Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

A Build System for Complex Projects: Part 2

Ibs is an invisible build system that doesn't require any build files

August 04, 2009
URL:http://www.drdobbs.com/tools/a-build-system-for-complex-projects-part/219000123

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).

A Build System for Complex Projects: Part 1
A Build System for Complex Projects: Part 2
A Build System for Complex Projects: Part 3
A Build System for Complex Projects: Part 4
A Build System for Complex Projects: Part 5

This is the second in a series of articles that explores an innovative build system for complicated projects. Part 1 discussed build systems in general, problems with existing build systems in particular, and presented an ideal/invisible build system -- ibs. In Part 2, I dive into the internals of ibs and explain how it works.

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, and NetBeans projects. It is focused on C/C++ projects, but can be extended to additional languages and other projects types.

Architecture

Ibs has a generic core, an object model, and build-system-specific code (helper modules and templates). There is a clear separation between the common generation code and the build-system-specific parts. The object model includes the following classes:

- **Project**. The **Project** class is the unit of work. Each software system is just a list of projects with some dependencies between them. In the ibs world, a project always has an associated project directory and contains files that reside under its root directory, and possibly subdirectories too. **Project** maintains information such as the project name, its root directory, its dependencies, and its templates (see below). The files are discovered automatically.
- **Template**. Ibs can generate build files for multiple build systems. Each build system has its own set of build files. In some cases, it is just a single file such as a Visual Studio .vcproj file, and in other cases such as NetBeans nbproject, it can be a build directory that contains multiple files. Each project is associated with a set of templates that can be used to generate all the necessary build files.

  The **Template** class has a skeleton text file with placeholders and a substitution dictionary that can be applied to generate the target build file.
- **BaseHelper**. The generic part of ibs doesn't know anything about the particular build systems it can generate, but it knows what it needs to know. **BaseHelper** is a base class that encapsulates this meta-knowledge. For each build system you want to support, you need to write a proper helper subclass that can provide the necessary information along with specific build file templates.The helper is responsible for getting all the necessary templates, preparing the substitution dictionary, and collecting files for each project.
- **Generator**. The generator is the engine of ibs. It understands the regular directory structure and knows how to build every top-level directory and all its subdirectories. Once it populates a project list and generates proper build files for each project, it maintains a map of top-level directories to project [template] types.
- **Dependency Analyzer**. The dependency analyzer module is C/C++-specific. It analyzes **#include** files and calculates the dependency graph of the various projects. In general, the only dependencies (at the C/C++ level) are static libraries that must be built before they can be linked with executables and dynamic libraries that depend on them.

Implementation

It's time to look at the implementation of the core ibs. Let's examine the build_system_generator.py script that contains most of the object model, the DependencyAnalyzer.py module that contains the dependency analyzer, and the various project templates.

build_system_generator.py

Let's start with the **main** function. Its job is to parse command-line arguments and hand them over to the **generate_build_files**() function. There is currently only one command-line argument -- **root_dir**. The **root_dir** is the directory that contains ibs and the source tree. It defaults to the parent directory, which works if you run it from the ibs directory (where the script is located). In addition, the **save_projects** argument is passed. If it's True, the generated build files are saved. If it's False, the build files are generated but not saved, which is useful during development and debugging.

```
def main(argv):
  """Parse command-line arguments and call generate_build_files()
```

```
  """
  import getopt
  optionSpec = ['root_dir=']

  save_project_files = True

  root_dir = os.path.join(script_dir, '..')

  try:
    opts, args = getopt.gnu_getopt(argv, '', optionSpec)
  except Exception, e:
    print "Error parsing command line: %s" % e
    usage()

  if len(args) > 0:
    usage()

  for (option, val) in opts:
    if option == '--root_dir':
      root_dir = val

  # Convenience. Allows use of "~" in pathnames and other things.
  root_dir = os.path.abspath(os.path.normpath(os.path.expanduser(root_dir)))

  generate_build_files(root_dir, save_project_files)

if __name__=='__main__':
  main(sys.argv[1:])
```

The **generate_build_files()** function calls its nested **get_platform()** function to figure out what platform it is running on (using the Python "platform" standard library module). Based on the platform, it decides what build system to target (Visual C++ 2008 or NetBeans 6). In your build system you may want to employ different logic to determine the target build system or pass it from the outside as a command-line argument.

After the target build system has been determined, the appropriate helper module is instantiated. All the helper modules reside in the 'helpers' subdirectory and are named <build system>_helper.py. For example, the Visual C++ 2008 helper is "ibs/helpers/VC_2008_helper.py". Future articles will take a closer look at these helper modules and drill into specific build systems. This article stays at the generic build-system level.

The proper helper is imported dynamically via the **__import__** statement. This is a useful Pythonic technique to implement plug-ins. Note that additional target build systems can be easily added by moving the build system selection code outside and by simply adding more helper modules for the new build systems.

The next step is to select the project template's directory. The project templates for each build system reside in a sub-dir of 'project_templates' and are named after the build system. For example, the NetBeans 6 templates dir is: "ibs/project_templates/NetBeans_6".

A helper module is instantiated using an unusual syntax:

```
helper = getattr(helper_module, 'Helper')(templates_dir)
```

The **getattr()** call gets the **Helper** attribute from the current helper module. Every helper module contains a class called **Helper** that implements the helper interface. Then, the retrieved **Helper** class, which also doubles as a factory function/constructor, is invoked with the templates dir as an argument, resulting in an instantiated **Helper** object.

The helper is used to create a **Generator** object **g**. Several generator methods are now invoked in a sequence: **generate_projects()**, **save_projects()**, and **generate_workspace_files()**.

```
def generate_build_files(root_dir, save_project_files):
  def get_platform():
    arch = platform.architecture()[0][:2]
    operating_system = platform.system().lower()
    processor = platform.processor()

    if operating_system == 'darwin' and arch == '32' and processor == 'i386':
      return 'darwin86'
    elif operating_system == 'linux':
      return 'linux' + arch
    elif operating_system == 'windows':
      return 'win' + arch
    else:
      raise Exception('Unsupported platform: ' + platform.platform())

  title()
  # Dynamically import the build system specific helper module
  p = get_platform()
  build_system = 'VC_2008' if p.startswith('win') else 'NetBeans_6'
  helper_name = build_system + '_helper'
  sys_path = sys.path[:]
  sys.path.insert(0, os.path.join(script_dir, 'helpers'))
  helper_module = __import__(helper_name)

  templates_dir = os.path.join(script_dir, 'project_templates', build_system)
  # Instantiate the Helper class
  helper = getattr(helper_module, 'Helper')(templates_dir)

  g = Generator(root_dir, p, helper)
  g.generate_projects()
  g.save_projects(save_project_files)
  g.generate_workspace_files()
```

Let's examine the **Generator** class. The constructor stores the root_dir, the platform, and the helper module. The helper module is used to execute all the steps that require specific build-system knowledge. This is sort of a Strategy pattern, if you will. The final line calls the internal **_populate_project_list()** method.

```
  def __init__(self, root_dir, platform, helper):
    self.root_dir = root_dir
    self.libs_dir = os.path.join(root_dir, 'src/hw')
    self.platform = platform
    self.helper = helper
    self.ignore = helper.get_ignore_string()
    self.projects = {}
    self._populate_project_list()
```

The **_populate_project_list()** method gets the template objects for various project types from the helper (this is build-system-specific) and create a mapping between each top-level directory (hw, dlls, apps, and test) and its template object.

Then it starts drilling down the source tree starting from each top-level dir (aka parent dir) using the **os.walk()** function. It ignores .svn directories and sub-dirs that contain the special **helper.skip_dir** string. For each found sub-dir, a **Project** object is created and added to the project's list. This process demonstrates the flexibility of the build system generator. You can freely add new projects as long as they belong to one of the known project types and reside under the proper parent dir.

```
  def _populate_project_list(self):
    """Populate the list of projects to be generated
    """
    title()

    # Map project parent directories to templates
    static_lib_template = self.helper.get_templates('static_lib')
    dynamic_lib_template = self.helper.get_templates('dynamic_lib')
    program_template = self.helper.get_templates('program')
    mapping = dict(hw=static_lib_template,
                   dlls=dynamic_lib_template,
                   apps=program_template,
                   test=program_template)
    skip_dir = self.helper.skip_dir
    for parent_dir, templates in mapping.items():
      title(parent_dir)
      for project_dir, s, files in os.walk(os.path.join(source_dir, parent_dir)):
        # Treat only sub-directories directly under the parent dir
        # as projects (except .svn or skip_dir)
        if '.svn' in project_dir or \
           (skip_dir and skip_dir in project_dir) or \
           os.path.basename(os.path.dirname(project_dir)) != parent_dir:
          continue

        assert not project_dir in self.projects
        #print project_dir

        project = Project(project_dir)
        project.generated = False
        project.templates = copy.deepcopy(templates)
        self.projects[project.path] = project
```

Now let's take a look at the **Project** class -- a simple container to hold the project information. The most exciting thing it does is call the **DependencyAnalyzer** to get its dependencies and sort them.

```
class Project(object):
  """The project class stores all build information of a single project

  A project has a name, a directory, a list of project templates
  and dependencies (other projects) that must be built before the project itself.
  """
  def __init__(self, path):
    self.path = path
    self.name = os.path.basename(path)
    self.templates = {}
    dependencies = \
      DependencyAnalyzer.get_project_dependencies(path,
                                                  'hw',
                                                  [source_dir])
    self.dependencies = sorted(dependencies)
```

What's next? Generating the build files for all the projects. This is done by the **Generator**'s **generate_project()**, which iterates over all the projects and builds them one after the other by calling **_generate_project()**:

```
  def generate_projects(self):
    title()
    for p in self.projects.values():
      self._generate_project(p)
```

But what about the dependency graph? The projects are built in the order they were inserted into the list. In fact, the **_generate_project()** method is a pretty smart cookie and it builds all its dependencies before embarking on building the project it was asked to build. This method does the heavy lifting, so I'll review it piece by piece. First, it determines if the project was already generated. **project.generated** is a Boolean attribute initialized to False; when **_generate_project()** is called for the first time on a project, it is set to True. The check ensures that a given project will only be generated once. This is necessary because any given project may be a dependency of multiple projects.

```
  def _generate_project(self, project):
    title(additional=os.path.basename(project.path))
    if project.generated:
      return

    project.generated = True
```

In the next stage, all the source and header files associated with a project are collected. This process is build-system-specific, so the helper is invoked to perform it. If

the project has no source files or header files, there is nothing to do and it bails out. Then the path separators (either \ or /) are replaced with the helper's separator character.

```
    # Collect source and header files
    project.header_files, project.source_files = self.helper.collect_files(project.path)

    # Bail out on projects with no source and/or header files
    if project.header_files == [] and project.source_files == []:
      return

    # Make sure the paths are correct
    project.source_files = self._fix_path_separators(project.source_files)
    project.header_files = self._fix_path_separators(project.header_files)
```

The next stage ensures all the dependencies have been generated and iterates over them. If a dependency project hasn't been generated yet, the **_generate_project()** method is invoked recursively to generate it. The code doesn't detect circular dependencies at the moment, but it will not get stuck in an infinite loop because of the "generated" check. This is more an issue of code hygiene and might point to other bugs.

```
    # Make sure all the dependencies are in the project list
    for d in project.dependencies:
      assert d in self.projects

    dependencies = [self.projects[d] for d in project.dependencies]

    # Make sure all dependencies have been generated
    # This is recursive. It terminates as long as we have no circular
    # dependencies
    # @todo detect circular dependency
    for dep in dependencies:
      if dep.generated:
        continue
      self._generate_project(dep)
```

Finally, the project itself is generated. This means iterating over the project templates (there may be multiple build files per project, each with its own template), asking the helper to prepare a substitution dictionary based on various project parameters, its source and header files, and its dependencies list, and then applying the substitution dictionary to the template. The end result is that each template object now contains a textual build file.

```
    # Generate the project build files by creating a substitution dict
    # and applying it to each template file
    h = self.helper
    for t in project.templates:
      #print t.relative_path
      d = h.prepare_substitution_dict(project.name,
                                      t.template_type,
                                      t.template_file,
                                      project.path,
                                      self.libs_dir,
                                      dependencies,
                                      project.source_files,
                                      project.header_files,
                                      self.platform)

      d['Name'] = project.name
      d['name'] = project.name.lower()
      d['Platform'] = self.platform

      t.substitution_dict = d
      t.apply()
```

The ultimate goal of the project generation is to apply the substitution dictionary with all its values to the template object. Let's see how it works using the **Template class**. The **__init__()** method stores all the information, reads its template file, and stores it as a string. The **apply()** method uses Python's **string.Template** object to actually perform the substitution. We will examine what a template file looks like in the next installment.

```
class Template(object):
  """The template class has a template file and a substitution dict

  Applying the substitution dict to template file results in a final build file
  """
  def __init__(self, template_file, relative_path, template_type):
    """
    @param template_file - absolute filename of the template file
    @param relative_path - relative path of this build file from the project dir
    @param template_type - static_lib, dynamic_lib or program
    """
    self.template_file = template_file
    self.relative_path = relative_path
    self.template_type = template_type
    self.template_text = open(template_file).read()
    self.substitution_dict = {}
    self.build_file = ''

  def apply(self):
    # Generate A build file from a template file
    # (using PEP-292 string.Template substitution)
    # convert the template_text to string in case its unicode
    # (VisualStudio .vcproj files)
    t = string.Template(str(self.template_text))
    self.build_file =  t.substitute(self.substitution_dict)
```

All the projects have been generated and the template objects contain all the generated build files. It's time to save the files via the **Generator**'s **save_projects()** method. The method iterates over all the projects and within each project, it iterates over all the templates. It compares the generated build file against the exiting build

file (if one exists), and if it's different and the **save** argument is True, it writes the new/modified build file.

```
  def save_projects(self, save):
    title()
    for p in self.projects.values():
      # Save build file only if modified
      for t in p.templates:
        modified = False
        filename = os.path.join(p.path, t.relative_path)
        if '%s' in filename:
          filename = filename % p.name
        if os.path.exists(filename):
          text = open(filename).read()
        else:
          text = None
          modified = True

        if text is not None:
          if t.build_file != text:
            modified = True

        # Save build file file if necessary and different
        if modified and save:
          parent_dir = os.path.dirname(filename)
          if not os.path.exists(parent_dir):
            os.makedirs(parent_dir)
          open(filename, 'w').write(t.build_file)
```

If **save** is False and the build file was modified, the method goes into debugging mode and compares the existing build file against the generated file line by line and prints the results. This functionality is a lifesaver when developing generation code for a new build system. You will see it in action when I discuss the specific build systems in future articles.

```
        # Check generated file
        if modified and not save:
          dirname, basename = os.path.split(filename)
          gen_filename = os.path.join(dirname, 'gen.' + basename)
          open(gen_filename, 'w').write(t.build_file)
          real_lines = open(filename).readlines()
          gen_lines = open(gen_filename).readlines()
          if real_lines != gen_lines:
            assert len(real_lines) ==  len(gen_lines)
            for i in range(min(len(real_lines), len(gen_lines))):
              if real_lines[i] != gen_lines[i]:
                print i
                print "'%s'" % real_lines[i]
                print "'%s'" % gen_lines[i]
                print
          if sorted(real_lines) != sorted (gen_lines):
            print filename
            print
            assert False
```

Last but not least is generation of the workspace files. A workspace is just a collection of projects. It's called a Solution in Visual Studio and project group. It may or may not be trivial depending on the particular build system. The helper is doing all the work here.

```
  def generate_workspace_files(self):
    title()
    self.helper.generate_workspace_files('hello_world',
                                          source_dir,
                                          self.projects)
```

As you can see, the generic build system has a lot of logic that can be used to generate build files for any build system. The specific work for each build system is nicely encapsulated in the helper modules.

DependencyAnalyzer.py

Let's examine the DependencyAnalyzer.py module. The entry point is the **get_project_depenencies()** function. It accepts a project_dir, the static libraries dir, the include search path, and a list of extensions. It returns a list of projects that the current project depends on. The algorithm is pretty simple: Get the dependencies of each file in the project (that has the right extension) and prune duplicates.

```
def get_project_dependencies(project_dir, libs_dir, search_path,
                             extensions=['.cpp', '.hpp', '.c', '.h']):
  """Get all the projects in the libs dir that the target project depends on

The algorithm gets the dependencies of every file in the current project and keep a list of all the directories the files reside in.

  @project_dir: the target project
  @libs_dir: the name of the static libraries parent dir (e.g. 'nta')
  @search_path: the list of include directories
  @extensions: the list of file extensions that are checked for dependencies
  """
  files = glob.glob(os.path.join(project_dir, '*.*'))
  all_dependencies = []
  for f in files:
    if not os.path.isfile(f):
      continue
    if not os.path.splitext(f)[1] in extensions:
      continue
    file_dependencies = []
    get_file_dependencies(f, libs_dir, file_dependencies, search_path)
    all_dependencies += file_dependencies
```

```
  temp = [os.path.dirname(f) for f in all_dependencies]

  dependencies = []
  for p in temp:
    if not p in dependencies:
      dependencies.append(p)

  dependencies.remove(project_dir)
  return dependencies
```

How do you find the dependencies of a file? Via a recursive scan of its **#include** statements. There is a little bit more going on, but the whole point of the exercise is to figure out what projects you depend on so you can build them. Various system **#include**s that have already built libraries are irrelevant. So, the prefix serves as a filter to limit the search to **#include** statements that include the prefix.

```
def get_file_dependencies(filename, prefix, file_dependencies, search_path):
  """Get all the projects in the prefix dir that the target file depends on

  All the filenames it #includes are extracted using get_file_includes().
  The dependencies of each dependency are extracted recursively.

  @filename: the target filename
  @prefix: the prefix of interesting dependencies
  @file_dependencies: the list of dependencies (grows as the function trundles along)
  """
  filename = os.path.abspath(filename)
  if filename in file_dependencies:
    return
  else:
    file_dependencies.append(filename)

  text = open(filename).read()
  includes = get_file_includes(text, prefix, search_path)

  includes = [i[0] for i in includes if starts_with_prefix(i[0], prefix, search_path)]
  for i in includes:
    if i not in file_dependencies:
      get_file_dependencies(i, prefix, file_dependencies, search_path)
```

The **get_file_dependencies()** function uses the helper function **get_file_includes()** to get all the relevant **#include** statements. It uses a regular expression to match every line in the file. The regular expression is compiled using Python's **re** module and works with both double quotes (") and angled brackets (< >) around the included file. It can also handle leading whitespace and any following whitespace or comments. The regex also uses groups -- the parts in the expression surrounded by braces as in (.*). This allows the extraction of the interesting parts directly without further parsing of each line.

The entire function is a nice example for using regular expressions in Python. The result of a successful match is an object that contains a list of groups that are stored as a 3-tuple in the results and filtered according to the prefix.

```
# Pick up both #include statements
# Also take care of comments following the #include statement
include_re = re.compile('\s*#include [<"](.*)[>"](.*)')

def get_file_includes(text, prefix, search_path):
  """Get the filenames from #include statements in a text

  The text usually comes from a source file. If prefix
  is not empty it will return only include statements
  whose content (following the first quote or angle bracket)
  begins with the prefix.

  The algorithm is to extract the relative filename using a regex
  and then scan the search path an try to append the relative filename
  and see if it exists.
  """
  includes = []
  lines = text.split('\n')
  for line in lines:
    m = include_re.match(line)
    if m is not None:
      includes.append((m.group(1), line, m.group(2)))

  results = []

  for i in includes:
    if not i[0].startswith(prefix):
      continue
    for d in search_path:
      full_path = os.path.join(d, i[0])
      if os.path.exists(full_path):
        results.append((full_path, i[1], i[2]))
        break

  return results
```

Conclusion

This article discussed the architecture and implementation of the generic core of the ibs. It explored the implementation and demonstrated several interesting aspects of the architecture and the code: separating generic logic from custom logic using lightweight plug-ins (dynamically loaded helper modules), using templates (object can manage text files with placeholder and substitution dicts) to generate build files, and automatic discovery of dependencies using regular expressions to match **#include** statements. The next article will delve into the implementation of a specific build system (NetBeans 6) within ibs and demonstrate how the sage development manager Isaac and the dedicated Bob "the Builder" use it to build their enterprise "Hello, World!" system.

- A Build System for Complex Projects: Part 1