echo "Extracted SF_SDK_VERSION: $SF_SDK_VERSION from SDK_NAME: $SDK_NAME"
else
    echo "Could not find sdk version from SDK_NAME: $SDK_NAME"
exit 1
fi

if [[ "$SF_SDK_PLATFORM" = "iphoneos" ]]
then
    SF_OTHER_PLATFORM=iphonesimulator
    SF_ARCHS=i386
else
    SF_OTHER_PLATFORM=iphoneos
#   We'd actually like to put armv6 code in the lib too, but modern XCode versions
#   won't generate armv6 code, so we'll have to settle for 7 and 7s.
    SF_ARCHS="armv7 armv7s"
fi

if [[ "$BUILT_PRODUCTS_DIR" =~ (.*)$SF_SDK_PLATFORM$ ]]
then
    SF_OTHER_BUILT_PRODUCTS_DIR="${BASH_REMATCH[1]}${SF_OTHER_PLATFORM}"
else
    echo "Could not find platform name from build products directory: $BUILT_PRODUCTS_DIR"
exit 1
fi

# Build the other platform.
xcodebuild -project "${PROJECT_FILE_PATH}" -target "${TARGET_NAME}" -configuration "${CONFIGURATION}" -sdk ${SF_OTHER_PLATFORM}$
{SF_SDK_VERSION} BUILD_DIR="${BUILD_DIR}" CONFIGURATION_TEMP_DIR="${PROJECT_TEMP_DIR}/${CONFIGURATION}-$
{SF_OTHER_PLATFORM}" ARCHS="${SF_ARCHS}" $ACTION

# Combine the two static libraries into one fat binary and store it in the .framework
lipo -create "${BUILT_PRODUCTS_DIR}/${SF_EXECUTABLE_PATH}" "${SF_OTHER_BUILT_PRODUCTS_DIR}/${SF_EXECUTABLE_PATH}" -output "$
{BUILT_PRODUCTS_DIR}/${SF_WRAPPER_NAME}/Versions/A/${SF_TARGET_NAME}"

# To simplify library testing during development, copy debug builds back to project folder.
if [[ "$CONFIGURATION" = "Debug" ]]
then
#   Because there are symlinks in the framework, it's a mess trying to copy a new one over the old, we'll toss it first.
    rm -Rf ./"${SF_WRAPPER_NAME}"
    cp -R -f -p "${BUILT_PRODUCTS_DIR}/${SF_WRAPPER_NAME}" .
    echo "Copied debug build of $SF_WRAPPER_NAME from $BUILT_PRODUCTS_DIR to project dir"
else
    echo "The fresh-built non-debug $SF_WRAPPER_NAME is in $BUILT_PRODUCTS_DIR"
fi
# ----------------------------------------------------
```

Double clicking on "Run Script" allows you to edit the name of the step, try something like "Run assemble_fatbin.sh", useful in build logs.

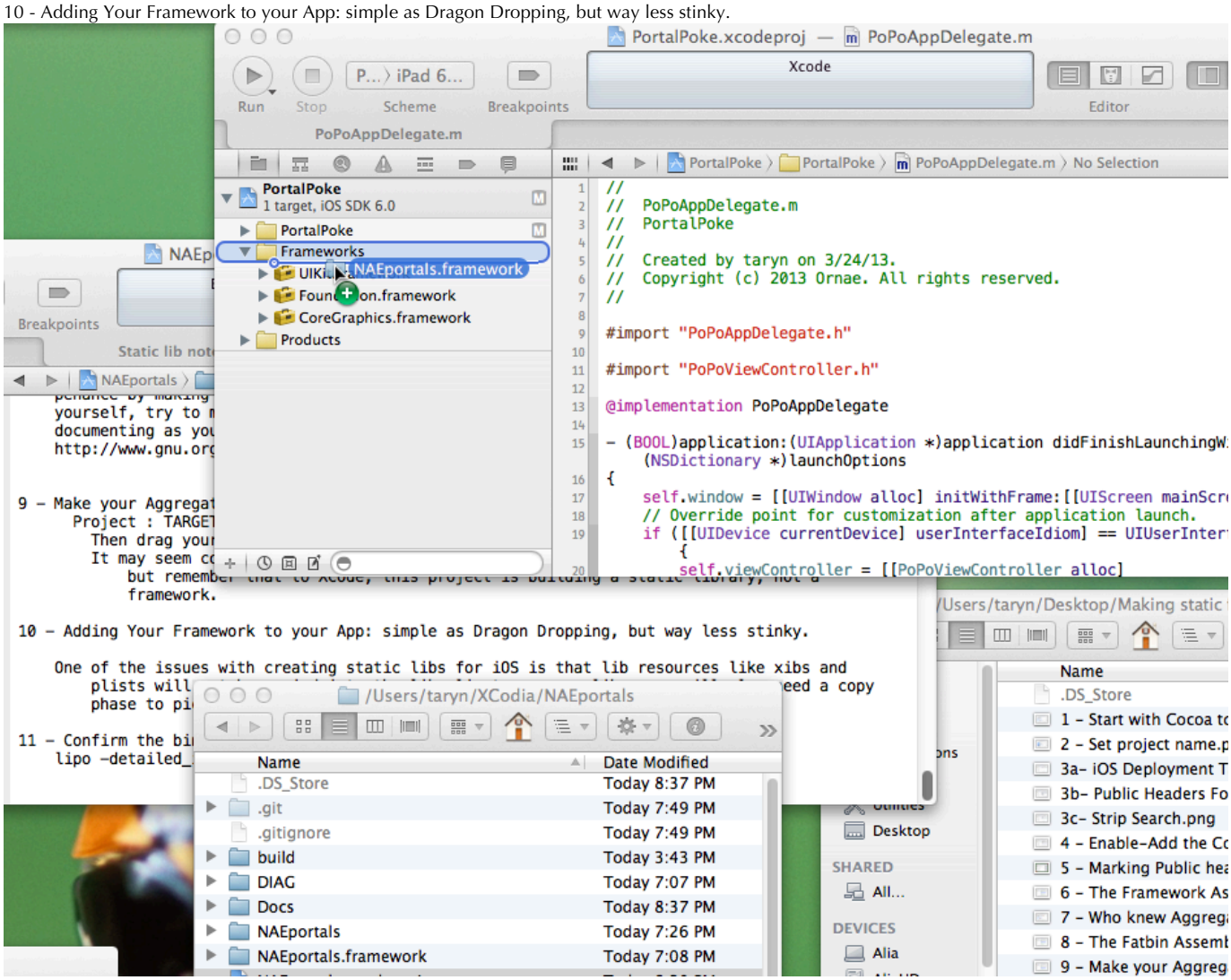
Shell scripting is truly awful. Nobody should be proud of writing stuff like this without extreme justification, since it is dangerous, obscure, and fragile. When using other's shell code, do penance by making sure you both understand and document it before using it. Writing it yourself, try to make sure you (and others) will understand it a week hence by heavily documenting as you go:) More on the peculiarities of bash's '=~' conditional here:
http://www.gnu.org/software/bash/manual/bashref.html#Conditional-Constructs



9 - Make your Aggregate target dependent on your Project target:
Project : TARGETS : FatFramework : Build Phases : Target Dependencies(expand)
Then drag your library into the the aggregate's Target Dependencies window.
It may seem counterintuitive that the Aggregate depends upon the Framework it is building but remember that to XCode, this project is building a static library, not a framework.



10 - Adding Your Framework to your App: simple as Dragon Dropping, but way less stinky.



11 - Confirm the binary is "fat" from the command line:
lipo -detailed_info <libname>.framework/Versions/A/<libname>

Note:
One of the issues with creating static libs for iOS is that lib resources like xibs and plists will not be copied into the lib clients, so any lib users will also need a copy phase to pick up the lib's GUI elements.