ICT Call 7
ROBOHOW.COG
FP7-ICT-288533

Deliverable D7.3:

Guidelines for software development

January 31st, 2013

Project acronym:	ROBOHOW.COG
Project full title:	Web-enabled and Experience-based Cognitive Robots that Learn Complex Everyday Manipulation Tasks
Work Package:	WP 7
Document number:	D7.3
Document title:	Guidelines for software development
Version:	1.0
Delivery date:	January 31st, 2013
Nature:	Report
Dissemination level:	Public
Authors:	Keith François (CNRS) Lorenz Mösenlechner (Uni-HB)

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement n^{O}288533 ROBOHOW.COG.

Contents

Chapter 1
Introduction

This document details some rules for future developed code. Keep in mind that the code will be read by humans and will be eventually modified by them (you may not be the last person to use it). It must be readable, understandable and reusable.

The most important rule about coding is the **consistency**. Hence, if a project already has its own format, do not impose yours, and try to adapt yourself. There are several reasons for that:

- Correcting everything will take you a long time that you may not have
- Temporarily mixing two coding styles is not a good thing to do.

Hence, if you are a visitor/occasional developer of a package, you may signal to the main developers that some rules have been broken. But don't do it by yourself, Prefer asserting the consistency inside one package rather than the consistency inside the whole project. However, when creating a new package, please follow those coding guidelines.

Chapter 2
General guidelines

- Ensure the consistency of each project (this is ***that*** important).
- Maintain the documentation up-to-date (use doxygen, sphinx...), and make your code understandable.
- Make the validation of the code easier by adding unit-tests.
- Use versioning tools (git or svn, depending on the type of data), tag versions that work and release often.
- Keep the code clean. Remove useless and dead code (especially if it is versioned).
- Do not neglect debugging tools (e.g. make the debug easier by using assertion (C++)).

2.1 General shape of package

All packages must contain the same information (author list, description, copyright...), but depending on the type of package (generic package or ROS package), those informations will be detailed in different locations.

2.1.1 Generic package

Packages independent from ROS should at least contain the following elements:

README(.md) The README(.md) file is the first file a newcomer will consult. The .md extension should be added if you use markdown to make the reading of the file easier.
It should define all the informations relative to the project:

- The purpose of the package.
- The use/compilation informations: on which OS does it work? What are the dependencies of the project (with the version if it matters).
- The related papers.
- The origin of the sources, and the project management web application (github, redmine ...)

It is important to detail on which OS the code has been tested, especially for C++. While using cmake and avoiding OS specific code or header inclusion are good practices, they do not automatically ensure the compatibility between the platforms. For example, on windows systems, there are some additional routines to export the symbols of the libraries. On Mac OS, the linkage verification is stricter than on ubuntu (where the linkage can be solved at the compilation). Also, the double precision handling may be different between 64 bits and 32 bits systems, which can also modify the behavior of a program.

The purpose is mainly to say to users that there are no maintainers for some OS, and that they will be on their own if they want to port the code.

Authors The list of contributing authors can be added at the top of each files. The order to keep is the alphabetical order of last name, then first name.

Since this can be a daunting task, an alternative is to define an AUTHOR file in the root directory of your package, in which you can indicate the corresponding(s) author(s) and list all the contributors and a summed up detail of their contribution. To simply list the contributors, you can use the following script:

```
#!/bin/sh
# Source: http://goo.gl/Y4UWR
set -e

# Get a list of authors ordered by number of commits
# and remove the commit count column
AUTHORS=$(git --no-pager shortlog -nse | cut -f 2-)
if [ -z "$AUTHORS" ] ; then
    echo "Authors list was empty"
    exit 1
fi

# Display the authors list and write it to the file
echo "$AUTHORS" | tee "$(git rev-parse --show-toplevel)/AUTHORS"
```

Copyright The full text statements of all licenses used in the project should be placed in the LICENSES directory at the top of the repository. If you add a package under a license that is not included in LICENSES, add the license statement.

The code developed in the frame of the robohow project should be open source. The licenses used in this purpose are the following ones (from the more restrictive to the least one).

- GPL[1], permits the use of the library only for free programs.
- LGPL[2] permits the use of the library in proprietary programs.
- BSD [3]

[1] http://www.gnu.org/licenses/gpl.html
[2] http://www.gnu.org/copyleft/lesser.html
[3] http://opensource.org/licenses/BSD-3-Clause

2.1.2 ROS package

The organization of a ROS package is slightly different from a generic package, since more informations are gathered in the ROS specific files manifest.xml and stack.xml. Hence ROS packages should at least contain the following elements:

manifest.xml or stack.xml The manifest.xml (for a package) / stack.xml (for a stack) files should detail the following informations:

- Author list
- Description
- License
- Url of the project.

As a result, please **DO NOT** create files such as AUTHOR, in order to avoid inconsistencies with the manifest.xml file.

README(.md) Since the description of the package is now in the manifest/stack files, the README file now contains less informations:

- The use/compilation informations: on which OS does it work? What are the dependencies of the project (with the version if it matters).
- The related papers.

2.2 File convention

2.2.1 Line endings

All text files must be normalized so that lines terminate in the unix style (LF). Mixture of termination styles will sooner or later create an issue in the versioning tool when the end-of-line character will change, and make the history unusable. Hence, avoid committing files that terminate in CRLF (windows ending) or CR (mac ending) since such a modification is likely to appear as a whole-file modification.

2.2.2 Naming convention

The names of the files should be in ascii, lower case, without space or numbers. The worst thing you could do is to have two files with the same name but with a different case: this leads to severe subversion issues on windows systems (one of the file erase the other automatically at the checkout of the repository).
Also, please avoid too long lines in the source code: respecting a maximum of 80 chars per line is good rule of thumb.

2.3 Clarify and document your code

Keep in mind that your code will be read, reused and (eventually) improved by humans. You want them to use your code wisely, and you don't have the time to help each of them personally.

Write the documentation and the code at the same time. Detail the how of your code, but don't forget the why. Knowing the goal of your code will help one to detect/correct wrong behavior.
Do not hesitate to detail any specific or unusual implementation (e.g. the use of the specific structure of a sparse matrix to gain computation time). Similarly, document your variables: units, specific bounds, legal values and give them *explicit* names (avoid `tmp` or random sequence of letters).
Please refer to the articles corresponding to the code, *especially* if the notations come from them.
Yet, do not overload your code by documenting the obvious. This can be avoided:

```
i+=1 //increase the index value of 1
```

2.4 Test a lot

Testing is the easiest way to check the behavior and good health of your code through its evolution. It allows to find what (and who) is responsible for a wrong behavior. Also, it can also be of help to understand the code.
The sooner the tests are added to the project, the easier it is to debug it. Similarly, the more automatized it is, the better.

- Do unit tests (verifies the behavior of the code)
- Do regression tests (verifies that the behavior of the code do not change with the evolution of the code)
- Do computation time tests to find bottleneck in your code (e.g. use callgrind in C++)
- Do memory handling tests (valgrind).

Automatize your tests using ctest or gtest[4].

2.5 Bug tracking - bug report

Eventually, you'll encounter some issues using code developed by someone else.
Do not hesitate to report an issue, and to signal the issue in the code with a TODO.
Please be explicit when you report an issue, and define the following informations: your goal (maybe there is another/safer way to do what you're trying to do), your OS (32 bits/64 bits), your version of ROS, the version of the packages you are using...

Prefer opening a ticket on the project management web application associated to the package, than contacting directly the person in charge of the code. You might not be the only one to encounter the problem and the resulting discussion might help someone else.

Help the developers to reproduce the bug, e.g. by creating a branch corresponding to your problem.

2.6 Feature request

Similarly to the bug report, prefer opening tickets on the corresponding website. You can help someone by doing this, especially when there are reasons not to implement the proposed feature.

[4] `http://www.ros.org/wiki/gtest`

2.7 Deprecation

As soon as there are users of your code, you have a responsibility not to pull the rug out from under them with sudden breaking changes. Instead, use a process of deprecation, which means marking a feature or component as being no longer supported, with a schedule for its removal. Give users time to adapt, which is usually one release cycle, then do the removal. Similarly, be thoughtful if you plan to change the ABI of your program.

Chapter 3
Versioning general settings

3.1 SVN or Git?

Use Git for text files, e.g. sources code (C, python, matlab, java...) or latex (.tex, .bib).
You may use SVN for huge binary-like files (.doc files, videos, pdf...). Indeed, using git to version huge files will drastically increase the size of the local git repository on the user's side. With svn, you can do only partial checkout, which can be practical.

Define the files to be ignored by using .gitignore or svn:ignore. Typically, **never version restricted access file** or automatically generated files (e.g. by a compiler). If you add by mistake such files, contact your administrator so that he can correct this quickly before anyone pulls it.

3.2 Commit message

It is important to keep the commits as small and simple as possible. Typically, create one commit per topic (bug fix, feature addition, documentation update...). Do NOT mix up bug fixing and feature addition, this will make the commit impossible to understand. This will increase the readability and make the cherry-picking and the debugging (e.g. with `git bisect`) easier. Hence, do not use `git commit -a`, prefer adding the files one by one or only chosen parts of files in your commit (using `git add -p`).

Convention Commit messages should be written in English in the present tense (so as to match up with commit messages generated by commands like `git merge` and `git revert`). They should have the following shape:

```
Brief description of the commit (< 60 characters)

Thorough description of the commit.
- item 1
- item 2
```

When correcting a bug, define the bug tracker reference (if any) and/or the bug origin or a small description of it. Please avoid putting simply "Remove bug".

When committing code on behalf of others use the `--author` option, e.g. `git commit --author "Emmett Brown <emmett.brown@bttf.org>"`, or thank the person in the message "Thanks to Emmett Brown for notifying."

Reverting commits Sometimes, a wrong commit can be pushed, by mistake or inattention. Before reverting a wrong commit, please contact its author so as to know if it is possible for him to correct his mistake or if he can revert himself. If the author cannot be reached and the revert is necessary and urgent (e.g. the main branch does not compile any more and external users need it), please indicate the reason of the revert. It can help the author of the commit to understand this decision, and avoid reverts of revert.

3.3 Tag

Tagging is a really handy way to ensure the coherence of a set of packages, e.g. by using pkg-config. The release tags have the shape vX[.Y[.Z[-descr]]] (eg. 1.0, 0.2.3-beta)

- X is the major version number. Change of major version number indicates that the code may not be backward compatible.
- Y is the minor version number. It indicates an improvement that should be backward compatible.
- Z is the patch version (bug fix or security fix)
- desc corresponds to the beginning of the git commit id.

It is also possible to define tags for specific purpose (experiments, paper...). In this case, they can have a different shape (eg: iros2012-FirstAuthorName).

Prefer tagging with a message, in order to get the correct tag with `git describe`.
Also, you can use the -s option to create a GPG-signed tag and certifies that *you* are the one that created the tag.

```
$ git tag -s v1.2.3 -m"Small description"
$ git describe
> v1.5.1-2-g59d2 # where 59d2 is the commit number.
$ git tag -v v1.2.3
object f148df68ed4f5c1fb7d35aae68342f6bb0ab7356
type commit
tag v1.2.3
tagger Emmett Brown <emmett.brown@bttf.org> 1357239557 +0100

Small Test
gpg:  Signature made Wed 21 Oct 2015 07:59:17 PM CET using RSA key ID 8D7EAAA1
gpg:  Good signature from "Emmett Brown <emmett.brown@bttf.org>"
```

For SVN, the tag command is:
`svn cp --parents <repo>/trunk <repo>/tags/v0.0.1`

ROS has a specific stack version policy [1]:

[1] `http://www.ros.org/wiki/StackVersionPolicy`

- 0.1: completely experimental, unreviewed
- 0.2: some review has occurred, and we are migrating this stack to more stable APIs
- 0.3: ready to start including with distributions, though definitely not stable
- 0.4-0.8: on stable development cycle towards 1.0 release (aka tick-tock)
- 0.9: 1.0 release candidate

3.4 Branch

The main branch, **master**, should **always** be usable (compilable and executable with unit tests successful). Another branch, **stable**, contains all the stable versions of the code.
Except from small modifications, all developments should be first realized in a separated branch (that may be published or remain local), tested, then merged with the master branch.

Convention The name of the branch should describe the purpose of the branch. Sorting the branches by sub-folders (i.e. using '/') increases the readability for a high number of branches.
Branches in the root folder (such as "master", "stable", "iros02"...) should be usable whereas work in progress should be in a sub-folder (e.g. topic/debug-dynamic, *developer-name*/dev for a personal branch).

Branch or tag? Use a tag if you are sure that the corresponding commit is usable and stable. Otherwise, use a branch, so as to allow adding patches afterwards, and make the update of local repository easier.

Merging branches: git rebase and git merge

Consider the project with two branches *initial* and *feature* on your local machine. Two commands, `git rebase` and `git merge`, can be used to gather the modifications realized in the two branches. They differ in the way the blending is realized, as depicted in Fig. 3.1:

- The merge process consists in keeping the history of both branches and creating an additional commit entitled "Merge branch ..." that gathers the modifications of both branches.
- The rebase process consists in reapplying all the commits of the current branch after the last commit of the target branch. For example:
  `emmet@pc:~/work (feature): git rebase initial`
  will stack all the commits of the feature branch after the commits of the master initial, and will rewrite the feature branch.

If you are afraid of losing the initial branch after a rebase (e.g. if one step of the rebase went wrong), do the rebase from a temporary branch
`emmet@pc:~/work (feature): git checkout -b rebase-feature`
`emmet@pc:~/work (rebase-feature): git rebase origin initial`
You can also retrieve the initial branch using `git reflog`.
The advantage of the rebase process is to keep the history readable and more linear. As a consequence, browsing the history and debugging (using `git bisect`) is simpler. In exchange, the initial history is rewritten (order of commit modified, commits squashed...).

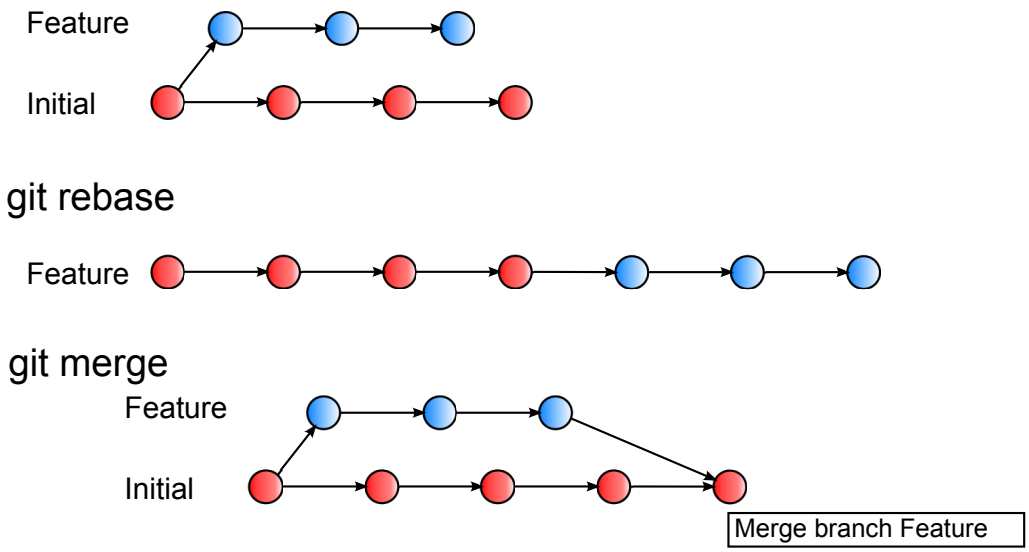


Figure 3.1: Difference between the commands *git merge* and *git rebase*. The first graph represents the original state of the git repository. The second one shows the result of the command *git rebase initial*, realized on the branch *feature*: the shape of the branch *feature* changes, so that every commit on this branch now appear after the commits of the init branch. The third graph shows the result of the command *git merge feature*, realized on the branch *initial*: an extra commit, gathering all the modifications realized on the branch feature, is added to the branch initial.

The advantage of the merge is that it keeps the original order of the commits. The main disadvantage is that the history graph will quickly grow into a bag knot, making debugging a nightmare, especially when several users develop on the same branch. Also, in case of conflict, the resulting commit is likely to be really obscure if some extra modifications have to be added.

In order to have a more linear history, it is better to use `git rebase`, especially for small differences (1 to 3 commits).

3.5 Communication with remote repository

Working with a remote repository introduces some subtleties and some extra good practices to follow. Since this repository is likely to be shared between several users / several places, it is important to ensure the good state and the coherency of the remote repository. An error realized on this repository may be hard to correct and is likely to propagate quickly. It is thus important to know what to do and what not to do.

Especially, the good use of the command *git rebase* requires a little practice, so if you are not familiar with this command, please ask the maintainer of the code to realize the merge/rebase operation for you.

3.5.1 Recommended practices

Forbidden practices Briefly, it is forbidden to modify a commit that has already been pushed. Such operations are likely to damage the remote repository and/or the repository of other users. In particular, the following practices should be prohibited:

1. Forcing a push. **Never** use the command `git push -f`
2. Changing a commit after pushing it. Once pushed, it is too late to do a `git commit --amend` (this commands allows to correct a commit by changing the commit message or the modifications included).
3. Rebasing a branch already on the remote repository, and pushing again on this branch, ie:
   ```
   git branch feature origin/feature # create the branch feature.
   git checkout feature # change to the branch feature
   git rebase master # change the order of commits
   git push origin feature # Push the modifications to the remote branch (not straightforward in most cases).
   ```

   The rebase process should be done only at the very end of the life cycle of a branch. Rebasing published branches can lead to duplicate commits in the shared repository. An example is given in section 3.5.5.

The only exception to the rule 1 is when a file that should not been distributed has been pushed (file containing a password, sensitive informations . . .). In this case, you can destroy the wrong commit by using `push -f` and/or contacting the main person in charge of the code to explain what happened.

Good practices Before pushing on the remote repository, it is important to make sure that all the required files are included, and that everything works well: clone your repository in a local folder (e.g. /tmp/), then compile and test it. This simple manipulation reduces small mistakes such as forgotten files.

3.5.2 Merging of branches

The life and death of a branch follows classicaly this scheme, illustrated by Fig. 3.2

1. Create the branch *topic/feature* based on the branch *start*
   ```
   git pull origin start
   git checkout -b topic/feature
   ```
2. Add some commits.
3. Occasionally, merge with the starting branch
   ```
   git pull origin/start
   ```
4. At the very end (before the merge), rebase against the starting branch
   ```
   git rebase origin/start
   ```
5. Do not forget to test the new branch.

6. Go back to the starting branch and **merge** with the branch to be deleted.
   ```
   git checkout start
   git merge topic/feature
   ```

7. Delete the branch
   ```
   # delete remote branch
   git remote prune feature
   git push origin :refs/heads/feature
   # delete local branch
   git branch -D feature
   ```

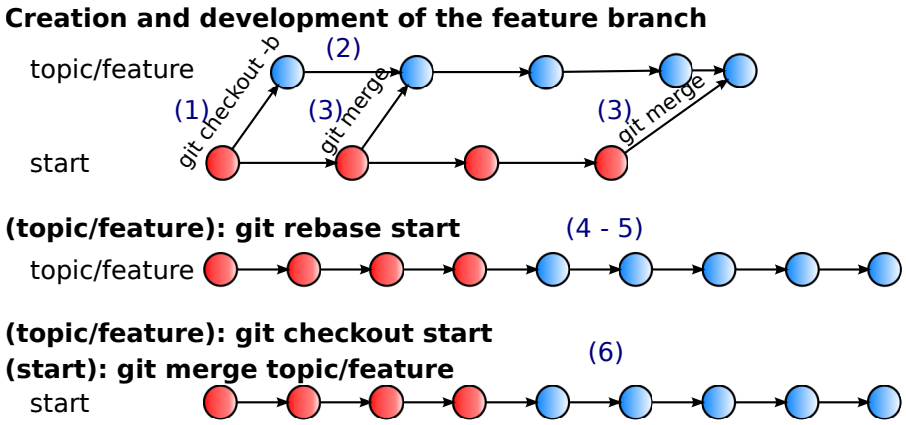


Figure 3.2: Life and death of a feature branch.

3.5.3 Rebasing during branch update

The rebase appends when one wants to gather two branches, but also during the update of a branch. When both the local and the remote repositories have been modified, the user has to update his locate repository before being authorized to push his modifications. Using the command `git pull` will implicitly realize a merge between the remote and the local branches if the pull is not feed-forward.
To avoid creating knots in the history graph, prefer using `git pull --rebase` instead of `git pull`. Note that to avoid (bad) surprises, you can do it in several steps:

1. `git pull origin current_branch` # do the merge: you can then check that everything went well.
2. `git rebase origin/current_branch` # stack your modifications after those in the remote repository.

Also, the following commands can be of use: `git fetch` checks if there are any modification of the remote repository.
`git pull --ff-only` pulls the remote modifications only if the pull is feed-forward (else fails).

3.5.4 Cleaning up

Do not forget to clean the dead branches of the remote repository from time to time.

```
# Delete tracking branch
git remote prune name_of_remote_branch
# Pour supprimer une branche :
git push origin :refs/heads/name_of_remote_branch
```

3.5.5 Warning on git rebase

As explained earlier, the git rebase is a powerful tool enabling to have a clean history, which make the debugging easier. However, when you rebase a branch, it is likely that its shape will differ of the shape of the corresponding branch on an remote repository. As a consequence, it is forbidden to continue working with this remote branch, because of readability reasons.

The figure 3.3 illustrates the evolution of the local branch *feature*, that is rebased on the branch *start* and merged again with the remote branch *origin/feature*. Before the rebase operation, the remote repository and the local repository are identical. We realize a *git rebase start* on the branch feature: the commits of the branch feature are now placed after the commits of the branch *start* (Figure 3.3B). The content of the commits "K" and "L" are the same[2], but their id are now different (illustrated by the change of color).

Starting now, the remote branch and the local branch have a totally different structure. Hence, it is not possible to push the branch feature on the remote branch (the push is not straightforward). In order to push on the remote branch, it would be necessary to update your local repository, that will create the tree illustrated in Figure 3.3C. As a result, the repository will contain the commits "K" and "L" twice, making the history unreadable.
Since it is no longer possible to communicate with the remote branch origin/feature, the best solution would now be to create a new remote branch with a different name (origin/feature2) or to merge the modification with the start branch (if it is the end of the branch cycle).

3.6 Structure of a GIT repository

The repository should follow the model proposed in [3]. Here is a short summary:

Main permanent branches The repository should have two permanent branches:

- The *master* branch, that contains commits as small as possible. This branch gives the detailed history of all the modification that have been realized on the repository.
- The *stable* branch, that contains released version only. They **must** work. The corresponding commits may be huge, since they correspond to squashes of commits that have been pushed in the develop branch.

[2] For simplification reasons, let's assume that no conflict occurred during the rebase
[3] http://nvie.com/posts/a-successful-git-branching-model/

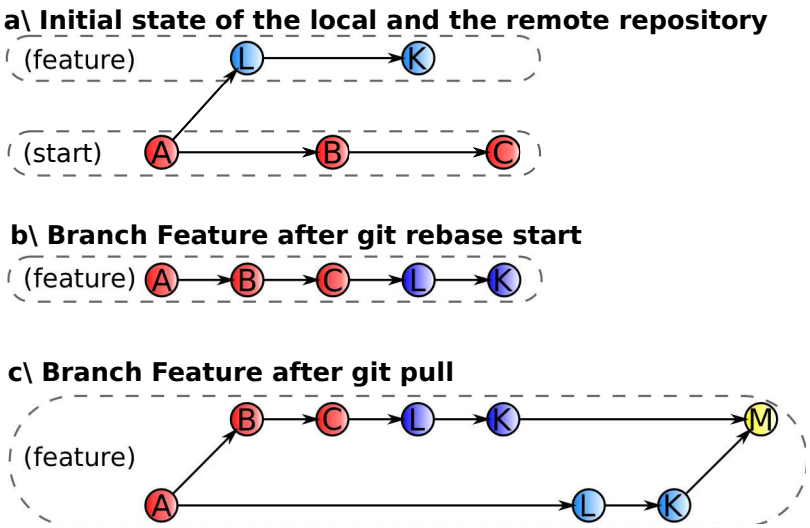


Figure 3.3: Wrong behavior: pulling a branch after rebasing it leads to the duplication of the history.

Extra permanent branches Other permanent branches can be created, corresponding to important demonstration or papers. Those repositories should be stored in the root of the git repository.

Temporary branches Next to those branches, other branches can be created for several reasons:

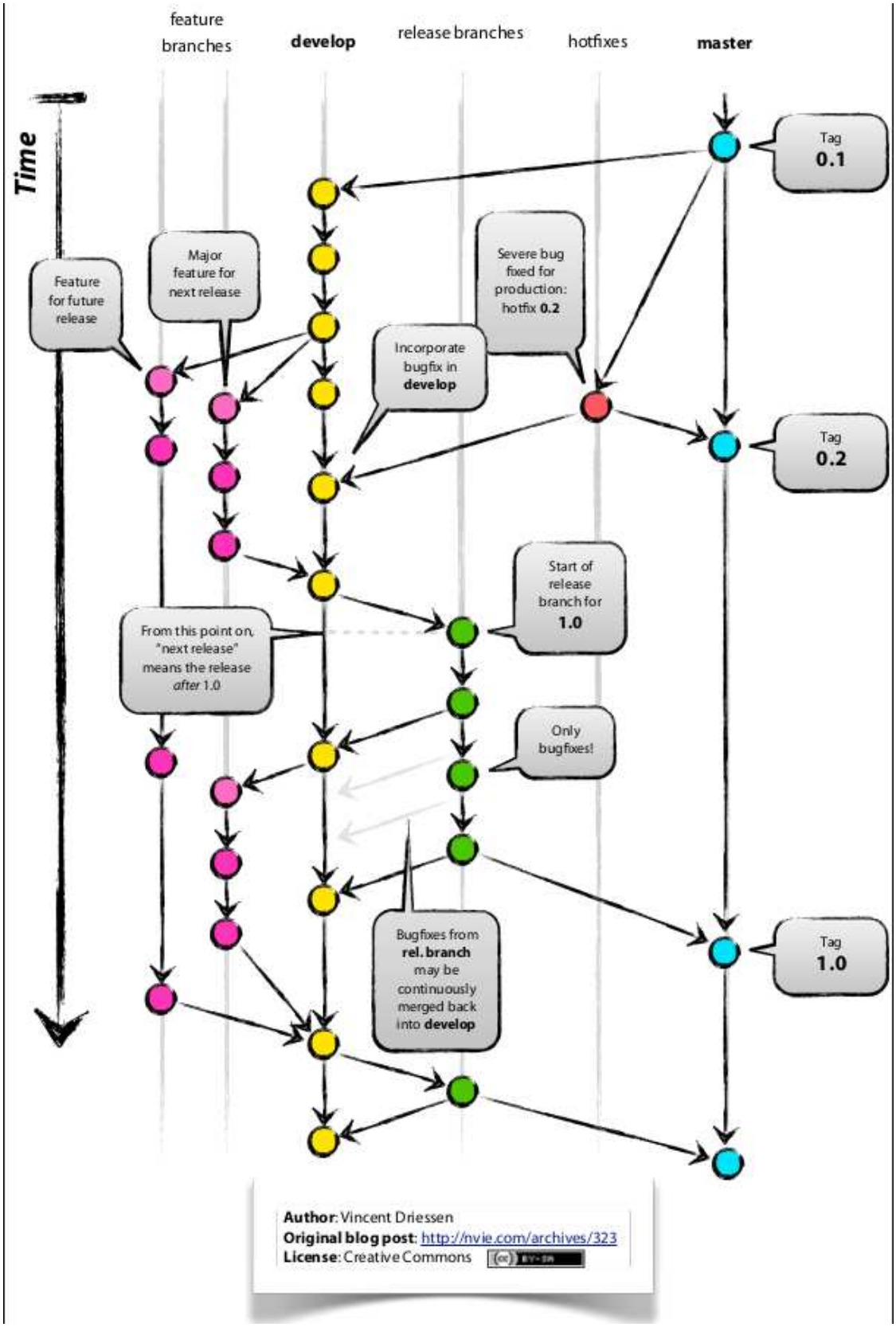
- Develop new features. At the end of the test process, the commits *must be rebased* with the release branches. Since the rebase can be tricky, please ask the code maintainer to do it if you are not familiar with this command.
- Develop a hotfix. Once done, that are merged with both the master and the develop branch.

Bug fixes Bug fixes can be cherry-picked into the stable branch.

3.7 Working in community

Modifying an external package If you have write access to the remote repository, you can commit your modification, push them in a new remote branch (`git push origin patch/improvement`), ask for verification from the main developers that will do/authorize the merge with the master branch.

If you do not have write access to the remote repository, commit the modification on your local repository and create the corresponding patches using the command
`git format-patch reference_commit -o out_folder`
e.g. `git format-patch origin/master -o patches` #
This command will convert all the commits from the reference_commit (here origin/master) to the last one into applicable commits, and store them in the folder *patches*


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/archives/323
License: Creative Commons

Thus, the main developers will be able to directly apply your patches using the command `git apply patch`

Do NOT do `git diff > patch.txt` This will create a file that is not directly usable, and the developers will have to recode your modifications.

Using others' packages Prefer working with stable branches or tagged versions, that are more likely to be stable, and are better reference points.

3.8 Sources

General guidelines
`https://wiki.duraspace.org/display/FCREPO/Git+Guidelines+and+Best+Practices`
`http://goo.gl/H5OnA`

Commit message shape
`http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html`

About the merge and the rebase
`http://darwinweb.net/articles/the-case-for-git-rebase`
`http://jeffkreeftmeijer.com/2010/the-magical-and-not-harmful-rebase`
`http://nvie.com/posts/a-successful-git-branching-model`

Chapter 4
ROS guidelines

The following rules are a copy of the slides created by Lorenz Mösenlechner [1].
The ROS version to be used is the **ROS Groovy** version (released in January 2013).
The ROS libraries are by default installed in the /opt/ros folder. **Never** edit those files. Prefer using ROS overlays [2] instead, and use the tool support *rosws* to create, modify and manage overlays. Multiple overlays with different versions can exist in parallel (e.g. one overlay for development, one for demos, one for experimenting with bleeding edge code of other people...).

4.1 ROS Overlays

Creation The creation of an overlay with rosws is realized as follows:

```
sudo apt-get install python-rosinstall
rosws init ~/fuerte /opt/ros/fuerte # Create a new overlay
# Load the created file setup.bash in .bashrc (optional)
# echo "source ~/fuerte/setup.bash" >> ~/.bashrc
```

Add packages To add a local directory (e.g. a sandbox for experimental packages), do:

```
mkdir ~/fuerte/sandbox
rosws set ~/fuerte/sandbox
```

Installation Install packages from a rosinstall file:

```
rosws merge robohow-cram.rosinstall
rosws update
```

Install a (released) stack from source:

```
roslocate info turtlebot | rosws merge -
rosws update
```

The rosinstall files correspond to a YAML description of repositories to install. They are ideal for repository snap shots and collaboration, and enable to specify versions, e.g:

```
- git:
   local-name:  cram_pl
   uri:  http://code.in.tum.de/git/cram-pl.git
   version:  0.1.5
- svn:
   local-name:  knowrob
   uri:  http://code.in.tum.de/pubsvn/knowrob/tags/latest
```

[1] https://robohow.org/_media/meetings/first-integration-workshop/ros-best-practices.pdf
[2] http://www.ros.org/wiki/fuerte/Installation/Overlays

4.2 Naming Conventions

File names

- Package names are lower case.
- Packages and stacks must not contain dashes ("-"), only underscores ("_").
- Messages, services and actions are named in camel case: `geometry_msgs/PoseStamped`
- Do not use the word "action" in an action definition (e.g.: Foo.action, not FooAction.action).

Topics, parameters, actions, services

- Nodes, topics, services, actions, parameters are all lower case with underscores as separator.
- Never use global names, always node local topic, service, action and parameter names.
- Use ros::NodeHandle handle("~")

Bad	**Good**
ros::NodeHandle nh;	ros::NodeHandle nh("~");
nh.advertise<Foo>("foo", 10);	nh.advertise<Foo>("foo", 10);
Topics: → /foo	Topics: → /node_name/foo

4.3 Best Practices

Topics vs. Services vs. Actions

- Use *topics* to publish continuous streams of data, e.g. sensor data, continuous detection results...
- Use *services* only for short calculations.
- Use *actions* for all longer running processes, e.g. grasping, navigation, perception, ...

ROS package

- ROS packages are cheap, create many.
- One package per functionality.
- Create separate packages that contain only messages, services and actions (separation of interface and implementation).
- Keep your dependencies clean:
  - only depend on what you need
  - specify all dependencies
  - do not use implicit dependencies
- Provide launch files.
- Group packages in stacks.

Misc

- Use standard data types when possible.
- Do not define matrix data types for transforms but use geometry_msgs/PoseStamped.
- Do not require a specific startup order for nodes: use `waitForService`, `waitForTransform`, `waitForServer`, ...
- Use ros::Time, ros::Duration and ros::Rate instead of system time.
- Do not use command line parameters but the ROS parameter server.
- Use rosconsole utilities for logging (ROS_INFO, ROS_DEBUG, ...).
- Never call cmake by hand in a package!

4.4 Third Party Libraries

- If possible, try to use libraries from Debian packages.
- Specify rosdep dependencies (tool for installing system packages).
- If you need to compile a library from source create a ROS wrapper package that downloads and compiles the package.
- Do not use sudo in wrapper packages.
- Do not require manual system wide installations.
- Do not copy libraries into packages that need them.

4.5 Collaboration

- Provide a rosinstall file.
- Create a short Wiki page for each package
- Document what the node does.
- Document topics, services and actions that are required and provided.
- Document ROS parameters and their default values.
- Data can be recorded and exchanged using bag files.

4.6 Relase of the software

To realize a release of your software, it is mandatory to

- Test your software using willowgarage buildfarm (see `http://www.ros.org/wiki/regression_tests` for more details).
- Use the ros release system `http://www.ros.org/wiki/release`. It will create the binary deb and the source deb (repackaged tarball of the source) for you.

4.7 ROS Bag Files

Recording of a bag: `rosbag record <topic> <topic> ...`
Play a bag: `rosbag play foo.bag`
Play a bag using recorded time (important when stamped data and TF was recorded):
`rosbag play --clock foo.bag`

Chapter 5
Code guidelines

The integration in ROS is one possible objective of the code. As a result, the first guidelines proposed are those of ROS. However, please be aware that this integration is not mandatory: one can want its code to be used outside of the ROS framework and independently from any middleware.
Some general guidelines have already been given in the section 2. The following rules refine them for some coding languages. Some pointers will be given for extra reading.

5.1 CPP guidelines

The cpp guidelines are those given by ROS. Hereafter are gathered some of the most important guidelines:

Convention The files are named in lower case, using underscores:

- C++ source files have a .cpp extension
- header files a .h extension
- template declaration a .hxx extension
- template definition a .t.cpp extension

Example:
`my_package/include/my_package/foo_bar.h`
`my_package/src/foo_bar.cpp`

- Files defining a class have the name of the class.
- C++ classes/types are normally named in camel case:
  `class FooBar ... ;`
- Functions are in `camelCase`, and their arguments are under_scored
  `int exampleMethod(int example_arg);`
- Variables are `under_scored`.
- Constants are all capital: `GRAVITY_FORCE`.

Indentation The indentation should follow the bloc structure of the program.
Put only one instruction by line.

Variables Make your variable name explicit. Avoid one-letter variables, except for integral iterator variables (i, j, k). Avoid using tmp, aaaa or meaningless random sequence of letters. An exception is tolerated if the notation come from a paper (matrix name etc).

Namespace Use namespaces to scope your code and gather classes/methods that belong to a same package or topic. Avoid defining global variables / functions. Define them in an appropriate namespace or in an anonymous namespace in the cpp file to limit their effect.

Headers The headers must be protected by multiple inclusion by using #ifndef guards

```
#ifndef PACKAGE_PATH_FILE_H
#define PACKAGE_PATH_FILE_H

#endif  //PACKAGE_PATH_FILE_H
```

Keep the headers light: use forward declaration.
Never add "using namespaces" in headers

Class Attributes of class should have a final underscore to distinguish them from other variables. Forbid the copy of classes when necessary by making the copy constructor private or making the class inherit from boost::noncopyable
Do not abuse of the friendship.

Function Output arguments to methods / functions (i.e., variables that the function can modify) are passed by pointer, not by reference.
Heavy object should be passed by const reference, except for small data (bool, char, int, double)
`int exampleMethod(const FooThing & input, BarThing* output);`
Limit the number of arguments of a function to 5.

Be a const addict Add const whenever it is possible: in the parameters of a function (input/output) and for methods that must not modify the class attributes.
Return elements by const reference.
Do not return by const copy (it does not make sense).

```
const std::vector<double> & getValue() const; // Good
const std::vector<double>   getValue() ; // Bad
```

Preprocessor macros should be avoided. Prefer inline functions, enums, and const variables to macros.

Debugging and testing Use assertions: assert or ROS_ASSERT (for code on ROS).
Use Ros messages: ROS_FATAL, ROS_ERROR, ROS_WARN, ROS_INFO, ROS_DEBUG.
Use callgrind, valgrind.

Exceptions Exceptions are the preferred error-reporting mechanism, as opposed to returning integer error codes.
Always document what exceptions can be thrown by your package, on each function / method.
Do not throw exceptions from destructors.
Do not throw exceptions from callbacks that you do not invoke directly.

When your code can be interrupted by exceptions, you must ensure that resources you hold will be deallocated when stack variables go out of scope. In particular, mutexes must be released, and heap-allocated memory must be freed. Accomplish this safety by using smart pointers.

Forbidden practices Do NOT ignore warnings. If some warnings are way too verbose (e.g. boost on windows), silent them one by one.

Sources
Ros: `http://www.ros.org/wiki/CppStyleGuide`
google: `google-styleguide.googlecode.com/svn/trunk/cppguide.xml`
Forbidden Practices: `http://www.strauss.za.com/sla/code_std.asp`

5.2 Python

Follow the python guidelines given by ROS[1].

Indentation Use 4 spaces per indentation level.

Parentheses should be use sparingly: do not use them in return statements or conditional statements unless using parentheses for implied line continuation.

Imports should usually be on separate lines, e.g.:

```
 Yes:  import os
       import sys
 No:   import sys, os
```

It's okay to say this though:

```
from subprocess import Popen, PIPE
```

Note on compatibility Use the REAL types returned by the function you use.

1. Example 1: Python/C API[2]
   `Py_ssize_t PyDict_Size(PyObject *p)`
   If you use this function you **MUST** use a Py_ssize_t and not an integer type chosen by you. Failing to do so will create conversion errors on different platforms.

2. Example 2: Matrix Libraries and STL containers [3]

   A vector size method has the following API: `size_type size() const;`

   Which means that the size *MUST* be stored into a `std::vector<T>::size_type variable.`

Package/Module Names All python code must be placed within a module namespace. ROS exports your Python source directory to be on the path of any of your dependencies, so it is important not to accidentally clobber someone else's import. We strongly recommend that this module name be the same as your ROS package name. There are two recommended code layouts:

1. Small modules with no msg/srvs

```
 packagename
 |-  src/
       |- packagename.py
 |-  scripts/
       |- non-exported python files
```

[1] `http://www.ros.org/wiki/PyStyleGuide`
[2] `http://docs.python.org/c-api/dict.html`
[3] `http://www.cplusplus.com/reference/stl/vector/`

2. Module with msgs/srvs

```
packagename
|-  src/
    |-  packagename/
          |- __init__.py
          |- yourfiles.py
|-  scripts/
    |-  non-exported python files
```

The more complicated layout for msg/srv files is required as the Python msg/srv generators will need to generate files into your package's namespace.
In the rare case that you cannot place your source code in src/ (e.g. thirdparty code), you can override the Python export path of your package by editing your manifest.
See the next section for description of node files

Node Files In ROS, the name of a node type is the same as its executable name. Typically, for python files, this means including `#!/usr/bin/env python` at the top of your main script, and having the main script's name identical to node's name.
If your node is simple, this script may contain the entire code for it. Otherwise, the node file will likely do an import packagename and invoke code there.
NOTE: we strive to keep ROS-specific code separate from reusable, generic code. The separation of 'node files' and files you place in src/packagename helps encourage this.

Sources
ros: `http://www.ros.org/wiki/PyStyleGuide`
google: `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`

5.3 XML: robot model

Indentation Use 2 spaces for indentation.

Robot model For now, the models of the robots are provided under the xml ros format. Ros Enhancement Proposals are active for mobile robots[4] and humanoid robots[5] (that provides extra information on vision, feet). These styles should be adopted to write the models of robots and humanoid robots (especially for Romeo).

5.4 CMake

- Use cmake to generate projects.
- When dealing with a great number of sources files (> 8), list them in a SourcesLib.cmake. Otherwise you can list them in a CMakeFiles.txt
- Use List(APPEND a ${b}) rather than set(a ${a} ${b})

[4] `http://www.ros.org/reps/rep-0105.html`
[5] `http://www.ros.org/reps/rep-0120.html`