Software Products As Objects

Kevin Jameson
Realcase Software Research

Last Updated November 1997

Abstract

This paper describes our experiences in modeling entire software products (trees of software files) as objects. Container **pnodes** (product nodes) have user-defined Internet-unique names, data types, and methods (operations). Pnodes can contain arbitrary collections of software files that represent programs, libraries, documents, or other software products. Pnodes can contain multiple software products, so that header files, libraries, and program products may all be stored within one pnode. Pnodes can contain **views** that list other pnodes in order to form large conceptual structures of pnodes. Typical pnode-object **methods** include: fetching and storing into version controlled repositories; dynamic analysis of pnode contents to generate makefiles of arbitrary complexity; local automated build operations; Internet-scalable distributed repository synchronizations; Internet-scalable, multi-platform, distributed build operations; extraction and generation of online API documentation, spell checking of document pnodes, and so on. Since methods are user-defined, they can be arbitrarily complex. Modelling software products as objects provides a large amount of effort leverage, since one person can define the methods and many people can use them in extensively automated ways.

Contents

1 Introduction

This paper describes some results of our research into modelling entire software products as objects. This paper presents our conceptual models, summarizes our implementation architecture, gives examples of typical applications of our system, discusses some of the problems that we have solved, and lists some simple benefits of our approach.

Our experience over the past several years, has convinced us of the broad spectrum utility of this approach. This claim is based on an experience base that includes (to the nearest order of magnitude): 100K's of build cases, involving 1K's of pnodes, written in 10's of languages, stored in 10's of repositories, compiled on 10's of platforms, by 100's of users, at 10's of commercial sites across the Internet.

1.1 Terminology

Here are some definitions for terms used in this document:

- A **platform** is any unique combination of hardware, operating system, compiler, linker, compilation options, etc. In general, a platform is any set of unique processing options.
- A **product** is a named collection of software. Examples of products include libraries, binaries, collections of include files, documents, or whatever you can store in a computer filesystem hierarchy. Products may or may not be building blocks of larger products.
- A **pnode** (product node) is a physical filesystem representation of a product. Conceptually, pnodes are modelled as objects that have unique identity, a data type, internal meta-data, and a set of user-defined methods or operations. Physically a pnode is a collection of files organized in a traditional hierarchical filesystem directory structure.
- A **pnode method** is some computational operation performed on the pnode contents from the pnode level (as distinct from pointing to an individual file and operating on it alone). Examples of pnode methods include creation, checkout, makefile generation, automated builds, release operations, spell checks (on document pnodes only), static code analyses, and other user-defined operations.

2 Pnode Object Structure and Contents

Pnodes are physically organized as trees in traditional hierarchical filesystems. The minimum content of a pnode is a *pnode.dbi* (database information) file. In practice, there is also a hidden subdirectory that holds cache information for the repository tools that manipulate pnodes.

2.1 Pnode Directory and File Structure

Here is a sample pnode structure for the "Hello world" program after compiling on one platform named *sol25.plt*:

```
hello                             - top level directory
hello/.zzrcase                    - repository cache info directory
hello/.zzrcase/dbsfile.srv
hello/.zzrcase/dbsfile.anc
hello/s                           - source file directory
hello/s/hello.c                   - source file for hello.c
hello/pnode.dbi                   - pnode object meta-data file

hello/sol25.plt                   - build platform #1 (solaris 2.5)
hello/sol25.plt/makefile          - auto-generated makefile
hello/sol25.plt/makefile.pid      - auto-generated makefile
hello/sol25.plt/hello.o           - intermediate build files
hello/sol25.plt/hello             - final software product

hello/aix415.plt                  - build platform #2 (aix4.1.5)
...
hello/N.plt                       - build platform N
...
```

2.2 Pnode Meta-Data File (*pnode.dbi*)

Each pnode contains a meta-data file that specifies interesting information about the pnode and its contents.

Here is a DBI file from a the simple "Hello world" pnode. This file declares the pnode name, its authoritative home repository, and its data type. The product section of the DBI file defines the product name, type, and a short description of the product.

```
# $ Id: pnode.dbi 1.1 1997/11/27 20:31:16 irmsd Exp $

pnode                  hello
pnode_home             examples:rns.kwj.realcase.com:hello
```

```
pnode_type             rc_default_program

... other global pnode attributes
end_pnode

product                hello
product_type           rc_default_program
product_desc           A demonstration hello world pnode.

... other per-product attributes
end_product
```

Pnode types and product types have different names and functions. Coincidentally, the two user-defined types have the same name in this example.

For example, the pnode type defines (among other things) the build order of this pnode within a collection of other pnodes. This ensures that library pnodes are built before program pnodes that use the libraries.

2.3 Repository Name System (RNS) Names

Pnodes are given Internet-unique RNS names that contain an embedded Internet DNS (Domain Name System) name. The unique DNS name of the authoritative repository host machine is what gives RNS names their uniqueness.

A pnode RNS name functions for pnodes like an URL does for retrieving web documents—it provides an Internet-unique name for referencing the filesystem containing the pnode contents.

The structure of an RNS name is designed to be useful in multiple applications. For example,

- In our Internet Repository Management System (which provides distributed, version-controlled repository services), RNS names look like this: *Repository-Name:DNS-Name:Pnode-Name*.
- In our Internet Software Deployment System (which deploys software files into named directory trees), RNS names look like this: *Tree-Name:DNS-Name:/path/name/to/file/or/dir*.
- In our Internet Task Management System (which manages software change requests), RNS names look like this: *Task-Database-Name:DNS-Name:CR000112*.

The main benefit of the RNS system is that it provides a way to reference arbitrarily complex collections of software files in a location independent, platform independent way.

3 Pnode Object Methods And Operations

In an object modelling sense, one of the most noticeable things about pnode methods (operations) is that they are (and must be) supplied at runtime by external agents that are not part of the pnode object.

External "pnode method" agents can be simple programs that perform simple functions, or complex software subsystems that provide complex pnode manipulation services.

For example, here are some external agents that we use on our pnodes:

1. An MPCM (**Multi-Platform Code Management**) toolset consisting of tens of programs that together provide local pnode workspace services such as makefile generation, automated building, and automated installation into team or site installation directories.
2. An IRMS (**Internet Repository Management System**) that provides Internet-scalable version control, configuration management, and synchronized repository services for pnode objects.
3. An IABS (**Internet Automated Build System**) that provides Internet-scalable, multi-platform, multi-machine, multi-language, multi-* support for performing user-defined computational processes on pnodes fetched from remote repositories.
4. An ISDS (**Internet Software Deployment System**) that provides Internet-scalable software deployment services for products built from pnodes.

In all cases, external agents that manipulate pnode objects understand pnode architecture and content at an appropriate conceptual level for the tasks that are being performed.

4 Context Sensitive Knowledge Base

Much of the "intelligence" possessed by external method programs is stored in an IKBS (**Internet Knowledge Base System**).

The current knowledge base contains of approximately 1000 small files that specify information about pnode methods and processes and site/team/individual system option preferences. In addition, the knowledge base can contain an arbitrary number of reusable templates for program code, documents, or other reuse-oriented computer files.

The labor required to configure the system for default operation is not extensive. For example, the initial system configuration is straightforward, and can be completed in minutes by knowledgeable people. After initial configuration, reasonable amounts of customization labor are required to add reasonable amounts of custom knowledge.

The knowledge base contains (among other things) user-defined pnode object types, user-defined pnode product types, user-defined pnode object methods and processes, and user-defined program code templates.

The knowledge base is organized using five levels of specialization to partition knowledge by search rules, contexts, virtual platforms, accessing program, and database tag name. This allows users to control where to look (search rules), why to look (context), and what platform to look under (virtual platforms) when programs look up data.

One interesting aspect of the knowledge base is that it supports user-defined *contexts*, such as “debug”, “integration”, or “production”. Use of contexts permits users to change the behavior of pnode methods at runtime, simply by providing different context names for lookups in the knowledge base.

4.1 Drivers of Knowledge Base Complexity

We have spent almost a decade building and implementing a software productivity infrastructure based on pnode object models. Our experience has been that the three most important drivers of system complexity are scale, diversity, and customization.

Scale means “more of the same”, or “more of a similar kind of thing.” Scale is a quantitative effect. Increases in scale can often be treated by a corresponding increase in the capacity of existing mechanisms, such as more computing horsepower, more people, more network bandwidth, etc.

Diversity means “different stuff”, or “more of a different kind of thing.” Diversity is a qualitative effect. Increases in diversity cannot usually be treated with existing processing mechanisms. Instead, diversity usually requires the extension or modification of existing mechanisms to accommodate the new diversity.

Customization Diversity means that everything must have a default value, and one or more methods for customizing that value on a semi-permanent, context-driven, or per-invocation basis. Diversity in override methods is also important. Customization diversity is a qualitative effect.

Our experience has shown that at least hundreds of significant technical and automation issues can be found in the overall problem of managing large numbers of software components using pnode object models. For a longer list of related issues and solution strategies, see *The Multi-Platform Productivity Problem* `<http://www.realcase.com/papers/mpprob.html>`.

5 Example Applications

The pnode object model is very general, and can be usefully applied to many computational processes in the software development and IS industries.

Here are some sample applications that have been implemented using the pnode object model:

- **Local automated makefile generation.** The **getmakes** program analyses the content and structure of the pnode object, analyzes the current build context and system build environment, and then calculates a complete set of pnode-specific makefile command sequences to implement pnode methods.
- **Real Time Internet Builds Upon Checkin.** When a pnode object is modified and checked in to a remote repository, the repository detects the checkin and issues a rebuild order to the automated build system. The build system checks out the pnode object and runs a user-defined build/install procedure on the pnode in real time, sending a status report to the originator of the checkin.
- **Automated Nightly Baseline Rebuilds.** A view pnode (containing a list of RNS names of other views and pnodes) can be written to model arbitrarily large collections of software products. The entire view can be rebuilt from scratch using a single command to the automated build system. The build system can fetch pnodes from distant repositories and utilize distant machines for rebuilding if appropriate.
- **Automated Regression Testing.** Pnode views can be used to model sets of regression tests in the same way that they can be used to model collections of software programs. This approach is used to checkout, build, and perform regression tests on over 2000 small pnodes every night. The pnodes mentioned here each contain a header file, a library, a small client program, and a small server program.
- **Automated Release Building.** We use the pnode object model to build and package our standard software release. To the nearest order of magnitude, a single command to the automated build system builds and packages approximately 10,000 source files into 100 programs that run on 10 different platforms.
- **Automated Workstation Configuration.** In this example, pnode object models were used to solve the problem of setting up project-specific workstations with specific versions of in-house programs, databases, and third party software products. Once modelled, external pnode methods could reliably install the modelled software in only a few hours. The software typically included products such as Oracle, Sybase, and various collections of third party tools and libraries.
- **Geographically Distributed Development.** In this example, pnode objects and synchronized repositories were used to solve the problem of a distributed development

team. Master repositories were maintained in one city, and synchronized to another city in real time as checkins and repository modifications were made.

- **Simplified Development Operations.** The pnode object model makes it possible to use simple mouse clicks to perform large scale operations on large numbers of pnodes. Importantly, the knowledge burden that developers must maintain about software pathnames, compilers, options, and complex build processes is greatly reduced in areas modelled by pnode objects and pnode methods.

- **Automatic Generation of Websites.** As a final example, we have used the pnode object models and methods to model and build our whole website from one pnode view.

6 Simple Benefits

Here is a short list of simple benefits derived from the pnode object model:

- Software assets are

    1. modelled as pnode objects, in a uniform way.
    2. named individually, and stored in named repositories.
    3. managed in version-controlled repositories.

- Reuse is enhanced by unique pnode RNS names and convenient reuse tools.

- Extensive automation support for all user-defined pnode operations.

- Significantly enhanced operational simplicity for all common pnode operations such as create, checkout, build, checkin, and rebuild, release, deploy, and install.

- Management of distributed projects is made easier by automated synchronization of product repositories, and real-time automatic rebuilds of products as new checkins are made. This keeps all parties closely synchronized with each other in real time.

7 Sample Command Line Scenarios

The following set of typical commands shows the user-interface simplicity offered by pnode object models that are accompanied by intelligent pnode methods (tools). All these commands are valid in practice.

```
# create a new pnode in a remote repository and check it out locally
irms newprod kwj_papers:rns.realcase.com:pnode_as_object

# change the type of pnode to sgml document
# changes = change string
cd pnode_as_object
changes rc_default_unknown rc_doc_sgml pnode.dbi

# generate standard platform compilation directories for sgml pnode types
# xpp = expand platforms
xpp go

# instantiate a template sgml document
cd s
stub -l rcdoc pnode_as_object > pnode_as_object.sgml

# generate a makefile and compile the sgml into html, postscript, and dvi
# rcgo executes its command arguments in the proper platform directory
rcgo getmakes go
rcgo make all

# edit the document to contain desired material
<edit pnode_as_object.sgml and sub-sgml files>

# check for spelling errors, and fix them
rcgo make spell
<edit doc based on spell checker report>

# rebuild the doc into html, postscript, dvi to ensure things work
rcgo getmakes go
rcgo make all

# lock all files in the pnode for checkin
irms lock ::

# checkin all locked files, and all new files, ignoring .bak, etc files
irms ciu ::
<type in checkin msg interactively>

# rebuild and release the doc from scratch
rcgo make all release
```

The set of commands shown above is typical for pnodes of arbitrary size and complexity. We know of pnodes containing thousands of files that are processed with identical commands.

It is important that there are no language sensitive, platform sensitive, or process sensitive commands in the example above. All such information is hidden inside the pnode and knowledge base, where normal system users are not exposed to it. Their knowledge burden is thus significantly reduced.

The next example shows a different, yet still simple sequence of commands that we use to manually build our entire software release of approximately 200 programs and documents, on approximately 10 platforms.

```
# sample commands to manually build a release from a pnode view

# check out the view
irms corf releases::spi45

# expand and build the view by recursively checking out pnodes,
cd spi45
make v_expand_tree

# generate makefiles for the whole tree, on all supported platforms
rcgo -set all_unix make v_getmakes_tree

# sequentially visit all platforms and build/release the products
rplt make v_build_tree v_release_tree
```

This final example shows the IABS command that performs the release rebuild shown above (all products on all platforms).

```
# tell the IABS automated build system to build the entire release
# of all products and documents on all platforms
iabs releases::spi45 rebuild release
```

All of the pnode operations in the examples above are also available through a GUI interface, with the exception of the manual editing changes made to source files.

Using a GUI to perform operations such as these is initially quite striking for most people because so much can be accomplished with so few mouse clicks and 1 RNS name.

8 Conclusion

This paper has described a method for modelling whole software products as objects, and has summarized some of the benefits and applications of the model.

Our experience with pnode object models has convinced us of both the user-interface simplicity they provide for pnode operations, and of the extensive automation benefits to be gained by manipulating the pnode object with external programs (methods).

The modelling power of the pnode object model is extensive. User-defined pnode types, product types, and pnode operations can be used to effectively model software products of many different kinds.

Pnodes, and pnode views, can represent and store software collections of arbitrary size, in arbitrary languages, in arbitrary combinations, on arbitrary platforms, and so on. Pnodes can contain meta-data about the pnode contents, and can be conveniently manipulated using automated user-defined processes.

It is difficult to communicate in writing the look and feel of an infrastructure that is based on pnode object models such as the ones described here. Programmers can perform many common—and complex—development tasks by simply invoking a named operation (method) on an RNS name (a pnode object).

Importantly, the programmers can perform these operations without knowing the location of the source pnode, the language that the pnode source files are written in, the platforms that the pnode compiles on, the location or options for the local platform compiler, or any other such details.

The knowledge burden of these kinds of details is completely removed from everyone but the person that configures the knowledge base. This is a very significant economic benefit, directly proportional to the number of people that use the system.

We think that pnode object models are a good example of how advanced systems might model and manage millions of reusable software components and products in the future.

9 Further Information

For current information, see **www.realcase.com**. The website contains several other papers and the user manuals for the external pnode method systems mentioned in the paper. The author can be contacted at **jameson@realcase.com**.