

# Overview of Manifests

January 2018

Michael Benjamin, mikerb@mit.edu  
Department of Mechanical Engineering, CSAIL  
MIT, Cambridge MA 02139

---

<b>1</b>	<b>Manifests Overview</b>	<b>1</b>
1.1	An Example Manifest File . . . . .	2
1.2	Motivation . . . . .	2
<b>2</b>	<b>Components of a Manifest</b>	<b>3</b>
2.1	Minimal Components of a Manifest . . . . .	3
2.2	Optional Components of a Manifest . . . . .	4
<b>3</b>	<b>Two Manifest Related Programs in the MOOS-IvP Tree</b>	<b>4</b>
3.1	The manifest_test Utility . . . . .	4
3.2	The manifest_wiki Utility . . . . .	5
<b>4</b>	<b>Further Detail on File Syntax</b>	<b>5</b>
4.1	Multi-line Entries . . . . .	5
4.2	Comments and Blank Lines . . . . .	6
4.3	Module Types . . . . .	6
4.4	Author, Contact and Organization Fields . . . . .	7
4.5	The BornDate Field . . . . .	7
4.6	The DocURL and Distro Fields . . . . .	7
<b>5</b>	<b>Including Lines of Code Information in Manifests</b>	<b>8</b>
5.1	The Free Code Analysis Utility - sloccount . . . . .	8
5.2	Generating a LOC File to Augment Manifest Information . . . . .	9
<b>6</b>	<b>Creating Your Own Manifest Set</b>	<b>10</b>
6.1	Conventions for Organizing A Manifest Set . . . . .	10
6.2	A Group Manifest . . . . .	11
6.3	Checking Your Manifest Set . . . . .	12
6.4	How to Share your Manifest Set . . . . .	12
<b>7</b>	<b>Full Description of the manifest_test Utility</b>	<b>12</b>
7.1	Content Input . . . . .	12
7.2	The Full Set of Test Options . . . . .	12
7.3	The Verbose Mode . . . . .	13
7.4	The manifest_test Return Value . . . . .	13

---

## 1 Manifests Overview

A *manifest* is a short description of a software module, in a particular syntactic format. It also contains enough information to enable contact with the current maintainer of the code, and determine

the software dependencies associated with the code. The goal is to increase the likelihood of code sharing by allowing a quick determination if an existing suitable module may already exist.

## 1.1 An Example Manifest File

Here is an example manifest for the IvP Helm:

```
module    = pHelmIvP
type      = MOOS App
author    = Michael Benjamin
contact   = mikerb@mit.edu, issues@moos-ivp.org
org       = MIT
thumb     = A behavior-based helm with multi-objective optimization based
           (IvP) action selection.
depends    = lib_helmivp, lib_contacts, lib_behaviors-marine,
           lib_behaviors-colregs, lib_behaviors, lib_bhvutil, lib_ivpbuild,
           lib_ivpcore, lib_ivpsolve, lib_geometry, lib_apputil, lib_mbutil,
           lib_logic, lib_genutil, lib_MOOS

group     = Core Autonomy
borndate  = 020101
doc_url   = http://oceanai.mit.edu/ivpman/apps/pHelmIvP
license   = GPL
distro    = moos-ivp.org

synopsis  = pHelmIvP is a behavior-based autonomous decision-making
           MOOS application. It consists of a set of behaviors reasoning over a
           common decision space such as the vehicle heading and speed.
           Behaviors are reconciled using multi-objective optimization with the
           Interval Programming (IvP) model.
```

More info on syntax is given below in Sections 2 and 4. There is also a command line utility which can quickly confirm the validity of manifest file:

```
$ manifest_test myfile.mfs
```

The `manifest_test` utility is discussed in Section 7.

## 1.2 Motivation

Manifests accomplish two things:

- For *publicly available* code, manifests help users navigate code options, to hopefully quicken the process of finding what is needed, before re-writing it, perhaps unnecessarily.
- For *non publicly available* code, manifests help users learn about the existence of code, to facilitate a direct connection between potential user and code author, for direct sharing of beta code, or licensing of proprietary or classified code.

The latter may be the more important of the two. First, existing publicly available code is more likely to have user documentation. Interested users can simply read the documentation. The manifests in this case only make the search process quicker, which is still of significant value.

The real value lies in manifests of non publicly available code. To begin with, this codebase may be much larger, comprising many dozens of work-years of effort, on code that may have substantial field-trial experience. For many organizations, publicly releasing code is difficult. It may contain proprietary intellectual property, or it may be related to a classified project. Release of code publicly is also a hurdle many code authors are uncomfortable with until those proverbial last features are built, or until the documentation has finally been written. These hurdles in practice often are never surmounted, especially when the motivation is to release the code publicly to unknown users.

A *manifest* is powerful because it requires several orders of magnitude less work than actual documentation, and it does not require or even imply public release of code. But it does allow potential users to learn about other non public efforts and reach out to the manifest author(s) to make a direct connection. This is especially relevant to work at government labs. Government lab work is (a) very likely non public, (b) paid with tax dollars which should be motivation enough to re-use substantial investments in code development, and (c) very likely sharable directly between labs or projects.

Finally, a note to project managers (and this could be a project manager at a government or industry research lab, or an academic advisor). A huge impediment to progress on the five year scale or higher, is the inability to re-use code. This is largely due to the fact that code is typically written on the 1-3 year scale, for a particular grant or thesis project. The original author often is just not around after a few years. An important way to mitigate this problem is if other users (later students or later projects) take an interest in the code *as it emerges*. This hand-off of custodianship is more likely to happen if the original module's manifest emerges and is socialized earlier in the development process.

A forum for listing manifests is great beginning to ensure the gems of prior projects are further refined in later projects. Steadily building solid capabilities through well organized software may have a huge impact on a lab or research group's ability to advance marine autonomy in the 5-15 year time-scale. If this vision is properly conveyed to each code developer, it may motivate code development to take on a different mind-set from the outset - to build something meaningful that outlives the current deadline.

## 2 Components of a Manifest

A *manifest* is simply a text file. A *manifest set* is a collection of manifests, perhaps in many files, or all in a single file. A manifest has several fields, some of which are mandatory (if they are to be accepted in utilities that manage them and generate web content), and some components are optional.

### 2.1 Minimal Components of a Manifest

The following are the absolute bare-minimum fields for a module manifest.

- **module:** The name of the module.
- **contact:** Email or other means for contacting the author or current maintainer the code.
- **thumb:** A short one sentence description.

## 2.2 Optional Components of a Manifest

The following are additional fields which also really should be included. However, if the module name and one-line description above, also includes a valid point of contact, the below information can at least be asked by direct contact. Here are the additional fields:

- **type:** MOOS App, Command line utility, GUI utility, Library, Behavior.
- **author:** One or more authors (comma-separated, Firstname Lastname).
- **depends:** A list of library dependencies.
- **group:** One or more groups that relate to this module.
- **borndate:** Best guess date on when development began on this module.
- **doc\_url:** A URL where one may find documentation for this module.
- **license:** License under which this code is available for sharing.
- **distro:** Name of the distribution if this code is part of one.
- **synopsis:** A 1-2 paragraph description of the code module.

## 3 Two Manifest Related Programs in the MOOS-IvP Tree

Although a manifest is just a text file, the goal is to support a utility for auto-generating web pages, from a folder of individual manifest files, into cross-correlated web page content. The web pages help sort modules by groups, show common dependencies and also provide estimates of work year efforts and lines of code per module and groups of modules.

To this end, there are two command-line utilities:

- **manifest\_test:** A utility for validating that one or more manifest files are syntactically correct and have the minimum required information.
- **manifest\_wiki:** A utility for generating a set of wiki pages with information pulled from the whole set of manifest files.

### 3.1 The manifest\_test Utility

The first utility, **manifest\_test**, is a convenience utility for manifest authors. It will check for syntax and semantic correctness of manifest files. Ideally this utility is used by all manifest authors prior to sharing their content. This utility is included in the MOOS-IvP codebase in releases after 17.7, and in the development trunk as of this writing. So it should already be in your shell path if other MOOS-IvP applications are there. It can be run as below:

```
$ manifest_test full/path/to/my/manifest_files/*.mfs
```

It will run a handful of tests. The first three reveal errors that need to be fixed. The later tests reveal warnings that probably should be fixed. A successful test output would look like:

```

$ cd full/path/to/my/manifest_files/
$ manifest_test *.mfs *.loc --all
===== TEST BEGIN =====
Phase 1: All files readable with valid syntax:    PASS
Phase 2: All manifests have a unique module name: PASS
Phase 3: All manifests have all required fields:  PASS
Phase 4: All manifests have all optional fields:  PASS
Phase 5: All manifest dependency libs are known:  PASS
Phase 6: All manifests have lines-of-code stats:  PASS
===== TEST COMPLETE =====

```

More detail is given on this utility in Section 7. If any part fails, the test can be re-invoked in verbose mode to find out precisely what has been flagged.

### 3.2 The manifest\_wiki Utility

The second utility, `manifest_wiki`, is used for auto-generation of web pages. While this is also part of the MOOS-IvP distribution, it primarily does its thing on the web server hosting MOOS-IvP to generate web content as manifest files change. The typical user or author of manifest content will never need to run this. But the end goal is to generate valid manifest file content as input to `manifest_wiki` to produce rich manifest web content. All of the manifest web pages on the [moos-ivp.org](http://moos-ivp.org) website were produced by `manifest_wiki`.

## 4 Further Detail on File Syntax

You may have noticed, by the example manifest file Section 2, that manifest files are not written in XML format. They are written in a simpler, flat format. Each component of a manifest is of the form:

```
keyword = value
```

For example:

```
module   = pHelmIvP
type     = MOOS App
```

### 4.1 Multi-line Entries

Most items fit on a single short line. For items with longer lines, the long line may be broken up into several lines in the file, for better readability, by indenting the continuing line(s) with one or more white space. For example the following is treated as one long single line:

```
synopsis = pHelmIvP is a behavior-based autonomous decision-making
         MOOS application.
```

It is equivalent to:

```
synopsis = pHelmIvP is a behavior-based autonomous decision-making MOOS application.
```

A note of caution: the following line would produce an error:

```
synopsis = pHelmIvP is a behavior-based autonomous decision-making  
MOOS application.
```

In this case the continuing line is not indented, and the text "MOOS application" is not a valid keyword = value format.

## 4.2 Comments and Blank Lines

Comment lines have the characters "/" as the first two non-white space characters. Blank lines simply ignored.

For example, the following two snippets are equivalent:

```
// This is the manifest for the pHelmIvP MOOS application  
module    = pHelmIvP  
  
// The Core Autonomy group has all things helm related  
group     = Core Autonomy  
synopsis = pHelmIvP is a behavior-based autonomous decision-making  
          MOOS application. It consists of a set of behaviors reasoning over a  
          common decision space such as the vehicle heading and speed.  
  
// And we have more to say about this here:  
Behaviors are reconciled using multi-objective optimization with the  
Interval Programming (IvP) model.
```

The second snippet represents the same data:

```
module    = pHelmIvP  
group     = Core Autonomy  
synopsis = pHelmIvP is a behavior-based autonomous decision-making  
          MOOS application. It consists of a set of behaviors reasoning over a  
          common decision space such as the vehicle heading and speed.  
Behaviors are reconciled using multi-objective optimization with the  
Interval Programming (IvP) model.
```

## 4.3 Module Types

Each manifest has a type field. Currently the below five types cover all modules in the MOOS-IvP code base.

- MOOS App: For example pLogger, pHelmIvP and iSay.
- Behavior: For example BHV\_Waypoint, BHV\_Loiter and BHV\_AvdColregs.
- Command Line Utility: For example alogscan, alogrm and ippsolve.

- GUI Utility: For example `alogview`, `zaic_hdg` and `zaic_peak`.
- Library: For example `lib_contacts`, `lib_helmivp` and `lib_mbutil`.

Currently, listing a type other than the above five will not result in a warning when running `manifest_test`. When `manifest_wiki` is invoked to create web pages, it will use the group information accordingly, e.g, creating a page for all behaviors, and all libraries. In grouping, the type information is case insensitive.

#### 4.4 Author, Contact and Organization Fields

Each manifest has a `author` field. For a single author, the format is simply "first-name last-name". Multiple authors are separated by a comma.

```
author = Jane Doe
author = Jane Doe, Joaquin Phoenix
```

Presently no sorting or web page generation is done by author name. In the cases where the author is not the maintainer, use the `contact` field to indicate the current maintainer of the software. This field should be a comma-separated list of email addresses:

```
contact = jake@gmail.com, mary@cs.college.edu
```

The organization name is similarly specified by a simple string, comma-separated if multiple organizations are involved:

```
org = MIT, Oxford
```

Presently no sorting or web page generation is done by organization. This will likely change in the near future.

#### 4.5 The BornDate Field

Each manifest has a `borndate` field, indicating the earliest date of any file in the module. Admittedly this may only be a rough measure in many cases, especially if substantial modifications or additions occurred well after the first file was created. But nevertheless it can provide useful information to external users. The date is in the format of six numbers `YYMMDD`.

#### 4.6 The DocURL and Distro Fields

The `doc_url` field should be a valid URL link to either a web page or a PDF.

The `distro` may indicate the name of a software tree if the code is publicly available or is in any form for later private sharing. It could also be a git or scm URL.

## 5 Including Lines of Code Information in Manifests

Applications require varying degrees of effort to write, and having a feel for this aspect of a module may help gauge whether to adopt, modify or re-write a module. The most common metrics are

- Logical lines of code
- Files of code
- Workyears invested

Manifest files may be augmented with one or more instances of an additional type of file, a lines-of-code, or .loc file. This file contains the above three pieces of information for each software module. For example:

```
module=uFldMessageHandler, loc=385, foc=5, wkyrs=0.07
module=uFldNodeBroker, loc=517, foc=5, wkyrs=0.10
module=uFldNodeComms, loc=790, foc=5, wkyrs=0.16
module=uFldShoreBroker, loc=515, foc=5, wkyrs=0.10
module=uHelmScope, loc=1301, foc=9, wkyrs=0.26
module=uLoadWatch, loc=349, foc=5, wkyrs=0.07
```

There are four columns of data - the module name, the number of logical lines of code, the number of files, and the estimated number of work years of effort.

When the full manifest set is processed by `manifest.wiki`, it will include this information for each module in the resulting web content. For many applications, the heavy lifting in the implementation is done in the dependent libraries. In these cases, the three metrics should also include the dependent libraries to provide a more accurate picture. The `manifest.wiki` utility will ingest both the lines-of-code and dependency information to generate two values. For example, with the `pHelmIvP` MOOS app, over 95% of the implementation is in the libraries:

```
pHelmIvP:
  Lines of Code:  1,701 (with libraries: 66,947)
  Files of Code:   7   (with libraries: 573)
  Work Years:     0.35 (with libraries: 14.8)
```

### 5.1 The Free Code Analysis Utility - `sloccount`

Counting the logical lines of code is preferred to just counting the lines in a file, with the `wc -l` utility for example. The former does not count comments and blank lines. And disregards other style issues like whether a developer uses a whole line for an open or close brace in a for-loop. To count the logical lines of code, we use `sloccount`. This utility is available in common package managers in MacOS and Linux.

To run `sloccount`, simply provide the directory as a command line argument. By default, it will produce a report similar to the following:



```

$ cd moos-ivp/ivp/src/
$ sloccount lib_mbutil
Creating filelist for lib_mbutil
Categorizing files.
Finding a working MD5 command....
Found a working MD5 command.
Computing results.

SLOC      Directory      SLOC-by-Language (Sorted)
3315      lib_mbutil      cpp=3315

Totals grouped by language (dominant language first):
cpp:      3315 (100.00%)

Total Physical Source Lines of Code (SLOC)          = 3,315
Development Effort Estimate, Person-Years (Person-Months) = 0.70 (8.45)
  (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                  = 0.47 (5.62)
  (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule) = 1.50
Total Estimated Cost to Develop                     = $ 95,093
  (average salary = $56,286/year, overhead = 2.40).

SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'

```

Files of code may be insightful. For example, you might be leery of a module with 5,000 lines of code in one file. The level of effort in terms of work years is trickier to measure. We use the

## 5.2 Generating a LOC File to Augment Manifest Information

While `sloccount` provides all the information needed to make a `.loc` file as described in Section 5, it would require a lot of tedious work to make a `.loc` file by hand. Here we describe a shell script used for auto-generating such a file.

This script is available in the MOOS-IvP trunk in `moos-ivp/scripts/`, and in MOOS-IvP releases after 17.7.

```

$ cd moos-ivp/scripts
$ cat slocgen.sh

```

```
#!/bin/bash
# This is slocgen.sh (M. Benjamin, MIT, December 2017)

for D in *; do
  if [ -d "${D}" ]; then
    LOC='sloccount "${D}" 2> /dev/null | fgrep "cpp:" | awk -F' ' '{print $2}''
    FOC='sloccount --filecount "${D}" 2> /dev/null | fgrep "cpp:" | awk -F' ' '{print $2}''
    WYR='sloccount "${D}" 2> /dev/null | fgrep "Development" | awk -F' ' '{print $7}''
    printf "module=${D}, loc=${LOC}, foc=${FOC}, wkyrs=${WYR}\n"
  fi
done
exit 0
```

This script assumes that all the source code is in a single folder, e.g., `src/`, and that all source code sub-folders are to be included. If you have a different arrangement, you can simply replace the `for D in *; do`, which looks at all sub-folders of the current directory, with something like:

```
for D in {pFooBar iUtility pSomething}; do
```

Finally, remember that the `slocgen.sh` script does not actually create a file. But this is generally accomplished by redirecting the output to a file:

```
$ cd moos-ivp-mytree/src
$ slocgen.sh > mytree.loc
```

## 6 Creating Your Own Manifest Set

This section is a checklist guide for manifest authors. A minimal manifest set is just a few lines of text for each software module. But richer manifests are encouraged, especially those that contain concise synopses, and statistics from `sloccount`.

### 6.1 Conventions for Organizing A Manifest Set

A manifest set is a collection of manifest (`.mfs`) files, and lines-of-code (`.loc`) files. A reasonable way to organize this is to have a single directory with all files. The manifest software can also handle having all manifests in a single file. Likewise for the lines-of-code data. So minimally there will be one `.mfs` file. More typically a manifest set will be comprised of several `.mfs` files, one for each module, and one `.loc` file.

The manifest set for MOOS-IvP is part of the MOOS-IvP distribution. If you have that downloaded, you can just check out the directory `moos-ivp/manifests`. If you don't have the tree handy, or if you have an older release, you can pull down just the manifest set from `svn` with:

```
$ svn co https://oceanai.mit.edu/svn/moos-ivp-aro/trunk/manifests
```

## 6.2 A Group Manifest

A single group manifest is a very good idea. This allows you to generate a short description of your organization or research group. Likely all your manifests will include this group name in each manifest group list. Remember a manifest can belong to more than one group. So a manifest can belong to the group "ACME Autonomy Lab", *and* the group "Simulation" for example. Here is an example group manifest:

```
//=====
module   = Simulation
type     = group
author   = Michael Benjamin
contact  = mikerb@mit.edu
org      = MIT
synopsis  = The Simulation Toolbox contains a number of tools for supporting
           multi-vehicle missions where each vehicle is connected to a shoreside
           community. This includes both simulation and real field experiments.
           It also contains a number of simulated sensors that run on off-board
           the vehicle on the shoreside.

group    = Simulation
doc_url  = http://oceanai.mit.edu/ivpman/tools
distro   = moos-ivp.org
```

The above group manifest describes a set of modules with common *function*. Another type of group manifest would describe modules from a common *organization*:

```
//=====
module   = StateU Robot Lab
type     = group
author   = Prof. Lab Leader
contact  = lab_leader@stateu.edu
org      = StateU
synopsis  = The StateU Robot Lab specializes robotic software.
           This software toolbox contains several applications and utilities
           suitable for fielding marine robots.

group    = StateU Robot Lab
doc_url  = http://robots.stateu.edu/main_page
distro   = https://github.com/StateURobotLab/moos-ivp_apps
```

The organization manifest is particularly useful if your group is creating a manifest set to be included on the moos-ivp.org web site. Your modules will have a single link on the left-hand main menu, which will link to a single page with the above group manifest information. On this page will be a link to all modules belonging to this group.

Note: While group manifest files and software manifest files have the very same structure and syntax, out of convenience the group manifest files may have the `.gfs` suffix instead. Files with this syntax are ingested by `manifest.test` and `manifest.wiki` exactly the same.

### 6.3 Checking Your Manifest Set

Use the `manifest_test` command line tool (Section 7) to verify the correctness and completeness of your manifest set. The goal is to pass as many or all of the tests in this utility as possible.

### 6.4 How to Share your Manifest Set

The goal, in writing this document, is to include your manifest set on the web pages on the [moos-ivp.org](http://moos-ivp.org) web site. Perhaps the simplest way is to send us a tar file of your manifest set. Alternatively, if your manifest set is available via version control, e.g., svn or git, with anonymous read-only access, we can make provisions to check out a local copy onto the [moos-ivp.org](http://moos-ivp.org) web server. Contact us if you have questions. We're open to other ideas.

## 7 Full Description of the `manifest_test` Utility

More details on the `manifest_test` utility. This utility is designed for manifest authors to help catch broken, missing or inconsistent items in a manifest set, before exporting for sharing.

### 7.1 Content Input

The utility accepts any number of manifest or lines-of-code files as input to the command line. Typical usage would be to run this utility from within the directory where all your manifest and lines-of-code files live.

```
$ cd full/path/to/my/manifest_files/  
$ manifest_test *.mfs *.loc
```

Additional files may be specified by providing the full path name. For example, to include all the MOOS-IvP manifest files:

```
$ cd full/path/to/my/manifest_files/  
$ manifest_test *.mfs *.loc ~/moos-ivp/manifests/*.mfs ~/moos-ivp/manifest/*.loc
```

The latter may be desirable since Phase 5 of the manifest test is to check that all named library dependencies are known/valid. If any of your manifests link to MOOS-IvP libraries, this allows this test to proceed without warnings.

### 7.2 The Full Set of Test Options

There are six phases of tests supported. The first three are always conducted. Phase 1 will confirm that all manifest and lines-of-code files given on the command line exist, are readable and contain valid syntax. Phase 2 will confirm that no two manifests contain the same module name. The module name is regarded as the *key* in the set of manifests and must be unique. Phase 3 checks that all manifests contain the minimum fields described in Section 2.1. By default, only the first three phases are conducted when invoking `manifest_test`:

```
$ cd full/path/to/my/manifest_files/
$ manifest_test *.mfs *.loc
===== TEST BEGIN =====
Phase 1: All files readable with valid syntax:    PASS
Phase 2: All manifests have a unique module name: PASS
Phase 3: All manifests have all required fields:  PASS
```

The Phase 4 test is optional and is invoked with the `--warnings` or `-w` command line option. This test checks whether all manifests contain all the optional fields described in Section 2.2. Since they are optional fields, it is quite likely they do not.

The Phase 5 test is optional and is invoked with the `--depends` or `-d` command line option. This test checks whether all library dependencies listed in any of the manifests, is a known library. A mis-spelled library name may result in an inaccurate description of an app in terms of lines-of-code or work-years since the mis-spelled library would not be included. In some cases a module may have a "system" library dependency, e.g., `lib.fltk`. To avoid raising warnings for such libraries, the `manifest_test` utility allows you to specify a colon-separated list of libraries to ignore:

```
$ cd full/path/to/my/manifest_files/
$ manifest_test *.mfs *.loc --ignore_libs=lib_fltk:lib_fltk_gl
```

The last, Phase 6 test, is optional and is invoked with the `--stats` or `-s` command line option. This test checks whether all modules have sloccount statistics: lines-of-code, files-of-code, and work-years. This information is typically found in one or more `.loc` files.

All three optional tests can be invoked simply with the `--all` or `-a` command line option:

```
$ cd full/path/to/my/manifest_files/
$ manifest_test *.mfs *.loc --all
```

If any of the first three tests fail, the `manifest_test` utility will exit. Failures in the last three tests will not result in the early exit of the program.

### 7.3 The Verbose Mode

If one of the `manifest_test` parts fail, you may can find more information by re-running with the `--verbose` command line option. It will produce a much more detailed report of all phases.

### 7.4 The `manifest_test` Return Value

The `manifest_test` utility will have a return value of 0 if all invoked phases pass. It will have a value of 1 otherwise. This allows the `manifest_test` utility to work within a shell script, possibly requiring a successful test before updating web pages.