Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

A Build System for Complex Projects: Part 3

Generating a full-fledged NetBeans build system for a non-trivial system involves multiple projects -- and more

September 21, 2009
URL:http://www.drdobbs.com/architecture-and-design/a-build-system-for-complex-projects-part/220100417

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

This is the third article in a series of articles that explore an innovative build system for complicated projects. Part 1 and Part 2 discussed build systems in general and the internals of the ideal build system that can integrate with existing build systems. This article will explain in detail how the ibs can generate build files for NetBeans 6.x, which is a fantastic free cross-platform IDE that supports multiple programming languages.

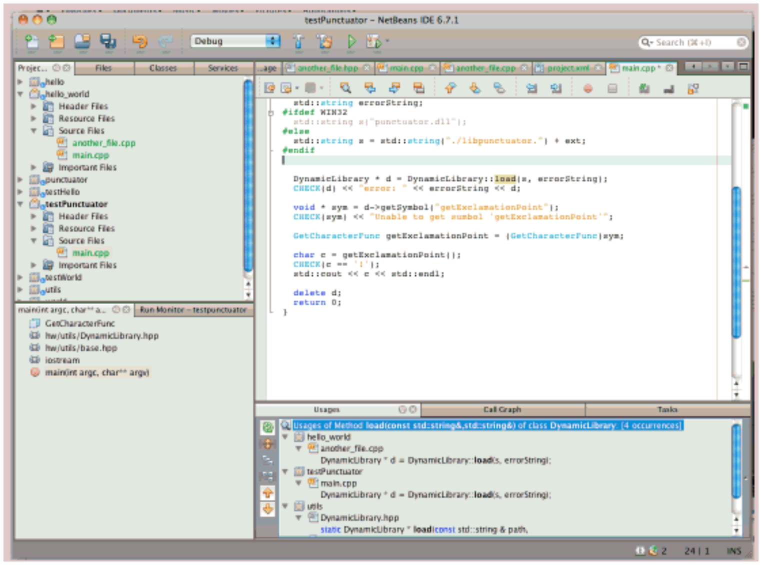
To recap: ibs is an invisible build system that doesn't require any build files. It relies on a regular directory and conventions to infer build rules and it detects dependencies automatically. It generates build files for other IDEs or build systems like Makefiles, Visual Studio solutions, or NetBeans projects. It is focused on C/C++ projects, but can be extended to additional languages and other projects types.

Generating a NetBeans Build System

Back in the Hello world project (Enterprise Platinum Edition!!!) trenches Bob and the other developers picked NetBeans as their primary Mac and Linux IDE. NetBeans started as a Java IDE, but grew in leaps and bounds to become a component-based platform for application development in addition to an IDE that supports all the common programming languages in wide use today. The latest version is NetBeans 6.7.1 and it supports C/C++ development very well and even Python via a special early access.

The task facing Bob is to figure out the structure of the NetBeans build files for C/C++ and implement the build system specific parts (the helper and the project templates) to embrace NetBeans into the ibs.

Bob did some reading and poking around and discovered that NetBeans itself is a Makefile-based (build system generator. The way it works is that each project's directory has a common Makefile that includes a bunch of auto-generated sub-makefiles. The NetBeans user interface allows you to add files to every project and set dependencies between projects. All this information is stored in several files that NetBeans uses to generate the proper sub-makefiles. Figure 1 shows the NetBeans IDE with the various Hello World projects.



```
hello
hello_world
    Header Files
    Resource Files
    Source Files
        another_file.cpp
        main.cpp
    Important Files
punctuator
testHello
testPunctuator
    Header Files
    Resource Files
    Source Files
        main.cpp
    Important Files
testWorld
utils
```

```
main(int argc, char** a...
GetCharacterFunc
hw/utils/DynamicLibrary.hpp
hw/utils/base.hpp
iostream
main(int argc, char** argv)
```

```
    std::string errorString;
  #ifdef WIN32
    std::string s("punctuator.dll");
  #else
    std::string s = std::string("./libpunctuator.") + ext;
  #endif

    DynamicLibrary * d = DynamicLibrary::load(s, errorString);
    CHECK(d) << "error: " << errorString << d;

    void * sym = d->getSymbol("getExclamationPoint");
    CHECK(sym) << "Unable to get symbol 'getExclamationPoint'";

    GetCharacterFunc getExclamationPoint = (GetCharacterFunc)sym;

    char c = getExclamationPoint();
    CHECK(c == '!');
    std::cout << c << std::endl;

    delete d;
    return 0;
  }
```

```
Usages of Method load(const std::string&,std::string&) of class DynamicLibrary: [4 occurrences]
  hello_world
    another_file.cpp
      DynamicLibrary * d = DynamicLibrary::load(s, errorString);
  testPunctuator
    main.cpp
      DynamicLibrary * d = DynamicLibrary::load(s, errorString);
  utils
    DynamicLibrary.hpp
      static DynamicLibrary * load(const std::string & path,
```

Figure 1

The NetBeans Build System Anatomy

Let's take a look and see what all these files are about. As a running example Bob suggests the the 'hello' project. This is a static library that contains two files: hello.cpp and hello.hpp. If you are unfamiliar with make-based build systems, you may want to take a small detour and read about it here.

The project Makefile

In the 'hello' directory there is the common Makefile. This file is the same for every project (on a given platform). It sets some environment variables and contains a bunch of make targets and most importantly includes the project implementation makefile (generated based on the actual content of the project). Every target (a step in the build process) has an empty pre and post targets that allow you to customize the build process by executing actions before and/or after each target. The original file has elaborate comments that explain exactly what targets are available and how you can override them. Here is an edited version without the comments and some of the targets of the Mac OS X Makefile:

```
# Environment
MKDIR=mkdir
CP=cp
CCADMIN=CCadmin
RANLIB=ranlib

# clean
clean: .clean-pre .clean-impl .clean-post

.clean-pre:
# Add your pre 'clean' code here...

.clean-post:
# Add your post 'clean' code here...


# all
all: .all-pre .all-impl .all-post

.all-pre:
# Add your pre 'all' code here...

.all-post:
# Add your post 'all' code here...


# include project implementation makefile
include nbproject/Makefile-impl.mk
```

nbproject

The Makefile resides in the project's directory. All the other build files reside in a sub directory called nbproject. There is nothing special about it. It is just convenient to have all the build files in their own directory and not cluttering the project directory. The Makefile is the exception due to limitations of the make program.

Generated make files

NetBeans generates three "sub" make files that are included by the main project Makefile: Makefile-impl.mk, Makefile-Debug.mk and Makefile-Release.mk. Makefile-impl.mk is included directly by the main Makefile and it invokes either Makefile-Debug.mk or Makefile-Release.mk depending on the current active configuration, which is controlled by the **$CONF** environment variable. In the NetBeans IDE you may select what configuration is active.You may also create your own configurations and they will be available for activation just like the built-in Debug and Release configurations with their own Makefile-*your configuration*.mk file. Here is the Makefile-impl.mk file of the testPunctuator project:

```
#
# Generated Makefile - do not edit!
#
# Edit the Makefile in the project folder instead (../Makefile). Each target
# has a pre- and a post- target defined where you can add customization code.
#
# This makefile implements macros and targets common to all configurations.
#
# NOCDDL

# Building and Cleaning subprojects are done by default, but can be controlled with the SUB
# macro. If SUB=no, subprojects will not be built or cleaned. The following macro
# statements set BUILD_SUB-CONF and CLEAN_SUB-CONF to .build-reqprojects-conf
# and .clean-reqprojects-conf unless SUB has the value 'no'
SUB_no=NO
SUBPROJECTS=${SUB_${SUB}}
BUILD_SUBPROJECTS_=.build-subprojects
BUILD_SUBPROJECTS_NO=
BUILD_SUBPROJECTS=${BUILD_SUBPROJECTS_${SUBPROJECTS}}
CLEAN_SUBPROJECTS_=.clean-subprojects
CLEAN_SUBPROJECTS_NO=
CLEAN_SUBPROJECTS=${CLEAN_SUBPROJECTS_${SUBPROJECTS}}

# Project Name
PROJECTNAME=testPunctuator
```

```
# Active Configuration
DEFAULTCONF=Debug
CONF=${DEFAULTCONF}

# All Configurations
ALLCONFS=Debug Release

# build
.build-impl: .validate-impl
  @#echo "=> Running $@... Configuration=$(CONF)"
  ${MAKE} -f nbproject/Makefile-${CONF}.mk SUBPROJECTS=${SUBPROJECTS} .build-conf


# clean
.clean-impl: .validate-impl
  @#echo "=> Running $@... Configuration=$(CONF)"
  ${MAKE} -f nbproject/Makefile-${CONF}.mk SUBPROJECTS=${SUBPROJECTS} .clean-conf


# clobber
.clobber-impl:
  @#echo "=> Running $@..."
  for CONF in ${ALLCONFS}; \
  do \
      ${MAKE} -f nbproject/Makefile-$${CONF}.mk SUBPROJECTS=${SUBPROJECTS} .clean-conf; \
  done

# all
.all-impl:
  @#echo "=> Running $@..."
  for CONF in ${ALLCONFS}; \
  do \
      ${MAKE} -f nbproject/Makefile-$${CONF}.mk SUBPROJECTS=${SUBPROJECTS} .build-conf; \
  done


# configuration validation
.validate-impl:
  @if [ ! -f nbproject/Makefile-${CONF}.mk ]; \
  then \
      echo ""; \
      echo "Error: can not find the makefile for configuration '${CONF}' in project ${PROJECTNAME}"; \
      echo "See 'make help' for details."; \
      echo "Current directory: " `pwd`; \
      echo ""; \
  fi
  @if [ ! -f nbproject/Makefile-${CONF}.mk ]; \
  then \
      exit 1; \
  fi

# help
.help-impl:
  @echo "This makefile supports the following configurations:"
  @echo "    ${ALLCONFS}"
  @echo ""
  @echo "and the following targets:"
  @echo "    build  (default target)"
  @echo "    clean"
  @echo "    clobber"
  @echo "    all"
  @echo "    help"
  @echo ""
  @echo "Makefile Usage:"
  @echo "    make [CONF=<CONFIGURATION>] [SUB=no] build"
  @echo "    make [CONF=&gltCONFIGURATION>] [SUB=no] clean"
  @echo "    make [SUB=no] clobber"
  @echo "    make [SUB=no] all"
  @echo "    make help"
  @echo ""
  @echo "Target 'build' will build a specific configuration and, unless 'SUB=no',"
  @echo "    also build subprojects."
  @echo "Target 'clean' will clean a specific configuration and, unless 'SUB=no',"
  @echo "    also clean subprojects."
  @echo "Target 'clobber' will remove all built files from all configurations and,"
  @echo "    unless 'SUB=no', also from subprojects."
  @echo "Target 'all' will will build all configurations and, unless 'SUB=no',"
  @echo "    also build subprojects."
  @echo "Target 'help' prints this message."
  @echo ""
```

The structure of this file is very uniform. Every command is implemented in the same way (except .help that just echos the help text to the screen). It always invokes eventually the configuration specific make file. Commands may operate on all configurations and on sub-projects too (on by default). For example the default .build command (if you just type 'make') is:

```
# build
.build-impl: .validate-impl
  @#echo "=> Running $@... Configuration=$(CONF)"
  ${MAKE} -f nbproject/Makefile-${CONF}.mk SUBPROJECTS=${SUBPROJECTS} .build-conf
```

Let me decipher this line-noise that makes sense only to make-savvy people. The name of the command is **.build-impl**. It will execute the **.validate-impl** command, skip the commented out **echo** command (if you remove the # it will print the text between the double quotes) and run 'make' again on the file nbproject/Makefile-${CONF}.mk (CONF is the active configuration, which is Debug in this case, unless you specified CONF=Release when you run 'make'). Finally it will execute the .build-conf command that is defined in the Makefile-${CONF}.mk. This command builds all the sub projects (if necessary) and finally build the project itself by invoking the C++ compiler and linker.

It sounds complicated and it is complicated. This is the cleanest make-based system I have seen with good separation of concerns, very uniform structure and great extensibility. Most make-based build systems are simply a mess. The nice thing about NetBeans is that it takes care of all the messy parts and lets you work entirely at the IDE level, but still allows you to extend the build process at the makefile-level if you need to do something special.

Let's take a look at the Makefile-Debug.mk file:

```
#
# Generated Makefile - do not edit!
#
# Edit the Makefile in the project folder instead (../Makefile). Each target
# has a -pre and a -post target defined where you can add customized code.
#
# This makefile implements configuration specific macros and targets.

# Environment
MKDIR=mkdir
CP=cpj
CCADMIN=CCadmin
RANLIB=ranlib
CC=gcc
CCC=g++
CXX=g++
FC=

# Include project Makefile
include Makefile

# Object Directory
OBJECTDIR=build/Debug/GNU-MacOSX

# Object Files
OBJECTFILES= \
  ${OBJECTDIR}/main.o

# C Compiler Flags
CFLAGS=

# CC Compiler Flags
CCFLAGS=
CXXFLAGS=

# Fortran Compiler Flags
FFLAGS=

# Link Libraries and Options
LDLIBSOPTIONS=../../hw/utils/dist/Debug/GNU-MacOSX/libutils.a

# Build Targets
.build-conf: ${BUILD_SUBPROJECTS} dist/Debug/GNU-MacOSX/testpunctuator

dist/Debug/GNU-MacOSX/testpunctuator: ${BUILD_SUBPROJECTS}

dist/Debug/GNU-MacOSX/testpunctuator: ${OBJECTFILES}
  ${MKDIR} -p dist/Debug/GNU-MacOSX
  ${LINK.cc} -o dist/Debug/GNU-MacOSX/testpunctuator ${OBJECTFILES} ${LDLIBSOPTIONS}

${OBJECTDIR}/main.o: main.cpp
  ${MKDIR} -p ${OBJECTDIR}
  $(COMPILE.cc) -g -I../.. -o ${OBJECTDIR}/main.o main.cpp

# Subprojects
.build-subprojects:
  cd ../../hw/utils && ${MAKE}  -f Makefile CONF=Debug

# Clean Targets
.clean-conf: ${CLEAN_SUBPROJECTS}
  ${RM} -r build/Debug
  ${RM} dist/Debug/GNU-MacOSX/testpunctuator

# Subprojects
.clean-subprojects:
  cd ../../hw/utils && ${MAKE}  -f Makefile CONF=Debug clean
```

This file includes the project's Makefile (which includes the Makefile-impl.mk), defines a bunch of variables that point to different tools like C and C++ compilers and it also defines the dependencies of the project (in this case just the hw/utils sub-project). Note the **.build-conf** command that I mentioned earlier when I discussed the **.build-impl** command from Makefile-impl.mk. So, there is a lot of interplay between the various make files. This is done in the interest of separating fixed logic like command invocation from very dynamic parts like the files that are contained in a project and its dependencies and also providing clear extension points (the.pre and .post commands in the main Makefile). The bottom line is that most developers don't even need to know that there is a make-based build system underneath and can just stay at the IDE level. Build administrators can automate the build process using this clean and standard make interface and people with special needs can customize the build process very elegantly using the extension points provided by .pre and

.post targets, as well as add new targets.

Project Metadata Files

Make files are okay (I wouldn't say great) for running a build, but they are not very easy to parse and modify. NetBeans uses two XML files to maintain the project information and dynamically generates the make files from these files.

configurations.xml

This is the file. It contains most of the project information used for generating the make files as well as some GUI information such as the logical folders displayed in the IDE for each project and what files reside in them. It is a typical XML file. The root element is called **configurationDescriptor** and it contains a **logicalFolder** element called "root that contains nested **logicalFolder** elements for "HeaderFiles", "ResourceFiles", "SourceFiles" and "ExternalFiles". Each logicalFolder element may contain **itemPath** elements for the files in this folder. Here is the logical folders of the **testPunctuator** project:

```
<?xml version="1.0" encoding="UTF-8"?>
  <configurationDescriptor version="45">
    <logicalFolder name="root" displayName="root" projectFiles="true">
    <logicalFolder name="HeaderFiles"
             displayName="Header Files"
             projectFiles="true">
    </logicalFolder>
    <logicalFolder name="ResourceFiles"
             displayName="Resource Files"
             projectFiles="true">
    </logicalFolder>
    <logicalFolder name="SourceFiles"
             displayName="Source Files"
             projectFiles="true">
      <itemPath>main.cpp</itemPath>
    </logicalFolder>
    <logicalFolder name="ExternalFiles"
             displayName="Important Files"
             projectFiles="false">
      <itemPath>Makefile</itemPath>
    </logicalFolder>
    </logicalFolder>
```

The configurations.xml file also contains elements for the encoding the make file for the project (in case you want to change its name for some reason):

```
  <sourceEncoding>UTF-8</sourceEncoding>
  <projectmakefile>Makefile</projectmakefile>
```

Then comes the all important "confs" elements that contains "conf" elements for each configuration. Each "conf" elements contains a "toolset" element and a "compileType" elements that contains various tools. Each tool has its own set of elements and attribute that translate directly to make file tool settings:

```
<confs>
    <conf name="Debug" type="1">
      <toolsSet>
        <compilerSet>GNU|GNU</compilerSet>
        <platform>4</platform>
      </toolsSet>
      <compileType>
        <ccCompilerTool>
          <includeDirectories>
            <directoryPath>../..</directoryPath>
          </includeDirectories>
        </ccCompilerTool>
        <linkerTool>
          <linkerLibItems>
            <linkerLibProjectItem>
              <makeArtifact PL="../../hw/utils"
                            CT="3"
                            CN="Debug"
                            AC="true"
                            BL="true"
                            WD="../../hw/utils"
                            BC="${MAKE}  -f Makefile CONF=Debug"
                            CC="${MAKE}  -f Makefile CONF=Debug clean"
                            OP="dist/Debug/GNU-MacOSX/libutils.a">
              </makeArtifact>
            </linkerLibProjectItem>
          </linkerLibItems>
        </linkerTool>
      </compileType>
      <item path="main.cpp">
        <itemTool>1</itemTool>
      </item>
    </conf>
    <conf name="Release" type="1">
      <toolsSet>
        <compilerSet>GNU|GNU</compilerSet>
        <platform>4</platform>
      </toolsSet>
      <compileType>
        <cCompilerTool>
```

```
          <developmentMode>5</developmentMode>
        </cCompilerTool>
        <ccCompilerTool>
          <developmentMode>5</developmentMode>
          <includeDirectories>
            <directoryPath>../..</directoryPath>
          </includeDirectories>
        </ccCompilerTool>
        <fortranCompilerTool>
          <developmentMode>5</developmentMode>
        </fortranCompilerTool>
        <linkerTool>
          <linkerLibItems>
            <linkerLibProjectItem>
              <makeArtifact PL="../../hw/utils"
                            CT="3"
                            CN="Release"
                            AC="false"
                            BL="true"
                            WD="../../hw/utils"
                            BC="${MAKE}  -f Makefile CONF=Release"
                            CC="${MAKE}  -f Makefile CONF=Release clean"
                            OP="dist/Release/GNU-MacOSX/libutils.a">
              </makeArtifact>
            </linkerLibProjectItem>
          </linkerLibItems>
        </linkerTool>
      </compileType>
      <item path="main.cpp">
        <itemTool>1</itemTool>
      </item>
    </conf>
  </confs>
```

There is one "conf" element for each configuration (in this case Debug and Release).

project.xml

The project.xml file holds additional information like the project type, project dependencies and the file extensions of different file types. I'm not sure why this information should go in a separate file, but that's how it is. Here is the project.xml file of the hello_world application project itself:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <project xmlns="http://www.netbeans.org/ns/project/1">
    <type>org.netbeans.modules.cnd.makeproject</type>
    <configuration>
      <data xmlns="http://www.netbeans.org/ns/make-project/1">
        <name>hello_world</name>
        <make-project-type>0</make-project-type>
        <make-dep-projects>
          <make-dep-project>../../hw/hello</make-dep-project>
          <make-dep-project>../../hw/utils</make-dep-project>
          <make-dep-project>../../hw/world</make-dep-project>
        </make-dep-projects>
        <c-extensions/>
        <cpp-extensions>cpp</cpp-extensions>
        <header-extensions>hpp</header-extensions>
      </data>
    </configuration>
  </project>
```

Project Group

In addition to the individual project files NetBeans support a higher level of organization called a project group. A project group is simply a collection of projects that can be loaded together into the NetBeans IDE. The hello world system is also a project group that contains many projects. The ibs can generate such a project group on behalf of the user and update it automatically when new projects are added or removed.

NetBeans keeps a lot of information in the user's home directory in a hidden directory called ".netbeans". The project groups are stored in the following directory:

```
~/.netbeans/6.7/config/Preferences/org/netbeans/modules/projectui/groups
```

Note the 6.7 version number following the .netbeans directory. You may have multiple versions of NetBeans installed on your machine and their preferences are stored separately. Each project group has a file under the projectui sub-directory called **<project group name>.properties**. There are three kinds of projects groups: free group, master + dependencies and folder group (all projects under a root directory). The hello_world project group is a folder group. Here is the hello_world.properties file:

```
name=hello_world
kind=directory
path=file\:/Users/gsayfan/Documents/Invisible.Build.System/src
```

Pretty simple, really. In addition there is another important file called:

```
~/.netbeans/6.7/config/Preferences/org/netbeans/modules/projectui/groups.properties.
```

This file determines the active project group if there are multiple project groups. Its format is very simple too and to make a group active you just need to have this line in the file:

```
active=<project group name>
```

In case of the hello_world project group it is:

```
active=hello_world
```

The NetBeans Helper

The NetBeans Helper class is responsible for implementing all the code that is NetBeans-specific. The generic build_system_generator.py script is using this helper to generate all the NetBeans project files (inside the nbproject directory) for each project and a project group that includes all the generated project. Let's take a closer look at this class. The first thing it does is import some useful system modules and then import the BaseHelper and Template classes from the build_system_generator module (as well as the 'title' function for debugging purposes):

```
#!/usr/bin/env python
import os, sys, string
from pprint import pprint as pp

sys.path.insert(0, os.path.join(os.path.abspath(os.path.dirname(__file__)), '../'))
from build_system_generator import (BaseHelper,
                                    Template,
                                    title)
```

Then the **Helper**class is defined. This is the class that the build system generator module is using to customize the build system generation for each specific target (NetBeans 6.x in this case). The **__init__()** method accepts the **templates_dir**, which is the path to the root of the templates directory used to generate all the build files. It also initializes the separator ('/') and line separator ('\n') to Unix values to make the generated files fit well in their intended environment. The **skip_dir** is used to tell the recursive drill-down code that looks for projects in sub-directories to ignore directories called 'nbproject' (which is the special sub-directory used by NetBeans to store the build files). The other methods this class implements are: **get_templates()**, **prepare_substitution_dict()**, and **generate_workspace_files()**.

```
class Helper(BaseHelper):
  """NetBeans6 helper
  """
  def __init__(self, templates_dir):
    BaseHelper.__init__(self, templates_dir)
    self.sep = '/'
    self.linesep = '\n'
    self.skip_dir = 'nbproject'

  def get_templates(self, template_type):
    ...

  def prepare_substitution_dict(self, ...):
    ...
    def generate_workspace_files(self, name, root_path, projects):
    ...
```

get_templates()

The **get_templates()**method is pretty simple. For each build file there is a corresponding template file. These template files are just skeleton of real build files, with some place holders. You will see all the template files soon enough. The **get_templates()** method just iterates over all the template files (located in the nbproject) and adds a template for the Makefile in the project directory itself. For each such build a file a **Template** object is generated. Finally the list of **Template** objects is returned.

```
  def get_templates(self, template_type):
    result = [Template(os.path.join(self.templates_dir,
                                    template_type,
                                    'Makefile'),
                                    'Makefile',
                                    template_type)]

    nb_project = os.path.join(self.templates_dir, template_type, 'nbproject')
    assert os.path.isdir(nb_project)
    for f in os.listdir(nb_project):
      project_file_template = os.path.join(nb_project, f)
      if not os.path.isfile(project_file_template):
        continue

      filename = os.path.join(nb_project, f)
      relative_path = '/'.join(['nbproject', f])
      result.append(Template(filename, relative_path, template_type))
    return result
```

prepare_substitution_dict()

This method is the heart of **NetBeans6_Helper** class. It is responsible for creating a substitution dictionary that contains all the values to be substituted into the templates of each build file. This is not so trivial because some place holders are supposed to be replaced by dynamic content that is generated on the fly. In addition, as you saw earlier NetBeans has quite a few build files. The **prepare_substitution_dict()** method has several nested function to assist in prepare the substitution dictionary for each one of them. The nested functions are:

- **prepare_makefile**() for generating the Makefile-Debug.mk and Makefile-Release.mk files
- **prepare_configurations_xml**() for generating configurations.xml
- **prepare_project_properties**() for generating project.properties
- **prepare_project_xml**() for generating project.xml

The substitution dict for the project's main Makefile is empty because it is a generic file that doesn't have any place holder and the substitution dict for the Makefile-impl.mk file contains only the name of the project so no helper function is necessary. Here is the code of the method (without the nested functions). It accepts a long list of arguments that the various nested functions use to generate the proper values. The operating system and the dynamic library extension are also determined here. This method is called multiple times with different template names (each template_name corresponds to a build file) and **prepare_substitution_dict**() calls the proper nested function or generates the dict directly (for Makefile and MakeFile-Impl.mk).

```
def prepare_substitution_dict(self,
                                            project_name,
                                    project_type,
                                            project_file_template,
                                            project_dir,
                                            libs_dir,
                                            dependencies,
                                            source_files,
                                            header_files,
                                            platform):
if platform.startswith('darwin'):
  operating_system = 'MacOSX'
  ext = 'dylib'
elif platform.startswith('linux'):
  operating_system = 'Linux'
  ext = 'so'

temaplate_name = os.path.basename(project_file_template)
if temaplate_name ==  'Makefile':
  return {}

if temaplate_name ==  'Makefile-Debug.mk':
  return prepare_makefile('Debug', operating_system)

if temaplate_name ==  'Makefile-Release.mk':
  return prepare_makefile('Release', operating_system)

if temaplate_name == 'Makefile-impl.mk':
  return dict(Name=os.path.basename(project_dir))

if temaplate_name == 'configurations.xml':
  return prepare_configurations_xml(operating_system)

if template_name == 'project.properties':
  return prepare_project_properties()

if temaplate_name == 'project.xml':
  return prepare_project_xml(dependencies)

assert False, 'Invalid project file template: ' + temaplate_name
return {}
```

Now, let's examine one of nested functions. I chose the **prepare_makefile**() function because it is not trivial. The keys in its substitution dictionary are: 'ObjectFiles', 'CompileFiles', 'LinkCommand', 'LDLIBSOPTIONS', 'BuildSubprojects', 'CleanSubprojects', 'OperatingSystem' and 'DynamicLibExtension'. Some of these are simple strings like 'OperatingSystem' and 'DynamicLibExtension'. Others are much more complicated like 'CompileFiles', which is a list of compile commands where each command itself requires a template with substitution values such as 'File', 'CompileFlag', 'Platform' and 'FPIC'. The **link** command depends on the project type and **ldliboptions** depends on the platform. Here is the code:

```
    def prepare_makefile(conf, operating_system):
      compile_flag = '-g' if conf == 'Debug' else '-O2'

      d = dict(Name=project_name)
      object_file_template = '  ${OBJECTDIR}/%s.o \\\n'
      object_files = ''
      for f in source_files:
        f = os.path.splitext(os.path.basename(f))[0]
        object_files += object_file_template % f

      # Flag for dynamic libraries
      fpic = '-fPIC  ' if project_type == 'dynamic_lib' else ''

      # Get rid of last forward slash
      if len(object_files) > 2:
        object_files = object_files[:-3]
      d['ObjectFiles'] = object_files

      compile_file_template = \
        '$${OBJECTDIR}/${File}.o: ${File}.cpp \n' + \
        '\t$${MKDIR} -p $${OBJECTDIR}\n' + \
        '\t$$(COMPILE.cc) ${CompileFlag} -I../.. ${FPIC}-o $${OBJECTDIR}/${File}.o ${File}.cpp\n\n'

      t = string.Template(compile_file_template)
      compile_files = ''
      for f in source_files:
        f = os.path.splitext(os.path.basename(f))[0]
        text = t.substitute(dict(File=f,
                                 CompileFlag=compile_flag,
                                 Platform=platform,
                                 FPIC=fpic))
        compile_files += text

      # Get rid of the last two \n\n.
      compile_files = compile_files[:-2]
      d['CompileFiles'] = compile_files

      link_command = ''
      if project_type == 'dynamic_lib':
        if platform.startswith('darwin'):
          link_command = '${LINK.cc} -dynamiclib -install_name lib%s.dylib' % project_name
        else:
          assert platform.startswith('linux')
          link_command = '${LINK.c} -shared'
        d['LinkCommand'] = link_command

      ldlibsoptions = ''
      if dependencies != []:
        ldliboption_template = '../../hw/%s/dist/%s/GNU-%s/lib%s.a'

        ldlibsoptions = ' '.join([ldliboption_template % \
                        (dep.name, conf, operating_system, dep.name)
                        for dep in dependencies])
        if operating_system == 'Linux':
          ldlibsoptions += ' -ldl'
      d['LDLIBSOPTIONS'] = ldlibsoptions

      build_subproject_template = '\tcd ../../hw/%s && ${MAKE}  -f Makefile CONF=%s'
      clean_subproject_template = build_subproject_template + ' clean'

      build_list = [build_subproject_template % (dep.name, conf) for dep in dependencies]
      clean_list = [clean_subproject_template % (dep.name, conf) for dep in dependencies]
      d['BuildSubprojects'] = '\n'.join(build_list)
      d['CleanSubprojects'] = '\n'.join(clean_list)
      d['OperatingSystem'] = operating_system
      d['DynamicLibExtension'] = ext

      return d
```

Note, that there are better ways to accomplish this task. There are several third-party template languages like Genshi, Mako, Tempita and Jinja. These template engines can handle the nested templates that **prepare_makefile()** generates manually in a much more natural way. The code could have been much shorter and concise. I made a deliberate decision to use only standard Python libraries in the interest of keeping the scope of this project limited. Choosing a particular template language/engine would have made the code shorter, but required the reader to understand an additional language and might antagonize fans of other template languages.

The other **prepare_XXX()** nested functions are all very similar to **make_makefile()** although some of them generate XML files and another one generate a properties file (INI file like).

generate_workspace_files()

This method is responsible for generating the project groups in the user account. The reason the method is called **generate_workspace_files()** is that the method is defined in the generic **Helper** base class and **NetBeans6_Helper** is just overriding it. So, the NetBeans-specific term "Project Group" is not used here. The code itself is pretty simple. It either creates or updates the proper .properties files that dictate the contents of the project groups as explained earlier:

```
  def generate_workspace_files(self, name, root_path, projects):
    """Generate a NetBeans project group for all the generated projects

    """
    base_path = \
      '~/.netbeans/6.7/config/Preferences/org/netbeans/modules/projectui'
    base_path = os.path.expanduser(base_path)

    if not os.path.exists(base_path):
      os.makedirs(base_path)

    # Create a project group
    groups_path = os.path.join(base_path, 'groups')
    if not os.path.exists(groups_path):
      os.makedirs(groups_path)

    text = """\
name=%s
kind=directory
path=file\:%s"""
    group_filename = os.path.join(groups_path, name + '.properties')
    open(group_filename, 'w').write(text % (name, root_path))

    # Make it the active project
    text = 'active=' + name
    open(os.path.join(base_path, 'groups.properties'), 'w').write(text)
```

The NetBeans Project Templates

The substitution dictionaries are very important of course, but they can't do much by themselves. Each build file is generated by substituting the values from the proper dictionary into the proper template file.

As you recall NetBeans can build three types of projects: static library, dynamic library and a program. for each one of them there are templates of all the build files. A few templates are the same for some or all project types, so an identical copy is kept for each one. The templates are organized in the following file system structure:

```
project_templates
  NetBeans_6
    dynamic_lib
      Makefile
      nbproject
        configurations.xml
        Makefile-Debug.mk
        Makefile-Impl.mk
        Makefile-Release.mk
        project.properties
        project.xml
    program
      Makefile
      nbproject
        ...
    static_lib
      Makefile
      nbproject
        ...
```

This regular structure mimics the structure of the build files inside a project directory and allows the generic part of ibs to apply the substitution dicts to the templates blindly and end up with the correct build file in the correct place. Note, the project.properties file that wasn't mentioned earlier. This is an empty file that doesn't seem to have a role in C++ projects, but I keep it there to be consistent with NetBeans.

To create the template files I simply took the various NetBeans build files and replaced anything that was project-specific (like the source files or list of dependencies) with a place holder.Let's examine a couple of template files. Here is the main part of the Makefile-Debug.mk of the 'program' project type:

```
# Link Libraries and Options
LDLIBSOPTIONS=${LDLIBSOPTIONS}

# Build Targets
.build-conf: $${BUILD_SUBPROJECTS} dist/Debug/GNU-${OperatingSystem}/${name}

dist/Debug/GNU-${OperatingSystem}/${name}: $${BUILD_SUBPROJECTS}

dist/Debug/GNU-${OperatingSystem}/${name}: $${OBJECTFILES}
        $${MKDIR} -p dist/Debug/GNU-${OperatingSystem}
        $${LINK.cc} -o dist/Debug/GNU-${OperatingSystem}/${name} $${OBJECTFILES} $${LDLIBSOPTIONS}

${CompileFiles}

# Subprojects
.build-subprojects:
${BuildSubprojects}

# Clean Targets
.clean-conf: $${CLEAN_SUBPROJECTS}
        $${RM} -r build/Debug
        $${RM} dist/Debug/GNU-${OperatingSystem}/${name}

# Subprojects
.clean-subprojects:
${CleanSubprojects}
```

The placeholder are expressions of the form **${Place holder}**. This is the format used by the **string.Template** class. Unfortunately, this convention is used by make files a lot too for environment variable, defined symbols and make variables. So, when a **$** sign is part of the make file it is escaped by additional **$** sign. For example, **$${MKDIR}** will not be treated as a place holder by the ibs.

Here is a simpler template of the project.xml file. It is just a bunch of XML with two place holders for the name of the project and its dependencies:

<?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://www.netbeans.org/ns/project/1"> <type>org.netbeans.modules.cnd.makeproject</type> <configuration> <data xmlns="http://www.netbeans.org/ns/make-project/1"> <name>${Name}</name> <make-project-type>0</make-project-type> ${MakeDepProjects} </data> </configuration> </project>

Testing the NetBeans Generated Build System

Bob was pleased with ibs and he decided to put it to the test. His plan to make sure it works on Mac OS X and on Linux (Kubuntu 9.04). For each target OS Bob generated all the build files using ibs and then built all the projects both from the command line using make and from the NetBeans IDE itself. Then he proceeded to run various tests and programs.

Generate all the build files

Generating the build files is as simple as launching the build system generator. The program displays some simple progress information as it goes through the different stages.

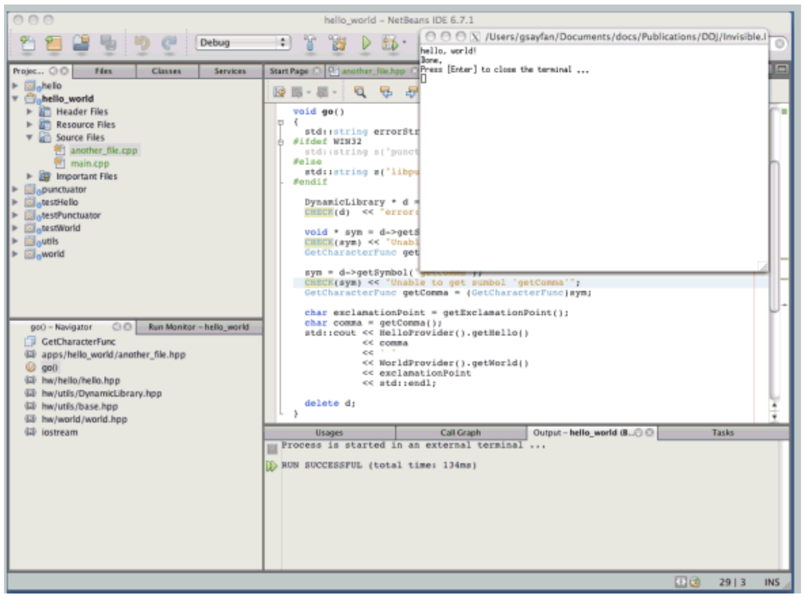
```
{root dir}/ibs >  py build_system_generator.py
--------------------
generate_build_files
--------------------
----------------------
_populate_project_list
----------------------
----
test
----
----
dlls
----
----
apps
----
--
hw
--
-----------------
generate_projects
-----------------
----------------------
_generate_project world
----------------------
--------------------------
_generate_project testWorld
--------------------------
----------------------
_generate_project utils
----------------------
----------------------
_generate_project hello
----------------------
---------------------------
_generate_project punctuator
---------------------------
----------------------
_generate_project utils
----------------------
-------------------------------
_generate_project testPunctuator
-------------------------------
----------------------------
_generate_project hello_world
----------------------------
--------------------------
_generate_project testHello
--------------------------
-------------
save_projects
-------------
------------------------
generate_workspace_files
------------------------
```

Bob did a quick sanity check to verify that the nbproject directory was indeed created for each project under the src directory:

```
{root dir} > find src -name nbproject
src/apps/hello_world/nbproject
src/dlls/punctuator/nbproject
src/hw/hello/nbproject
src/hw/utils/nbproject
src/hw/world/nbproject
src/test/testHello/nbproject
src/test/testPunctuator/nbproject
src/test/testWorld/nbproject
```

At that point Isaac (the sage) heard the good news and came into the room. He wanted to personally supervise on the testing of ibs, which will soon be responsible for building the entire "Hello World - Enterprise Edition" system.

The next stage was to actually build the system using the ibs-generated build files. For starters Bob fired up NetBeans on Mac OS X, built and ran the hello_world application. This caused all its dependencies to be built and finally the application ran in its little terminal window and indeed printed the vaunted "hello, world!" message (see Figure 2).



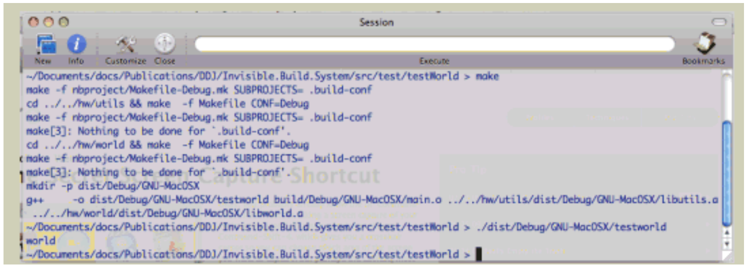
```
/Users/gsayfan/Documents/docs/Publications/DDJ/Invisible.I
hello, world!
Done.
Press [Enter] to close the terminal ...
```

```
void go()
{
  std::string errorStr
#ifdef WIN32
  std::string s("punct
#else
  std::string s("libpu
#endif

  DynamicLibrary * d =
  CHECK(d)  << "error:

  void * sym = d->getS
  CHECK(sym) << "Unabl
  GetCharacterFunc get

  sym = d->getSymbol("
  CHECK(sym) << "Unable to get symbol 'getComma'";
  GetCharacterFunc getComma = (GetCharacterFunc)sym;

  char exclamationPoint = getExclamationPoint();
  char comma = getComma();
  std::cout << HelloProvider().getHello()
            << comma
            << ' '
            << WorldProvider().getWorld()
            << exclamationPoint
            << std::endl;

  delete d;
}
```

```
Process is started in an external terminal ...

RUN SUCCESSFUL (total time: 134ms)
```

Figure 2

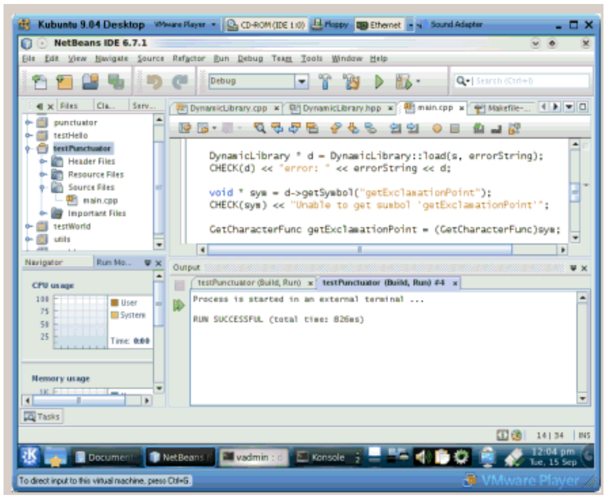
Isaac was duly impressed, but not fully convinced yet. He asked Bob how ibs can handle automated tests and running outside of the NetBeans IDE. Bob was more than happy to comply and demonstrated how the test_world program can be built using the standard 'make' command from a terminal window and then executed (see Figure 3).



```
~/Documents/docs/Publications/DDJ/Invisible.Build.System/src/test/testWorld > make
make -f nbproject/Makefile-Debug.mk SUBPROJECTS= .build-conf
cd ../../hw/utils && make  -f Makefile CONF=Debug
make -f nbproject/Makefile-Debug.mk SUBPROJECTS= .build-conf
make[3]: Nothing to be done for `.build-conf'.
cd ../../hw/world && make  -f Makefile CONF=Debug
make -f nbproject/Makefile-Debug.mk SUBPROJECTS= .build-conf
make[3]: Nothing to be done for `.build-conf'.
mkdir -p dist/Debug/GNU-MacOSX
g++     -o dist/Debug/GNU-MacOSX/testworld build/Debug/GNU-MacOSX/main.o ../../hw/utils/dist/Debug/GNU-MacOSX/libutils.a
 ../../hw/world/dist/Debug/GNU-MacOSX/libworld.a
~/Documents/docs/Publications/DDJ/Invisible.Build.System/src/test/testWorld > ./dist/Debug/GNU-MacOSX/testworld
world
~/Documents/docs/Publications/DDJ/Invisible.Build.System/src/test/testWorld >
```

Figure 3

Isaac commended Bob on a job well done, but Bob wasn't done. He knew that Isaac was an old Unix hand and he proceeded to demonstrate the ibs-generated build files on Kubuntu 9.04 (in a VM). First he built the libPunctuator project in the NetBeans IDe and then the testPunctuator project (see Figure 4).

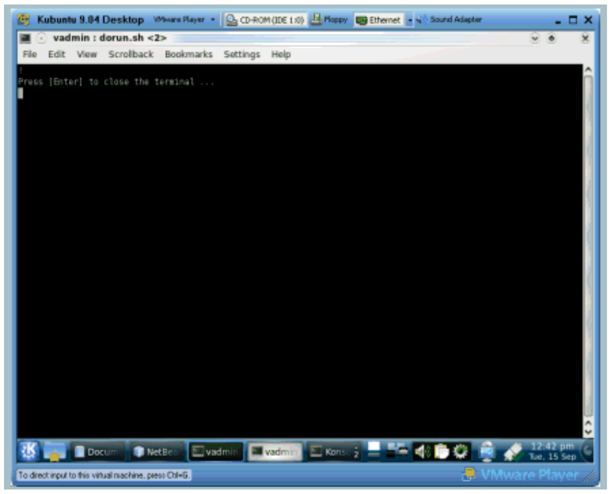


```
DynamicLibrary * d = DynamicLibrary::load(s, errorString);
CHECK(d) << "error: " << errorString << d;

void * sym = d->getSymbol("getExclamationPoint");
CHECK(sym) << "Unable to get symbol 'getExclamationPoint'";

GetCharacterFunc getExclamationPoint = (GetCharacterFunc)sym;
```

```
Process is started in an external terminal ...

RUN SUCCESSFUL (total time: 826ms)
```

Figure 4

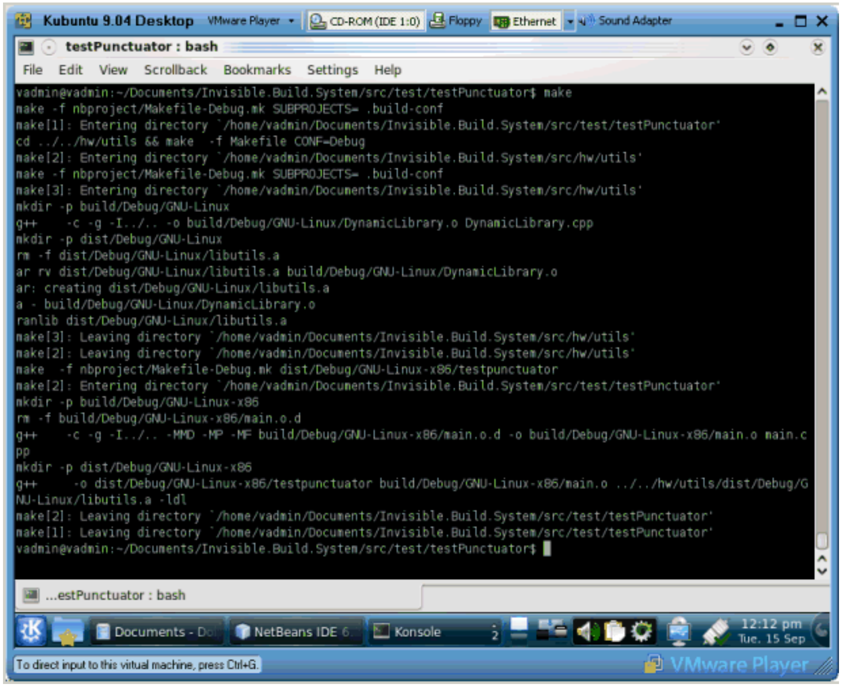
Bob was careful to copy the "libpunctuator.so" shared library to the directory of the testpunctuator program because the test always tries to load the shared library from the current working directory. They both noted with interest that on Kubuntu 9.04 programs run in an external terminal window (see Figure 5) as opposed to the internal NetBeans window on the Mac OS X.



```
Press [Enter] to close the terminal ...
```

Figure 5

Then, Bob demonstrated building from Kubuntu's terminal (see Figure 6).



```
vadmin@vadmin:~/Documents/Invisible.Build.System/src/test/testPunctuator$ make
make -f nbproject/Makefile-Debug.mk SUBPROJECTS= .build-conf
make[1]: Entering directory `/home/vadmin/Documents/Invisible.Build.System/src/test/testPunctuator'
cd ../../hw/utils && make  -f Makefile CONF=Debug
make[2]: Entering directory `/home/vadmin/Documents/Invisible.Build.System/src/hw/utils'
make -f nbproject/Makefile-Debug.mk SUBPROJECTS= .build-conf
make[3]: Entering directory `/home/vadmin/Documents/Invisible.Build.System/src/hw/utils'
mkdir -p build/Debug/GNU-Linux
g++    -c -g -I../.. -o build/Debug/GNU-Linux/DynamicLibrary.o DynamicLibrary.cpp
mkdir -p dist/Debug/GNU-Linux
rm -f dist/Debug/GNU-Linux/libutils.a
ar rv dist/Debug/GNU-Linux/libutils.a build/Debug/GNU-Linux/DynamicLibrary.o
ar: creating dist/Debug/GNU-Linux/libutils.a
a - build/Debug/GNU-Linux/DynamicLibrary.o
ranlib dist/Debug/GNU-Linux/libutils.a
make[3]: Leaving directory `/home/vadmin/Documents/Invisible.Build.System/src/hw/utils'
make[2]: Leaving directory `/home/vadmin/Documents/Invisible.Build.System/src/hw/utils'
make  -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux-x86/testpunctuator
make[2]: Entering directory `/home/vadmin/Documents/Invisible.Build.System/src/test/testPunctuator'
mkdir -p build/Debug/GNU-Linux-x86
rm -f build/Debug/GNU-Linux-x86/main.o.d
g++    -c -g -I../.. -MMD -MP -MF build/Debug/GNU-Linux-x86/main.o.d -o build/Debug/GNU-Linux-x86/main.o main.c
pp
mkdir -p dist/Debug/GNU-Linux-x86
g++     -o dist/Debug/GNU-Linux-x86/testpunctuator build/Debug/GNU-Linux-x86/main.o ../../hw/utils/dist/Debug/G
NU-Linux/libutils.a -ldl
make[2]: Leaving directory `/home/vadmin/Documents/Invisible.Build.System/src/test/testPunctuator'
make[1]: Leaving directory `/home/vadmin/Documents/Invisible.Build.System/src/test/testPunctuator'
vadmin@vadmin:~/Documents/Invisible.Build.System/src/test/testPunctuator$
```

Figure 6

Isaac felt his concerns melting away. He was now convinced that ibs is the way to go to make "Hello World - Enterprise Edition" the best hello world application on Microsoft Windows too on the way to [Hello] World Domination!

Conclusion

In this article you saw ibs in action, generating a full fledged NetBeans build system for a non-trivial system that involves multiple projects, static libraries, shared libraries, applications and test programs. ibs handled well multiple target operating systems (Mac OS X and Kubuntu 9.04) and allowed building and testing from the NetBeans IDE or externally from a terminal window. Bob demonstrated ibs successfully to Isaac his manager and in the next episode, Bob will try to make ibs build the "Hello World - Enterprise Edition" system on the Windows OS.