Retail passion. Innovative solutions.

Retek® Predictive Application Server™ 11.1

Standards for Developing RPAS Extensions

Corporate Headquarters:

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
USA

888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000

Fax:
+1 612 587 5100

European Headquarters:

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax:
+44 (0)20 7563 46 10

Customer Support

Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Retek customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

Contact Method	Contact Information
E-mail	support@retek.com
Internet (ROCS)	rocs.retek.com Retek's secure client Web site to update and view issues
Phone	+1 612 587 5800

Toll free alternatives are also available in various regions of the world:

Australia	+1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus)
France	0800 90 91 66
Hong Kong	800 96 4262
Korea	00 308 13 1342
United Kingdom	0800 917 2863
United States	+1 800 61 RETEK or 800 617 3835
Mail	Retek Customer Support Retek on the Mall 950 Nicollet Mall Minneapolis, MN 55403

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step-by-step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

Contents

Introduction

From its inception, the 11.*x* series of the **Retek Predictive Application Server** (**RPAS**) platform has been designed to provide an enormous amount of flexibility for its customers. One of the features of RPAS that makes it flexible is its framework for handling extensions, in library or executable form, to its core functionality.

To foster the development of extensions to RPAS, Retek provides a software development kit (SDK) with each release of RPAS. This kit consists of a collection of C++-based include headers, shareable object libraries and documentation. Together, these development components enable the RPAS customers to develop, test and deploy additional behavior for their RPAS-based production environments.

However, great care must be taken to maintain the functional integrity and performance of the total solution when developing these extensions to RPAS. In fact, this development process should attempt, wherever appropriate, to mirror the development of RPAS itself as much as possible.

This document outlines the requirements and procedures for building extensions in a process conforming to Retek standards. In particular, this document lists and describes the software tools required for each platform, including instructions on how to procure and build them when necessary. It also covers the commands, variables and file structures of the RPAS build system. Finally, the document describes the contents of the RPAS distribution itself.

Summary of basic requirements

Proper skill sets

From the outset, an RPAS extensions developer needs to have a solid knowledge of the **C++** programming language in its modern, standardized form. In particular, familiarity with the C++ Standard Template Library (**STL**) is a must. The developer should also have a working knowledge of the utilities mechanisms provided by a **POSIX-compliant** command-line interface (**CLI**) environment to develop software. This CLI is supplied by default in most UNIX-based operating systems, and it can be added onto a Windows-based environment.

Installation of RPAS (11.0.4 or higher)

To develop extensions to RPAS, the RPAS client/server platform itself should be installed on one or more of the desired target platforms. Retek currently provides development support for RPAS versions 11.0.4 and higher.

A recent operating system platform

The development of RPAS itself takes place on relatively recent versions of the operating systems that Retek explicitly supports. Details of these operating systems are listed below:

Sun Solaris for SPARC

Retek supports and develops on **Solaris 8** for Sun's **SPARC**-based platforms. Please note that Retek cannot guarantee the behavior of RPAS extensions development on Solaris 9 or higher. Furthermore, Retek does *not* support any versions of Solaris for the Intel IA32 (x86) platform.

IBM AIX 5L

Retek supports and develops on **AIX 5L** (version 5.1) for IBM's **PowerPC**-based platforms. Retek cannot guarantee the behavior of RPAS extensions development on version 5.2 or higher of AIX.

Hewlett-Packard HP-UX 11i for PA-RISC

Retek supports and develops on **HP-UX 11i** (version B.11.11) for HP's **PA-RISC**-based platforms. Retek cannot guarantee the behavior of RPAS extensions development on other versions of HP-UX. Furthermore, Retek does *not* support any releases of HP-UX for the Intel IA64 (Itanium) platform.

Retek *especially* recommends the application of any HP-UX patches involving the dynamic linker/loader (**dld**). Failure to do so will prevent the successful execution of the RPAS server and related utilities.

Microsoft Windows 2000

While Retek does support **Windows 2000** (Win2K) as a development, testing and demonstration platform, it does *not* currently support Win2K as a *deployment* platform. Additionally, Retek cannot guarantee the behavior of RPAS, its extensions, or the development thereof, in Windows XP Professional or the Windows 2003 line of servers. Furthermore, Retek cannot recommend the development of RPAS extensions in Windows XP Home Edition, Windows NT or the MS-DOS-based operating environments (Windows 98/98SE/Me).

Access to a Korn shell command-line environment

UNIX platforms

All of the UNIX-based platforms that Retek supports have the Korn shell (**ksh**) installed by default. Furthermore, all of these platforms are set up by default to associate this particular shell with newly created user accounts.

Windows platforms

While most Windows platforms do not have a POSIX-compliant CLI installed, there are several add-on products, including the downloadable Windows Services for UNIX package from Microsoft, which can provide this feature. However, Retek can only guarantee the behavior of developing RPAS extensions using the Korn shell environment supplied with the **MKS Toolkit** from Mortice Kern Systems (www.MKS.com).

A suitable C++ compiler

UNIX platforms

For UNIX-based platforms, Retek uses and recommends **g++**, the C++ compiler that is included with the open source-based GNU Compiler Collection (**GCC**, http://GCC.GNU.org). Retek uses specific versions of GCC depending on the platform, and will only support these specific versions. As newer versions of RPAS are released, Retek may opt, at its discretion, to upgrade the version of GCC required to develop extensions. Retek cannot guarantee the behavior of extensions developed with versions of GCC more recent that those outlined below, and it is believed that those later versions would be incompatible with official RPAS releases.

Here are the versions required for each platform:

- Solaris – GCC 3.3
- AIX – GCC 3.3.2
- HP-UX – GCC 3.3

Windows platforms

For Windows platforms, Retek uses and recommends **CL.EXE**, the C++ compiler that is included with **Visual C++ 6.0**, an integrated development environment (IDE) from Microsoft. Visual C++ is available either as a stand-alone product or as part of **Visual Studio 6.0** (http://MSDN.Microsoft.com/vstudio/previous/vs6/). Retek cannot guarantee the behavior of extensions developed with the C++ component of Visual Studio. NET 2002 or 2003.

A suitable assembler (UNIX systems only)

The **g++** compiler depends upon a platform-specific assembler as part of its compilation process. For some systems, the system-provided assembler is required; for others, the GNU assembler (**gas**) is required. For the platforms needing **gas**, Retek uses and recommends the assembler that is included in version **2.14** of the GNU **binutils** collection. Here is a summary of what Retek requires each UNIX platform to have:

- Solaris – GNU (gas)
- AIX – system-supplied (as)
- HP-UX – GNU (gas)

A suitable linker

The C++ development paradigm requires the use of a system linker/loader utility to bind together information from C++ intermediate objects as well as static and dynamic libraries to create executable images and shared objects. The type of linker that Retek uses and recommends for development is platform-dependent. For some systems, the system-supplied linker is required; for others, the GNU linker (**gld**) is required. For the platforms needing **gld**, Retek uses and recommends the linker that is included in version **2.14** of the GNU **binutils** collection. Here is a summary of what Retek requires each platform to have in the way of linkers:

- Solaris – GNU (gld)
- AIX – system-supplied (ld)
- HP-UX – system-supplied (ld)
- Windows – supplied with Visual C++ (LINK.EXE)

The GNU Make (gmake) utility

The RPAS Makefile system is highly complex; for example, one of its capabilities is a run-time determination of the platform upon which a library or executable is being built. Because of the need to perform advanced build management functions, Retek depends upon GNU Make (**gmake**), an enhanced open source version of the standard UNIX-based build management utility **make**. Retek currently uses version **7.1** of **gmake** across all RPAS platforms. However, Retek does not guarantee the behavior of the extensions build process if version 8.0 or above of **gmake** is used in that process.

The RPAS Makefiles employ heavy usage of **gmake**-specific syntax. Therefore, legacy versions of **make** typically included in UNIX platforms (as well as the MKS Toolkit) will *not* suffice for RPAS extensions development. This same rule applies for the **NMAKE.EXE** utility included with Visual C++.

Installation of required utilities

GNU Compiler Collection

Introduction

The GNU Compiler Collection (GCC) was one of the first open source software projects in the world. It began its life as the GNU C Compiler, developed in the late 1980s in the laboratories of the Free Software Foundation. Retek chose GCC for its cross-platform support and consistency, adherence to open standards, and benefit-to-cost ratio.

Obtaining

Binaries

Even though the GNU Project does not distribute official binaries of their software products, several third-party organizations have stepped in to do so. These organizations provide installable binary packages of GCC for various platforms, making them available for free downloading or for purchase at a nominal cost. Some of the more popular vendors and distribution sites are listed at http://GCC.GNU.org/install/binaries.html. One site on that list, **SunFreeware**, carries binaries for several versions of Solaris (the website lists it as only carrying Solaris 2 binaries).

When obtaining the GCC binaries for your platform of choice, please be sure to heed the operating system and GCC version recommendations in the **Requirements** section of this document. If by chance, you are unable to find the version of GCC that suits your particular platform, you may need to install the most recent version of GCC that you can find for your platform in order to build the version your platform requires (see below).

Source

Retek recommends that pre-built binaries of the GNU utilities be installed wherever possible. However, there will be instances where this is simply not possible. For these cases, GCC will need to be built directly from the official GNU source code distribution.

Since GCC is an open source project, it can be built from scratch. However, GCC *does* require a C compiler to be installed and operable in order to build a bootstrapped version of GCC. The C compiler can either come from another version of GCC or a vendor-supplied development package. Please be aware, however, that some of the C compilers included with certain operating system distributions are based on pre-ANSI-standard notions of the C language. These types of compilers will prove insufficient for building the GCC bootstrap compiler.

Source “tarballs” (i.e., compressed file archives) of GCC are available directly from GNU or a mirror archive; see http://GCC.GNU.org/releases.html for more details.

Most source distribution sites make GCC available in two forms: a giant tarball with every programming language supported by GCC, and smaller individual tarballs representing each language component of GCC. Retek recommends downloading only the C (core) and C++ components to minimize the amount of time and storage space required to build the compilers.

Installation

Binaries

Most of the binary distributions of GCC are already packaged such that they can be easily installed by a particular platform's package management system (e.g., the HP-UX Depot mechanism). Exceptions to this are when the package is in a compressed archive (tarball) format. However, this type of package is also relatively easy to install.

Source

GCC includes a special script used to configure settings that affect the way GCC will be built. The script, named **configure**, works by taking a series of one or more user-supplied switches and parameters that, in turn, help the configuration script in its discoveries about the capabilities of the target platform. Please consult http://GCC.GNU.org/install/configure.html for more information on using this script.

In particular, Retek has determined that there are at least four (4) main categories of build configuration switches that will need consideration:

Installation target location

With the **--prefix** parameter, one can change the default location target of the installed GCC files. Alternately, each type of file directory (**bin**, **include**, **lib**) can be individually specified.

Linker and assembler

This part can be tricky: for the platforms that require a GNU-specific assembler and/or linker (see Requirements), the **--with-gnu-as** and **--with-gnu-ld** switches will be needed as appropriate. However, if a GNU utility is specified in the configuration but the compiler finds a non-GNU version of that utility, the compiler will get confused! If desired, one can use the **--with-as** and **--with-ld** parameters to lock in a particular variant of the appropriate utility.

Specific programming languages

When building from the GCC "full distribution" tarball, several other languages besides C and C++ (e.g., Java, Fortran, and Ada) will also be built by default! To avoid building these extra languages, the configuration script provides the **--enable-languages** parameter for specifying only the language compilers necessary for one's situation. For example, to build only C and C++, **--enable-languages=c,c++** should be specified.

Platform-specific switches

For AIX, these additional configuration parameters should also be specified:

```
--enable-shared --enable-threads=posix --disable-nls
```

For HP-UX, the **--enable-shared** configuration switch should also be specified.

Configuration

Configuring GCC, after it has been installed, is very easy. Only one environment variable absolutely has to be changed: the **PATH** variable, (preferably) prepended with the location of the GCC binaries. However, it is also generally a good idea to prepend the particular operating system's shared path variable with the location of the GCC shared libraries. The name of the variable depends on the platform:

- Solaris – LD_LIBRARY_PATH
- AIX – LIBPATH
- HP-UX – SHLIB_PATH

Usage

Once GCC has been properly obtained and installed, the **g++** compiler should be ready to go. While most development interactions will occur through the use of the gmake command, it is generally a good idea to ensure that **g++** is operable. A quick way to root out any problems is to enter the **g++ -v** command to retrieve version and configuration information on the GNU C++ compiler. It's also a good idea to log out of the current command shell and log back in so that configuration changes to the executable and library paths are assured of taking effect.

Visual C++ 6.0 (Windows-only)

Introduction

Visual C++ is the official C++ development environment for Microsoft Windows. In addition to providing a command-line compiler and linker, VC++ provides an IDE for debugging and testing. Retek chose VC++ for its close association with the Microsoft Windows platform.

Obtaining

Visual C++ 6.0, as well as the Visual Studio 6.0 product which includes it, are available from a variety of third-party software vendors. Please contact your vendor(s) of choice for more information. As stated before, Retek cannot guarantee the behavior of RPAS extensions development using any version of Visual Studio. NET.

Installation

The installation for Visual C++ is relatively straightforward. Since it is a Windows-based product, the installation procedure is completely menu-driven. The default installation settings in the various menus are perfectly acceptable choices, but additional items can also be installed if desired.

However, after installing Visual C++, Visual Studio 6.0 Service Pack 5 *must* be installed before development work can begin. This service pack is available on Microsoft's Visual Studio 6.0 support site (http://MSDN.Microsoft.com/vstudio/previous/vs6/downloads).

After Service Pack 5 has been installed, one other file needs to be updated. In the directory containing the Visual C++ non-MFC header files (e.g., **C:\Program Files\Microsoft Visual Studio\VC98\Include**), there is a header file named **xtree**. To successfully build RPAS-based extensions, the contents of this file must be replaced with an updated version from **Dinkumware** (http://www.Dinkumware.com/xtree.txt). This updated version should also be called **xtree**.

Configuration

After the Visual C++ installation has completed, please ensure that the system variables **PATH**, **INCLUDE** and **LIB** have been properly updated. These variables may be examined either in the system settings panel, accessible in **Settings->Control Panel->System->Advanced->Environment Variables**, or in the MKS shell as UNIX-style variables.

Usage

Since most of the build process will revolve around the use of **gmake**, it's generally a good idea ensure the accessibility of the command-line development utilities within an MKS Korn shell. To test, simply launch a Korn shell, then enter the **cl** and **link** commands with no arguments. If the responses that are generated do not indicate Microsoft copyrights, the **PATH** system variable will need to be changed. Specifically, the location of the Visual C++ executables (e.g., **C:\Program Files\Microsoft Visual Studio\VC98\Bin**) will need to be moved to the beginning of the variable's value.

The GNU assembler (gas) and linker (gld)

The GNU assembler and linker are part of the GNU Binutils project (http://Sources.RedHat.com/binutils). Binary distributions of these utilities are available from the same places as GCC. If these utilities absolutely have to be built from source, it should be considerably easier to do so with these utilities than with GCC.

Please consult the Requirements section above to determine if your platform requires one or both of these utilities.

GNU Make (gmake)

The GNU Make build management utility (http://www.GNU.org/software/make) is also available in binary form from the same sources as GCC. This utility has been stable long enough such that building from source should be unnecessary for most platforms.

Obtaining **gmake** for Windows, however, can be a bit trickier. Retek recommends obtaining **gmake** for the **Cygwin** environment (www.Cygwin.com). A possible complication of this Cygwin version is that it requires obtaining a minimal subset of the Cygwin environment itself. If one does go this route, please remember to save off the value of the SHELL environment variable that the MKS Toolkit installation sets. It is believed that the Cygwin installation will clobber this variable. Also, Cygwin installs the GNU Make utility under the name **make**, so a **gmake** alias that points to the Cygwin **make** will need to be added to a user's MKS environment.

Building extensions with the RPAS Makefile system

The development of RPAS extensions revolves around the usage of certain Makefiles that Retek has provided in each RPAS distribution. This section describes how to incorporate these Makefiles into your own project's **Makefile** to build your extensions in a standard, consistent manner.

Additionally, this section describes how to set up your development project's **Makefile** to include unit tests that aid in assessing the quality of the functional components for each extension you have written. Retek strongly recommends that the unit-testing paradigm be employed as an integral part of RPAS extensions development. Retek also strongly recommends the use of **CppUnit** (http://CppUnit.SourceForge.net) for the unit-testing framework.

Directory structure

Before setting up a **Makefile**, it needs to be stored in a directory. Retek specifically recommends placing each extension to be developed in its own development directory. If multiple extensions are being developed for a particular solution, each of these directories could reside in a parent directory, where a centralized **Makefile** could, in turn, recursively invoke the **Makefile** of each extension's subdirectory.

Retek also recommends creating, within each extension's development directory, two subdirectories: one that contains the source code for the extension, and another that contains the source code for any unit tests associated with the extension. The advantage to this method is that the same **Makefile** can manage the builds for both of these subdirectory types.

To clear up any confusion, here are some sample directory specifications to consider:

```
/home/janedoe/RPASSolution
/home/janedoe/RPASSolution/Extension1
/home/janedoe/RPASSolution/Extension1/src
/home/janedoe/RPASSolution/Extension1/unittest
/home/janedoe/RPASSolution/Extension2
/home/janedoe/RPASSolution/Extension2/src
/home/janedoe/RPASSolution/Extension2/unittest
```

Sample Makefile

The following text represents the contents of a **Makefile** for a typical extensions development situation where a shared library is being developed. The variables in this **Makefile** are described later in this section.

```
# Sample Makefile for a shared library build
RPAS_HOME=/vol.nas/u00/builds/release/rpas/11.0.4.3/aix_debug
SRC_ROOT=$(RPAS_HOME)/Source
TARGET = shared
TARGET_BINARY =
TARGET_SHARED = $(UT_LIB_PREFIX)srptmpl$(SHARED_LIB_SUFFIX)
UT_DIR = unittest

# include some architecture specific options
include $(SRC_ROOT)/Makefile.arch

INSTALL_SHARED_LIB_DIR = $(INSTALL_DOMAIN_LIB_DIR)

INCDIR += -I$(RPAS_HOME)/include \
          -
I/vol.nas/u02/replenish/srp/11.2/build/aix/Libraries/AIP/AliCommon \
          -I. \
          -I./srp

SHARED_RETEK_LIBS = $(LFLAG_PRE)rpas11$(LFLAG_POST)

OBJS = srp/ManageOrderStatusSRPTemplate.o \
       srp/ManageOrderReviewSRPTemplate.o \
       srp/CriteriaReview.o \
       srp/SrpTemplatesFactory.o \
       srp/SrpTemplatesVersion.o \
       srp/WHReview.o \
       srp/WHConf.o \
       srp/SelectWH.o \
       srp/SRPMaintTemplate.o \
       srp/SrpTwoTree.o \
       srp/SelectDays.o \
       srp/Utilities.o \
       srp/EnterDays.o

UT_RETEK_LIBDIR += $(L)$(RPAS_HOME)/lib
UT_RETEK_LIBS += $(LFLAG_PRE)rpas11$(LFLAG_POST)

UT_OBJS = unittest/SRPUTapp.o \
          unittest/WHReviewUT.o

include $(SRC_ROOT)/Makefile.rules
```

Environment variables

RPAS_HOME

This variable indicates the fully qualified directory location of the particular distribution of RPAS on the system where extensions are to be developed. If RPAS has been configured for use in your UNIX account or Windows command shell, this should already be set.

Variables for the Makefile

The following list of **Makefile** variables are used by the RPAS Makefile system to make the process of developing RPAS extensions relatively painless for the developer. Please refer to the **Makefile** example above for the proper assignment syntax of these variables.

RPAS_HOME

This variable serves the same purpose as the environment variable (see above for description). If this variable doesn't exist in the **Makefile**, the **gmake** command will use the environment variable. If the variable exists both in the **Makefile** and the environment, the setting in the **Makefile** will override the environment variable during the **gmake** invocation.

SRC_ROOT

This variable points to the directory location of the RPAS Makefile system files. This is typically set to **$(RPAS_HOME)/Source**.

TARGET

This variable can be set to one of two possible values: **binary** for executable images, and **shared** for shareable intermediate object libraries.

TARGET_BINARY

When the **TARGET** variable is set to **binary**, this variable should contain the base filename of the executable image target to be created.

TARGET_SHARED

When the **TARGET** variable is set to **shared**, this variable should contain the base filename of the shared object library target to be created.

When developing extensions, this variable should be set to **$(UT_LIB_PREFIX)*yourLibName*$(SHARED_LIB_SUFFIX)** , where ***yourLibName*** represents the unique, developer-chosen part of the library name. The prefix and suffix variables are automatically determined by the platform where the library is being built.

For example, if **ProductOpt** was chosen for ***yourlibname***, **gmake** will determine that the library name is **libProductOpt.so** on AIX and **ProductOpt.dll** on Windows.

UT_DIR

This variable should contain the directory containing source code for any unit tests that have been developed for the extension. This directory may be specified relative to the directory location of the **Makefile**.

INSTALL_SHARED_LIB_DIR

This variable should contain the directory location that a target shared object library file is to be installed in. For most development situations, this should be set to **$(RPAS_HOME)/applib**.

The system administrator for the development platform should configure this directory to allow developers to insert extension libraries (but *not* to overwrite the other libraries in the directory). If this is not possible, a private copy of the RPAS distribution will need to be created for this purpose.

INCDIR

This variable should contain a space-separated list of include directives, where each directive is of the form

-IincludeDirectory

where ***includeDirectory*** represents the location of a directory that the C++ compiler should use in looking for header files.

This variable should contain, at a minimum, the location of the header files included with the RPAS distribution. This location is typically **$(RPAS_HOME)/include**.

OBJS

This variable should contain a space-separated list of intermediate object filenames. The filenames may be specified relative to the directory location of the **Makefile**. The RPAS Makefile system uses this variable for determining both the source-to-object and the object-to-target dependencies.

UT_RETEK_LIBDIR

This variable should contain a space-separated list of library directives, where each directive is of the form

$(L)libDirectory

where ***libDirectory*** represents the location of a directory that contains libraries necessary for functionality of your unit tests (if you have them). This variable should only be appended to, and it should always contain **(L)(RPAS_HOME)/lib** as its first directive. It should also contain a directive representing the location of the directory containing the CppUnit library.

UT_RETEK_LIBS

This variable should contain a space-separated list of library directives, where each directive is of the form

$(LFLAG_PRE)libName$(LFLAG_POST)

where ***libName*** represents the core name of a library necessary for functionality of your unit tests (if you have them). This variable should only be appended to, and it should always contain **$(LFLAG_PRE)rpas11$(LFLAG_POST)** as its first directive.

UT_OBJS

This variable should contain a space-separated list of unit test intermediate object filenames (separate from the intermediate objects that comprise the extension library or executable). The filenames may be specified relative to the directory location of the **Makefile**.

Placement of Makefile includes

Makefile.arch

This include file heuristically discovers the platform on which **gmake** is being invoked, and it sets certain internal critical variables accordingly. This file should always be included *after* the setting of **RPAS_HOME**, **SRC_ROOT**, **UT_DIR** and the **TARGET*** series of variables. However, it should also be included *before* the setting of **INSTALL_SHARED_LIB_DIR**, **INCDIR**, **OBJS** and the other **UT_*** variables.

Makefile.rules

This include file generates the basic dependency rules and supplies the ***make*** invocation with its eventual target(s). It should always be included at the end of a **Makefile**.

Using the RPAS Makefile system

Once the development directory structure has been set up with the proper Makefiles to properly manage the build process, you will be ready to build your project!

To streamline the process of building, Retek has provided a series of **gmake**-based keywords to build each necessary component of your extensions' development, testing and deployment. These keywords drive the generation and placement of build-time products. The most common ways of using these keywords are summarized below:

gmake package all

This builds a version of the RPAS extension library or executable within the development directory for each extension. This type of build contains debugging information but no optimizations.

gmake package install

This is the same as above, except that it copies the library or executable to the default installation target directory, making permissions adjustments as needed. This type of build is suitable for problem resolution and normal testing.

gmake package release {all|install}

This builds an optimized-for-performance version of the RPAS extensions library or executable. This type of build is suitable for performance testing and deployment. It is not suitable for debugging since it contains no debugging information.

gmake package test

This builds any unit tests associated with the RPAS extension(s). The code specific to the unit tests is built with debugging information but no optimizations.

gmake clean

This removes any intermediate and final products of the build process, allowing for a complete build of the extension(s) and unit tests from scratch.

Anatomy of an RPAS distribution

This section describes the contents of the core RPAS distribution. Please refer to the directory listing relative to the setting of the **RPAS_HOME** variable.

applib

This subdirectory contains standard and custom RPAS extension libraries. These shared libraries are typically registered to an RPAS domain with a **regtemplate** or **regfunction** command. However, some of these libraries are actually support libraries upon which the registered libraries are directly dependent.

bin

This subdirectory contains all the RPAS-related executable binaries and shell scripts. These executables include several commands for creating, manipulating and querying RPAS domains. This directory also includes the two main RPAS server daemons (**DomainDaemon** and **RpasDbServer**).

domain

This subdirectory contains all the "bootstrap" files necessary for creating an RPAS domain from scratch. It includes the minimal set of necessary database files. It also includes several fixed-width flat files used to initialize the core hierarchies, dimensions, positions, and measures as well as a set of localized messages.

include

This subdirectory contains the header files necessary for the development of RPAS extensions. This directory itself has separate subdirectories for each functional cluster of header files. Retek recommends including only this directory (as opposed to each individual subdirectory) in an extension Makefile. The headers may be accessed with the standard C/C++ notation, e.g.,

```
#include <rpas/String.h>
```

The top-level **include** directory contains only one file, **FlexLexer.h**, a file that is normally included with **flex**, an open source-based lexical analyzer. This particular file has been modified by Retek to conform to its special development needs. Only the **Parser.h** header in the **rpas** subdirectory includes **FlexLexer.h**, which should never be *directly* included in RPAS extension code.

Since each **include** subdirectory covers a unique functional area, those directories are described further below:

boost

This directory contains header files from **Boost** (www.Boost.org), an open source-based set of extensions to the standard C++ libraries. While you might be able to use Boost headers that were installed on your development platform prior to the installation of RPAS, please be aware that Retek only supports the usage of version 1.30.2. Specifically, Retek cannot guarantee the behavior of RPAS extensions built with Boost version 1.31 or above. Also note that since Retek does not use or distribute any Boost-related libraries, Retek does not redistribute any of the Boost non-header source files.

rpas

This directory contains header files that, collectively, provide the RPAS API. There are several subdirectories within this directory as well.

rtkapp

This directory contains header files that provide the API for the **DynamicTemplate** class, which is used to construct an RPAS workbook template from a configuration file.

lib

This subdirectory contains the RPAS library as well as any libraries that it depends upon (including the Zlib compression library).

Source

This subdirectory contains the files that comprise the RPAS Makefile system.

Patches and release notes

As with any software release, Retek will provide, from time to time, a series of patches intended to add any critically needed functionality to RPAS as well as remedy any defects found within the release. A set of release notes that details these changes, including the impact on the application programming interface (API), necessary upgrades to compilers and other development utilities, and other aspects of RPAS behavior, will be included for each one of these patches. It is the responsibility of the RPAS extensions developer to ensure that their installation of RPAS has the latest changes applied and to change their RPAS extensions code accordingly. Retek will strive to minimize changes to the currently existing API, but there may be occasional circumstances when an element of the API has to be changed.

The RPAS API

Retek supplies RPAS with a C++-based application programming interface (API) suitable for creating custom extensions for an RPAS-based solution. This API specifically allows developers to create custom functions and procedures by using a number of publicly-available classes. While other supporting classes in the RPAS hierarchy may change interfaces, Retek intends to keep the classes listed herein as stable as possible, changing them only when absolutely necessary.

Note: All RPAS classes, templates and variables mentioned below are assumed to reside within the **rpas** C++ namespace and have a corresponding C++ header file in **$RPAS_HOME/include** unless otherwise specified.

Base classes

String

The **String** class is perhaps the most-used class in the RPAS API, as several methods of the other RPAS classes expect **String** parameters. Its functionality is a superset of the C++ STL class **std::string**. As an example, one can perform an "addition" operation on a **String** to achieve string concatenation:

```
String s1("This string");
s1 += " has been added to.");
```

In addition to the normal comparison and substring extraction methods, a number of utility methods for converting to and from other data types such as **int**, **double** and **bool** have been provided. There is also a method specifically geared for filling in the parameters of a locale-specific message label.

StringConvertException

This exception-handling class is intended for catching problems when converting a String object to/from another type, e.g.:

```
try
{
  int i = String("This is not a number").toInt();
}
catch (StringConvertException &e)
{
  // error-handling here
}
```

File

The **File** class encapsulates the lower-level, operating system-specific file descriptor handle, which in turn identifies a unique I/O session performed on a particular file or directory. Methods for opening, closing, reading, writing, and erasing the file are provided and typically encapsulate the operating system's file descriptor API. However, **File** goes a few steps further by providing methods that can list, or even copy, the contents of a file directory. **File** also provides several methods to lock, unlock, and check the lock status of the associated file.

FilePath

Typically associated with **File** (but also used by several other classes), the **FilePath** class encapsulates the fully-qualified name path for a file residing in a filesystem employing a hierarchical directory structure. This type of structure is indigenous in UNIX- or Windows-based operating environments.

A key advantage of the **FilePath** class is the automatic handling of the character sequence for separating directory hierarchies (e.g., '/' in UNIX-style environments, '\' in DOS and Windows). The **FilePath** class can also "flatten" a path, where shorthand references to a current or parent directory are taken out to create a fully-qualified filename. Additionally, several methods to manipulate the filename extension (e.g., "**.txt**") and query a file path's directory and base filename are provided.

ArgReader

The **ArgReader** class encapsulates the contextual list of arguments that were specified as part of the invocation of a new RPAS process. For example, when

loadMeasure –d /home/rpas/myDomain –meas TwPStrCnt

is invoked, the **loadMeasure** process can create an **ArgReader** object that will hold the –d and –meas parameters and allow easier access to them that that afforded by the traditional C-style **argc**/**argv** method. The ArgReader class can be further used to stipulate the valid parameters and switches (which are parameterless directives such as **–noWarn**) as well as dictate which of those *must* be specified on the command line.

Though much of the argument processing logic (aside from that specified above) must be performed by the creator of the ArgReader object, ArgReader does automatically handle one type of argument: the **–loglevel** class, which specifies the level of granularity for statements generated through the **Logger** class (see below).

InvalidArgException

This exception will be thrown if the **ArgReader** object notices a parameter or switch in the process's parameter list that has not been previously registered with the **ArgReader** object.

DateTime

The **DateTime** class encapsulates the operating environment-specific API for representing a specific point in time (timestamp) down to the millisecond. After the **DateTime** object has obtained a valid value, it may be manipulated by special accessor methods that can adjust the timestamp on any level of granularity, from years down to milliseconds. **DateTime** objects may also be compared for equality and chronological order, and they may also be queried for leap-year status.

A DateTime object, by default, captures the current value of the system's internal clock. However, DateTime objects are also frequently used as RPAS position values.

DateTimeComponents

This companion class to **DateTime** allows a developer to access, in both a read-only and a read-write fashion, the individual date and time elements that comprise a **DateTime** timestamp. The various levels that can be accessed are: year, month, day within the month, hour, minute, second, and millisecond. A **DateTime** object can generate a **DateTimeComponents** object, and *vice versa*.

DateTimeFormat

The **DateTimeFormat** class, also a companion class to the **DateTime** class, formats a **DateTime** timestamp and generates a character **String** value suitable for display to users and for parsing by back-office batch processes. A suitable default format is provided. However, developers can provide alternate formats to further refine the display ability of the timestamp.

The timestamp display format is specified as a **String** object containing one or more of the following placeholder tokens into which the appropriate timestamp values are substituted. All values are formatted as two digits unless otherwise specified:

- **%Y** Four-digit Gregorian calendar year
- **%y** Two least-significant digits of year (e.g., **04** is 2004)
- **%m** Month (e.g., **01** is January)
- **%B** Varying-length full name of the month
- **%h** Three-character abbreviation of the month name
- **%d** Day of the month (e.g., **01** is the first day of the month)
- **%H** Hour of the day (e.g., **22** represents 10 PM)
- **%M** Minutes past the hour
- **%S** Seconds past the current minute
- **%s** Three-digit milliseconds past the current second
- **%%** Treated as a single-character percent literal

The format string can also contain non-token characters to enhance the readability of the formatted string.

For example, a format string of

```
%D %h %y
```

will yield the formatted string **02 Jul 04** for a timestamp occurring on July 2, 2004, and a format string of

```
%m/%D/%Y %H:%M:%S.%s
```

will yield the formatted string **07/02/2004 10:30:00.001** for a timestamp of one millisecond past 10:30 AM on the same day.

Logger

The **Logger** class, or more specifically, the **Logger** singleton object accessor **rpas::logger()**, can be used to generate non-interactive, "back-channel" feedback regarding the behavior and/or performance of one or more sections of C++ code. The logger() accessor behaves like an output stream, so it can initiate a chain of stream insertion operations, e.g.:

```
logger() << debug << "The query returned" << i
         << endl;
```

Note the **debug** operand at the beginning of the insertion operation. This is one of several logger stream modifiers available. They allow developers to specify the granularity of the log message (basic, none, error, warning, information, debug, and profile with basic and **none** for always-logged to **profile** for logging at the deepest logging level). For controlling the formatting of the output stream all standard stream modifiers are supported.

RpasVersionMacros.h header

Various C-preprocessor-style macros are defined in the header file RpasVersionMacros.h so that libraries and binary executables will have a foolproof mechanism for self-discovering the version of RPAS they were built with. There are two different constructors for the RpasVersion object. One of the constructors is designed for the Rpas Library itself and does not include the parameter show below that contain additional in their name. For libraries and code written outside of Rpas that requires additional version information, this additional information can be provided as show in the example. The parameters to the constructor that have additional in their name provide the version information for the library or code written outside of the Rpas while the other parameters provide the Rpas library version this library or code was complied against.

To include this information into a custom extension library, a C++ source file with the following contents must be included (the developer can substitute more appropriate text in the “name of your library here” space):

```
#include "rpas/RpasVersion.h"

namespace
{

static rpas::RpasVersion

   rpasVersion
   (
      "name of your library here",
      ADDITIONAL_VERSION_MAJOR,
      ADDITIONAL_VERSION_REVISION,
      ADDITIONAL_VERSION_MODTIME,
      ADDITIONAL_VERSION_BUILD_DATE,
      RPAS_VERSION_MAJOR,
      RPAS_VERSION_REVISION,
      RPAS_VERSION_MODTIME,
      RPAS_VERSION_BUILD_DATE,
      RPAS_VERSION_CLIENT_API,
      RPAS_VERSION_MSPL_API,
      RPAS_VERSION_COM_MSG_FORMAT
   );

}


extern "C"  const rpas::RpasVersion&  __LibraryVersion()
{
   return rpasVersion;
};
```

To display the version information in a binary, the code would look something similar to this:

```
if (args["version"] == ArgReader::TRUE_ARG())
{
   RpasVersion  rpasVersion
   (
      "name of your binary here",
      ADDITIONAL_VERSION_MAJOR,
      ADDITIONAL_VERSION_REVISION,
      ADDITIONAL_VERSION_MODTIME,
      ADDITIONAL_VERSION_BUILD_DATE,
      RPAS_VERSION_MAJOR,
      RPAS_VERSION_REVISION,
      RPAS_VERSION_MODTIME,
      RPAS_VERSION_BUILD_DATE,
      RPAS_VERSION_CLIENT_API,
      RPAS_VERSION_MSPL_API,
      RPAS_VERSION_COM_MSG_FORMAT
   );

   logger() << basic << endl << rpasVersion
            << endl;

   return 0;
}
```

RPAS Functions

Standard Functions

See the *RPAS 11 Rule Functions* document for a description of standard functions.

Custom Functionality

RPAS supports five techniques for implementing custom functions and procedures. This section contains a brief summary of each technique.

Derive from ExpressionFunction<T>

- Very simple to implement, because there is just one class to create.
- Insulates you from working directly with the RPAS calculation engine.
- Good for data-driven functions working with a single data type.
- Supports full flexibility to use other operations in the same expression
- Supports vector mode and/or dropping the calendar dimension (function with vector arguments returning non-vector result)
- No access to underlying intersection or dimension information.
- All arguments and the return value must be the same type.
- An example follows below:

```
class SliceVarFn : public ExpressionFunction<double>
{
public:

   SliceVarFn()
   {
   }

   virtual ~SliceVarFn() { }

   virtual rpas::String functionName() const
      { return "ut_slicevar"; }
   virtual int minArgCount() const
      { return 2; }
   virtual int maxArgCount() const
      { return 2; }
   virtual ExpressionFunctionBase* clone() const
      { return new SliceVarFn(*this); }
```

```
      virtual bool canDropCalendarDimension() const
         { return true; }


      virtual void eval

      (  const typename ExpressionFunction<double>

         ArgumentVectorType& args,
               typename ExpressionFunction<double>

         ResultVectorType& results

      ) const

      {

         // Just populate the first result. Typically with
         // this function we will be dropping the
         // calendar dimension anyway.

         results.at(0) = args.at(0).at(static_cast
            <SimpleVectorWrapper<double>::size_type>(
               args.at(1).at(0)));


      }


      virtual SemanticInformation::EvaluationModeType
         evaluationMode() const

      {

         return SemanticInformation::VECTOR;

      }

   };
```

Derive from DynamicExpressionFunction

This is a specialized version of ExpressionFunction, which supports mixed types for arguments and return values. Types may be determined at runtime.

- Can be used to implement simple function name overloading. For example, you could write a ‘max’ function which returns reals if you pass it reals, strings if you pass it strings, etc.
- Functions coded this way run slightly slower than the equivalent function using ExpressionFunction (typically 10-15% slower)
- An example follows below:

```
class DecompressMeasureFn : public
   DynamicExpressionFunction

{

public:


   DecompressMeasureFn() { }

   virtual ~DecompressMeasureFn() { }
```

```
virtual String functionName() const
   { return "ut_decompressMeasure"; }
virtual int minArgCount() const
   { return 1; }
virtual int maxArgCount() const
   { return 1; }


virtual MeasureType returnType() const
{ return MeasureType:: real(); }


virtual bool convertType(int argIndex)
{ return false; }


virtual ExpressionFunctionBase* clone() const
{
   return new DecompressMeasureFn(*this);
}


virtual bool checkArgType(
   const MeasureType& argType,
   const int argNum)
{
   switch(argNum)
   {
      case 0:
         // Nothing to check - we can handle any
         // type.


         // Remember the type of argument 1 – this
         // will be the return type.
         _returnType = argType;
         break;
      default:
         return false;
   }
   return true;
}
```

```
    virtual void eval
    (
      Const
      DynamicExpressionFunction::ArgumentVectorType&
      args,
      DynamicExpressionFunction::ResultVectorType&
      results
    ) const
    {
       // Loop over argument 1, populating the
       // corresponding entries in the
       // output vector with the last non-error value
       // entry from the input.
       ArrayCell lastNonErrorValue = errorValue();
       for (ArgumentType::size_type i = 0;
            i < args.at(0).size();
            ++i)
       {
          ArrayCell currentValue = args.at(0).at(i);
          if (currentValue != errorValue())
          {
             lastNonErrorValue = currentValue;
          }

          results.at(i) = lastNonErrorValue;
       }
    }

    virtual SemanticInformation::EvaluationModeType
    evaluationMode() const
    {
       return SemanticInformation::VECTOR;
    }
};
```

Derive from ParseNode/EvalNode

This is the low-level RPAS API for implementing functions. It is more powerful, but is also more complex.

- Has access to intersection and dimension information as well as the error and ignore channels (for example, to report an error that is trapped by a prefer statement).
- Like ExpressionFunction, functions written this way can be combined with other operations in an expression.
- Should only be used if ExpressionFunction and DynamicExpressionFunction don't meet your requirements.
- The RPAS "first" keyword is implemented this way. See /rpas/11/AppLibs/RpasFunctions/rpas/IndexFirst*.h and IndextFirst*.cpp.

Derive from MultiResultFunctionParseNode/MultiResultFunctionEvalNode

This is similar to ParseNode/EvalNode but intended for implementing functions that return multiple results. Supports multiple return values, one primary and any number of secondaries. This is a very efficient way to compute groups of closely related values within a single function, rather than having one single-valued function per result.

- Each return value can be a different type. Some return values can have a calendar dimension and some not. Computation of each secondary result can be individually controlled simply by changing the expression text (using named arguments).
- Although not fully automated, a helper function assists in implementing vector mode and dropping the calendar dimension.
- Can be partially combined with other operations in an expression (each function argument can be an expression but result must be assigned directly to the LHS).
- Three examples are available:

Function	Location and description
sumdiff	/rpas/11/RpasTestUtility/rpas/SumDiff*.h and SumDiff*.cpp. Very simple multi-result function.
getlastdigit	/rpas/11/Libraries/Expression/unittest/GetLastDigitExpr.cpp. Multi-result function returning three results, each of a different type.
priceprofile1/priceprofile2	/rpas/11/Libraries/Expression/unittest/PriceProfileExpr.cpp: Two samples in one, both multi-result functions returning three results, some with a calendar dimension and some without. priceprofile1 drops the calendar dimension for its primary result. priceprofile2, which derives from priceprofile1, doesn't.

Implementing a function using ParseNode/EvalNode

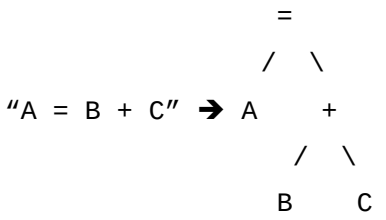
At the most complex, the implementation of a function for RPAS 11.0 requires implementing four classes deriving from:

- ParseNodeFactory
- ParseNode
- EvalNodeFactory
- EvalNode.

To understand which of these interfaces you need to implement, a basic understanding of the life cycle of an expression is necessary. All expressions in RPAS 11.0 start as strings (for example, "A = B + C" or "A = lag(B)"). The Parser object parses these strings and if no syntax errors are encountered a Parse Tree is built. This tree is then checked for semantic errors and if none are found the Expression is considered to be valid. A valid expression may then in turn be evaluated in either full (that is, data load/batch mode) or incremental (workbook update) mode. Either way the ParseTree is fed through a collection of EvalNodeFactories and an Eval Tree is generated, which in turn is used to perform the calculations.

For example, assume that A, B, and C are real valued measures in a domain with a "sku_str_week" intersection.

The expression "A = B + C" is transformed into a parse tree



where "=" is an AssignParseNode with two children: a VarParseNode for measure A and a BinaryOpParseNode for addition. The BinaryOpParseNode also has two children: VarParseNodes for measures B and C. Once this tree is built from the expression string, the recursive legalSubTree algorithm is run on it to detect semantic errors. This algorithm is a recursive, post-order tree traversal of the tree that stores the intersection and the type of the node before return.

In the above example, the legalSubTree algorithm is executed on the AssignParseNode that then calls the legalSubTree function on the VarParseNode for A. This node retrieves the type and base intersection of measure A and returns *true* (that is, successful validation); if measure A did not exist, the function would have returned *false*. Upon the return from VarParseNode A, the AssignParseNode calls the legalSubTree function on the BinaryOpParseNode for +, which calls the legalSubTree function on the VarParseNodes for B and C. Each of these stores its intersection and type, retrieved from the MeasureStore that holds them and returns *true* (since they exist). Upon the return from the VarParseNode's legalSubTree function, the BinaryOpParseNode queries them for their intersections and types. If the types and the intersections are compatible (that is, they are the same type, and one intersection is above or equal to the other), then the BinaryOpParseNode stores its intersection and type and returns. Finally, the AssignParseNode queries the BinaryOpParseNode and the VarParseNode for A for their types and intersections and if they are valid (that is, they are the same type, and intersection A is below or equal to intersection of BinaryOpParseNode +) it returns *true*.

When the calling expression receives a true from the legalSubTree function it knows that:

- The sub-tree can be legally evaluated at the base.
- Each node in the tree knows its type.
- Each node in the tree knows its intersection.

Then, when the expression is asked to evaluate at some intersection, two phases occur in the tree. The first uses the parse tree's general-purpose query device to determine which instances of the measure need to exist to allow calculation.

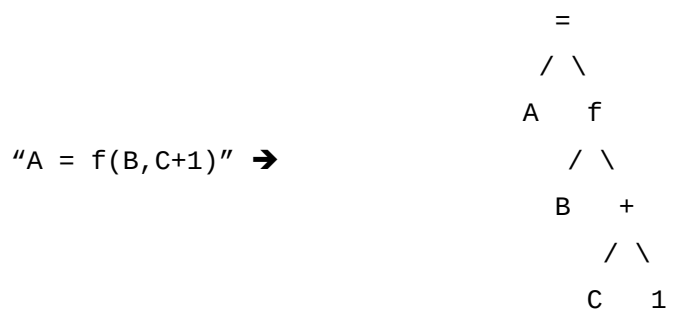
Once these arrays are generated, the parse tree is used by the Expression class' buildEvalTree method to build a tree to evaluate this expression at the given intersection. The resulting eval tree exists only for the length of the evaluation and then is discarded. The tree is specific to the expression and the intersection at which to evaluate the expression.

Forming the tree occurs by invoking the getEvalNode function of the ParseNode. This function uses the stored EvalNodeFactory pointer to construct an eval node, passing itself as an argument to the factory. Each factory invokes the call on the children of the node to get their EvalNode implementations and finally returns the EvalNode for the ParseNode that invoked the method.

Continuing with the example above, the getEvalNode is called on the AssignParseNode (Note: This is handled by the root class ParseNode and should not be overridden). This function calls the AssignEvalNodeFactory function getEvalNode using the pointer stored in the ParseNode from the AssignParseNode's chaining constructor. The factory then calls getEvalNode on its children (VarParseNode A and BinaryOpParseNode +), which use their factories and recursive invocation to build the eval tree.

All nodes in the eval tree are descendants of the EvalNode class, which implements two principle recursive tree functions: eval and evalNA. Each node also allocates a buffer of strongly typed space to be used to hold the value of the calculation at that node. Each node should be able to run in Single Cell mode (calculates one cell at a time) or Vector mode (calculates a vector of cells at a time). Evaluation takes place using similar recursive techniques. Both algorithms descend the tree traveling to the leaves of the tree (which are either constants or VarEvalNodes), setting those EvalNode's buffers to the correct values. Upon the return of all a node's children's eval calls, the parent calculates its value based on the children's calculated values, stores this result in its buffer, and returns. Finally, the AssignEvalNode stores the result in the left hand side's VarEvalNode, and eventually in the array on disk.

Given this overview of the processes of the expression library, it is easy to see how a function fits into the language. A function will become an internal node, or in some rare cases a leaf, in both the parse and eval trees. Consider the expression string "A = f(B,C + 1)", it parses as follows:



Note the following about function *f*:

- This example takes two arguments. A function can take, 1, 2, … arguments.
- Any type of ParseNode, including other functions, can be a child.

The following table describes the interfaces used to write your own function.

Interface	Description
ParseNodeFactory	This interface must be implemented to allow the parser to recognize your function and create ParseNodes of the appropriate type. The factory stores the name of the function. The factory is registered using a static interface on the Parser class.
ParseNode	This interface interacts with the expression to provide information about the function before evaluation time. The ParseNode stores a pointer to the EvalNode factory. legalSubTree must be implemented.
EvalNodeFactory	This interface converts a ParseNode into a corresponding specific EvalNode. It should allocate the eval node .Eval nodes are often, but not always, implemented as templates (depending on whether they need to support multiple types).

EvalNode	This interface is used to calculate actual values.
	It should allocate and de-allocate appropriate sized buffers.
	Its computed results should match the buffer type specified when the eval node was initialized.
	It should implement eval and evalNA.
	Note: Custom functions are not allowed to generate an "ignore" condition—only the built-in RPAS "if" function can do this. But custom functions *are* responsible for propagating the ignore channel from their child nodes (that is, the eval nodes representing the arguments to the function).
	Important: When an EvalNode is destroyed, it is responsible for deleting its child eval nodes. For example, from RpasFunctions:
	`~AttributeEvalNode()`
	`{`
	`delete _child;`
	`}`

Implementing a multi-result function (MultiResultFunctionParseNode/MultiResultFunctionEvalNode)

Similar to ParseNode/EvalNode-based functions, multi-result functions are created by creating four classes:

Interface	Description
ParseNodeFactory	<Same as ParseNode/EvalNode-based functions.>
MultiResultFunctionParseNode	This interface interacts with the expression to provide information about the function before evaluation time. MultiResultFunctionParseNode derives from ParseNode. In addition to its ParseNode-related responsibilities, implementations of this interface must implement: • validateSecondaryResultLabel: This function checks the LHS label to make sure it matches one of the expected label values.
MultiResultFunctionEvalNodeFactory	This interface converts a ParseNode into a corresponding specific EvalNode. MultiResultFunctionEvalNodeFactory derives from EvalNodeFactory. • It should allocate the eval node .Eval nodes are often but not always implemented as templates (depending on whether they need to support multiple types). • Note that the “factory” function is named buildMultiResultFunctionEvalNode instead of getEvalNode as it is on EvalNodeFactory.
MultiResultFunctionEvalNode	This interface is used to calculate actual values. MultiResultFunctionEvalNode derives from EvalNode. • It should allocate and de-allocate appropriate sized buffers. • Its computed results should match the buffer type specified when the eval node was initialized. • In addition to its normal EvalNode-related functionality, a class deriving from MultiResultFunctionEvalNode

Interface	Description

	implements getEvalNode, acting as a factory for SecondaryResultFunctionEvalNodes, which serve as a temporary holding area for the secondary function results that are being assigned to a measure. The class needs to retain a pointer to each secondary eval node it creates in order to implement eval and evalNA (see example code, referenced in a previous section).
	• It should implement eval and evalNA, storing *only* the secondary results that are actually being stored (see the previous bullet point).
	• If the multi-result function is designed to support mixed intersections, it is advisable to verify this in the constructor – i.e. that the vector results have size > 1 and the non-vector results have size = 1. RPAS itself does not check for this.
	• **Important:** When a MultiResultFunctionEvalNode is destroyed, it is responsible for deleting its child eval nodes. For example, from Expression/unittest:
	`template <class T>`
	`GetLastDigitEvalNode<T>::~GetLastDigitEvalNode()`
	`{`
	`std::for_each(_children.begin(), _children.end(), rpas::DeleteObject());`
	`}`

Named Parameters

On the left-hand side of the equation, a variable may be preceded by a label. For example:

```
frcst:f, int:i = autoes(sales)
```

In a function, a measure may be preceded by a label. For example:

```
f = autoes(sales, alpha:a)
```

In both cases, the ParseNode will have the label attribute set to the value before the colon. If no attribute is set, a numeric string corresponding to the position will be returned as the label.

Measures

Registering Measures

Measures must be registered. See **Error! Reference source not found.** for more information.

Measure Properties

The following is a reference for the measure properties that are supported in RPAS 11.0. Note that some of the RPAS 9.4 measure properties (referred to in RPAS 9.4 as measure attributes) are deprecated and are handled differently, mainly as a result of the new calculation engine.

General Measure Properties

Property	Description	Updateable
label	Label of measure. Defaults to measure name. Unique name of measure for display in client (in grid and measure dialogs). In the past, labels were sometimes non-unique, but now they are guaranteed to be unique. There is no maximum size limit, but you should keep grid display limitations in mind when creating a measure label. (History: displayed in wb but not in measure dialogs.)	Updateable
description	Potentially long, descriptive summary of measure.	Updateable
type	Base types: int, real, string, date, boolean. The numeric representations of these types are 1, 2, 3, 4 and 6 respectively. **Note:** This is a required property for adding a measure.	Not updateable

Property	Description	Updateable
navalue	Suggested value for unpopulated (na) cells. Defaults according to type. Initially, the base array of a measure is defined with this naval; however, this is subject to change in order to optimize the compression factor.	Not updateable
baseint	Base intersection for measure. Concatenation of all dimension names (4 characters, suffixed by '_' as necessary). Dimensions are ordered according to the hierarchy to which they belong. In [Measure]SFX array, baseint is prefixed by "I". Simply 'I' for scalar measures. **Note:** This is a required property for adding a measure.	Not updateable
defagg	Default aggregation method (see Rule Functions document for master list) Note that changes to defagg will only be reflected in new or rebuilt workbooks.	Updateable
defspread	Default spread method Note that changes to defspread will only be reflected in new or rebuilt workbooks.	Updateable
allowedaggs	Permitted aggregation methods. Syntax is as "MAX MIN TOTAL"	Updateable
basestate	Editable at base level: read, write. Default to read.	Updateable
aggstate	Editable at aggregate levels: read, write. Default to read.	Updateable
db	Database where measure is stored in domain (in future, a partitioned database which comprises many partitions). If database is NA then measure is not stored. It specifies the relative path from domain root. Syntax is as "data/test".	Not updateable

Property	Description	Updateable
viewtype	Indicates if the measure is calculated when viewed. If the view type is not 'none' then the measure must not be materialized (ie. not stored in workbook or domain) and must appear on the LHS of one and only one expression in a rule group (ie. the view expression). Synchronized view types are maintained immediately at cell edit time. View types (and their corresponding integer values) include: none (0) – measure is stored view_only (1) – measure is calculated when viewed (aggstate and basestate must be read-only) sync_first_lag (2)– period 1 from first measure (no calendar), periods 2..N from second measure 1..N-1 (lag) [eg. bop ⇔ os & eop] sync_lead_last (3)– periods 1..N-1 from first measure 2..N (lead), period N from second measure (no calendar) [eg. eop ⇔ bop & cs] sync_first (4) – get period 1 from measure (similar to first agg type along calendar dimension) [eg. os ⇔ bop] sync_last (5) – get period N from measure (equivalent to last agg type along calendar dimension) [eg. cs ⇔ eop]	Updateable
syncwith	Comma separated list of measures used for synchronization. Depends on view type.	Updateable
insertable	Indicates whether measure is visible to user for inclusion in workbooks (used in addition to measure security). Default to true.	Updateable
refreshable	Indicates whether measure is refreshable in workbook	Updateable

Property	Description	Updateable
range	Specify suitable range for the measure at edit time. For picklist measures (indicated with ui_type property) these are the displayed/possible values. For numeric values, the syntax is as Lower Bound : Upper Bound. For strings, it is the maximum length. For picklist measures (both numeric and string) values can be explicitly listed as “a,b,c,d…”; additionally labels can be specified for each value such as “valuea(labela), valueb(labelb), …” where the labels will be displayed in the user interface.	Updateable
ui_type	Indicates whether the measure is a picklist or not.	Updateable
materialized	There are two possible values for materialized: Persistent: a "normal" measure Display: a measure that is only calculated when it is displayed (e.g. in the client). Only the displayed values are calculated, rather than the whole measure, which significantly improves performance with these measures. Display-only measures have three major restrictions: 1. Must have agg type "recalc" 2. Cannot appear on the RHS of any rule 3. Cannot be edited.	Not updateable
lowerbound	A measure that provides minimum bounding values for this measure’s data cells.	Not updateable
upperbound	A measure that provides maximum bounding values for this measure’s data cells.	Not updateable
tokenmeas	Values are true or false to indicates if the measure is a token measure	Not updateable

Notes:

- When a recalc measure is registered, its aggstate and basestate are set to user-specified value no matter what is specified for defspread field.
- When a measure with defagg "PopCount" or "NobCount" is registered, this measure's defspread is set to NONE, aggstate is set to read only no matter what user specifies for this field.
- For measures that don't fit into the above descriptions, if their defspread is NONE, their aggstate is read only all the time.

Measure Loading Properties

Property	Description	Updateable
loadint	Intersection at which measure is loaded (for ovr, inc loads). Default to *baseint*. Changing loadint will make it impossible to back out previous data loads.	Not updateable
filename	File used to load measure (suffix of ovr, inc, clr). Default to *measure name*.	Updateable
start	Start column of data (defaults to column after dimensions).	Updateable
width	Width of data (defaults by type). Note: For Boolean types, width can be >= 1, but only the first character is read.	Updateable
loadstokeep	History of loads (that may be backed out). Default to *6*.	Updateable
loadagg	Used when aggregating from load intersection.	Updateable
stageonly	When true, data is not loaded into measure, but left in a staging array for custom processing. Default to *false*.	Updateable
clearint	Intersection at which to clear measure data (for clr loads).	Updateable
purgeage	Limits the history of data in a measure along the time dimension (purge occurs during load measure process). Default to *na*.	Updateable

Virtual Measures

In RPAS, certain measures–Opening Stock (OS), Begin of Period (BOP), End of Period (EOP), and Closing stock (CS)–have a predefined relationship across time that is always true for predictive applications. Assuming that data is stored across time from a starting point to an ending point, this relationship can be characterized as follows:

1 OS is equal to the first element of BOP across the time dimension.

2 The relationship also states that *n*th + 1 element of BOP is equal to the *n*th element of EOP across the time dimension.

3 The relationship also states that the last element of EOP across the time dimension is equal to CS.

In order to support this predefined relationship between OS, BOP, EOP, and CS, the RPAS 11 client supports the concept of virtual measures. Virtual measures are defined by a set of rules that allow the client to calculate the data for virtual measures from stored measures. This set of rules is created to implement the relationship between OS, BOP, EOP, and CS. For any given implementation, either OS and EOP are stored measures with BOP and CS virtual measures, or BOP and CS are stored measures with EOP and OS virtual measures

Editing

If a virtual measure is edited, the edit will be translated to the corresponding cell address of the stored measure that is related to the virtual measure. The calculation engine cannot handle virtual measures being changed.

Protection

When the read/write state of a virtual measure is calculated, the cell address of the stored measure corresponding to the cell address of the virtual measure is used.

Deferred Calculation

When the deferred calculation queue is applied to a visible grid, all stored measures that have a cell edit that corresponds to a virtual measure address that is visible are displayed in the visible cell of the virtual measure.

Non-materialized measures

Non-materialized measures are measures that are temporary or transient in nature, and are not persisted to the domain or workbook. They are always assumed to be of recalc aggregation type. They are designed to be performance enhancing for two major reasons:

- Non-materialized measures are calculated only when required.
- Non-materialized measures are not persisted to the domain or workbook

Expressions for normal (materialized) measures are always calculated when the rule is affected and the expression is selected. This is not the case with non-materialized measures, which are only calculated when required, for example during the fetch process for display-only non-materialized measures.

Display-only non-materialized measures

RPAS supports *display only non-materialized measures*. They provide significant performance improvements when used for 'performance indicators' that cannot be manipulated and are used for viewing only. The following requirements must be observed to utilize display only non-materialized measures.

- The measure cannot appear on the RHS of any expression
- The measure must be registered with Recalc aggregation type and follow the rules for recalc measures.
- The measure must be registered with no db.

The measure must be registered with materialized type "display".

Token Measures

Example

```
@a = @b + c / d     ,where a and b are token measures.
```

Assumptions

- Token measures are string-type scalar measures. They cannot appear on the rhs or lhs of expressions without the indirection operator @.
- Token measures cannot be aggregated or calculated. An expression including non-indirected token measures (without @ prefix) will not be valid. (Expression parser will not validate such expressions.)
- Each token measure must be initialized with the name of a valid non-token measure. This must be done before running processes that create tokenized expressions. Examples of such processes are alert-manager and domain configuration processes.
- RPAS provides a C++ api and a utility called **regTokenMeasure** for assigning values to token measures. This api will register non-existing token measures and populate them with the provided values.
- Token measures do not need to be registered by RPAS applications. Registration of these measures is done internally by **regTokenMeasure** utility. Please refer to the domain utility regTokenMeasure.

C++ API For Setting Token Measures

MeasureStoreTools class contains the interface for registering token measures.

```
MeasureStoreTools::registerTokenmeasure(MeasureStore& store,
const std::map<String, String>&= tokenValuePairs);
```

The input map is a collection of token-value pairs, while the first argument is the name of the token measure and the second argument is the name of the reference measure.

Measure Attributes

Several RPAS applications require the ability to register sets of mostly identical measures that very from each other slightly based on application specific attributes. RPAS supports this ability by introducing the concept of "measure attribute", i.e., single level roll-ups along data hierarchy above the measure dimension. Measure attributes are valid dimensions in the domain and their name should not consist more than 4 characters. The interfaces for dynamic registration of measure attributes and using a set of attributes to allow existence testing, creation, and retrieval of measures are defined.

Measure Attribute Interface

The basic RPAS measure hierarchy is the "data" hierarchy and by default no longer contain any roll-ups at creation time. Instead applications will be able to register desired one level higher aggregates of measure as needed. It is recommended for efficiency that all such attributes be registered during the applications initial registration into the RPAS base.

Attributes can be any useful higher level-grouping device. Several examples of potential attributes include role, version, metric, forecast level, forecast date, etc. In order to promote readability it is further enforced that attributes should not share a name with the information components associated with a measure (i.e., label, type, baseint, navalue, defagg, defspread, allowedaggs, range, refreshable, db, insertable, basestate, aggstate, stageonly, filename, loadint, clearint, loadstokeep, start, width, loadagg, purgeage, viewtype, syncwith, description, ui_type).

Finally, due to the fact that the attributes are higher-level roll-ups of the measure hierarchy, each attribute must be registered with a default value. The default value will be the higher-level position that any measure will belong to when no specific value is assigned to that measure for that attribute.

Registration and removal of attributes are through the MeasureStore.

Function	Description
void registerAttribute(const AliName& attName, const AliName& defVal, const String& attributeLabel, const String& defValueLabel)	Register a measure attribute
void unregisterAttribute(const AliName& attName)	Unregister a measure attribute

Once the attribute is registered two functions are used to manipulate the positions present in the dimension.

Function	Description
void addAttributePosition (const AliName& attName, const AliName& posName, const String& posLabel)	Add an measure attribute position
void removeAttributePosition(const AliName& attName, const AliName& posName, const String& lastMeasName = "")	Unregister a measure attribute position. The "lastMeasName" is the last measure that rolls up to "posName", but is going to change to another rollup for the attribute.

A position may be harmlessly added for an attribute multiple times, although it will only appear once in the dimension position list for the attribute. Any attempt to remove an attribute position that is in use as a roll-up for a measure except the measure “lastMeasName” (if it is not empty string) will throw a MeasureAttributeException exception.

Attribute Controlled Measure Interface

One general basic types call is supported in this interface. This call mimics a section of the more traditional measure registration interface. In fact the primary difference between the attribute based interface and the existing interfaces is the use of an attribute-value pair map instead of a measure name as the vehicle for referring to a measure. For convenience the following type is added to the MeasureProperties class:

```
typedef std::map<AliName, AliName> MeasAttrCollection;
```

Based on this type the following procedures are available to register a measure through MeasureStore:

Function	Description
Measure& registerMeasure(const MeasureProperties& propertiesMap);	Register a measure where the attributes for this measure can be prior set by calling rollups(const MeasAttrCollection& rollups) function on the object “propertiesMap”

The parameter propertiesMap encapsulates the information components associated with a measure. Three components, measure name, type and baseintersection, are required for constructing such an object. Other information components will be set to default values if they are not explicitly provided. If the “name” component in the propertiesMap object is an empty string, it will be set to a unique measure name at registration time.

Removal of measures uses the following interface:

Function	Description
void unregisterMeasures(const MeasAttrCollection& description)	Remove all measures that fit the description

Retrieval of measures uses the following interfaces:

Function	Description
String::VectorT lookupMeasures(const MeasAttrCollection& escription)	Remove all measures that fit the description
String::NoCaseSetT measuresWithoutAttribute(const AliName& attName, const AliName& attValue)	Return the names of all the measures that don't' roll up to the specified attribute position
String::NoCaseSetT measuresWithAttribute(const AliName& attName, const AliName& attValue)	Return the names of all the measures that roll up to the specified attribute position

All of these functions could throw MeasureAttributeExcepiton to indicate a bad map, with the details manifested in the exception message.
For an existing measure, the following functions of Measure class can be used to set and retrieve measure attribute information, on the basis that measure attributes and positions have been properly registered and added, otherwise, MeasureAttributeException exception will be thrown.

Function	Description
void setRollups(const MeasureProperties::MeasAttrCollection& description)	Specify the hierarchy roll up information for a measure
void setRollup(const AliName& attName, const AliName& posName)	Make a measure to roll up to a specific measure attribute position
MeasureProperties::MeasAttrCollection getRollups() const	Retrieve the hierarchy roll up information of a measure
AliName getRollup(const AliName& attName) const	Retrieve the position a measure rolls up to for a specified measure attribute dimension

MeasureStore Class Library

In RPAS 11, the MeasureStore class, and its subclasses Domain and Workbook, form the basis for the manipulation of data stored in RPAS and the construction of viewable subsets of that data. The domain represents the primary data storage of RPAS, and a workbook represents a temporary data cache for both display and manipulation purposes. Within a single process, there can be at most one Domain object and one Workbook object in existence simultaneously. A single Domain object exists for the duration of a process. Several workbooks may exist within the lifetime of a single process.

A measure store, associated with either a domain or a workbook, contains interfaces to all the primitives necessary to manipulate and use measures. These tasks include:

- Registering and unregistering measures
- Navigating hierarchies and dimensions
- Retrieving and setting the properties of existing measures
- Retrieving arrays that correspond to a measure instance.

The first step to performing any of the above tasks is to get a reference to the appropriate measure store. Three static functions are used to retrieve the appropriate references:

Function	Description
MeasureStore& MeasureStore::current();	Returns a reference to the active workbook measure store, if one exists; otherwise returns a reference to the domain measure store.
Domain& Domain::current();	Returns a reference to the domain measure store.
Workbook *Workbook::current();	Returns a reference to the active workbook measure store. If there is no active workbook, returns NULL.

Workbooks and Workbook Templates

Most user interaction from the RPAS client is done using workbook templates and workbooks. A workbook template can be used to configure and build a workbook that is opened in the RPAS client or to configure and initiate a batch process that is executed on the server.

The implementation of all workbook templates in RPAS 11 must be in C++. **RPAS 11 does not support the MSPL procedures that were used to define and build workbook templates and workbooks in previous versions of RPAS.**

Workbook Templates

An example of a simple workbook template class that does absolutely nothing is shown in Figure 1.

Note: The numbers in parentheses at the end of a line are used as reference markers in the discussion of the code.

Figure 1 – Example Workbook Template

```
#include <rpas/templates/WorkbookTemplate.h>
#include <rpas/commdata/WizardFinish.h>

class EmptyTemplate : public rpas::templates::WorkbookTemplate
(1)
{
public:

   EmptyTemplate()
   {
      using rpas::templates::WorkbookTemplateGroup;

      _name = “EMPTY_EXAMPLE”;                          (2)
      _label = “Empty RPAS Template”;
      _group = WorkbookTemplateGroup(“RPAS_EXAMPLES”, “RPAS
Examples”);          (3)
   }

   virtual ~EmptyTemplate()                              (4)
   {
   }

   virtual void registerTemplate()                       (5)
   {
```

```
    }

    virtual rpas::commdata::WizardFinish* finish()        (6)
    {
       // only override this function if your template is doing
something other
       // than building a workbook
       return NULL;
    }

    virtual void initializeWizard()                         (7)
    {
    }


};
```

All workbook template classes must inherit from WorkbookTemplate (1). The constructor for the class must set the *_name*, *_label*, and *_group* member variables as indicated. The *_name* variable must be a valid RPAS identifier (2). The first parameter for the WorkbookTemplateGroup constructor is an RPAS identifier specifying which template group this template will be contained within (3). The destructor for a workbook template class must be declared to be virtual (4). A workbook template class with only a constructor and destructor as in Figure 1 is functional, although not very useful. It can be registered with a domain and will appear in the New Workbook dialog in the RPAS 11 client application.

Before a template can be used by the RPAS 11 client, it must be registered with the domain to which the client is connected. This registration must be done only once to add a new template to a domain. It should also be done if the template implementation has been updated. Template registration can be done in a program or by using the regtemplate utility. The most common method will probably be to use the regtemplate utility. The regtemplate utility takes two arguments. One is the path to the domain and the second is the name of the shared library that contains the template implementation (more on compiling workbook template classes into a shared library later). The shared library is assumed to be in a directory called *lib* in the domain.

To register a workbook template in a program, use the TemplateManager::registerTemplate functions. For example, the following statement registers all the templates in the MyTemplateLib library:

```
TemplateManager::instance().registerTemplate(“MyTemplateLib”);
```

When a workbook template is registered with the domain, a function called registerTemplate() is called. If your template must do some initialization at registration time, you can override this function (5).

If your workbook template does not build a workbook but instead invokes a task on the server, you can override the finishWizard() function (6). This function is called at the end of the wizard process or immediately after a user selects the template from the RPAS client if the template has no wizard dialogs.

Wizard Dialogs

A workbook template can optionally be configured by a set of wizard dialogs. There are two basic approaches for defining a set of wizard dialogs for a workbook template. The first and simplest approach is to override the initializeWizard() function in your template class. The code in Figure 2 shows a batch template with two wizard pages.

Figure 2 – Example Workbook Template with Wizard Pages

```
#include “TestPage1.h”
#include “TestPage2.h”
#include <rpas/templates/WorkbookTemplate.h>
#include <rpas/commdata/WizardFinish.h>


class BatchTemplateWithWizard : public
rpas::templates::WorkbookTemplate
{
public:


   BatchTemplateWithWizard ()
   {
      using rpas::templates::WorkbookTemplateGroup;


      _name = “WIZ_EXAMPLE”;
      _label = “RPAS Wizard Example Template”;
      _group = WorkbookTemplateGroup(“RPAS_EXAMPLES”, “RPAS
Examples”);
      properties().hasWizardPages(true);                (1)
   }


   virtual ~BatchTemplateWithWizard ()
   {
   }


   virtual rpas::commdata::WizardFinish* finishWizard()    (5)
   {
      // only override this function if your template is doing
something other
      // than building a workbook
      return NULL;
   }
```

```
   virtual void initializeWizard()                          (2)
   {
      clearWizardPages();                                   (3)
      appendWizardPage(new TestPage1());                    (4)
      appendWizardPage(new StandardTwoTreePage("PROD", "All
Products",
                                                "Selected Products",
                                                "STYL", "STYL"));
   }


};
```

In the constructor for the class, you must specify that this template has wizard pages (1). In the initializeWizard() function (2), you must instantiate each wizard dialog and pass it to the template using the appendWizardPage() function (4). The order of the appendWizardPage calls defines the order the wizard pages will be presented to the user. You should also call clearWizardPages() (3) at the top of the initializeWizard() function to make sure that any previously cached wizard pages are destroyed. initializeWizard() is called each time a user selects your template from within the RPAS client. Note that the wizard pages need to be allocated using the new operator (4). They cannot be allocated as local variables at this time. You also do not need to worry about deallocating the wizard page objects. The RPAS framework will take care of them for you.

Note: The implementation of the wizard page classes (TestPage1 and TestPage2) is explained after discussing the second method for defining a set of wizard dialogs for your template.

The simple method for defining a set of wizard dialogs for a workbook template allows you to define a static chain of dialogs to be presented to the user. At the time the user selects the template, you must know enough to be able to create the set of dialogs. If this is not the case (for example, if the next dialog in the sequence depends on selections made in previous dialogs), then you cannot use the initializeWizard() method discussed in the previous paragraph. For finer control of the wizard dialog process, you need to provide implementations for the moveToNextWizardPage(), moveToPreviousWizardPage(), getCurrentWizardPage(), getFinishButtonState(), getBackButtonState(), and getNextButtonState() functions. You may optionally implement the cancelWizard(), finishWizard(), and initializeWizard() functions. A description of each of these functions follows.

The initializeWizard() function takes no arguments and returns no value. This function is called before any wizard pages are needed after a user selects your template from the RPAS client. You should do any initialization or preparation needed for your wizard pages in this function.

The moveToNextWizardPage() function takes no arguments and returns a pointer to a WizardPage object. You are responsible for allocating and deallocating the WizardPage object. This function is called before the first wizard page is displayed and also when the Next button is clicked on a page. In this function, you can use any algorithm you choose to determine which wizard page should be displayed next. You can build them dynamically, look them up from a cached set of pages, use the user's selections on previous pages, etc.

The moveToPreviousWizardPage() function also takes no arguments and returns a pointer to a WizardPage object. Again, you are responsible for allocating and deallocating the WizardPage object. This function serves a similar purpose as moveToNextWizardPage(), except it is only called when the Back button is clicked on a wizard page.

The getCurrentWizardPage() function takes no arguments and returns a pointer to a WizardPage object. It must return the currently visible wizard page. This function can be called at any time by the RPAS 11 framework.

The getBackButtonState() and getNextButtonState() functions take no arguments and return a Boolean value. If the value is true, then the corresponding button on the wizard page (Back or Next) will be enabled. If the value is false, the button will be disabled. These functions are called following calls to both moveToNextWizardPage() and moveToPreviousWizardPage().

The getFinishButtonState() function also take no arguments and return a Boolean value. This value should be true for the last wizard page of a wizard page process. If the value is false, additional wizard pages must follow. This functions is called following calls to both moveToNextWizardPage() and moveToPreviousWizardPage(). The getFinishButtonState() function has this semantic difference with the other Button State functions to allow the client application to implement an AutoFinish feature. The AutoFinish feature allow a user to hit the finish button prior to the last wizard page if the user has previously made selections for the wizard page process.

The finishWizard() function is called after a user clicks the Finish button on a wizard page. The cancelWizard() is called after the user clicks the Cancel button. You can implement these functions to complete or abort your template process as appropriate.

Wizard Pages

Each wizard dialog is an instance of a WizardPage subclass. Figure 3 shows examples of classes that define the wizard dialogs used in Figure 2.

Figure 3 – Example Wizard Page

```
#include <rpas/templates/WizardPage.h>

class TestPage1 : public rpas::templates::WizardPage     (1)
{
public:
   virtual ~TestPage1()                                  (2)
   {
   }

   virtual bool initialize()                              (3)
   {
      // set the virtual size of our page
      setDimensions(50, 75);                            (4)

      // add a label (non-editable static text field)
```

```
      addText("label1", "Field 1:", "right", 0, 7, 8, 0);
(5)

      // add an editable text field                        (6)
      addEdit("EditField", "", 9, 7, 8, 0);
      setText("EditField", "Edit me...");

      // add a drop down list field and select the second element
(7)
      addDropDownList("DropDownList", "", 9, 11, 8, 20);
      rpas::String::StringVectorT ddLabels =
         rpas::String("Choice 1/Choice 2/Choice 3").unconc("/");
      rpas::String::StringVectorT ddValues =
         rpas::String("1 2 3").unconc();
      setListContents("DropDownList", ddValues, ddLabels);
      setSelection("DropDownList", ddValues[1]);

      // add a checkbox field and check it                 (8)
      addCheckbox("Checkbox", "Check Me", "", 9, 17, 8, 0);
      setSelected("Checkbox", true);

      // add a list box with a scrollbar and select the second
element          (9)
      addListbox("ListBox", "scrollbar", 9, 23, 8, 8);
      rpas::String::StringVectorT lbLabels =
         rpas::String("List 1/List 2/List 3").unconc("/");
      rpas::String::StringVectorT lbValues =
         rpas::String("1 2 3").unconc();
      setListContents("ListBox", lbValues, lbLabels);
      setSelection("ListBox", lbValues[1]);

      // add a couple of radio buttons and select the second one
(10)
      addRadioButton("Radio1", "Radio 1", "", 9, 35, 8, 0);
      addRadioButton("Radio2", "Radio 2", "", 9, 39, 8, 0);
      setSelected("Radio1", false);
      setSelected("Radio2", true);

      // add a box around the radio buttons
```

```
      addGroupbox("Group1", "Group Label", "", 8, 32, 10, 15);
(11)

      // add a Dropdown Hierarchy control              (12)
      addDropdownHierarchy("DropdownHier", "", 25, 10, 20, 0);
      addHierPositionOption("DropdownHier", "element1", "Element
1");
      addHierPositionOption("DropdownHier", "element2", "Element
2");
      addHierPositionOption("DropdownHier", "subElement1a", "Element
A",
                            "element1");
      addHierPositionOption("DropdownHier", "subElement1b", "Element
B",
                            "element1");
      addHierPositionOption("DropdownHier", "subElement2a", "Element
A",
                            "element2");
      setSelection("DropdownHier", "subElement1b");

      // add a Tree control with a dummy hierarchy and checkboxes
(13)
      addTree("Tree", "checkboxes", 25, 20, 15, 15);
      addHierPositionOption("Tree", "element1", "Element 1");
      addHierPositionOption("Tree", "element2", "Element 2");
      addHierPositionOption("Tree", "subElement1a", "Element A",
"element1");
      addHierPositionOption("Tree", "subElement1b", "Element B",
"element1");
      addHierPositionOption("Tree", "subElement2a", "Element A",
"element2");
      setSelection("Tree", "subElement1b");

      // add a TwoTree control                         (14)
      addTwoTree("LeftTree", "menu checkboxes", 2, 50, 10, 15,
                 "RightTree", "menu", 20, 50, 10, 15);
      addHierPositionOption("LeftTree", "element1", "Element 1");
      addHierPositionOption("LeftTree", "element2", "Element 2");
      addHierPositionOption("LeftTree", "subElement1a", "Element A",
"element1");
```

```
      addHierPositionOption("LeftTree", "subElement1b", "Element B",
"element1");
      addHierPositionOption("LeftTree", "subElement2a", "Element A",
"element2");
      addHierPositionOption("RightTree", "element1", "Element 1");
      addHierPositionOption("RightTree", "element2", "Element 2");
      addHierPositionOption("RightTree", "subElement1a", "Element
A", "element1");
      addHierPositionOption("RightTree", "subElement1b", "Element
B", "element1");
      addHierPositionOption("RightTree", "subElement2a", "Element
A", "element2");\


      return true;                                          (15)
   }


};
```

Each WizardPage subclass must provide a virtual destructor (2). It may also override the initialize() function (3) to define the controls that are on the page. The initialize() function is called just before the page must be displayed by the RPAS client. You may also choose to create all the controls when in the constructor of your class. The difference between these two approaches is that creating and configuring the controls in the initialize() function will allow you to change the configuration each time the page is displayed. Creating and configuring the controls in the constructor of your WizardPage subclass causes this configuration to happen only once (when the page is created and added to the template via the appendWizardPage() function).

Before adding any controls to your page, you need to define the size of the page. This is done with the setDimensions() function (4). setDimensions() specifies an imaginary width and height of the page space occupied by your controls. It does not specify pixels, inches, or any other graphical unit of measurement. Treat the arguments to setDimensions() as generic units. If you call setDimensions(50, 100) and specify that a control starts at [25, 50] (see descriptions of the controls below for details on how to specify the location of a control), then the control will start halfway across and halfway down the dialog when it is opened by the client.

Wizard Page Controls

The WizardPage class implements functions for creating controls, setting control properties, and querying selections from controls. The available controls include text, edit, radio button, checkbox, groupbox, listbox, dropdown, dropdown hierarchy, tree, and two tree. A discussion of each control follows.

Note: Code references are to Figure 3 for the controls.

Text Control

The text control is used for labels or other non-editable text fields. To add a text control to a wizard page, call the addText() function. See line (5) in Figure 3. This function takes seven arguments:

Argument	Description
name	The name of the text control. The name must be unique for the wizard page.
text	The text displayed for the text control.
style	A space-separated string describing the style of the control. Valid styles include: left – left justify the text (default) right – right justify the text center – center the text singleline – do not wrap the text across lines. **Note:** Only one type of justification (left, right, or center) may be specified.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension()function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

Edit Control

The edit control is a single-line or multi-line text field that the user can edit. To add an edit control to a wizard page, call the addEdit() function. See line (6) in Figure 3. The addEdit function takes six arguments:

Argument	Description
name	The name of the edit control. The name must be unique for the wizard page.
style	A space-separated string describing the style of the control. Valid styles include: left – left justify the text (default) right – right justify the text center – center the text multiline – allow the control to have more than one line of text password – display asterisks (*) in the field as the user types number – allow only numeric characters to be entered. **Note:** Only one type of justification (left, right, or center) may be specified. Either password or number can be entered, but not both.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

To initialize or change the text that is in an edit control, use the setText() function. This function takes the name of the control, followed by a string that will be displayed in the control.

To get the text entered by a user from an edit control, use the getText() function. The getText() function takes two arguments. The first is the name of the control. The second is a reference to a String that is filled with the text in the control. For example, after executing the following, the data variable will contain the text that is in the EditField control.

```
String data;
getText(“EditField”, data);
```

Radio Button Control

The radio button control is used to display a mutually exclusive set of options. Currently, all radio buttons on a wizard page are grouped together logically. Only one of them may be selected at a time. To add a radio button control to a page, call the addRadioButton() function. See line (10) in Figure 3. The addRadioButton() function takes seven arguments.

Argument	Description
name	The name of the text control. The name must be unique for the wizard page.
label	The text displayed next to the radio button.
style	A space-separated string describing the style of the control. Valid styles include: left – left justify the control (default) right – right justify the control center – center the control multiline – allow the control to have more than one line of text flip – control is selected with the right mouse button rather than the left mouse button startgroup – marks the radio button as the first in the group **Note:** Only one type of justification (left, right, or center) may be specified.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

To initialize or change the state of a radio button control (selected or unselected), use the setSelected() function. This function takes the name of the control, followed by a Boolean (true selects the control, false unselects the control).

To get the selection state of a radio button, use the getSelected() function. This function takes two arguments. The first is the name of the control. The second is a reference to a Boolean that is set to the state of the control. For example, after executing the following, the data variable will contain the selection state of the Radio1 control.

```
String data;
getSelected(“Radio1”, data);
```

Checkbox Control

The checkbox control is used to display a set of options that can be simultaneously selected. To add a checkbox control to a wizard page, use the addCheckbox() function. See line (8) in Figure 3. This function takes seven arguments.

Argument	Description
name	The name of the checkbox control. The name must be unique for the wizard page.
label	The text displayed next to the checkbox.
style	A space-separated string describing the style of the control. Valid styles include: left – left justify the control (default) right – right justify the control center – center the control multiline – allow the control to have more than one line of text flip – control is selected with the right mouse button rather than the left mouse button **Note:** Only one type of justification (left, right, or center) may be specified.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

As with radio button controls, initialize or change the state of a checkbox control (selected or unselected) with the setSelected() function. Retrieve the selection state of a checkbox control with the getSelected() function.

Groupbox Control

A groupbox control is used to draw a labeled box on a wizard page. This control can be added to a page by calling the addGroupbox() function. See line (11) in Figure 3. This function takes seven arguments.

Argument	Description
name	The name of the groupbox control. The name must be unique for the wizard page.
label	The text displayed next to the groupbox.
style	A space-separated string describing the style of the control. Valid styles include: left – left justify the control (default) right – right justify the control center – center the control **Note:** Only one type of justification (left, right, or center) may be specified.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

The first is the unique name of the control. The second is the label displayed on the control.

The third argument is a string containing one of “left” (left justify the label), “right” (right justfiy the label), or “center” (center justify the label). These are the only valid styles for a groupbox control.

The last four arguments specify the left edge, top edge, width, and height of the control. These values are relative to the virtual size of the page set with the setDimensions() function.

Listbox Control

The listbox control displays a list of strings that can be selected by the user. The listbox can be scrollable and supports single or multiple selections. To add a listbox control to a page, use the addListbox() function. See line (9) in Figure 3. The addListbox() function takes six arguments.

Argument	Description
name	The name of the listbox control. The name must be unique for the wizard page.
style	A space-separated string describing the style of the control. Valid styles include: sort – alphabetically sorts the strings in the list multiple – allows multiple simultaneous selections scrollbar – displays a scrollbar on the control **Note:** The default is unsorted.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

To initialize the contents of a listbox, use the setListContents() function. This function takes as arguments the name of the control, a list of strings that specify the values returned to the wizard page when the user makes a selection, and a list of strings that are the labels that the user sees. The labels and values can be the same. The type for the string list variables is StringVectorT.

To initialize the selection for a listbox, you can use either the setSelection() or the setSelections() function. Both functions take the name of the control as the first argument. The setSelection() function takes a single string that indicates which value is to be selected in the control. This function works with both single and multiple selection lists. The setSelections() function takes a set of strings (StringSetT) and is used to initialize the selections for multiple selection lists.

To get the selections from a listbox control, call the getSelections() function. This function takes two arguments. The first is the name of the control. The second is a reference to a StringSetT that is filled with the selections in the listbox. For example, after executing the following, the data variable will contain the selections in the ListBox control.

```
StringSetT data;
getSelections(“ListBox”, data);
```

Dropdown Control

A dropdown control is similar to a listbox control, except that it typically only displays one line showing the currently selected item in the control. It also only supports a single selection at a time. This control can be added to a wizard page by calling the addDropdown() function. See line (7) in Figure 3. This function takes six arguments:

Argument	Description
name	The name of the dropdown control. The name must be unique for the wizard page.
style	A space-separated string describing the style of the control. Valid styles include: sort – alphabetically sorts the strings in the list scrollbar – displays a scrollbar on the control **Note:** The default is unsorted.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

As with a single selection listbox control, use the setListContents() function to initialize the labels and values for the control. Use the setSelection() function to initialize the selection for a dropdown control. Use the getSelection() function to retrieve the selection from this control.

Dropdown Hierarchy Control

A dropdown hierarchy control allows manipulation and selection of list elements from a tree structure. This control can be added to a wizard page by using the addDropdownHierarchy() function. See line (12) in Figure 3. This function takes six arguments:

Argument	Description
name	The name of the dropdown hierarchy control. The name must be unique for the wizard page.
style	A space-separated string describing the style of the control. Valid styles include: base – ??? any – checkboxes – displays a scrollbar on the control
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.

Argument	Description
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

To initialize the hierarchy displayed in a dropdown hierarchy control, use the addHierPositionOption() functions. There are two versions of this function. The first takes three arguments (name of the control, value of the item in the control, and label of the item in the control) and is used to add top-level nodes in the tree. The second version of addHierPositionOption() takes four arguments (name of the control, value of the item in the control, label of the item, and name of the parent item). This version is used to add child nodes to the tree.

To initialize the selection for this control (this control only allows a single selection), use the setSelection() function as with dropdown controls. Retrieve the current control selection using the getSelection() function.

Tree Control

A tree control is essentially a dropdown hierarchy control that is displayed as a listbox. It can also have knowledge of hierarchies and dimensions, supporting a fairly rich user interface for manipulating the display of hierarchical elements. To add a tree control to a wizard page, call the addTree() function. See line (13) in Figure 3. This function takes six arguments:

Argument	Description
name	The name of the tree control. The name must be unique for the wizard page.
style	A space-separated string describing the style of the control. Valid styles include: menu – enables the right mouse button menu used for advanced navigation and control of the tree checkboxes – displays elements in the tree with checkboxes next to them rather than highlighting the entire item when selecting multiselect – allows multiple items to be selected **Note:** By default, tree controls have a scrollbar, allow multiple selections, do not display checkboxes, and do not display a menu.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.

Argument	Description
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

Initialize the hierarchy, set the initial selections, and query the current set of selections as you do with a dropdown hierarchy control, using the addHierPositionOption(), setSelection(), and getSelections() functions. Because the tree control supports multiple selections, you may also use the setSelections() function to select more than one element in the tree.

Another set of tree control functions allows a set of dimensions and their hierarchical relationships to be defined. The most powerful way to use a tree control is to use it to display, navigate, and select pieces of a dimension hierarchy. Use the addTreeDimension() function to add a dimension to a tree control. This function takes four arguments. The first argument to this function takes the name of the control. The second and third arguments are the name and label of a dimension. The fourth argument is the name of the child dimension. Alternatively, if you wish to add all of the dimensions in a hierarchy to a tree control, use the loadRollups() function. This function takes the name of the control and the name of the hierarchy as arguments and adds all dimensions in the hierarchy to the control.

After the dimensions and their hierarchical relationships are specified for a tree control, you should define the relationships between the positions in the dimensions. You can do this by using the addHierPositionOption() functions as in the example in Figure 3. An easier way to do this is to use the loadHierarchy() function. This function takes the control name, the name of a dimension that defines a starting point in the hierarchy (the rollup), and the name of a hierarchy. It will then start at the rollup dimension and work its way to the root of the dimension hierarchy, adding dimension positions and defining their relationships as it walks the tree. When the function is finished, the tree control should display a graphical representation of the dimension hierarchy.

Tree controls currently assume that a rollup is defined. A rollup is a dimension in a hierarchy that can be found in the dimension dictionary of the domain the RPAS client is connected to. The rollup indicates level in a hierarchy of dimensions at which elements of a tree control are being viewed and manipulated. The example code in Figure 3 constructs an artificial tree structure and does not specify a rollup. The name of the dimension defining the rollup for a tree control is set by using the setInitialRollup() function. The first argument to this function is the name of the tree control. The second is the name of the rollup.

Two-Tree Control

A two-tree control is a special variant of the tree control. It is displayed as two related tree controls. The left tree control is typically used to display all the possible selections that can be made from a hierarchy structure. The right tree displays only those elements of the left tree that are selected. Use the addTwoTree() function to add this control to a wizard page. See line (14) of Figure 3 as an example.

The addTwoTree() function takes two sets of six arguments (12 total). The first six define the left tree and the second six define the right tree.

Argument	Description
name	The name of the tree control. The name must be unique for the wizard page.
style	A space-separated string describing the style of the control. Valid styles include: menu – enables the right mouse button menu used for advanced navigation and control of the tree checkboxes – displays elements in the tree with checkboxes next to them rather than highlighting the entire item when selecting multiselect – allows multiple items to be selected **Note:** By default, tree controls have a scrollbar, allow multiple selections, do not display checkboxes, and do not display a menu.
left edge	The left edge of the control, relative to the size of the page set using the setDimension() function.
top edge	The top edge of the control, relative to the size of the page set using the setDimension() function.
width	The width of the control, relative to the size of the page set using the setDimension() function.
height	The height of the control, relative to the size of the page set using the setDimension() function.

It is not too likely that you will directly use a two-tree control. The easiest way to use this control is to add one or more StandardTwoTreePage objects as wizard pages in your workbook template. See line (4) in Figure 2 for an example of this. To create a StandardTwoTreePage, provide the name of a dimension hierarchy, a label for the left tree, a label for the right tree, the initial rollup for the left tree, and the initial rollup for the right tree when constructing the page. The StandardTwoTreePage class handles all the details of building the tree structure from this specification.

Saving Wizard Page Data

All of the data entered or selected on a wizard page by the user is permanently saved when the wizard page is exited by clicking on the Next or Finish buttons. The next time the page is visited by the user, the previously saved data will be automatically restored. If you do not want to restore previously saved values, you can override the restoreControlDataArray() function (implement it as an empty function to ignore any previously saved page data). You can also override both saveControlDataArray() and restoreControlDataArray() if you have unique save and restore requirements.

Finishing a Batch Template

When a user reaches the end of the set of wizard dialogs for a template and clicks the Finish button, the finishWizard() function will be called on your template. See line (5) in Figure 2 for an example. If your template does not override this function, the RPAS 11 framework will assume that a workbook is being built from your template and will try to do so (see the next section for a discussion of this mechanism). If you want your template to do something else (invoke some one-time processing on the server, update some configuration data stored on the server, etc), you need to override the finishWizard() function.

If you are overriding the finishWizard() function, it is likely that you will want to get data entered by the user into some of the wizard page controls. If you have given all of your wizard pages to the RPAS 11 framework (via the appendWizardPage() function), then you can get access to a wizard page in a couple of different ways. The first is to call the getWizardPage() function. This function takes the name of a wizard page and returns a pointer to a WizardPage object or NULL if one with that name is not found. The name for a wizard page can be set using the id() function before calling appendWizardPage(). If a wizard page does not have an id set before appendWizardPage() is called, it is assigned an id. The assigned id is a number (starting with 0) that indicates the order of the wizard page in the sequence of pages for your template.

If you are managing the wizard pages for your template outside the RPAS 11 framework (not using appendWizardPage()), then you are responsible for figuring out how to find the pages you need to perform your finishWizard() tasks.

Once you have access to a WizardPage object, you can query the values of controls on the page by using a function appropriate to the type of control: getSelected() for check boxes and radio buttons, getText() for edit controls, getSelection() for dropdown and dropdown hierarchy controls, and getSelections() for listbox and tree controls.

Building Workbooks

If the purpose of your workbook template is to build a workbook, you should not override the finishWizard() function in your template class. The finishWizard() function in the base WorkbookTemplate class attempts to build a workbook based on configuration information it gets from your template. You can override the following functions in your WorkbookTemplate subclass to somewhat control the workbook build process (see Figure 4 for code examples).

Figure 4 — Example Workbook Template That Builds Workbook

```
#include <rpas/templates/WorkbookTemplate.h>
#include <rpas/measurestore/Workbook.h>

class TestWorkbookTemplate : public WorkbookTemplate
{
public:

   TestWorkbookTemplate()
   {
      using rpas::templates::WorkbookTemplateGroup;

      _name = “RPAS_TEST”;
      _label = “RPAS Test Template”;
      _group = WorkbookTemplateGroup(“RPAS_EXAMPLES”, “RPAS
Examples”);
      properties().hasWizardPages(true);
   }

   virtual ~TestWorkbookTemplate()
   {
   }

   void registerTemplate()
   {
      using namespace rpas::rule;
      using namespace rpas::measurestore;
      using namespace rpas::acumate;
      using rpas::calcengine::CalcEngine;

      TransactionWrap txn;
```

```
      // register our b and c measures in the domain
      Domain& domain = Domain::current();
      rpas::String measDbName = “data/rpastest”;
      domain.unregisterMeasure(“b”);
      domain.registerMeasure(“b”, “b”, MeasureInfo::IntegerType,
                             “sku_str_week”, Metric(“b”),
measDbName);
      domain.unregisterMeasure(“c”);
      domain.registerMeasure(“c”, “c”, MeasureInfo::IntegerType,
                             “sku_str_week”, Metric(“c”),
measDbName);

     // put some dummy data in the domain measure arrays
      rpas::String measDbPath = rpas::String(“../../”) + measDbName;
      Database measDataDb = Database::restore(measDbPath,
Ali::Write);
      Array domainB = Array::restore(measDataDb, “b%1”, Ali::Write);
      domainB->set(DimSlice(“week”, “w01_1997”) + DimSlice(“sku”,
“sku_10000008”)
                   + DimSlice(“str”, “str1000”) + DimSlice(“info”,
“info”), 1);
      Array domainC = Array::restore(measDataDb, “c%1”, Ali::Write);
      domainC->set(DimSlice(“week”, “w01_1997”) + DimSlice(“sku”,
“sku_10000008”)
                   + DimSlice(“str”, “str1000”) + DimSlice(“info”,
“info”), 2);

      // create and register the rules for loading, calculating, and
committing all
      // of our measures. measure a in these rules is local to the
workbook (not
      // registered in the domain).
      Rule r1 = Rule::registerNewRule();
      Rule r2 = Rule::registerNewRule();
      Rule r3 = Rule::registerNewRule();

      r1.registerExpression(“a=b+c”);
      r1.registerExpression(“b=a-c”);
      r1.registerExpression(“c=a-b”);
      r2.registerExpression(“b=master(b)”);
```

```
      r3.registerExpression(“c=master(c)”);

      RuleGroup loadRules = RuleGroup::registerNewGroup();
      loadRules.registerRule(r1);
      loadRules.registerRule(r2);
      loadRules.registerRule(r3);
      setLoadRuleGroup(loadRules);

      RuleGroup calcRules = RuleGroup::registerNewGroup();
      calcRules.registerRule(r1);
      setCalcRuleGroup(calcRules);

      Rule r4 = Rule::registerNewRule();
      Rule r5 = Rule::registerNewRule();
      r4.registerExpression(“domain(b)=b”);
      r5.registerExpression(“domain(c)=c”);
      RuleGroup commitRules = RuleGroup::registerNewGroup();
      commitRules.registerRule(r4);
      commitRules.registerRule(r5);
      setCommitRuleGroup(commitRules);

      txn.commit();
   }

   virtual rpas::String::StringVectorT getDomainMeasureNames()
(1)
   {
      // build a StringVectorT by unconcatenating a space separated
string.
      // measures b and c should be copied from the domain into
workbooks
      // built from this template
      return String(“b c”).unconc();
   }

   virtual void registerLocalMeasures(rpas::measurestore::Workbook&
wb)          (2)
   {
      using namespace rpas::measurestore;
```

```
      // measure a is local to the workbook
      wb.registerMeasure("a", "a", MeasureInfo::IntegerType,
                         "sku_str_week", Metric("a"),
wb.info().name());
   }

   virtual void
registerWorkbookWindows(rpas::measurestore::Workbook& wb)        (3)
   {
      using namespace rpas::measurestore;
      using rpas::String;

      String::StringVectorT measNames;
      measNames.push_back("a");
      measNames.push_back("b");
      measNames.push_back("c");

      // create a single MeasureSheet in this wookbook with all
three measures
      // (domain b and c, and local a). the sheet is at intersection
sku,str,week.
      MeasureSheet sheet("TESTWIN", "RPAS Test",
                         measNames, "sku_str_week");
      wb.addSheet(sheet);

      // create the one and only window for this workbook. set its
sheet and
      // visible measures to be a, b, and c.
      WorkbookWindow window("TESTWIN", "RPAS Test");
      wb.addWindow(window, sheet, String("a b c").unconc());
   }

   virtual void formatWorkbook(rpas::measurestore::Workbook& wb)
(4)
   {
      using namespace rpas::measurestore;

      // get the one and only window in this workbook
```

```
      WorkbookWindow window = wb.getWorkbookWindow("TESTWIN");

      // put the PROD hierarcy at SKU level on the X axis
      HierarchyFormat prod("PROD", "sku", HierarchyFormat::X_AXIS);
      window.formatHierarchy(prod);

      // put the LOC hierarchy at the STR level on the Y axis
      HierarchyFormat loc("LOC", "str", HierarchyFormat::Y_AXIS);
      window.formatHierarchy(loc);

      // put the MEAS info first on the Z axis
      HierarchyFormat meas("MEAS", "info", HierarchyFormat::Z_AXIS,
0);
      window.formatHierarchy(meas);

      // stack the CLND hierarchy at the WEEK level second on the Z
axis
      HierarchyFormat clnd("CLND", "week", HierarchyFormat::Z_AXIS,
1);
      window.formatHierarchy(clnd);
   }

   virtual void initializeWizard()
   {
      clearWizardPages();
      appendWizardPage(new StandardTwoTreePage("PROD",
                                               "Available Products",
                                               "Selected Products",
                                               "STYL",
                                               "STYL"));
      appendWizardPage(new StandardTwoTreePage("LOC",
                                               "Available Stores",
                                               "Selected Stores",
                                               "STR",
                                               "STR"));
      appendWizardPage(new StandardTwoTreePage("CLND",
                                               "Available Dates",
                                               "Selected Dates",
```

```
                                                    “WEEK”,
                                                    “WEEK”));
        }

    };
```

getDomainMeasureNames()

This function takes no arguments and returns a list of measure names (see line 1 in Figure 4). This list specifies which measures from the domain should be copied into the workbook being built. RPAS 11 currently has the constraint that a workbook must have at least one measure from the domain.

describeDimensionDict()

This function takes a reference to a DimensionDictDescription object. Currently, this object only allows you to specify ranges for dimension hierarchies. You should call the rangeHierarchy() function on this object if you want to specify a range for a hierarchy. The rangeHierarchy() function takes the name of a hierarchy, the name of a dimension in the hierarchy that defines the level at which the hierarchy is ranged, and a set of positions in the dimension. The set of positions will be used to compute which pieces of the dimension hierarchy will be copied into the workbook. If you choose not to override this function, the RPAS 11 framework will call the describeDimensionDict() function on each page that has been registered with the framework via the appendWizardPage() function. By default, a WizardPage will do nothing in response to this call. The StandardTwoTreePage, however, will initialize a range on the hierarchy managed by the page. Either your WorkbookTemplate subclass or your WizardPage subclass can override this function to provide special behavior.

registerLocalMeasures()

This function (2) is called after all of the domain measures have been copied into the workbook. It takes a reference to a Workbook object and can be used to register measures that are local to the workbook. Use the registerMeasure() function on the Workbook object to do this.

registerWorkbookWindows()

Use this function (3) to define and configure the measure sheets. To create a measure sheet, construct a MeasureSheet object with the name of the sheet, a label, a list of workbook measure names that are on the sheet, and the base intersection of the sheet. Call the addSheet() function on the workbook for each sheet you create. Construct WorkbookWindow objects with the name of the window and a label. Call the addWindow() function on the workbook to specify which windows belong to which measure sheets and also which measures are initially visible in the window.

formatWorkbook()

After the measure sheets and workbook windows have been created, the RPAS 11 framework calls the formatWorkbook() function in your template (4). In formatWorkbook(), you can retrieve WorkbookWindow objects and format the layout of hierarchies in the window using the formatHierarchy() function on the window. The formatHierarchy() function takes a HierarchyFormat object that you can construct with the name of the hierarchy, the name of the initial dimension viewable in the hierarchy, and the axis of the window the hierarchy initially appears on (X, Y, or Z). The last argument of the HierarchyFormat constructor is optional and indicates the order of hierarchies on an axis if more than one hierarchy is stacked on the axis. At some point, this function will allow more window formatting to be specified.