oep Documentation

Release 0.0.1

Brian Bingham

April 27, 2016

Contents:

OEP: 1 Title: OEP Purpose and Guidelines Author: Brian Bingham Status: Active Type: Process Content-Type: text/x-rst Created: 14-Apr-2016 Post-History: 14-Apr-2016

1 OEP Purpose and Guidelines

1.1 1.1 What is an OEP?

OEP stands for OER Enhancement Proposal. An OEP is a design document providing information to the OER community, or describing a new feature for OER or its process environment.

The OEP process is based on the ROS REP process which is based on the Python PEP process. We are thankful to the ROS and Python contributors for providing a process, tools and templates for community participation in a design process.

This initial document is based on a search-and-replace of REP 1 by Ken Conley. Over time, it will incorporate OER-specific changes to this process. The Author field of this document has been changed in order to denote reponsibility for maintenance, not credit for original authorship.

1.2 1.2 OEP Types

There are three kinds of OEP:

1. A **Standards Track** OEP describes a new feature or implementation.
2. An **Informational** OEP describes a design issue, or provides general guidelines or information to the OER community, but does not propose a new feature. Informational OEPs do not necessarily represent a OER community consensus or recommendation, so users and implementors are free to ignore Informational OEPs or follow their advice.
3. A **Process** OEP describes a process surrounding OER, or proposes a change to (or an event in) a process. Process OEPs are like Standards Track OEPs but apply to areas other than the language itself. They may propose an implementation, but not to the codebase; they often require community consensus; unlike Informational OEPs, they are more than recommendations, and users are typically not free to ignore them. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in development. Any meta-OEP is also considered a Process OEP.

1.3 1.3 OEP Work Flow

The OEP editors assign OEP numbers and change their status.

The OEP process begins with a new idea for OER. It is highly recommended that a single OEP contain a single key proposal or new idea. Small enhancements or patches often don't need a OEP and can be injected into the development work flow with a patch submission to the OEP issue tracker. The more focussed the OEP, the more successful it tends to be. The OEP editor reserves the right to reject OEP proposals if they appear too unfocussed or too broad. If in doubt, split your OEP into several well-focussed ones.

Each OEP must have a champion – someone who writes the OEP using the style and format described below, shepherds the discussions in the appropriate forums, and attempts to build community consensus around the idea. The OEP champion (a.k.a. Author) should first attempt to ascertain whether the idea is OEP-able. Posting to the oer-users list (coming soon) is the best way to go about this.

Once the champion has asked the OER community as to whether an idea has any chance of acceptance, a draft OEP should be presented to oer-users. This gives the author a chance to flesh out the draft OEP to make properly formatted, of high quality, and to address initial concerns about the proposal.

Following a discussion on oer-users, the draft OEP should be sent to the oer-users list (coming soon). The draft must be written in OEP style as described below, else it will be sent back without further regard until proper formatting rules are followed.

If the OEP editor approves, he will assign the OEP a number, label it as Standards Track, Informational, or Process, give it status "Draft", and create and check-in the initial draft of the OEP. The OEP editor will not unreasonably deny a OEP. Reasons for denying OEP status include duplication of effort, being technically unsound, not providing proper motivation or addressing backwards compatibility, or not in keeping with the OER philosophy. The MDFN (Malevolent Dictator for Now, TBD) can be consulted during the approval phase, and is the final arbiter of the draft's OEP-ability.

As updates are necessary, the OEP author can check in new versions if they have GIT commit permissions, or can email new OEP versions to the OEP editor for committing.

Once a OEP has been accepted, the reference implementation must be completed. When the reference implementation is complete and accepted by the MDFN, the status will be changed to "Final".

A OEP can also be assigned status "Deferred". The OEP author or editor can assign the OEP this status when no progress is being made on the OEP. Once a OEP is deferred, the OEP editor can re-assign it to draft status.

A OEP can also be "Rejected". Perhaps after all is said and done it was not a good idea. It is still important to have a record of this fact.

Some Informational and Process OEPs may also have a status of "Active" if they are never meant to be completed. E.g. OEP 1 (this OEP).

1.4 1.4 What belongs in a successful OEP?

Each OEP should have the following parts:

1. Preamble – RFC 822 style headers containing meta-data about the OEP, including the OEP number, a short descriptive title (limited to a maximum of 44 characters), the names, and optionally the contact info for each author, etc.
2. Abstract – a short (~200 word) description of the technical issue being addressed.
3. Copyright/public domain – Each OEP must either be explicitly labelled as placed in the public domain (see this OEP as an example) or licensed under the Open Publication License.
4. Specification – The technical specification should describe the syntax and semantics of any new feature.
5. Rationale – The rationale fleshes out the specification by describing what motivated the design and why particular design decisions were made. It should describe alternate designs that were considered and related work, e.g. how the feature is supported in other languages.

   The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

7. Backwards Compatibility – All OEPs that introduce backwards incompatibilities must include a section describing these incompatibilities and their severity. The OEP must explain how the author proposes to deal with these incompatibilities. OEP submissions without a sufficient backwards compatibility treatise may be rejected outright.

8. Reference Implementation – The reference implementation must be completed before any OEP is given status "Final", but it need not be completed before the OEP is accepted. It is better to finish the specification and rationale first and reach consensus on it before writing code.

   The final implementation must include test code and documentation.

1.5 1.5 OEP Formats and Templates

All OEP are expected to be formatted in reStructuredText with UTF-8-encoding. reStructuredText OEPs allow for rich markup that is still quite easy to read. REP 12 contains instructions and a template [2] for reStructuredText.

There is a Python script that converts OEPs to HTML for viewing on the web. reStructuredText OEPs are parsed and converted by Docutils code called from the script.

1.6 1.6 OEP Header Preamble

Each OEP must begin with an RFC 822 style header preamble. The headers must appear in the following order. Headers marked with "*" are optional and are described below. All other headers are required.

```
  OEP: <oep number>
  Title: <oep title>
  Version: <svn version string>
  Last-Modified: <svn date string>
  Author: <list of authors' real names and optionally, email addrs>
* Discussions-To: <email address>
  Status: <Draft | Active | Accepted | Deferred | Rejected |
           Withdrawn | Final | Replaced>
  Type: <Standards Track | Informational | Process>
* Content-Type: <text/plain | text/x-rst>
* Requires: <oep numbers>
  Created: <date created on, in dd-mmm-yyyy format>
* ROS-Version: <version number>
  Post-History: <dates of postings to ros-users>
* Replaces: <oep number>
* Replaced-By: <oep number>
* Resolution: <url>
```

The Author header lists the names, and optionally the email addresses of all the authors/owners of the OEP. The format of the Author header value must be

> Random J. User <address@dom.ain>

if the email address is included, and just

> Random J. User

if the address is not given. For historical reasons the format "address@dom.ain (Random J. User)" may appear in a OEP, however new OEPs must use the mandated format above, and it is acceptable to change to this format when OEPs are updated.

[2] REP 12, Sample reStructuredText REP Template (http://ros.org/reps/rep-0012.html)

If there are multiple authors, each should be on a separate line following RFC 2822 continuation line conventions. Note that personal email addresses in OEPs will be obscured as a defense against spam harvesters.

Note: The Resolution header is required for Standards Track OEPs only. It contains a URL that should point to an email message or other web resource where the pronouncement about the OEP is made.

While a OEP is in private discussions (usually during the initial Draft phase), a Discussions-To header will indicate the mailing list or URL where the OEP is being discussed. No Discussions-To header is necessary if the OEP is being discussed privately with the author, or on the ros-users email mailing lists. Note that email addresses in the Discussions-To header will not be obscured.

The Type header specifies the type of OEP: Standards Track, Informational, or Process.

The format of a OEP is specified with a Content-Type header. The only supported values is "text/x-rst", which designates reStructuredText encoding (see OEP 12 [2]).

The Created header records the date that the OEP was assigned a number, while Post-History is used to record the dates of when new versions of the OEP are posted to ros-users. Both headers should be in dd-mmm-yyyy format, e.g. 14-Apr-2016.

Standards Track OEPs must have a OER-Version header which indicates the version/distribution of OER that the feature will be released with. Informational and Process OEPs do not need a OER-Version header.

OEPs may have a Requires header, indicating the OEP numbers that this OEP depends on.

OEPs may also have a Replaced-By header indicating that a OEP has been rendered obsolete by a later document; the value is the number of the OEP that replaces the current document. The newer OEP must have a Oeplaces header containing the number of the OEP that it rendered obsolete.

1.7 1.7 Auxiliary Files

OEPs may include auxiliary files such as diagrams. Such files must be named `oep-XXXX-Y.ext`, where "XXXX" is the OEP number, "Y" is a serial number (starting at 1), and "ext" is replaced by the actual file extension (e.g. "png").

1.8 1.8 Reporting OEP Bugs, or Submitting OEP Updates

How you report a bug, or submit a OEP update depends on several factors, such as the maturity of the OEP, the preferences of the OEP author, and the nature of your comments. For the early draft stages of the OEP, it's probably best to send your comments and changes directly to the OEP author. For more mature, or finished OEPs you may want to submit corrections to the OEP issue tracker so that your changes don't get lost. If the OEP author is a ROS developer, assign the bug/patch to him, otherwise assign it to the OEP editor.

When in doubt about where to send your changes, please check first with the OEP author and/or OEP editor.

OEP authors who are also ROS committers can update the OEPs themselves by using "git commit/push" to commit their changes.

1.9 1.9 Transferring OEP Ownership

It occasionally becomes necessary to transfer ownership of OEPs to a new champion. In general, we'd like to retain the original author as a co-author of the transferred OEP, but that's really up to the original author. A good reason to transfer ownership is because the original author no longer has the time or interest in updating it or following through with the OEP process, or has fallen off the face of the 'net (i.e. is unreachable or not responding to email). A bad reason to transfer ownership is because you don't agree with the direction of the OEP. We try to build consensus around a OEP, but if that's not possible, you can always submit a competing OEP.

If you are interested in assuming ownership of a OEP, send a message asking to take over, addressed to both the original author and ros-users. If the original author doesn't respond to email in a timely manner, the OEP editor will make a unilateral decision (it's not like such decisions can't be reversed :).

1.10 1.10 OEP Editor Responsibilities & Workflow

All OEP-related correspondence should be sent (or CC'd) to <ros-users@lists.ros.org>.

For each new OEP that comes in an editor does the following:

- Read the OEP to check if it is ready: sound and complete. The ideas must make technical sense, even if they don't seem likely to be accepted.
- The title should accurately describe the content.
- Edit the OEP for language (spelling, grammar, sentence structure, etc.), markup (for reST OEPs), code style (examples should match OEP 8 & 7).

If the OEP isn't ready, the editor will send it back to the author for revision, with specific instructions.

Once the OEP is ready for the oepository, the OEP editor will:

- Assign a OEP number (almost always just the next available number, but sometimes it's a special/joke number, like 666 or 3141).
- List the OEP in OEP 0 (in two places: the categorized list, and the numeric list).
- Add the OEP to GIT.

  The command to check out a read-only copy of the repository is:

```
git clone https://github.com/ros-infrastructure/oep.git
```

- Send email back to the OEP author with next steps (post to ros-users).

Updates to existing OEPs also come in to ros-users@lists.ros.org. Many OEP authors are not GIT committers yet, so we do the commits for them.

Many OEPs are written and maintained by developers with write access to the ROS codebase. The OEP editors monitor the oep-commits list for OEP changes, and correct any structure, grammar, spelling, or markup mistakes we see.

The editors don't pass judgement on OEPs. We merely do the administrative & editorial part. Except for times like this, there's relatively low volume.

Resources:

- Getting Involved With ROS
- ROS Developer's Guide

1.11 1.11 References and Footnotes

1.12 1.12 Copyright

This document has been placed in the public domain.

OEP: 2 Title: Python Style Guide Author: Brian Bingham Status: Active Type: Process Content-Type: text/x-rst Created: 14-Apr-2016 Post-History: 14-Apr-2016

2 Python Style Guide

2.1 2.1 External Python Style Guides

For OER projects it is recommended that developers follow Google's Python Style Guide: https://google.github.io/styleguide/pyguide.html which appears to be consistent with PEP8 https://www.python.org/dev/peps/pep-0008/

2.2 2.2 Reusable Code

Andy wants to, "iterate toward the best method to support using the script as a program, importing it into another script, executing while developing and executing when installed. Some of this is design and some is packaging."

2.3 2.3 Common Modules

- Use **argparse** for command line arguments.
- Use **pytest** for unit tests.
- Use the **logging** module verbosity control. We want to "be able to redirect log to file, modify log level of only selected functions/classes without modifying the code, make sure our library code behaves properly and some others" To support this it would be nice to point to a working/ideal example.

OEP: 3 Title: Naming Conventions and Repository Structure Author: Brian Bingham Status: Active Type: Process Content-Type: text/x-rst Created: 14-Apr-2016 Post-History: 14-Apr-2016

CHAPTER 3

3 Naming Conventions and Repository Structure

3.1 3.1 Organization of Repositories on Bitbucket

Currently bitbucket is hosting all OER repositories. Within bitbucket the following three hierarchical organizations are supported:

1. **Teams**: All OER repositories should be owned by the *noaarov* team.
2. **Projects**: Within the OER team, repositories are organized into projects. Each repository is associated with one, and only one, project. The naming of projects is fluid, but the following standard project associates are
   (a) *oer* is the highest level project for general purpose software tools. For example, this repository of OEPs is in the oer project because it is applicable to all OER projects. If a software package is useful in multiple projects it should be a part of the oer project.
   (b) *d2* is a vehicle specific project for D2. Software packages that are specific to this vehicle should be a part of this project.
   (c) *minirov* is a vehicle specific project for the minirov. Software packages that are specific to this vehicle should be a part of this project.
3. **Users**: In general, OER projects should not be owned by individual users. In the past the OER team has used the rov337 bitbucket user as a common owner for all repositories. Moving forward it is recommended that the team move all repositores to the noaarov team.

3.1.1 3.1.1 Permissions

In the short term it seems to make sense have all OER developers be **Administrators** of the **noaarov** bitbucket team. In the future, as the team grows, this policy may prove to be overly permissive, but for now everyone should have full rights.

All repositories should be public.

3.2 3.2 Repository and Package Naming Conventions

Individual sofware components are typically organized as Python packages and the source code is stored in a repository. Components will also typically have executables associated with them that make use of the package and associated modules.

The naming convetions we use are adaptations of PEP8 , The Hitchhiker's Guide to Python and the Packaging Python Libraries chapter of Dive Into Python 3 which are the guidelines for PyPI.

3.2.1 3.2.1 Package Naming

Unless there is a good reason not to, package names should follow these constraints:

- All lowercase.
- Unique. If you think your name might not be unique, search for it in the PyPi and ROS indicies.
- When multiple words are needed, an underscore should separate them. Don't use hypens or spaces. (Note that PEP8 says, "use of underscores is discouraged". We are less prescriptive about this. You should choose between a singleword solution and multiple_word solution based on readibility.)

In general package/repository names should become increasingly specific as you read from left to right. This arrangement supports effective sorting by name.

3.2.2 3.2.2 Repository Naming

The repository should use the same name as the package, except that they repository substitutes dashes (-) for underscores (_).

3.2.3 3.2.3 Executable Naming

Executables should also use dashes as word separators.

3.2.4 3.2.4 Example

Here is an example of how we might organize a repository called `'altimeter-valport-lcm'` which contains the package `'altimeter_valeport_lcm'`. The package contains the module `'altimeter_valeport_lcm.parser'`

```
altimeter-valeport-lcm/
-- altimeter_valeport_lcm
|   -- parser.py
|   -- __init__.py
|   -- __main__.py
-- README.rst
-- setup.py
```

The setup.py file contains a line such as

```
entry_points = {
      'console_scripts':
        ['altimeter-valeport-lcm=altimeter_valeport_lcm.__main__:main'],
        },
```

which generates the executable `'altimeter-valeport-lcm'`.

3.3 3.3 Naming Conventions by Component Type

3.3.1 3.3.1 User Interfaces

I'll leave this to Andy

3.3.2 3.3.2 Libraries and Utilities

Code that is generally useful across multiple other packages/modules should typically be named with three underscore-separated parts: **TYPE_DESCRIPTION** where

- TYPE is one of {utils, lib, defs} where
  - **utils** denotes a collection utility programs. Utilities are most often standalone exectuable programs meant to be used by a developer as opposed to libraries/modules which are meant to be used within another program. For example, “utils_lcm_python” might contain executable python programs that
  - **lib** denotes libraries or modules that contain OER shared objects, functions and methods that are commonly used in other OER programs. For example “lib_timestamp_python” might contain python modules for dealing with OER-compliant timestamps.
  - **defs** denotes definition files that are common to multiple OER packages. This is where message and service definitions should reside. For example “defs_minirov_lcm” would be expectd to contain LCM message definitions that are specific to the MiniRov project. “defs_general_lcm” might be expected to contain the most basic, broadly applicable LCM definitiosn for OER software.
- DESCRIPTION is self explanatory

3.3.2.1 Examples

- utils_lcm
- defs_lcm_general

3.3.3 3.3.3 Drivers

In the context of OER software “drivers” are the layer of software between hardware devices (e.g., external sensors, onboard I/O, etc.) and the internal communications (e.g., LCM or ROS).

Driver names should typically consist of three parts, each separated by an underscore: **TYPE_NAME_COMMUNICATION** where

- TYPE is the general description of the class of device with which your driver will interface. The TYPE might be the general name of the sensor (e.g. compass, imu) or the thing being measured (e.g., ctd, depth, altitude, analogin). The some cases a single device measures multiple things, e.g., an OceanServer compass module that also reports depth or the Valport altimeter that also reports pressure (depth). In most cases there is a primary type that should be used, e.g., “cmpass_os_lcm_cpp” or “altimeter_valeport_ros_python”.
- NAME is the specific description of the particular device with which your driver is designed to communicate. The NAME might be the mdel of the device (e.g., tcmxb) or the vendor of (e.g., rdi, rowe). The NAME should be specific enough to map to the communications interface that your driver implements. For example, using “ctd_seabird_lcm_python” may not sufficiently specific because Sea-bird makes a a bunch of different CTDs with a variety of CTD sensors. Based on the name “ctd_seabird_lcm_python” we would expect that this package/driver is capable of communicating with a variety of devices. If your driver is specific to a particular model and interface type, say for the SBE 49 FastCAT RS232 interface, you should include that specificity in the NAME: e.g., “ctd_seabird49_lcm_python”
- COMMUNICATION is the middleware communication used by the driver. As of this writing this will be one of {lcm,ros}

3.3.3.1 Examples

- dvl_rdi_lcm
- dvl_rowe_ros
- ctd_seabird49_lcm
- altimeter_valeport_lcm

3.4 3.4 Organization of Package/Module within a Repository

See the Packaging Python Libraries chapter of Dive Into Python 3 which are the guidelines for PyPI.

3.5 3.5 Packaging and Release Distribution

OER is still working towards a standard solution for packaging and distributing Python projects. At the time of writing there are at least two possibles:

1. We could use devpi as a private packaging, testing and release server http://doc.devpi.net/ This would entail using setuptools for building and distributing packages https://pythonhosted.org/setuptools/setuptools.html If we use LCM this would likely be an attractive method.
2. We could use ROS packages to achieve the same (or similar) functionality.

For now this issue is tabled until Andy has time investigate further. Note that how we organized repositories depends on this choice!

3.6 3.6 Naming Conventions for Communications

The guidelines below apply to the communication middleware used in OER projects. It would be beneficial if the conventions applied equally to either LCM or ROS.

3.6.1 3.6.1 Message Type Naming Conventions

Coming soon

3.6.2 3.6.2 Channel/Topic Naming Conventions

Coming soon

OEP: 4 Title: Licensing and Copyright for OER Software Author: Brian Bingham Status: Active Type: Process Content-Type: text/x-rst Created: 14-Apr-2016 Post-History: 14-Apr-2016

CHAPTER 4

4 Licensing and Copyright for OER Software

4.1 4.1 Why Licensing is Important

Because I said so

4.2 4.2 Recommended License

TBD

4.2.1 4.2.1 Why it is Recommended

Just because.

4.3 4.3 Copyright

TBD

4.3.1 4.3.1 Copyright Boilderplate

TBD

CHAPTER 5

Indices and tables

- genindex
- modindex
- search