The WIT Developer's Handbook:

A Reference Manual for the Development and Maintenance of WIT

Robert Wittrock

Revised April 11, 2022

This document is available on GitHub at:

`rjwittrock/WIT/doc/wit-dh/wit-dh.pdf`

1. Introduction

The target audience of this document is a “WIT developer”: a person whose has responsibility for the on-going development and maintenance of WIT. The purpose of this document is to provide the information that would be essential for someone to do a proper job of developing and maintaining WIT, at least at a rudimentary level.

This document covers the following three software libraries, which together comprise the WIT asset:

- MCL
- WIT
- WIT-J

Note: At present, WIT-J has not been brought up to date in the open-source version of WIT on GitHub, so there is no need to maintain it.

As you can see from the Table of Contents, most of the sections of this document are devoted to “How to”s: explanations of how to perform various important well-defined tasks: building code, testing code, making certain kinds of revisions to the code. The remaining sections generally give overviews of various aspects of the WIT asset. Deep algorithmic explanations are not given.

Prerequisite Knowledge:

This document assumes that the reader is already familiar with WIT from the point of view of a WIT application programmer. Specifically, the following knowledge is assumed:

- WIT concepts and capabilities. See [WIT Intro] and [WIT Guide].
- The WIT API. See [WIT Guide].
- GitHub

Size of the Code

It may be helpful to be aware of the scale of the source code discussed by this document:

Table 1: Lines of Source Code

Library	Main Code	Test Code	Total Code
MCL	5K	1K	6K
WIT	142K	11K	153K
WIT-J	27K	7K	34K
Total	174K	19K	193K

2. What is MCL?

As a WIT application programmer, you might not know what MCL is. MCL ("Message Class Library") is a library of C++ classes whose purpose is to make it easy for a program to create, manage, and issue "messages": textual output. It was created in the late 90s by members of the WIT team: designed by JP Fasano, myself, and Weiwen Zhao, and initially implemented by Weiwen. We originally intended for it to be used by as many software tools as possible, but in fact it was only used by two: WIT and SCE.

MCL is considered to be part of the WIT asset and is maintained along with the WIT code. For example, if WIT is to be ported to a new platform, then MCL must also be ported to that platform and it must be ported before WIT. Unlike WIT, the MCL code has been stable for a long time: Changes are made to it are infrequently and usually just for porting purposes.

Relationship to other WIT software:

- WIT depends on MCL, directly.
- SCE depends on MCL, directly.

MCL does not depend on WIT or SCE.

3. How to Build MCL

MCL can be built on the following platforms:

- Linux
- Windows

The Linux platform is similar to Unix. It is called a "Unix-like platform" in this document.

The port of MCL to Windows uses a special platform that we worked out, called "Windows SDK", which is a hybrid two distinct sets of software development tools: the Microsoft Windows SDK and the GNU MSYS environment. For an explanation of how to build MCL on the Windows SDK platform, see [Win SDK].

How to Build MCL on Linux, 32 Bit

First, you need to get the following directories from the WIT Fork on GitHub:

- `config`
- `mcl`

Next, you need to set up some environment variables. You normally specify these in your login profile, in my case, `~/.bash_profile`. The first environment variable to set is:

```
WIT_HOME
```

This is the pathname of the directory where you have put your WIT directories from GitHub. For example, the MCL source code would be located at `$WIT_HOME/mcl/src`. In my case, I have it set as:

```
export WIT_HOME=/home/rjw/main/wit/wit-projects
```

The next environment variable to set is:

```
PLATFORM
```

This is the "platform ID" that we have defined for each platform on which we can build WIT-related software. In this case, the correct value is `linux`:

```
export PLATFORM=linux
```

Next, you select an `exe_type`. `exe_type` is a makefile variable used by all of the makefiles in our WIT-Projects directory. It specifies which kind of build is being requested by a make command. (I think `exe_type` means "executable type"). There are 4 allowed values for `exe_type`:

- `debug`
  - Generates "debug" code: code that can be processed by a debugger.
  - Does run-time error checking: asserts, array indexes, etc.
  - This is the `exe_type` normally used for developing and testing the code.

- `released`
  - Generates optimized code.
  - This is the `exe_type` that's to be used for released/production code, especially code that's to be deployed externally.
- `fast`
  - Generates optimized code.
  - In the case of MCL, this is the same as `released`. (In the case of WIT, there's a distinction.)
- `profile`
  - Generates profileable code, code that be processed by a profiler.

Next, you CD to the build directory for MCL on Linux:

```
cd $WIT_HOME/mcl/linux
```

Next, you issue the "make" command, e.g.:

```
make exe_type=debug all
```

Notes:

- You can use any `exe_type` .
- The "`all`" target builds the MCL Unit Tester, which also causes the MCL library to be built.
- Alternatively, you can use the "`unitTest`" target.
- To build the MCL library without the Unit Tester, use the "`libmcl.a`" target.
- You can use the `-j[N]` argument to get N asynchronous compiles. This is a faster build, if your machine has more than one core. For example:

```
make exe_type=debug -j4 all
```

Before building MCL, it might be a good idea to issue the following from the MCL build directory:

```
make clean
```

This removes all object, library, and executable files for MCL on this platform, so that you can be sure that they will not contribute to the current build.

Sometimes it's necessary to issue the following (from the MCL build directory) before building MCL:

```
make depend
```

This updates the list of source code dependencies that will be used by make when building MCL. Specifically, it revises the MCL makefile, `$WIT_HOME/mcl/linux/Makefile`, replacing its dependency list with a new one, based on the `#include` statements that it finds by analyzing the MCL source code. The dependency list is the part of the makefile that begins after the following line:

```
# DO NOT DELETE THIS LINE -- make depend depends on it.
```

and continues to the end of the file.

Generally you only need to issue `make depend` when a change is made to the MCL source code (which is rare).

4. How to Test MCL

The MCL Test is implemented on all of our platforms: Linux and Windows SDK. It should be used under the following circumstances:

- If you are upgrading to a new compiler level or a new environment of some kind, you need to perform the test on that platform.
- If you change any MCL code, you need to perform the test on all platforms.

The procedure is as follows:

First, you build the MCL `unitTest` executable with `exe_type=debug`. (See Section 3.)

- The `debug` `exe_type` is used so that asserts will be activated for the test.
- On Windows SDK, the `unitTest` executable is called `unitTest.exe`.

Next, run the `unitTest` executable:

- `cd $WIT_HOME/mcl/$PLATFORM`

  (You're probably already there from the build.)
- `./unitTest`

Normally, the test will produce a large amount of output, most of which will fly off the screen, concluding with the following line:

```
All tests completed successfully.
```

If it terminates with that line, the test is successful; if it terminates any other way, it's unsuccessful.

If the test is unsuccessful

The MCL unit test mostly consists of many test outputs that are compared with expected output. If the comparison fails, the program terminates with a message showing what was different. That message can then be used as the starting point of an investigation into the underlying problem.

5. How to Build WIT

Like MCL, WIT can be built on the following platforms:

- Linux
- Windows SDK

For an explanation of how to build WIT on the Windows SDK platform, see [Win SDK].

How to Build WIT on Linux

First, you need to get the following directories from the WIT Fork on GitHub:

- `config`
- `mcl`
- `wit`

Next, you build MCL. (See Section 3.)

For WIT, you keep the environment variables `WIT_HOME` and `PLATFORM` as they were for MCL. In addition, there are 4 more environment variables that apply only to WIT: `WIT_COIN_HOME`, `WIT_CPLEX_HOME`, `WIT_CPLEX_LIB_SUBDIR`, and `WIT_GEN_PARSER`.

`WIT_COIN_HOME`.

If COIN is to be embedded into WIT, `WIT_COIN_HOME` should be defined as the path where COIN files are located. For example, `OsiSolverInterface.hpp` should be located at:

```
$WIT_COIN_HOME/include/coin/OsiSolverInterface.hpp
```

If COIN is not to be embedded into WIT, `WIT_COIN_HOME` should be left undefined, or defined as the null string.

`WIT_CPLEX_HOME`.

If CPLEX is to be embedded into WIT, `WIT_CPLEX_HOME` should be defined as the path where CPLEX files are located. In particular, `cplex.h` should be located at:

```
$WIT_CPLEX_HOME/include/ilcplex/cplex.h
```

If CPLEX is not to be embedded into WIT, `WIT_CPLEX_HOME` should be left undefined, or defined as the null string.

`WIT_CPLEX_LIB_SUBDIR`

Normally, the makefiles know where to look for the CPLEX library (based on `WIT_CPLEX_HOME` and the platform). But in some cases, it is necessary to override the default location used by the makefile and specify a different location. To do so, set `WIT_CPLEX_LIB_SUBDIR` to the subdirectory where the CPLEX library is located, so that its full path name is given by:

```
$WIT_CPLEX_HOME/lib/$WIT_CPLEX_LIB_SUBDIR/libcplex.a
```

Otherwise, leave this variable unset and the makefile will use its default location for the CPLEX library.

`WIT_GEN_PARSER`

In order to read WIT data files, WIT makes use of a pair of language processing utilities called Flex and Bison.

(modern versions of Lex and Yacc). To be precise, Flex and Bison are not invoked when WIT is reading a WIT data file; instead, they are invoked by the WIT makefile, when WIT is being compiled. Flex and Bison read in a set of input files that define a language and then output C or C++ source code that "parses" the language. The resulting C/C++ source code is called a "parser": it reads a file in the defined language and executes the actions specified in the file according to the language.

In WIT's case, the input to Flex and Bison is the following two files:

- `witLexer.l` (input to Flex)
- `witParse.y` (input to Bison).

These two files are part of WIT's modifiable source code: the developer is free to alter them as needed. The output of Flex and Bison are the following three "parser" files:

- `witLexer.C` (generated by Flex)
- `ytab.h` (generated by Bison).
- `witParse.C` (generated by Bison).

The parser files constitute WIT's generated source code: the developer must never alter them manually. All five of these files are located in the `$WIT_HOME/wit/src` directory.

The makefile calls Flex and Bison to generate the parser files according to the usual rules of the make utility, i.e., whenever the input files have a newer time stamp than the output files, subject to the following two restrictions:

- They are only generated on the "linux" platform. This allows us to avoid dealing with the various different versions of Flex and Bison that are out there. The files generated on Linux are used on all platforms. Note that this arrangement implies that the input files, `witLexer.l` and `witParse.y` should only be modified on the Linux platform. If you modify them on any other platform, the parser files will not get regenerated and WIT's behavior will not reflect your changes.
- They are only generated, if the environment variable `WIT_GEN_PARSER` is set to YES. If WIT is being built on Linux by a WIT application programmer, there is no need to regenerate the parser files, so by default, they are not generated. This default behavior occurs if `WIT_GEN_PARSER` is left unset. If you are developing/maintaining WIT, then on Linux, you should set `WIT_GEN_PARSER`:

```
export WIT_GEN_PARSER=YES
```

The parser generation makefile logic is located in: `$WIT_HOME/wit/linux/Makefile`.

Next, you select an `exe_type` from the valid possibilities: `debug`, `released`, `fast`, or `profile`. These have the same properties as in the MCL case, but with one additional aspect that applies only to WIT: If `exe_type=debug`, `fast`, or `profile`, the resulting code is considered to be built in "development mode". When WIT is in development mode, the following two undocumented aspects of WIT are enabled:

- The stand-alone executable has a set of "development control parameters", i.e., control parameters that are recognized only in development mode.
- There is an API function, `witInterpretDevCmd`, that only accepts invocations in development mode.

WIT's development mode has the following purposes:

- It gives the WIT developer access to WIT capabilities that are not needed by or desirable for application programmers, and therefore avoids the need for user documentation, rigorous testing, etc. These development capabilities are mostly debugging and testing capabilities. Most of WIT's stand-alone executable tests (described later in this document) use development control parameters.

- It allows the WIT developer the ability to add new partially implemented capabilities to WIT, while keeping them unavailable in production builds of WIT. This is done by requiring the use of a special development control parameter or a special call to `witInterpretDevCmd` before the new capability can be invoked. I use this technique virtually every time I add a new capability to WIT.

The purpose of `exe_type=fast` is to create a code-optimized development build of WIT.

Normally, `exe_type` should be set to the same value for MCL and WIT.

Next you CD to the build directory for WIT on Linux:

```
cd $WIT_HOME/wit/linux
```

Next, you issue the "make" command, e.g.:

```
make exe_type=debug all
```

Notes:

- You can use any `exe_type` .
- The "`all`" target builds the WIT Stand-Alone Executable, which also causes the WIT library to be built.
- Alternatively, you can use the "`wit`" target.
- To build the WIT library without the Stand-Alone Executable, use the "`libwit.a`" target.
- You can use the `-j[N]` argument to get N asynchronous compiles. This is a faster build, if your machine has more than one core. For example:

```
make exe_type=debug -j4 all
```

As with MCL, before building WIT, it might be a good idea to issue the following from the WIT build directory:

```
make clean
```

This removes all object, library, and executable files for WIT on this platform, so that you can be sure that they will not contribute to the current build.

As with MCL, it's also sometimes necessary to issue the following before building WIT, to update the list of source code dependencies that will be used by make when building WIT:

```
make depend
```

Generally you only need to issue `make depend` when a change is made to the WIT source code.

6. How to Test WIT: The API Test

There are 5 procedures for testing WIT:

- The API Test: A general test of WIT; uses the API.
- The No-Solver Test: A test of WIT without COIN or CPLEX embedded; uses the API.
- The COIN-Only Test: A test of WIT with COIN embedded, but not CPLEX; uses the API.
- The CPLEX-Only Test: A test of WIT with CPLEX embedded, but not COIN; uses the API.
- The Stand-Alone Test: A test of WIT's various capabilities; uses the Stand-Alone Executable.

This section will describe the API Test.

The WIT API test is implemented on all of our platforms: Linux, z/Linux, AIX, and Windows SDK. It should be used under the following circumstances:

- If you are upgrading to a new compiler level or a new environment of some kind, you need to perform the test on that platform.
- If you upgrade either of the prerequisites to WIT (MCL or CPLEX), you need to perform the test on all platforms.
- If you change the WIT code or the API Test code, you need to perform the test on all platforms.

The procedure is as follows:

First, you must add the following directory to your PATH:

```
$WIT_HOME/wit/scripts
```

The script for the API test is in this directory and it invokes other scripts in this directory.

Next, you build WIT with `exe_type=debug`. (See Section 5.)

- The `debug` `exe_type` is used so that asserts and other error checking will be activated for the test.

Next, you invoke the WIT API test script:

```
wit-api-test
```

This script builds and runs the following four WIT application programs:

- `apiAll` Invokes each of WIT's API functions at least once.
- `msgAll` Causes each of WIT's "testable" messages to be issued.
- `extOptTest` Tests the external optimizing implosion capability.
- `errTest` Tests WIT's response to various error conditions.

The source code and data for this test (as well as the outputs of it) are stored in the following directory:

```
$WIT_HOME/wit/api-test
```

The script prints a few high-level progress indicators to the screen (like "Running apiAll"), but directs most of its output to the following file:

```
$WIT_HOME/wit/api-test/api-out-new.txt
```

The script then runs a filtered diff, comparing this file against the following file:

```
$WIT_HOME/wit/api-test/api-out-$PLATFORM.txt
```

For example, `api-out-linux.txt`. This comparison file is platform-specific, because there are many minor differences in the output on different platforms, due to different floating point representations.

The diff output is stored as:

```
$WIT_HOME/wit/api-test/api-out-diff.txt
```

If the diff file is empty, the script will print the following lines and terminate:

```
No Significant differences were found.

The WIT API test was successful.
```

If it terminates with those lines, the test is successful.

If the diff file is not empty, then the test is unsuccessful and the script terminates with the following lines:

```
Significant differences were found. See:

   $WIT_HOME/wit/api-test/api-out-diff.txt

The WIT API test was unsuccessful.
```

where `$WIT_HOME` is expanded to its value.

7. How to Test WIT: If the API Test is Unsuccessful

If the WIT API test is unsuccessful, the first thing you need to do is to perform the other WIT tests for further insight. But for now, let's consider what needs to be done with respect to the API test.

You basically need to look at the api-out-diff file, the api-out-new file and the api-out-$PLATFORM file and figure out what the problem is. Doing that, of course, requires a true understanding of how WIT works, which is beyond the scope of this section. Anyway, once you have determined the exact nature of the problem, there are essentially three kinds of actions to take:

1. You might decide that the new output is correct and should become the new basis of comparison. Perhaps you changed the code and the new output is just what you expected. In this case, you need to save the new output on all platforms. To save the new output on the current platform, you take the following steps:
   - Remove the `api-out-$PLATFORM.txt` file.
   - Rename (i.e., mv) the `api-out-new.txt` file to `api-out-$PLATFORM.txt`.
   - Rerun the API test.
     - The test should be successful.
   - Commit the revised `api-out-$PLATFORM.txt` file to the WIT fork on GitHub.
     - Note that GitHub will ignore `api-out-new.txt` and `api-out-diff.txt`. This is specified in the file `$WIT_HOME/wit/api-test/.gitignore`.

   Then you need to run the API test on each of the other platforms and (assuming it makes sense) save the new output on each platform.

2. You might decide that the new output is correct, but is too variable to be used as the basis of comparison. For example, it might involve a run time. In this case, you need to add to the filter in the “diff” command that's in the `wit-api-test` script, so that the unavoidable differences will be ignored. This is the current diff command in the script:

```
diff                                          \
  -I "^Build Date:"                           \
  -I "^Run Started At:"                       \
  -I "^         Build Date:"                  \
  -I "^         Run Started At:"              \
  -I "^Elapsed real time ="                   \
  -I "^  Real time             ="             \
  -I "^Total (root+branch&cut) ="             \
  -I "^Total crossover time ="                \
  -I "^Total time on"                         \
  -I "^Barrier time ="                        \
  -I "^Presolve time ="                       \
  -I "^Found feasible solution after"         \
  -I "^Total time for automatic ordering ="   \
  -I "^Root relaxation solution time ="       \
   api-out-$PLATFORM.txt                      \
   api-out-new.txt                            \
 > api-out-diff.txt                           \
```

   The -I lines are the filter. See documentation on the diff command for how use the -I option.

Then you'll need to take these steps:

- Rerun the API test.
  - Get a successful result.
- Commit the revised script to GitHub.
- Run the API test on the other platforms.

3. You might decide that the new output is incorrect. In this case:
   - Revise the WIT source code and/or the API test source code as needed.
   - Rerun the API test.
     - Get a successful result.
   - If WIT code was changed, run other WIT tests as appropriate.
   - Commit the revised code to GitHub.
   - Run the appropriate WIT tests on the other platforms.

Of course, you may have a combination of the above cases and therefore need to do a combination of the corresponding actions. (Case 3 is likely to require saving the new output.)

8. How to Add to the API Test

Normally when you add new functionality to WIT, you also need to add new tests. This section will explain how to add new tests to the API Test.

- `apiAll`:
  - Since this program is supposed to invoke each of WIT's API functions at least once, you need to revise `apiAll` any time you add a new API function. Here are some suggestions for how to do this:
    - It's probably a good idea to find something similar in `apiAll` and use it as an example.
    - If your new functionality involves more than one API function, it's best to collect them together in one or more functions that implement a small use case for the new capability.
    - Generally, you'll want the test to produce some indicative output so that if something changes, the change has a chance to show up in the diff output.
    - Note that if you add a call to `witOptImplode` or `witStochImplode` to `apiAll`, the call must be conditional on the value of the `cplexEmbedded` attribute. The `apiAll` code retrieves the value of this attribute from WIT and stores it in a global variable, `cplexEmbedded`.
- `msgAll`:
  - This is explained in section 21.
- `extOptTest`:
  - You probably won't need to add to this test.
- `errTest`:
  - The purpose of `errTest` is to test WIT's response to error conditions that cannot be produced running the Stand-Alone Executable.
  - Note that each error test case is implemented as a separate invocation of the `errTest` program, because the errors being tested normally cause the program to terminate.
  - To add a new case to `errTest`, perform the following steps:
    - Determine the index of the test case: N, where N-1 is the highest test case index that already exists. For example N=72.
    - Add a test case function, called `testCase{N}`, e.g., `testCase72`.
    - You can use other `testCase` functions as examples.
    - Usually the `testCase` function will end with a call to `terminationExpected`, which prints a special error message if it is called. Normally the execution of a `testCase` function will not reach its last line, because WIT will terminate with an error message when it detects the error that's being tested. The call to `terminationExpected` prints the appropriate error message if that goes wrong and the last line is reached.
    - The `errTest` program accepts a single string argument whose value is the index of the text case being requested. The main program contains a long if-then-else chain to call the requested test case function. You need to add an “else if” clause to this if-then-else chain, calling your test case function. For example:

```
    else if (theArg == "72")
        testCase72 ();
```

- The if-then-else chain concludes with an error message indicating the largest test case index allowed. You need to increase the index that it prints.
- Finally, add a call to `errTest` to the `wit-api-test` script in `$WIT_HOME/wit/scripts`, passing your test case index as the argument. For example:

```
./errTest 72                    2>&1 | cat >> api-out-new.txt
```

9. How to Test WIT: The Thread Test

At present, the Thread Test is not being used or maintained.

10. How to Test WIT: The No-Solver Test

The WIT No-Solver Test is a test of WIT without any solver (i.e. COIN or CPLEX) embedded, using the API. Its purpose is to guard against the (unlikely) event that the code has somehow been changed in such a way that it works OK when a solver is embedded, but fails when no solver is embedded. It is implemented on both the Linux and Windows platforms. It should be used under the following circumstances:

- If you alter any WIT code that is conditional on whether or not COIN or CPLEX is embedded into WIT, you need to perform the test.
- If you alter the code in a way that is expected to alter the output of WIT, you'll need to perform the test so that you can revise the expected output, if necessary.
- If you alter the `apiAll` code, you'll need to perform the test so that you can revise the expected output, if necessary.

If you want an especially thorough test of WIT, you can perform the test for that purpose.

When you do this test, you should do it on both platforms (linux and winsdk).

The procedure (on each platform) is as follows:

First, as with the API test, the following directory must be in your PATH:

```
$WIT_HOME/wit/scripts
```

Next, you rebuild WIT with `exe_type=debug`, but without COIN or CPLEX embedded:

- `export WIT_COIN_HOME=`
- `export WIT_CPLEX_HOME=`
- `cd $WIT_HOME/wit/$PLATFORM`
- `make clean`
- `make exe_type=debug all`

Next, you invoke the WIT No-Solver Test script:

```
wit-no-solver-test
```

This script builds and runs `apiAll`. The `apiAll` program has many places where the code takes different action depending on whether or not COIN or CPLEX is embedded into the WIT library that it's using. (It uses WIT's global “coinEmbedded” and “cplexEmbedded” attributes for this purpose.) In this way, it is able to perform meaningful tests both with and without COIN or CPLEX.

The source code and data for this test (as well as the outputs of it) are stored in the following directory:

```
$WIT_HOME/wit/api-test
```

The script directs most of its output to the following file:

```
$WIT_HOME/wit/api-test/no-solver-out-new.txt
```

The script then runs a filtered diff, comparing this file against the following file:

```
$WIT_HOME/wit/api-test/no-solver-out-$PLATFORM.txt
```

The diff output is stored as:

```
$WIT_HOME/wit/api-test/no-solver-out-diff.txt
```

If the diff file is empty, the script will print the following lines and terminate:

```
No Significant differences were found.

The WIT No-Solver test was successful.
```

If it terminates with those lines, the test is successful.

If the diff file is not empty, then the test is unsuccessful and the script terminates with the following lines:

```
Significant differences were found. See:

   $WIT_HOME/wit/api-test/no-solver-out-diff.txt

 The WIT No-Solver test was unsuccessful.
```

After completing a successful No-Solver Test, you should restore WIT to a COIN-embedded and CPLEX-embedded build:

- `export WIT_COIN_HOME=`{Whatever}
- `export WIT_CPLEX_HOME=`{Whatever}
- `cd $WIT_HOME/wit/$PLATFORM`
- `make clean`
- `make exe_type=debug all`

If the No-Solver Test Is Unsuccessful

The procedure in this case is similar to the procedure for the API test.

How to Add to the No-Solver Test

Since the No-Solver test is based on apiAll, it gets revised whenever apiAll is revised. There's no need to do anything beyond this.

11. How to Test WIT: The COIN-Only Test

The WIT COIN-Only Test is a test of WIT with COIN embedded but not CPLEX, using the API. Its purpose is to guard against the (unlikely) event that the code has somehow been changed in such a way that it works OK when both solvers are embedded, but fails when only COIN is embedded. It is implemented on both the Linux and Windows platforms. It should be used under the following circumstances:

- If you alter any WIT code that is conditional on whether or not COIN is embedded into WIT, you need to perform the test.
- If you alter the code in a way that is expected to alter the output of WIT, you'll need to perform the test so that you can revise the expected output, if necessary.
- If you alter the `apiAll` code, you'll need to perform the test so that you can revise the expected output, if necessary.

If you want an especially thorough test of WIT, you can perform the test for that purpose.

When you do this test, you should do it on both platforms (linux and winsdk).

The procedure for the COIN-Only Test is very similar to that for the No-Solver Test, with the following differences:

- `export WIT_COIN_HOME=`{Whatever}
- The script is: `wit-coin-only-test`

The script directs most of its output to the following file:

```
$WIT_HOME/wit/api-test/coin-only-out-new.txt
```

The script then runs a filtered diff, comparing this file against the following file:

```
$WIT_HOME/wit/api-test/coin-only-out-$PLATFORM.txt
```

The diff output is stored as:

```
$WIT_HOME/wit/api-test/coin-only-out-diff.txt
```

Otherwise, the COIN-Only Test works the same way as the NoSolver Test.

12. How to Test WIT: The CPLEX-Only Test

The WIT CPLEX-Only Test is a test of WIT with CPLEX embedded but not COIN, using the API. Its purpose is to guard against the (unlikely) event that the code has somehow been changed in such a way that it works OK when both solvers are embedded, but fails when only CPLEX is embedded. It is implemented on both the Linux and Windows platforms. It should be used under the following circumstances:

- If you alter any WIT code that is conditional on whether or not CPLEX is embedded into WIT, you need to perform the test.
- If you alter the code in a way that is expected to alter the output of WIT, you'll need to perform the test so that you can revise the expected output, if necessary.
- If you alter the `apiAll` code, you'll need to perform the test so that you can revise the expected output, if necessary.

If you want an especially thorough test of WIT, you can perform the test for that purpose.

When you do this test, you should do it on both platforms (linux and winsdk).

The procedure for the CPLEX-Only Test is very similar to that for the No-Solver Test, with the following differences:

- `export WIT_CPLEX_HOME=`{Whatever}
- The script is: `wit-cplex-only-test`

The script directs most of its output to the following file:

```
$WIT_HOME/wit/api-test/cplex-only-out-new.txt
```

The script then runs a filtered diff, comparing this file against the following file:

```
$WIT_HOME/wit/api-test/cplex-only-out-$PLATFORM.txt
```

The diff output is stored as:

```
$WIT_HOME/wit/api-test/cplex-only-out-diff.txt
```

Otherwise, the CPLEX-Only Test works the same way as the NoSolver Test.

13. How to Test WIT: The Stand-Alone Test

The WIT Stand-Alone Test is a procedure that performs a very detailed test of WIT's capabilities by running the Stand-Alone Executable many times. It is implemented on the Windows SDK platform only. It should be used under the following circumstances:

- If you are upgrading to a new compiler level or a new environment of some kind on Windows SDK, you need to perform the test.
- If you upgrade either of the prerequisites to WIT (MCL or COIN or CPLEX), you need to perform the test.
- If you change WIT code, you need to perform the test.

The procedure is as follows:

First, in addition to the directories you need for WIT, you need to load the `wit-sa-test` directory from the WIT repository on GitHub. The `wit-sa-test` directory contains the following three top-level directories:

- `data` Contains the WIT input data files for the Stand-Alone Test, along with the corresponding saved output files.
- `params` Contains the control parameter files for the Stand-Alone Test.
- `scripts` Contains the scripts for the Stand-Alone Test.

Most of the input data files in the `data` directory were created by hand (by me) in order to test various aspects of WIT's capabilities as I was implementing them. The rest of them were contributed by the developers of WIT's main applications (GIView/Scorpio, SCE, ESO2), often documenting bugs that were subsequently fixed.

Next, you must add the following directory to your PATH:

```
$WIT_HOME/wit-sa-test/scripts
```

Next, you build the WIT Stand-Alone Executable with `exe_type=debug` (if you haven't already).

Next, you invoke the WIT Stand-Alone Test script:

```
wit-test-all
```

This script invokes a series of other scripts, each of which tests a different one of WIT's capabilities. For example, `wit-test-multi-route` tests the Multi-Route capability. As of this writing, `wit-test-all` invokes 28 different capability test scripts.

Each capability test script makes numerous calls to the script `wit-test-case`. This script performs a test of an individual test case for the capability. As of this writing, the Stand-Alone Test performs 707 individual case tests.

There are three arguments to the `wit-test-case` script:

- `$1` This is the subdirectory of `$WIT_HOME/wit-sa-test/data` in which the input data file for the test case is to be found. The file name is always `wit.data`.
- `$2` This the stem of the name of the parameters file.
- `$3` If this is empty or ≠ “save”, the test will be run.
  If this is “save”, the test will not be run; the results will be saved instead.

Consider for example:

```
wit-test-case multi-route/case02 heur-spl3
```

This invocation performs a case test using the following files as the input data file and control parameter file:

```
$WIT_HOME/wit-sa-test/data/multi-route/case02/wit.data
```

```
$WIT_HOME/wit-sa-test/params/heur-spl3.par
```

The output files for the case test are placed the following directory, created by the script:

```
$WIT_HOME/wit-sa-test/data/$1/$2/output-new
```

For example:

```
$WIT_HOME/wit-sa-test/data/multi-route/case02/heur-spl3/output-new
```

After the script runs the Stand-Alone Executable, it then runs a script `wit-test-case-diff` that performs a filtered diff on the output files, using the following directory as the basis of comparison:

```
$WIT_HOME/wit-sa-test/data/$1/$2/output-sav
```

Specifically, each file in the `output-sav` directory is compared to the corresponding file in the `output-new` directory with a filtered diff.

How to Tell Whether or Not the Stand-Alone Test was Successful

The Stand-Alone Test is considered to be successful if and only if each of the capability tests is successful. A capability test is considered to be successful, if and only if each of the filtered diffs for each of its case tests produces an empty diff output.

Specifically, when a capability test begins, it prints (to stdout) a heading identifying the test, preceded and followed by a line of underscores. For example:

```
_____________________________________________________________

Test of Multi-Route
_____________________________________________________________
```

When a capability test is successful, that's all it prints. So when the Stand-Alone Test is successful, this is what it prints:

```
_____________________________________________________________

Test of Multi-Route
_____________________________________________________________
_____________________________________________________________

Test of NSTN Build-Ahead
_____________________________________________________________
_____________________________________________________________

Test of ASAP Build-Ahead
_____________________________________________________________
{25 more capability test headings...}
```

If a capability test is unsuccessful, at least one of its filtered diffs produces a non-empty output. When the output of a filtered diff (from a particular case test) is non-empty, the diff output is printed, preceded by a heading indicating which files were being compared. For example, suppose a difference was found in file `soln.out`

when running a test on `nstn/case02/wit.data` with parameter file `heur-spl3.par`. The script would print it as follows:

```
=======================================================
diff
   nstn/case02/heur-spl3/output-sav/soln.out
   nstn/case02/heur-spl3/output-new/soln.out
=======================================================

55c55
<      0        7.000       10.000
---
>      0        6.000       10.000
```

Here is an example of the output of an unsuccessful Stand-Alone Test:

```
_____________________________________________________________

Test of Multi-Route
_____________________________________________________________
_____________________________________________________________

Test of NSTN Build-Ahead
_____________________________________________________________


=======================================================
diff
   nstn/case02/heur-spl3/output-sav/soln.out
   nstn/case02/heur-spl3/output-new/soln.out
=======================================================

55c55
<      0        7.000       10.000
---
>      0        6.000       10.000


=======================================================
diff
   nstn/case02/heur-spl3/output-sav/wit-log.txt
   nstn/case02/heur-spl3/output-new/wit-log.txt
=======================================================

105c105
< Permanent Commit: Part C, Period 0, Qty: 7.000
---
> Permanent Commit: Part C, Period 0, Qty: 6.000
_____________________________________________________________

Test of ASAP Build-Ahead
_____________________________________________________________
{25 more capability test headings...}
```

This output indicates that the NSTN Build-Ahead Test was unsuccessful, with differences in two files. All other capability tests were successful.

14. How to Test WIT: If the Stand-Alone Test is Unsuccessful

If the output of the Stand-Alone Test script is large, you'll need to redirect it to a file, so you can work with it.

Usually the Stand-Alone Test will be unsuccessful only on a proper subset of the capability tests. It's generally helpful to rerun the failing capability tests individually so you can work with them one at a time. To find the names of the capability test scripts, look at the `wit-test-all` script. In the example of the previous section, you would rerun the `wit-test-nstn` script.

Then you would study the files that are flagged as different along with the corresponding input data file. In the above case, you'd study `nstn/case02/wit.data` and the resulting `soln.out` and `wit-log.txt` files in the `output-sav` and `output-new` directories.

As with the API test, once you have determined the exact nature of the problem, there are essentially three kinds of actions to take:

1. You might decide that the new output is correct and should become the new basis of comparison. In this case, you need to save the new output of the particular capability test in question. To save the output of a capability test, you take the following steps:
   - Run the capability test script with "save" as the argument, e.g.:

     ```
     wit-test-nstn save
     ```

     - This cleans up the `output-new` directory a bit and then renames it to `output-sav`.
     - It does not run the test.
   - Rerun the capability test, to get a successful result.
   - Eventually, you will need to commit the new `output-sav` directory to GitHub.
   - To save the output of all of the capability tests, issue the following:

   ```
   wit-test-all save
   ```

2. You might decide that the new output is correct, but is too variable to be used as the basis of comparison. In this case, you need to take these steps:
   - Add to the filter (the -I lines) in the "diff" command that's in the `wit-test-case-diff` script.
   - Rerun the capability test, to get a successful result
   - Eventually check-in and deliver the revised script.
3. You might decide that the new output is incorrect. In this case:
   - Revise the WIT source code and/or the input data file as needed.
   - Rerun the capability test, to get a successful result
   - Run all appropriate WIT tests.
   - Eventually check-in and deliver the revised code/data.

Of course, you may have a combination of the above cases and therefore need to take a combination of the corresponding actions.

15. How to Add to the Stand-Alone Test

The Stand-Alone Test is the place to test any new functionality that you add to WIT, as long as it's something that can be tested by running the Stand-Alone Executable, including both "positive" functionality and error checking. The essential advantage of using the Stand-Alone Executable for testing is clarity: Each test case is defined by a WIT data file and a parameters file, whereas API-based test cases are defined by source code.

As you know, the Stand-Alone Test consists of a series of capability tests, each of which consists of a series of series of individual test cases.

To add a test case to an existing capability test, perform the following steps:

- Usually, you will create a new data file for the case. As explained in section 13, the pathname for a data file in the Stand-Alone Test must be of the following form:

```
$WIT_HOME/wit-sa-test/data/$1/wit.data
```

where $1 is the value of the first argument to the `wit-test-case` script.

- Sometimes, you can use an existing data file, with a new parameters file.
- Usually, you can use an existing parameters file. Once again, these are located at:

```
$WIT_HOME/wit-sa-test/params
```

- If none of the existing parameters files is appropriate for your test, create a new one.
    - In this case, copy the first two lines from an existing parameters file into your new one. These two lines set the `log_ofname` and `data_ifname` parameters to the values that are needed by the Stand-Alone Test scripts.
- Next, add an invocation of the `wit-test-case` script to the capability test script that you are working on, e.g.:

```
wit-test-case nstn/case21      heur-spl3 $1
```

    - Copying and modifying an existing line in the capability test script is a good way to do his.
- Run the modified capability test script.
    - It will print an error message, indicating that your `output-sav` directory does not exist. You can ignore this error message for now.
- Look at the content of the files in the relevant `output-new` directory and verify that the output is what you want. If it is not, modify the input files and repeat these steps, as needed.
- When the output is correct, run the capability test script with "save" as the argument, e.g:

```
wit-test-nstn save
```

- Run the capability test script again, the normal way. This time there should be no error message and no diff output.

If you are adding a whole new capability to WIT, you'll need to add a new capability test. To do this, perform the following steps:

- You'll probably need to add a new directory to the data directory, e.g.,

`$WIT_HOME/wit-sa-test/data/foo`

- Create a new capability test script, e.g.,

`wit-test-foo`

    - The name of the script doesn't have to match the name of the directory, but it often does.
    - It's probably best to copy and modify an existing capability test script.

- Add test cases to the new capability test script as needed, using the procedure given above.
- Add an invocation of the new capability test script to the Stand-Alone Test script, `wit-test-all`.
- Run the new `wit-test-all`.

16. How to Build and Test WIT-J

At present,WIT-J has not been brought up to date in the open-source version of WIT on GitHub, so there is no need to maintain it.

17. A Walk Through the Makefiles

One aspect of maintaining MCL, WIT, and WIT-J is the maintenance of their makefiles. The makefiles for WIT, etc. can be hard to follow, because in each case, the build process in question is defined not by just one makefile, but by several makefiles, nested together using `include` statements. To aid you in finding your way through these nested makefiles, this section will present a walk-through of one case of the WIT build process.

Suppose you are on the Linux platform and wish to build the WIT Stand-Alone Executable in debug mode. As explained in Section 5, you would set up some environment variables, cd to the `$WIT_HOME/wit/linux` directory, and then issue the following command:

```
make exe_type=debug all
```

What happens?

Assumptions:

- `WIT_COIN_HOME` is defined as null, so COIN is not to be embedded.
- `WIT_CPLEX_HOME` is defined as non-null, so CPLEX is to be embedded.
- `WIT_GEN_PARSER=YES`, so the parser is to be generated.
- MCL is built.
- Nothing in WIT has already been built.

The first thing that the make facility does is to read in all the relevant makefiles. The complete set of nested makefiles that are processed in response to the above command are as follows:

- `$WIT_HOME/wit/linux/Makefile`
  - This is the default makefile used by the make facility, i.e., `Makefile` on the current directory.
  - This makefile is responsible for the aspects of the build that are specific to both WIT and the Linux platform.
- `$WIT_HOME/config/p_linux.mk`
  - This makefile is responsible for the aspects of the build that are specific to the Linux platform, but are independent of the program being built (WIT, SCE, etc.).
- `$WIT_HOME/wit/src/appl.mk`
  - This makefile is responsible for the aspects of the build that are specific to WIT, but are platform-independent.
- `$WIT_HOME/config/ds.mk`
  - This makefile is responsible for the aspects of the build that are (mostly) independent of both the program being built and the platform. (Not all aspects are used in all cases.)
- `$WIT_HOME/config/t_debug.mk`
  - This makefile defines a few macros in a way that's specific to `exe_type=debug`.
- `$WIT_HOME/config/sfDefine.mk`
- `$WIT_HOME/config/legacy.mk`

These two makefiles define a few macros that are not used by the makefiles for WIT, MCL, or WIT-J, but are used for other programs in the WIT fork on GitHub.

- `$WIT_HOME/wit/src/sources.mk`
  - This makefile defines the macro `lib_objects`, which lists most of the object files that go into the WIT library.
- `$WIT_HOME/wit/src/appl_unix.mk`
  - This makefile is responsible for the aspects of the build that are specific to WIT and apply to all Unix-like platforms, i.e., all of our platforms except Windows SDK.
  - On Windows SDK, the makefile content that's functionally equivalent to `wit/src/appl_unix.mk` is contained in `wit/winsdk/Makefile`.

The actual nesting of included makefiles can be represented as follows, where indentation indicates file inclusion:

- `Makefile`
  - `p_linux.mk`
  - `appl.mk`
    - `ds.mk`
      - `t_debug.mk`
      - `sfDefine.mk`
      - `legacy.mk`
    - `sources.mk`
  - `appl_unix.mk`

After the make facility has read in all the nested makefiles, it processes the rules in them. (Actually, it might be concurrent; I'm not sure.) As you know, the processing is hierarchical, starting with the requested target and working downward through a tree of out-of-date prerequisite targets, processing each prerequisite target before its dependent target. The following is a representation of the tree of targets for building the WIT Stand-Alone Executable when nothing is already built, with indentation indicating the prerequisite relationships:

`all`: The primary target is the `all` target, whose rule is specified in `appl.mk`. This rule takes no action of its own, but its prerequisite is `wit`:

- **`wit`**: The rule for this target is specified in `appl_unix.mk`. It builds `wit` as an executable linked together from `wit.o`, and `libwit.a`, which are its prerequisites:
    - **`wit.o`**: This target is covered by the implicit rule for .o files, which compiles `wit.o` from `wit.C`. Actually, our makefiles don't specify this rule on linux; the make facility defines it based on macros that we define, e.g., `CXX=/usr/bin/g++` in `p_linux.mk`.
    - **`libwit.a`**: The rule for this target is specified in `appl_unix.mk`. It builds `libwit.a` as a static library formed from the .o files in `lib_objects`, `MclModule`, `CplexModule`, and `BuildDate.o`, which are its prerequisites:
        - **The .o files listed in `lib_objects`**: Most of these are covered by the implicit rule for .o files, which compiles them from the corresponding .C files. Two of the .o files, `witParse.o` and `witLexer.o`, have prerequisites: `witParse.C` and `witLexer.C`:
            - **`witParse.C`**: The rule for this target is specified in `Makefile`. It builds `witParse.C` from `witParse.y`, which is half of parser generation.
            - **`witLexer.C`**: The rule for this target is specified in `Makefile`. It builds `witLexer.C` from `witLexer.l`, which is the other half of parser generation.
        - **`Session.o`**: This .o file is listed in `lib_objects`, but uses an explicit rule, specified in `appl.mk`, which compiles `Session.o` from `Session.C` with special flags.
        - **`MclModule`**: The rule for this target is specified in `appl_unix.mk`. It builds `MclModule` as a pre-linked object file formed from the WIT object files that invoke MCL and the MCL library itself. Its prerequisites are the WIT object files that invoke MCL:
            - **The WIT object files that invoke MCL**: These are covered by the implicit rule for .o files.
        - **`CplexModule`**: The rule for this target is specified in `appl_unix.mk`. It builds `CplexModule` as a pre-linked object file formed from `CplexIf.o` and the CPLEX library. Its prerequisite is `CplexMgr.o`:
            - **`CplexIf.o`**: The rule for this target is specified in `appl.mk`. It compiles `CplexIf.o` from `CplexIf.C` with special flags.
        - **`BuildDate.o`**: The rule for this target is specified in `appl_unix.mk`. The `BuildDate.o` target depends on all other aspects of WIT, but is compiled like any other .o file. Its purpose is to note the date on which WIT was built and store it as a string that can be used by WIT.

18. Conventions of the WIT Source Code

Before we look at the specifics of the WIT source code, it will be helpful to note some of its conventions:

- Class names begin with "`Wit`", e.g. "`WitPart`".
- Any time a class is mentioned in a context that does not require the actual class name, the "`Wit`" prefix is dropped:
  - In comments, class `WitPart` is called `Part`.
  - A local variable of type `WitPart *`, might be called `thePart`.
  - A data member of type `WitPart *`, might be called `myPart_`.
  - The only exception to this is class `WitRun`, which is always called `WitRun`.
- Most classes are designed to have their instances accessed by pointers:
  - A typical local variable for a `Part` would be declared as "`WitPart * thePart;`", not "`WitPart thePart;`".
- A few classes are designed to have their instances accessed without pointers. These more mostly utility/collection classes, such as `FlexVec <Elem>`.
- Most classes do not support `const`, because in normal usage, `const` instances are not expected to arise.
  - Once again, the main exceptions to this are the collection classes; they usually do support `const`.
- The names of data members end with underscore, e.g., `myPart_`.
- WIT uses quite a few macros.
- Most of the time, a class will be declared in a header file whose name matches the class name, without the "`Wit`" prefix, e.g., `Part.h`.
  - Sometimes several very small class declarations are placed in a single header file.
- Some of the time, a class will be implemented (defined) in a .C file whose name matches the class name, without the "`Wit`" prefix, e.g., `Part.C`.
  - Often several small class implementations are placed in a single .C file.
  - The purpose of putting more than one implementation in the same .C file is to reduce the number of compilation units, to get faster compiles.
- When the implementation of a class is in a .C file whose name does not match the name of the header file containing its declaration, the main comment for the class declaration will give the name of the implementation file. For example, the main comment for the declaration of class `OptVar` contains:

```
// Implemented in OptMisc.C.
```

19. Structural Concepts

Before going any further, it will be helpful to define a few structural concepts that are relevant to WIT.

The WIT code consists primarily of many classes (hundreds). Most of WIT's classes are organized as subsystems. As defined in [patterns], a subsystem is “an independent group of classes that collaborate to fulfill a set of responsibilities”. A subsystem may contain one or more other subsystems that fulfill narrower sets of responsibilities, thus creating layers. WIT's subsystems tend to be layered.

An important concept in WIT's structure is “ownership”. Object theA will be considered to own object theB, if and only if theA has responsibility for:

- Maintaining a reference to theB
- Destroying theB

In most cases, the owning object will also have responsibility for creating the owned object.

A related concept is “holding”. Object theA will be considered to hold object theC, if and only if:

- theA is theC, or
- There is an object theB, such that:
    - theA owns theB, and
    - theB holds theC

(Note that this is a recursive definition.) “Hold” is the transitive and reflexive closure of “own”: An object holds itself and all the objects that it owns and all the objects that they own and so on.

Class C is called the “holder” of subsystem S, if and only if:

- there is at most one an instance of C in a WitRun, and
- that instance holds all the objects in the WitRun that are instances of classes in S.

Many of WIT's subsystems employ the Facade pattern. The definitive explanation of the Facade pattern is given in [Patterns], but it can be summarized as follows: You have a subsystem, and one of its classes, the facade class, functions as the single point of contact (or main point of contact) to the rest of the subsystem from objects outside the subsystem. When an object outside the subsystem needs to use the functionality of the subsystem, it normally invokes a member function of the facade class, which relays the request to other classes of the subsystem (internal classes). The basic purpose of using facades is to decouple the subsystems, to make each subsystem appear as simple as possible to the rest of the program. It's a kind of encapsulation at the subsystem level.

Actually, as defined in [Patterns], the facade class doesn't have to be a strict gate keeper of the subsystem; outside objects are allowed to access the internal classes directly, if necessary. In contrast, most of WIT's facades are “strict” facades: access to the internal objects from outside objects is impossible, because the pointers to the internal objects are not made available outside the subsystem. This creates strong subsystem encapsulation.

In WIT, a facade of a subsystem is always (I think) the holder of that subsystem, but a holder of a subsystem doesn't necessarily need to be a facade for it.

20. A Structural Overview of WIT

The section will present an overview of the structure of WIT. The approach will be to discuss the holder classes of the main subsystems of WIT, along with other related classes, starting from the outside and working in.

- `WitRun`:
  - A WIT application program will own one or more `WitRuns`.
  - As you know, most of the API functions take a `WitRun` pointer as the first argument. These functions are sometimes called the "outer API functions". For almost every outer API function, there is a corresponding public non-static member function of class `WitRun` with the same (or very similar) name. These are called the "inner API functions". The inner API function will have an argument list similar to the outer one, but without the `WitRun` argument. In almost all cases, the outer API function calls the matching inner one, invoking it on the `WitRun` and passing the other arguments.
  - An inner API function has responsibility for:
    - Error checking
    - Issuing relevant messages
    - Translating the arguments into terms suitable for internal WIT functions (and the reverse)
    - Calling the appropriate internal WIT function(s)
  - Because it implements almost the entire API, class `WitRun` is very large. For this reason, while its declaration is entirely in `WitRun.h`, its implementation is spread out across several files, each responsible for a different aspect of the API: `WitRun.C`, `globApi.C`, `optApi.C`, `stochApi.C`, `partApi.C`, `demApi.C`, `opnApi.C`, `bomApi.C`, `subApi.C`, `bopApi.C`, and `msgApi.C`.
  - Clearly, `WitRun` is the holder and strict facade for the whole of WIT.
  - More specifically, `WitRun` is the holder of the API subsystem, which implements the API.
- `ApiMgr`:
  - A `WitRun` owns one `ApiMgr`., part of the API subsystem.
  - Responsible for maintaining the state of API, particularly when entering or leaving an API function. Also handles exceptions thrown from within WIT.
- `Session`:
  - A `WitRun` owns one `Session`.
  - This is the holder of the Session Subsystem, which is responsible for all non-API aspects of WIT..
- `MsgFacility`:
  - A `Session` owns one `MsgFacility`.
  - This is the holder of the Message Subsystem, which is responsible for handling WITs messages. The classes of the message facility are the only classes that make direct calls to MCL. `MsgFacility` is a non-strict facade for the Message Subsystem.

- `Problem`:
  - A `Session` owns 0 or 1 `Problems`: 0 at first; 1 after `witInitialize` is called.
  - This is the holder of the Problem Subsystem, which is responsible for all non-API aspects of WIT that only apply after `witInitialize` has been called.
  - The Problem subsystem is the "main aspect" of WIT, once you get past the outer layers.
  - A `Problem` is the all-important central object of a `WitRun`: It has direct ownership of many important holder objects, and many objects in the Problem Subsystem have a pointer back to the `Problem` that holds them. It connects the internal objects of `WIT` together.
- `CompMgr`:
  - A `Problem` owns one `CompMgr`.
  - This is the holder of the Component Subsystem, which is responsible for the `Components` of a `WitRun`. (See below for the definition of a `Component`.)
  - A `CompMgr` owns all of the `Components` for a `WitRun`: It provides multiple forms of access to them and is responsible for deleting them for `witDeleteRun` and `witPurgeData`.
- `Component`:
  - Class `Component` is the base case for the Component classes of WIT, the classes whose instances represent parts, operations. etc., the "WIT Data Objects".
  - The following shows the inheritance hierarchy under class Component, where "class A inherits from class B" is indicated by B being indented under A:
  - `Component`
    - `GlobalComp` Maintains global data not for opt implosion
    - `OptComp` Maintains global data for opt implosion
    - `DelComp` Deletable Component (i.e., subject to `witPurgeData`)
      - `Node` Node in the BOM graph
        - `Part`
          - `Material` Material Part
          - `Capacity` Capacity Part
        - `Operation`
      - `BillEntry`
        - `ConsEntry` Consumption Bill Entry
          - `BomEntry` BOM Entry
          - `SubEntry` Substitute
        - `BopEntry` BOP Entry
      - `Demand`
  - A `CompMgr` owns one `GlobalComp` and and `OptComp`.

- A Problem owns one instance of each of the following subsystem holder classes:
  - `Preprocessor`:
    - Holder and facade for the Preprocessing Subsystem, which is responsible for preprocessing.
  - `Postprocessor`:
    - Holder and facade for the Postprocessing Subsystem, which is responsible for … well, you get the idea.
  - `HeurImploder`:
    - Holder and facade for the Heuristic Implosion Subsystem.
  - `EqAllocator`:
    - Holder and facade for the Equitable Allocation Subsystem.
  - `HeurAllMgr`:
    - Holder and facade for the Heuristic Allocation Subsystem.
  - `DetImploder`:
    - Holder and facade for the Deterministic Optimizing Implosion Subsystem.
  - `StochImploder`:
    - Holder and facade for the Stochastic Implosion Subsystem.
  - `PipMgr`:
    - Holder and facade for the PIP Subsystem.
  - `OrigMrpExp`:
    - Holder and facade for the MRP Subsystem.
  - `FSS`
    - Holder and facade for the FSS Subsystem.
  - `DataReader`:
    - Holder and facade for the Data Reading Subsystem, which is responsible for reading an input data file.
  - `DataWriter`:
    - Holder and facade for the Data Writing Subsystem, which is responsible for writing an input data file.
  - `SolnWriter`:
    - Holder and facade for the Solution Writing Subsystem, which is responsible for writing an solution files.

The following shows the high-level classes of WIT, where ownership is represented by indentation:

- WitRun
  - ApiMgr
  - Session
    - MsgFacility
    - Problem
      - CompMgr
        - Component
      - Preprocesser
      - Postprocessor:
      - HeurImploder:
      - EqAllocator:
      - HeurAllMgr:
      - DetImploder:
      - StochImploder:
      - PipMgr:
      - OrigMrpExp:
      - FSS:
      - DataReader:
      - DataWriter:
      - SolnWriter

Note: The Data Reading Subsystem was mostly written by Juan Lafuente. The documentation for his code is in GitHub at:

`WIT/doc/old/witDoc.txt`

`WIT/doc/old/witSyntx.txt`

The code has evolved somewhat since Juan wrote these documents, but their content is still relevant.

21. How to Add a Message to WIT

As you know, WIT produces its output by issuing "messages", which can be controlled in various ways by the application program. If you are making modifications to WIT, there's a good chance that you will need to add/modify/remove one or more of WIT's messages. This section will explain how to add a new message to WIT. (How to modify or remove a message should be easy enough to figure out.)

WIT works with messages by invoking MCL. A message is an object, an instance of the WIT class `WitMsg`, which inherits from the MCL class `MclMsg`. Adding a new message to WIT's output, normally consists of the following three steps:

1. Add code to create the message.
2. Add code to issue the message,
3. Add code to test the message, if possible.

To create a new message, you add code to the implementation of class `MsgBuilder` (in `MsgBuilder.C`). Here is an example:

```
makeMsg (
   "demandIdDdMsg",
   134,
   info_,
   "Part          \"%1$s\",\n"
   "Demand Stream \"%2$s\":");

currentMsg_->preceedingLineFeeds   (1);
```

This code is located in the `buildMsgs` function of class `MsgBuilder`.

The main code to be added is call to `makeMsg`, which is a member function of class `MsgBuilder`. The arguments to `makeMsg` are as follows:

- `theMsgID` ("demandIdDdMsg"):
  - This argument is of type `MsgID`, which is just a typedef for `const char *`. This will be the message's message ID, the string that uniquely identifies it internally among WIT's messages.
- `theExtNum` (134):
  - This argument (of type int) will be the message's external number, the number that uniquely identifies it externally among WIT's messages. By default, this number is printed when the message is issued, e.g., "`WIT0134I`". It is used as an argument to various API functions that manage the messages. Note that the messages must be created in order of increasing external number. If this condition is violated, `witNewRun` will print an error statement to `stderr` and terminate.
- `theLevel` (`info_`):
  - This argument (of type `MclLevel`) will be the message's message level. This concept is explained in [WIT Guide]. It can take any of these four values, defined as data members of class `MsgBuilder`:
    - `info_`
    - `warning_`
    - `severe_`
    - `fatal_`

- `textVal ("Part \"%1$s\",\n" "Demand Stream \"%2$s\":");`
    - This argument (of type `const char *`) will be the message's text string. This will be explained below.

By convention, the last characters of the message ID correspond to the message level as follows:

- Informational: “Msg”
- Warning: “Wmsg”
- Severe: “Smsg”
- Fatal: “Fmsg”

After the message is created with `makeMsg`, its attributes can be set by calling member functions on it. To make this convenient, there is a data member, `MsgBuilder::currentMsg_` (of type `Msg *`), whose value is the most recently created message. In the example, the following line sets an attribute of the message that was just created:

```
currentMsg_->preceedingLineFeeds   (1);
```

This causes one linefeed to be added to the beginning of the output of the message. In general, some of the message attributes belong to class `MclMsg` and some belong to class `WitMsg`. For more information, see the declarations of these two classes.

The main task of creating a message is to define its text string. The text string of a message works similarly to the format string of a `printf` statement, but with the following differences:

- The arguments to the text string are explicitly numbered. In the example above `%2$s` indicates that the second argument is to be inserted at that point in the text and it must be a string argument.
    - The original purpose of explicitly numbering the arguments was to allow the message texts to be translated into other languages, which use different word orderings, but it's pretty clear by now that that is unlikely to happen.
- The set of valid format flags is different than the flags for a `printf` statement. These are the valid format flags for MCL and WIT:
    - **b**: The argument must be of type `bool` or `int` or `size_t` or `long` and is displayed as a boolean.
    - **d**: The argument must be of type `int` or `size_t` or `long`.
    - **f**: The argument must be of type `double`.
    - **g**: The argument must be of type `double`.
    - **s**: The argument must be of type `const char *` or `WitString` or `WitMsgFrag`.
    - **m**: The argument must be of type `WitMsgFrag`. See below for an explanation of class `WitMsgFrag`.
- There is an option to specify a vector type argument, by putting a “v” after the argument number, for example:

    ```
    "   External supply volumes:%1v$8.0f"
    ```

- When “v” is specified, the argument must be a `WitVector` type, for example, `WitVector <double>`, if the “f” format is specified, as above.

- The full specification of an argument consists of:
  - %, followed by
  - the argument number, followed by
  - an optional v for vector, followed by
  - $, followed by
  - an optional field width, followed by
  - an optional precision, followed by
  - the format flag

After you have added the code to create a message, you need to add code to issue it. Here is an example of how to issue a message:

```
myMsgFac () ("demandIdDdMsg",
   demandedPartName (),
   demandName_);
```

This code is in `Demand.C`. The message identifies a `Demand` for `witDisplayData`.

Here are the steps to take for issuing a message.

- First, you specify a reference to the `MsgFacility` for the `Session` in which your code is operating:
  - In most cases you will be in the implementation of some class that inherits from class `ProbAssoc`. If so, you just invoke the `myMsgFac ()` function of class `ProbAssoc` and it returns a reference to the `MsgFacility`. That is what's being done in the example.
  - Some classes that do not inherit from class `ProbAssoc` have their own version of `myMsgFac ()`, so you can just use that version. (for example `WitRun` and `ApiMgr`)
  - Otherwise, you need to call `myMsgFac ()` on some instance of a class that has it.
- Next, you invoke the overloaded function call operator, "`()`", on the `MsgFacility`. This is the `MsgFacility`'s message issuing function:
  - The first argument to this function is the message ID of the message to be issued, e.g., "`demandIdDdMsg`".
  - The subsequent arguments to this function are the arguments to the message, e.g., `demandedPartName ()` and `demandName_`.
    - The message issuing function will accept up to 15 message arguments.

After you have added the code to issue a message, you may need to add a test for it to the testing procedure:

- In some cases, you're adding a message to code that's already being invoked by an existing test. In this case, the output of the message will show up in the "diff" output when you do that test. All you need to do then is to check that the output is as desired and then perform the appropriate output saving procedure as explained in section 7.
- Otherwise, if the output of the new message doesn't show up in the "diff" output of any test, you should modify the tests so that it shows up somewhere.
- If it's an error message that can be triggered from the Stand-Alone Executable, you should revise the Stand-

Alone Test, adding a new case that causes that error condition. See section 15.

- If it's an error message that cannot be triggered from the Stand-Alone Executable, you should revise the error test portion of the API Test, adding a new case that causes that error condition. See section 8.
- If it's not an error message (i.e., the message level is “informational” or “warning”), you should add code to `msgAll` that issues the message, if this is practical.
- If you have added a non-error message that is not issued from `msgAll`, `msgAll` will issue an message identifying your message as a “testable” message that it did not issue. If it's not feasible to have `msgAll` issue the new message (e.g. it only issues from the Stand-Alone Executable), you should turn off the “testable” attribute of the new message, by adding the following code just after the message was created with `makeMsg`:

```
currentMsg_->testable                (false);
```

This will keep `msgAll` from listing it as an unissued testable message. In this case, you should add it to the Stand-Alone Test, if possible.

Class `WitMsgFrag`

A `MsgFrag` (i.e. “message fragment”) is an object that stores a string (its “text”) and is intended to be used as a argument to a message. When a message is issued with a `MsgFrag` argument, the text of the `MsgFrag` is inserted into the output. The original purpose of using `MsgFrag`s as arguments instead of strings was to allow the text to be translated into other languages. But once again, this now seems unlikely to happen. Nevertheless, since the code has quite a few `MsgFrag`s, you may have to deal with them at some point.

A `MsgFrag` has a `MsgFrag` ID, which is a string that uniquely identifies it internally among WIT's `MsgFrags`. WIT's `MsgFrags` are created in the `buildMsgFrags` function of class `MsgBuilder`. The process is simpler than building a message and so further explanation will be given here.

To get an existing `MsgFrag` you invoke the `myFrag` function of class `MsgFacility`. This function takes one argument: the `MsgFrag` ID and returns the corresponding `MsgFrag`. You get the `MsgFacility` reference in the same way as you do for calling the message issuing function.

Here is an example of code that issues a message that uses a `MsgFrag` as an argument:

```
myMsgFac () ("categorySizeDdMsg",
   myMsgFac ().myFrag ("capacityFrag"),
   nCapacities);
```

This code is located in `Problem.C`. The text for the `capacityFrag` `MsgFrag` is “`Capacity`”, which is inserted into the output of the message.

22. How to Add an Attribute to WIT

When you add a new capability to WIT, usually, you'll need to add one or more attributes to control the new capability, or get new data in or out, etc. The procedure for adding a new attribute is well-defined and depends on whether or not the attribute is modifiable (i.e., its value can be explicitly set from the application program). The procedure given below will use the following hypothetical attribute as an example:

- Attribute: `exampleVol`
- Type: Vector of floats
- Owning Component Class: Operation
- Modifiable? Yes

To add a new attribute to WIT, perform the following steps:

- If the attribute is modifiable, add its default value function to the declaration of the owning class:
  - For example, in Operation,h:

    ```
    defaultValue (double defExampleVol, 0.0)
    ```

  - This is a static function member of the owning class.
  - Normally you use the `defaultValue` macro, as in the example. This creates the declaration of an inline default value function, so no further implementation is needed.
  - For a vector valued attribute, you usually just specify the default value as a scalar, as in the example.
  - Note that any type that is externally a float type is stored internally as the corresponding double type, as in the example.
  - The default value functions are declared in the same order as the data members for the attributes.
- Add the data member for the attribute to the declaration of the owning class:
  - For example, in Operation.h:

    ```
    WitDblFlexVec exampleVol_;
    ```

  - The name of the data member is the name of the attribute with an underscore suffix.
  - In the example, `DblFlexVec` is used as the type. This a space-efficient class for storing a time-vector of doubles: If the value is the same in all periods, it tends to store the vector as a scalar. It's the usual internal type of a “float vector” attribute. See FlexVec.h.
  - A comment explaining the attribute is placed under its data member declaration.
  - The attribute data members are declared in an order that seems “logical”: Input attributes are placed together. Opt implosion attributes are placed together, etc.

- Add the initializer for the attribute data member to the constructor of the owning class:
  - For example, in Operation.C:

    ```
    exampleVol_ (myProblem (), defExampleVol ()),
    ```

  - If the attribute is modifiable, you invoke the default value function for this; otherwise, just you just hard code the value.
  - The initializers go in the same order as the declarations.

- Add the access function for the attribute to the declaration of the owning class:
  - For example, in Operation,h:

    ```
    accessFunc (const WitDblFlexVec &, exampleVol)
    ```

  - This is a public member function that returns the value of the attribute.
  - Normally you use the `accessFunc` macro, as in the example. This creates the declaration of an inline access function, so no further implementation is needed. The name of the access function is the same as the name of the attribute.
  - Note that in the example, the returned value has a `const` reference type. This reflects the fact that class `DblFlexVec` deliberately avoids having a copy constructor, so as to prevent needless expensive copying.
  - The access functions are declared in the same order as the data members for the attributes.
- Add displaying code for the attribute to the `display` member function of the owning class:
  - For example, in `Operation::display:`

    ```
    myProblem ()->display ("exampleVolDdMsg", exampleVol_);
    ```

  - The `display` function of a Component class is the function that displays the value of the Component's attributes for `witDisplayData`.
  - This is done in a variety of ways; see the `display` functions for examples.
  - This will normally involve either adding a new message (as in the example) or modifying an existing one. See section 21, “How to Add a Message to WIT”.

- If the attribute is modifiable, add copying code for it to the `copyAttrsFrom` member function of the owning class:
  - For example, in `Operation::copyAttrsFrom`:

    ```
    exampleVol_ = theOpn->exampleVol_;
    ```

  - The `copyAttrsFrom` function of a Component class is the function that copies the value of each modifiable attribute from another instance of the Component class into the current instance. It is used by `witCopyData` and, e.g., `witCopyOperationData`.
  - The copying code is normally placed in the same order as the data members for the attributes.
- If the attribute is modifiable, add its "set" function to the owning class:
  - For example, in Operation.h:

    ```
    void setExampleVol (const WitIVRealArg &);
    ```

  - The "set" function has the following responsibilities:
    - Final validity checking of the argument value, via asserts.
      - Note that primary validity checking (via severe error messages) is the responsibility of the API, not this function.
    - Revising the state of the `WitRun`; for example, taking it out of the "preprocessed" state, if appropriate.
    - Setting the value of the data member to the argument value.
  - Note that in the example, the argument is of type `const IVRealArg &`. This is a type that is used by the inner API and will be explained later in this section.
  - The "set" functions are placed in the same order as the data members for the attributes.
- If the attribute is not modifiable, some other means of giving it a value from within WIT will need to be implemented.
  - This is done is a variety of ways, beyond the "rudimentary" scope of this document.
- If the attribute requires special treatment in stochastic mode, consider adding it to `StochAttMgr::makeStochAtts`.
  - This is rare.

- Add the inner API “get” function for the attribute:
  - For example, in WitRun.h:

```
void witGetOperationExampleVol (
   const char *         operationName,
   const WitOVRealArg & exampleVol);
```

  - The inner API “get” functions are declared in WitRun.h.
  - They are implemented in the Component-class-specific inner API source files, e.g. optApi.C.
  - The inner API “get” function has responsibility for:
    - Validity checking of the arguments, issuing severe error messages, as needed.
    - Issuing the relevant informational message(s).
    - Retrieving the requested value from the Component.
    - Setting the argument to the retrieved value.
  - In the example, the argument is of type `const OVRealArg &`. The name of this class means “Output Vector Real Argument”. It represents vector of real numbers for output. In the outer API, this corresponds to an argument of type `float * *` or of type `double * *`. When interacting with WIT internally, it acts like a vector of doubles. This is one of the classes that enables WIT to have two versions of its real valued API functions, one for floats and one for doubles.
  - The inner API “get” functions are ordered alphabetically.

- If the attribute is modifiable, add the inner API “set” function for the attribute:
  - For example, in WitRun.h:

```
void witSetOperationExampleVol (
   const char *         operationName,
   const WitIVRealArg & exampleVol);
```

  - The inner API “set” functions are declared in WitRun.h.
  - They are implemented in the Component-class-specific inner API source files, e.g. optApi.C.
  - The inner API “set” function has responsibility for:
    - Validity checking of the arguments, issuing severe error messages, as needed.
    - Issuing the relevant informational message(s).
    - Calling the Component's “set” method.
  - In the example, the argument is of type `const IVRealArg &`. The name of this class means “Input Vector Real Argument”. It represents vector of real numbers for input. In the outer API, this corresponds to an argument of type `const float *` or of type `const double *`. When interacting with WIT internally, it acts like a vector of doubles. This is one of the classes that enables WIT to have two versions of its real valued API functions, one for floats and one for doubles.
  - The inner API “set” functions are ordered alphabetically.

- Add the outer API “get” function for the attribute:
  - For example, in wit.h:

```
witReturnCode witGetOperationExampleVol (
   WitRun * const     theWitRun,
   const char * const operationName,
   float * *          exampleVol);
```

  - The following comments apply to all outer API functions, not just “get” functions for attributes:
    - The outer API functions are declared in wit.h and implemented in api.C.
    - An outer API function has responsibility for:
      - Calling member functions of class `ApiMgr`, to maintain the state of the API.
      - Calling the corresponding inner API function on the `WitRun` that's the first argument of the outer API function and passing the other arguments.
      - Doing all this in the context of a try block that will catch any exception thrown from within WIT. In particular, when WIT issues a severe or fatal error message (anywhere inside of WIT), it throws an exception that is to be caught by the outer API function's try block.
      - If an exception is caught, passing it to the `ApiMgr` for processing.
    - Since the above responsibilities are very uniform and must be fulfilled by hundreds of API functions, they are encoded into a macro: `STANDARD_API_IMPL`. Most outer API functions call this macro.
    - For example, in api.C:

```
witReturnCode witSetOperationExampleVol (
      WitRun * const      theWitRun,
      const char * const  operationName,
      const float * const exampleVol)
   {
   STANDARD_API_IMPL (
      theWitRun,
      witSetOperationExampleVol, (
         operationName,
         exampleVol))
   }
```

    - The outer API functions are ordered alphabetically.

- If the type of the attribute is a float/double type (as in the example), you actually need to add two versions of the output "get" function:
  - One version will use the `float` type. This version will have the ordinary name of the API function. The example above shows that case.
  - The other version will use the `double` type. This version will have the name with "Dbl" appended to the end, for example:

```
witReturnCode witGetOperationExampleVolDbl (
   WitRun * const     theWitRun,
   const char * const operationName,
   double * *         exampleVol);
```

  - The implementation of an outer API function that has double arguments uses a different macro to call the inner API function: `STANDARD_DBL_API_IMPL` .
- Both the `float` and the `double` outer API function call the same inner API function, which converts their `float/double` arguments into the appropriate "`RealArg`" arguments.

- If the attribute is modifiable, add the outer API "set" function for the attribute:
  - For example, in wit.h:

```
witReturnCode witSetOperationExampleVol (
   WitRun * const      theWitRun,
   const char * const  operationName,
   const float * const exampleVol);
```

  - The comments for the outer API "get" functions pretty much apply here, too (i.e., calling the macro, `float` vs. `double`, etc.).

- If the attribute is modifiable, add it to the Data Reading Subsystem:
  - For example, in SymTable.h:

```
static WitParseRC witSetOperationExampleVol();
```

  - And in SymTable.C:

```
{"set_operation_exampleVol", 1,
    WitSymTable::witSetOperationExampleVo l,
    {WitDRParmType::doubleVector}},
```

  - Also in SymTable.C:

```
WitParseRC WitSymTable::witSetOperationExampleVol()
   {
   ...
   }
```

  - This is the code that allows `witReadData` to read the attribute.
  - It's Juan Lafuente's code; I don't really know how it works.
  - Basically, you copy an existing case and modify it to your case.
  - All of this code is organized alphabetically.

- If the attribute is modifiable, add writing code for it to the `writeDataAttrs` member function of the owning class:
  - For example, in `Operation::writeDataAttrs`:

```
myDataWriter ()->writeVector (
   "exampleVol",
   exampleVol_,
   defExampleVol ());
```

  - The `writeDataAttrs` function of a Component class is the function that writes the value of each modifiable attribute to the message file for `witWriteData`.
  - The writing code is normally placed in the same order as the data members for the attributes.
- Add the following test code to `wit/api-test/apiAll.c`:
  - If the attribute is modifiable, add a call to the API "set" function for it.
  - If the attribute is modifiable and is of a `float/double` type, add a call to the `double` variant of the API "set" function for it.
  - Add a call to the API "get" function for it.
  - If the attribute is of a `float/double` type, add a call to the `double` variant of the API "get" function for it.
- Document the attribute in the WIT Guide Supplement:
  - Add an entry to Chapter 2, explaining the attribute.
  - If the attribute is modifiable, add a row for it to Table 6 in Chapter 4.
  - Add an entry for the API "get" function for it to Chapter 5.
  - If the attribute is modifiable, add an entry for the API "set" function for it to Chapter 5.
  - The "`double`" variants of API functions are not listed exhaustively in Chapter 5.
  - These are all organized alphabetically.
  - Mention the attribute and its API functions in the "Change History' in the Preface.
- Add the attribute to WIT-J.
  - At present, this can be skipped.

23. How to Add an Attribute to WIT-J

At present,WIT-J has not been brought up to date in the open-source version of WIT on GitHub, so there is no need to maintain it.

24. How to Add/Remove/Rename Source Files

As was indicated in the makefile walk-through (Section 17), one of the WIT makefiles defines a macro, `lib_objects`, that lists most of the WIT object file names. Thus if you add or remove or rename a WIT .C source file, you'll need to update this list. (Nothing needs to be done for .h files.) Here are the procedures:

The macro is in:

```
$WIT_HOME/wit/src/sources.mk
```

Adding a .C source file to WIT:

- Add its name to the `lib_objects` macro, with a `$(obj_suffix)` ending.
- Note that this list is kept in alphabetical order, to ease maintenance.
- Invoke `make depend`.

Removing a source file from WIT:

- Remove its name from `lib_objects`.
- Invoke both `make depend` and `make clean`.

Renaming a source file from WIT:

- Change its name in `lib_objects`.
- Move the name alphabetically.
- Invoke both `make depend` and `make clean`.

For MCL:

The procedures are the name. The macro is in:

```
$WIT_HOME/mcl/src/sources.mk
```

For WIT-J:

At present, this can be skipped.

References

[Patterns] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995. ISBN 0201633612.

[Win SDK] R. Wittrock. 2012. “The Windows SDK Platform for WIT Software Projects”.
Available on GitHub at:

`rjwittrock/WIT/doc/wit-winsdk.pdf`

[WIT Guide] Anon. 2022. “WIT, User's Guide and Reference” Release 9.0.
Available on GitHub at:

`rjwittrock/WIT/doc/wit.pdf`

[WIT Intro] R. Wittrock. 2013. “An Introduction to WIT: Watson Implosion Technology” Second Edition.
Available on GitHub at:

`rjwittrock/WIT/doc/wit-intro-2.pdf`