PhaROS - Towards Live Environments in Robotics

Noury Bouraqadi Santiago Bragagniolo
Luc Fabresse Jannik Laval

Version of May 20, 2014

This book is available as a free download from:

http://car.mines-douai.fr

Contents

Chapter 1

Abstract

ROS Is a open software integration framework for robots that is growing up every day and that it shows several success in its topic which includes commercial applications of robots. This time-slice will be oriented to overview how to program our own nodes for ROS In Pharo . Also we will show how much can we exploit our highly dynamic environment in this organic architecture. PhaROS started early 2012 with at L'Ecole des mines de Douai for a robot for human assistance project (Robulab Project). We started this project trying to put emphasis in the dynamism of Pharo language from the beginning. We are now in a process of stabilising and closing the all the basic functionalities, having support for the last three versions of ROS. This talk will provide knowledge about ROS environment, and how to quickly prototype the functionality you want to add on it without much knowledge.

Part I

This meets that

Chapter 2

Basic Concepts

2.1 ROS

Before we are able to explain benefits from PhaROS, we need to inform about what ROS is.

ROS is both, an integration framework for robotics and an architecture for robotics.

In terms of architecture, it propose a process layout, in terms of integration, it propose channels and protocols of communication.

ROS architecture proposal is a graph of processes with connections given by topic of interest.

ROS basic concepts

ROS counts with the following concepts:

- Nodes
- Topics
- Services

Node A node in ROS is a process that is susceptible to connect and be connected with other processes and to contribute to the robotic system. (Giving like that the outline of graph of process).

For more technical information about nodes, browse this link http://wiki.ros.org/Nodes

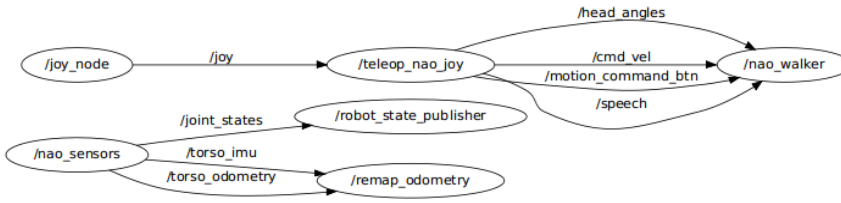


Figure 2.1: A ROS graph example. With circles the nodes and with arrows the topics

Topic The connections or relations between Nodes inside ROS architecture are by interest, so, they are called topics. A topic is a channel through is sent a kind of information. Nodes can join to a topic in concept of publishers or in concept of subscribers. All the information given through a topic by all the publisher nodes will arrive to all the subscriber nodes. One particular property of a topic is that topics are one way communication.

For more technical information about topics, browse this link http://wiki.ros.org/Topics

Service Beside the strong relation between Nodes (topics) we have also other way to relate: the Service. A service is a capability offered on demand. It requires a request and it offers a response in response. So topics are bilateral communications. Usually are related with changes of behaviour of the implementor node, access to information, resources management.

For more technical information about services, browse this link http://wiki.ros.org/Services

Adopting ROS for development

As we told before, ROS has two meanings, integration and architecture. So, to adopt ROS has two main impacts on our development, mainly, in that two branches.

Integration lines When we talk about integration, we talk about making several solutions work together. There is no way to make several persons to work together on something meaningful talking different languages. The same way, in order to make several pieces of human work to cooperate we need to share some standard uses. The first and easiest part of the standard is the protocol of communication, what includes handshake, discovery and data serialisation.

There is, however, a part less formal, what is about naming and typing communications, which is the hardest use to adopt, because is defined dynamically by the community, so, is a matter of usage.

Architecture lines In terms of architecture, is highly important to take in care the meaning of process graph. The graph of process is useful meanwhile is up-to-date. That means that the information should flow as fast as it needs. All the nodes count with that fact. In order to achieve that goal, nodes should be specific enough, in order to do not overload the process capability with too much Input/Output operations and to do not take too much time for generating the expected output.

This lines usually converge in nodes where the main process/thread runs with a rate (it runs n times per second), synchronising information from different inputs and informing output. Usually heavy work is available through services.

Benefits The main and quickly seen benefits is the amount of work already done and available that we can use. From algorithms of localisation to visualisation tools and ready to work simulators. The catalog promise to let your application have at least a prototype in not too much time.

The highly responsive community of ROS is growing up quickly, you can have answers to your problems quick if is not already answered (ROS count with an index of already made questions where anyone is welcome to participate. http://answers.ros.org/

Your work will be easy to integrate with actual technologies and integrate it again in the future will may be difficult but doable.

2.2 PhaROS

PhaROS is library and framework that allows you to write ROS nodes in the Pharo Smalltalk language (www.pharo.org).

Aiming

The aiming of PhaROS is to exploit the benefits of a highly dynamic image-based language (live environment) for the agile and prototype oriented development of ROS nodes and robotics solutions. Encourage code reusability and testing. An being also able to program and design in terms of high level abstractions, making possible to PhaROS to be used in robotics programming courses, without needing to know much about robotics in first time.

In order to make a goal-meeting analysis i will list this goals in a more concrete way

- Node prototyping
- Code reusability
- Testability
- Solution prototyping
- High level abstractions

Philosophy

Before talk about which abstractions do we have at hand, we want to propose a method of work that has gave us several satisfactions. We will make a more formal presentation in forward in the article, but first we introduce this ideas as guidelines of making up of available abstractions.

Do not reinvent the wheel This is a really usual thing in engineering, do not invent something new or worst, something old, during the engineering process. Check always if there is something already done to solve your problems, or part of them. Several times the need we have can be solved quickly by putting some nodes to work together. Several times it will not be the best idea from performance point of view, but again, you will gain better knowledge of the problem by using already existing solutions, allowing you to know that maybe performance was not critical on that issue or making you to understand where there are bottlenecks related with the problem it self.

Behaviour driven programming - BDP Even when it has things in common with the well known idea of Behaviour driven development of software engineering, ([1]), is not actually the same. You will realise quite soon that is not straight forward at all to test a robotic behaviour, and that stand for it goes quickly against prototyping. Then, we will not base our development on expected behaviour and easy to repeat simple behaviour, but in desired robotic behaviour.

Then we propose to think about which is the behaviour that you want for your robot for a particular scenario, localise the related sources of information available in ROS and start using this data as soon as possible, to understand quickly the relation in between you desires and the system. Several times you will need to cut behaviours in several smaller behaviours. But is

[1] **?**, .

quite important to do not implement more than the strictly needed in terms of stability and robustness.

To focus on the behaviour will allow you to understand which is the expected system before and after your program start to run, allowing you to make some architectural - layout tests.

Domain driven design This is an already old and well known idea in software, thats why we will not spend lines explaining it ([2]). We attach tightly to this idea when we are developing drivers and behaviours because it is what will make our code more reusable. Despite available abstractions, if we don't make things to make sense in the domain they exists, all the things we implement will be related with the punctual problem we are solving. So do never miss the domain picture of your solution.

[2] **?**, .

Chapter 3

Making up our environment

3.1 Installing Pharo

Before start with anything, we need to be sure we have installed all the needed libraries and dependencies needed by pharo. A default Pharo installation is usually an easy deal. But usually is not always. Be sure that you always install all the things it needs before moving forward.

Pharos installation tool Installing Pharo PPA

```
pharos@PhaROS:~$ sudo add−apt−repository ppa:pharo/stable
```

This adding of repository will make Pharo available from your ubuntu installation.

Pharos installation tool Installing Pharo PPA - Output

```
More info: https://launchpad.net/~pharo/+archive/stable
Press [ENTER] to continue or ctrl−c to cancel adding it

\gpg: keyring `/tmp/tmpa4dlup/secring.gpg' created
gpg: keyring `/tmp/tmpa4dlup/pubring.gpg' created
gpg: requesting key 708EAA85 from hkp server keyserver.ubuntu.com
gpg: /tmp/tmpa4dlup/trustdb.gpg: trustdb created
gpg: key 708EAA85: public key "Launchpad PPA for Pharo" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
OK
```

Pharos installation tool Apt-get Update

```
pharos@PhaROS:~$ sudo apt−get update
```

The output of this command is quite long and not very informative. Be sure that all the lines have one of the following shapes:

Pharos installation tool Apt-get Update

```
Hit http://extras.ubuntu.com raring Release
Get:1 http://ppa.launchpad.net raring Release.gpg [316 B]
Ign http://security.ubuntu.com raring−security/main Translation−en_US
```

If it appear any communication error, be sure you have your network well configured, and that you can reach the ubuntu servers.

Pharos installation tool Apt-get Update

```
pharos@PhaROS:~$ sudo apt−get install pharo−launcher:i386
```

The output of this last command will change depending on your architecture based ubuntu. For 32 bits, there are not much dependencies, for 64 bits there are a lot.

After this installation you will have available the following two command

- pharo-vm-x - Pharo vm for running images WITH graphical interface (x is for x11)
- pharo-vm-nox - Pharo vm for running images WITHOUT graphical interface (nox is for not x11)

Having all the dependencies installed, you can go to the next section.

3.2 Installing Pharos installation tool

Pharos installation tool idea is to make us easier all the life cycle of a robotic project. We will use it to avoid going to silly implementation details before being able to understand the core concepts.

Installing Pharos installation tool

```
pharos@PhaROS:~$ wget http://car.mines-douai.fr/wp-content/
uploads/2014/01/pharos.deb .
pharos@PhaROS:~$ sudo dpkg -i pharos.deb
```

To be sure the tool correctly installed execute the following command.

Pharos installation tool help

```
pharos@PhaROS:~$ pharos --help
```

The output for this should be something like

Pharos installation tool help output

```
pharos usage
  pharos install              - Installs a package
  pharos create               - Creates a new package based on an
archetype
  pharos create-repository    - Creates a smalltalk script for
creating your own repository
  pharos register-repository  - Register a new package repositry
  pharos list-repositories    - List registered repostiries
  pharos update-tool          - Check if there is any update of the
tool, and it update it.
  pharos ros-install          - Installs a ros
```

We will learn how to use this tool during the text. If you are eager for know it, there is a full explanation at the appendix Appendix C

3.3 Installing ROS with Pharos installation tool

One of the many things that Pharos installation tool does, is to make easier our life not just for beginners, but for advanced users as well, because try to allow the bureaucracy is not about being beginner ;)

Lets check which is the way to use the tool for this goal

Pharos installation tool help

```
pharos@PhaROS:~$ pharos --help
```

Pharos installation tool help output

```
pharos usage
    pharos install              - Installs a package
    pharos create               - Creates a new package based on an
archetype
    pharos create-repository    - Creates a smalltalk script for creating
 your own repository
    pharos register-repository  - Register a new package repositry
    pharos list-repositories    - List registered repostiries
    pharos update-tool          - Check if there is any update of the tool,
 and it update it.
    pharos ros-install          - Installs a ros
```

As we can see, there is a ros-install option that can be invoked. Lets check what can it does for us:

Pharos installation tool for ROS installation

```
pharos@PhaROS:~$ pharos ros-install --help
```

Pharos installation tool for ROS installation

```
usage: pharos ros-install
It installs a ROS in the machine. (Valid just for ubuntu).

Options:

--version          version of ROS. { fuerte | groovy | hydro }. Default: groovy.
--type-installation Type of ros installation { desktop | desktop-full | ros-base }.
   Default: desktop.
--configure     Indicates if the installation should be configured (modifiy .bashrc file )
   .{true|false} Default: true.
--create-workspace  Indicates if a workspace should be created. It will create a
   workspace at ~/ros/worskpace. { true|false }. Default: true.
```

The tool propose us several configurations, and an easy way to install by default (which is quite good by default)

Version: Specify the distribution of ROS we want to install (It should be compatible with our Ubuntu installation). By default it will try with Groovy.

Type-installation: Specify what do we want to load. Ros-base will load just the architecture things, desktop will download several graphic tools and some stacks, desktop-full even more tools and stacks. (Check in the wiki site of each distribution to know what will be downloaded if you need more information) By default it will download desktop.

Configure: Is a boolean value. If you are beginner, just left it with the default. It will modify .bashrc file of the current user, in order to have ROS always available.

Create-Workspace: Is a boolean value. If you are beginner, just left it with the default. In order to develop we need to have a place to place our code. This place is called workspace. And if this value is true, the tool will make up a new one in the standard location.

This command is safe for installing because it will tell you if the ROS that you want to install is compatible with your Ubuntu installation.

Running in an ubuntu 13.04 (Raring) , lets try to install a Fuerte distribution.

Pharos installation tool Installing incompatible version

```
pharos@PhaROS:~$ pharos ros-install --version=fuerte
```

Pharos installation tool Installing incompatible version - Output

```
fuerte version is not available for: raring
```

In order to have a rich example, i will install an hydro version in my raring ubuntu installation.

Pharos installation tool Installing Hydro in Ubuntu: Raring

```
pharos@PhaROS:~$ pharos ros-install --version=hydro
```

The output of this command is quite long, but i will stripe the most significant parts.

Pharos installation tool Installing Hydro in Ubuntu: Raring - Output

```
[sudo] password for pharos:
Hit http://extras.ubuntu.com raring Release
Hit http://security.ubuntu.com raring-security/main Sources
Hit http://fr.archive.ubuntu.com raring-updates Release
Hit http://fr.archive.ubuntu.com raring-backports Release
```

```
Hit http://security.ubuntu.com raring-security/restricted Sources
Hit http://extras.ubuntu.com raring/main Sources
Hit http://fr.archive.ubuntu.com raring/main Sources
Hit http://security.ubuntu.com raring-security/universe Sources
Hit http://extras.ubuntu.com raring/main i386 Packages
Hit http://fr.archive.ubuntu.com raring/restricted Sources
Hit http://security.ubuntu.com raring-security/multiverse Sources
Hit http://fr.archive.ubuntu.com raring/universe Sources
Hit http://fr.archive.ubuntu.com raring/multiverse Sources
Hit http://security.ubuntu.com raring-security/main i386 Packages
[...]
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
The following extra packages will be installed:
  binfmt-support blt build-essential cmake cmake-data collada-dom-dev
  collada-dom2.4-dp-base collada-dom2.4-dp-dev comerr-dev cpp
-4.4 debhelper
  dh-apparmor docutils-common docutils-doc doxygen doxygen-latex
dpkg-dev
  emacsen-common fakeroot ffmpeg fonts-lmodern fonts-lyx fonts-texgyre
  freeglut3 g++ g++-4.4 g++-4.7 gcc-4.4 gcc-4.4-base gccxml gir1.2-gtk
-2.0
  graphviz hddtemp html2text krb5-multidev latex-beamer latex-xcolor
[...]
libqt4-svg libqt4-test libqt4-xml libqt4-xmlpatterns libqtcore4 libqtgui4
  libssl1.0.0 libtiff5 libx11-6 libx11-xcb1 libxcb-dri2-0 libxcb-glx0
  libxcb-render0 libxcb-shm0 libxcb1 libxcursor1 libxext6 libxfixes3 libxi6
  libxinerama1 libxml2 libxrandr2 libxrender1 libxt6 libxxf86vm1 python2.7
  python2.7-minimal qdbus
57 upgraded, 675 newly installed, 0 to remove and 269 not upgraded.
Need to get 967 MB/983 MB of archives.
After this operation, 2 233 MB of additional disk space will be used.
Do you want to continue [Y/n]? Y
[...]
Get:2 http://fr.archive.ubuntu.com/ubuntu/ raring/main libgfortran3 i386
4.7.3-1ubuntu1 [326 kB]
Get:3 http://fr.archive.ubuntu.com/ubuntu/ raring/main libglade2-0 i386
1:2.6.4-1ubuntu2 [52,9 kB]
Get:4 http://fr.archive.ubuntu.com/ubuntu/ raring-updates/main libgnutls-
openssl27 i386 2.12.23-1ubuntu1.1 [21,9 kB]
Get:5 http://packages.ros.org/ros/ubuntu/ raring/main libpcl-kdtree-1.7 i386
1.7.0-2+raring1 [239 kB]
Get:6 http://fr.archive.ubuntu.com/ubuntu/ raring/main libgssrpc4 i386 1.10.1+
dfsg-4+nmu1 [57,5 kB]
...
Get:686 http://fr.archive.ubuntu.com/ubuntu/ raring/universe libbullet2.80 i386
2.80+sp1+dfsg-0ubuntu1 [754 kB]
```

```
    Get:687 http://fr.archive.ubuntu.com/ubuntu/ raring/universe libbullet-dev i386
 2.80+sp1+dfsg-0ubuntu1 [362 kB]
    Preconfiguring packages ...
    Fetched 967 MB in 14min 32s (1 109 kB/s)
    (Reading database ...
    Processing triggers for man-db ...
    Processing triggers for doc-base ...
    Processing 1 added doc-base file...
    Processing triggers for install-info ...
    [...]
    Processing triggers for libc-bin ...
    ldconfig deferred processing now taking place
    Processing triggers for python-support ...
    Creating symlink "/home/pharos/ros/workspace/src/CMakeLists.txt" pointing to
"/opt/ros/hydro/share/catkin/cmake/toplevel.cmake"
    -- The C compiler identification is GNU 4.7.3
    -- The CXX compiler identification is GNU 4.7.3
    -- Check for working C compiler: /usr/bin/cc
    -- Check for working C compiler: /usr/bin/cc -- works
    -- Detecting C compiler ABI info
    -- Detecting C compiler ABI info - done
    -- Check for working CXX compiler: /usr/bin/c++
    -- Check for working CXX compiler: /usr/bin/c++ -- works
    -- Detecting CXX compiler ABI info
    -- Detecting CXX compiler ABI info - done
    -- Using CATKIN_DEVEL_PREFIX: /home/pharos/ros/workspace/devel
    -- Using CMAKE_PREFIX_PATH: /opt/ros/hydro
    -- This workspace overlays: /opt/ros/hydro
    -- Found PythonInterp: /usr/bin/python (found version "2.7.4")
    -- Using PYTHON_EXECUTABLE: /usr/bin/python
    -- Using Debian Python package layout
    -- Using CATKIN_ENABLE_TESTING: ON
    -- Call enable_testing()
    -- Using CATKIN_TEST_RESULTS_DIR: /home/pharos/ros/workspace/build
/test_results
    -- Looking for include file pthread.h
    -- Looking for include file pthread.h - found
    -- Looking for pthread_create
    -- Looking for pthread_create - not found
    -- Looking for pthread_create in pthreads
    -- Looking for pthread_create in pthreads - not found
    -- Looking for pthread_create in pthread
    -- Looking for pthread_create in pthread - found
    -- Found Threads: TRUE
    -- Found gtest sources under '/usr/src/gtest': gtests will be built
    -- catkin 0.5.81
    -- BUILD_SHARED_LIBS is on
    -- Configuring done
    -- Generating done
```

```
-- Build files have been written to: /home/pharos/ros/workspace/build
Base path: /home/pharos/ros/workspace
Source space: /home/pharos/ros/workspace/src
Build space: /home/pharos/ros/workspace/build
Devel space: /home/pharos/ros/workspace/devel
Install space: /home/pharos/ros/workspace/install
####
#### Running command: "cmake /home/pharos/ros/workspace/src -
DCATKIN_DEVEL_PREFIX=/home/pharos/ros/workspace/devel -
DCMAKE_INSTALL_PREFIX=/home/pharos/ros/workspace/install" in "/home/
pharos/ros/workspace/build"
####
####
#### Running command: "make -j1 -l1" in "/home/pharos/ros/workspace/
build"
####
```

During this script we download all the required software of the chosen distribution, download some extra tools, configure and initialise the basic tools. And finally we make up a catkin package which is, as final step compiled.

3.4 Installing ROS by hand

Further we will see that PhaROS is a library that is abstract of installation, what means that we can run a node from a machine without ROS, but having a remote ROS installation. However we need to understand ROS architecture and runtime in order to understand how to interact with it. Therefore, for the firsts examples, we need to have a machine with a running ROS installation.

For this we strongly recommend to use a LTS distribution of Ubuntu, but previously to make the installation, be sure that the distribution you want to use is compatible.

Fuerte

- Ubuntu 10.04 (Lucid)
- Ubuntu 11.10 (Oneiric)
- Ubuntu 12.04 (Precise)

Groovy

- Ubuntu 11.10 (Oneiric)
- Ubuntu 12.04 (Precise)
- Ubuntu 12.10 (Quantal)

Hydro

- Ubuntu 12.04 (Precise)
- Ubuntu 12.10 (Quantal)
- Ubuntu 13.04 (Raring)

For this achievement, you can follow one of the next tutorials

- http://wiki.ros.org/fuerte/Installation/Ubuntu
- http://wiki.ros.org/groovy/Installation/Ubuntu
- http://wiki.ros.org/hydro/Installation/Ubuntu

Chapter 4

First Steps with PhaROS

4.1 Install PhaROS tool

This tool ease the installation and the use of ROS middleware. First of all, we need to install PhaROS-Tool, then we can use it to ease the configuration of our own nodes.

Download PhaROS-Tool

You can download PhaROS-Tool with:

```
$ wget http://car.mines-douai.fr/wp-content/uploads/2014/01/pharos.deb
```

Once downloaded just execute the following lines to install it and update it.

```
$ sudo dpkg -i pharos.deb
$ pharos update-tool
```

Executing the following line validate the correct installation and will give you an help.

```
$ pharos --help
```

The result is:

```
pharos usage
  pharos install          - Installs a package
  pharos create           - Creates a new package based on an archetype
  pharos create-repository    - Creates a smalltalk script for creating your own
      repository
```

```
pharos register−repository  − Register a new package repositry
pharos list−repositories    − List registered repostiries
pharos update−tool          − check if there is any update of the tool, and it update it.
pharos ros−install      − installs a ros
```

Each of these commands will be explained in this document.

Installing ROS with PhaROS-Tool: pharos ros-install

The following example line will install the desktop-full of the hydro version of ROS:

```
$ pharos ros−install −−version=hydro −−type−installation=desktop−full
```

You can install the version of ROS you want. This process works only on Ubuntu. The possible options are available executing this line:

```
$ pharos ros−install −−help
```

They are:

- –version : you define the version of ROS (fuerte | groovy | hydro). By default it is groovy.
- –type-installation : you can define the type of ros installation (desktop | desktop-full | ros-base). By default, it is desktop.
- –configure : It indicates if the installation should be configured. It modifies the .bashrc file. The value can be true or false. By default, it is true.
- –create-workspace : it indicates if a workspace should be created. It will create a workspace at /ros/worskpace. The value can be true or false. By default, it is true.

Knowing which repositories are registered: pharos list-repositories

The repositories listed are the places where your projects are saved. By default there are two repositories: the first one is the main repository of PhaROS, the second is the material for peripherics we already plugged on PhaROS. For example, there is a package for the turtlebot in this repository.

You can list repositories with the folowing command:

```
$ pharos list−repositories
```

By default, it will give you the two following repositories, and attached information.

```
--------
url = http://car.mines-douai.fr/squeaksource/PhaROS
user =
directory = PhaROSDeploymentDirectory
password =
repository = PhaROSDeploymentDirectory
--------
url = http://car.mines-douai.fr/squeaksource/Peripherics
user =
directory = PeriphericsDirectory
password =
repository = PeriphericsDirectory
--------
```

Adding a new repository: pharos register-repository

This command allows one to add one new repository. To use the ev3 ROS package, you have to register the following repository:

```
$ pharos register-repository --url=http://smalltalkhub.com/mc/JLaval/Ev3ROS/main
    --package=EV3ROSDirectory
```

The package EV3ROSDirectory contains all the metadata needed to build a ROS package. It is explained after in the chapter.

Tip: If your repository requires user/password for reading add –user=User –password=Password to the example. Pay attention that the User/Password will be stored in a text file without any security.

After adding the new repository, you can verify it is included it is available in the list of repository:

```
$ pharos list-repositories
```

Yes, it is !

```
--------
url = http://car.mines-douai.fr/squeaksource/PhaROS
user =
directory = PhaROSDeploymentDirectory
password =
repository = PhaROSDeploymentDirectory
--------
url = http://car.mines-douai.fr/squeaksource/Peripherics
user =
directory = PeriphericsDirectory
```

```
password =
repository = PeriphericsDirectory
———————
url = http://smalltalkhub.com/mc/JLaval/Ev3ROS/main
user =
directory = EV3ROSDirectory
password =
repository = EV3ROSDirectory
———————
```

Adding a repository would need some options. Here are the available options to register a repository:

- –url : this option is mandatory, it is the url of a monticello repository
- –package : this option is mandatory, it is the name of the Directory package in the monticello repository.
- –directory : This is the name of the class that works as directory. This class must implement *#includesPackage:* and *#deployUnitForPackage:*. By default, the value is the same as the package name.
- –user : it is the username to access the monticello repository. The default value is empty.
- –password : it is the password for the monticello repository. The default value is empty.

Installing a PhaROS package: pharos install

After having added the ev3 repository, as explained in the previous section, you can install the package Ev3 for ROS. This package is named ev3. The command should be run in the ros workspace. By default, the ROS workspace is in *~/ros/workspace/*.

```
$ cd ros/workspace/
$ pharos install ev3 ——version=3.0 ——ros—distro=hydro ——pharo—user=JLaval ——
      force—new
```

The previous line search the ev3 package in the available repositories. When it found it, it install it in a new pharo image. Here, we are using the version 3.0 of Pharo. It is configured for ROS Hydro, the version of ROS that I previously installed. I also configure the user name in the Pharo image. All the source code that I produce with this image will have as author the name JLaval. Finally, I force a new installation. It means that if the package already exists, it will be replaced and all the changes will be lost.

The usage of this command can be completed with the following options:

- –location : You can give the absolute path to a valid catkin workspace (not the source folder). The default value is the current directory.
- –silent : The values can be true or false. If silent is false you will be able to see the installation of the output image. The default value is true.
- –version : It indicates the version of Pharo to download. The default value is 2.0. It can be 1.4, 2.0 or 3.0.
- –ros-distro : It indicates the distribution of ROS that you want to use. By default it is groovy, but you can configure it with groovy, hydro or fuerte.
- –package : It indicates the name of the package to install (If you use «pharos install aPackage –package=otherPackage» the one to be installed is otherPackage).
- –package-version : It indicates the package version. By default it takes the latest one.
- –pharo-user : It indicates the user name for the result image.
- –force-new : It DELETES the package if it exists in the pointed location.

4.2 Run PhaROS software

To edit the source code, just run the following line. It will open the Pharo image used for your package.

```
$ rosrun ev3 edit
```

For example, you can browse the method EV3ROSPackage»scriptPublishColour in PhaROS. It is the script to run a ROS node that publishes the color sensor data.

Each scripts in PhaROS has the signature format like *scriptWord1Word2*. It begins with *script* followed by the words of the method.

Each script when compiled generates a .sh script. Like that, when you need to execute your script without touching the source code, you don't need to open PhaROS. You can access to the scripts generated by executing the following lines:

```
$ roscd ev3
$ ls image/script
```

The first line change the directory to the package ev3. The second line lists the available scripts. When writting these lines, there are 3 available scripts:

```
move_meanwhile_you_can.st
publish_colour.st
publish_sonar.st
```

The first script make the robot move meanwhile it detects an obstacle. The second provide a publisher for the color sensor. The third script publish the sonar data.

As expected we find the script that publish the color sensor data. The name given to the script is related to the format of the name given inside PhaROS. A method signature like *scriptWord1Word2* become *word1_word2.st*.

Now, you can run the scripts without seeing PhaROS environment as following:

```
$ rosrun ev3 headless publish_colour
```

or run the script with the environment:

```
$ rosrun ev3 pharos publish_colour
```

4.3 Create your ROS package

There are three steps to create your packages: first, you will create your PhaROS package, that contains your source code. Then, you will create a configuration that gives PhaROS the possibility to load your source code. Finally, you will create a method that contains the ROS Package information.

Create your package : pharos create

To create a Pharo image for developing your own software, you would use the command *pharos create*. In the following line, I am creating a phaROS configuration for the package *my_super_ros_package*.

```
$ cd ~/ros/workspace/
$ pharos create my_super_ros_package --version=3.0 --ros-distro=hydro --
    author=JLaval --author-email=jannik.laval@gmail.com
```

Pay attention that your package name follows the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.

The usage of this command can be completed with the following options:

- –help : It shows this text
- –location : You can give the absolute path to a valid catkin workspace (not the source folder). The default value is the current directory.
- –silent : The values can be true or false. If silent is false you will be able to see the installation of the output image. The default value is true.
- –version : It indicates the version of Pharo to download. The default value is 2.0. It can be 1.4, 2.0 or 3.0.
- –ros-distro : It indicates the distribution of ROS that you want to use. By default it is groovy, but you can configure it with groovy, hydro or fuerte.
- –archetype : It indicates the name of the archetype to use for creating things. By default, it is basic-archetype.
- –author : It indicates the name of the author.
- –maintainer : It indicates the name of the maintainer. The default value is the one indicated in author.
- –author-email : It indicates the email of the author.
- –maintainer-email : It indicates the email of the maintainer. The default value is the one indicated in author-email.
- –description : It indicates the description of the package.
- –pharo-user : It indicates the user name for the result image.
- –force-new : It DELETES the package if it exists in the pointed location.

Inside PhaROS environment, one package, *My_super_ros_packagePackage* was created with 2 classes inside: *My_super_ros_packagePackage* and *My_super_ros_packageNodelets*. Inside the class *My_super_ros_packagePackage*, you will find some example methods.

Create the configuration

Create the ConfigurationOf inside PhaROS, in a package ConfigurationOfMySuperRosPackage. To create a configuration in PhaROS, yo can find more documentation in the free book “Deep into Pharo” available on http://www.deepintopharo.com/. For example, for the ev3 package, I created this class:

```
Object subclass: #ConfigurationOfEv3ROS
   instanceVariableNames: 'project'
   classVariableNames: 'LastVersionLoad'
   category: 'ConfigurationOfEv3ROS'
```

This class contains the method *baseline10:*, which load my source code.

```
ConfigurationOfEv3ROS>>baseline10: spec
   <version:'1.0−baseline'>
   spec for: #common do: [
         spec blessing: #baseline.
         spec repository: 'http://smalltalkhub.com/mc/JLaval/Ev3ROS/main'.

         spec project: 'JetStorm' with: [
            spec
               className: 'ConfigurationOfJetStorm';
               file: 'ConfigurationOfJetStorm' ;
               version: #bleedingEdge;
               repository: 'http://smalltalkhub.com/mc/JLaval/JetStorm/main'.
         ].
         spec project: 'PhaROS' with: [
            spec
               className: 'ConfigurationOfPhaROS';
               file: 'ConfigurationOfPhaROS' ;
               version: #bleedingEdge;
               repository: 'http://car.mines−douai.fr/squeaksource/PhaROS'.
         ].

         spec package: 'EV3ROSPackage' with: [
            spec requires: #('JetStorm' 'PhaROS')
         ].

   ].
```

Create the ROS metadata: pharos create-repository

Creating a directory for your own project repository needs a st file. Then, you have to load it in a pharo image and personalize it.

For example, I used the following command line to create my file for Ev3.

```
$ pharos create−repository EV3ROS −−user=JLaval > eve.st
```

Then, I just file in the file.st in PhaROS. It will create a package EV3ROSDirectory that contains the meta data.

You can do it for My_super_ros_package:

```
$ pharos create−repository My_super_ros_package −−user=JLaval > myROS.st
```

It will create a package My_super_ros_packageDirectory that contains the meta data. This command can have two options:

- –user : gives the name of the user for generation by default is 'Generated'
- –output path to where the script will be dumped. By default is the standar output.

When the file is integrated in PhaROS, you can see the methods *EV3ROSDirectory class»archetypeEv3* and *EV3ROSDirectory class»ev3RosPackage*, respectively *My_super_ros_packageDirectory class»archetypeExample* and *My_super_ros_packageDirectory class»exampleMetacelloPackage* for My_super_ros_package. There are other files that are not necessary here. You can browse and discover them.

```
EV3ROSDirectory class>>archetypeEv3
  ^ {
       #archetype −> 'EV3ROS−archetype' .
       #description −> 'Minimum dependancies for using EV3+ROS' .
       #license −> 'MIT' .
       #version −> '0.1.0' .
       #maintainer −> ({
          #name−> 'JLaval' .
          #email −> 'jannik.laval@gmail.com'
       } asDictionary) .
       #author −> ({
          #name−> 'JLaval' .
          #email −> 'jannik.laval@gmail.com'
       } asDictionary) .
       #metacello −> ({
          #url −> 'http://smalltalkhub.com/mc/JLaval/ev3ros/main' .
          #configurationOf −> 'ConfigurationOfEv3ROS' .
          #package −> 'EV3ROSArchetype'
       } asDictionary)
     } asDictionary
```

```
EV3ROSDirectory class>>ev3RosPackage
  ^ {
       #name −> 'ev3' .
       #description −> 'ROS package for managing Ev3' .
       #license −> 'MIT' .
       #version −> '0.1.0' .
       #maintainer −> ({
          #name−> 'JLaval' .
```

```
            #email —> 'jannik.laval@gmail.com'
        }  asDictionary) .
        #author —> ({
            #name—> 'JLaval' .
            #email —> 'jannik.laval@gmail.com'
        } asDictionary) .
        #metacello —> ({
            #url —> 'http://smalltalkhub.com/mc/JLaval/Ev3ROS/main' .
            #configurationOf —> 'ConfigurationOfEv3ROS' .
            #package —> 'EV3ROSPackage'
        } asDictionary)
    } asDictionary
```

As you can see, each method represents the metadata used for the publication in ROS. You can use and complete your data for your own PhaROS package.

Finally, do not forget to change the method *EV3ROSDirectory class»deployUnitsMetadata* to call the correct configuration method.

```
EV3ROSDirectory class>>deployUnitsMetadata
  ^ {
    self archetypeEv3 .
    self ev3RosPackage.
  }
```

4.4 Create your ROS node source code

Last but not least, we can create the behavior in the PhaROS environment. In this section, I present the source code used by the three nodes: the first node publish the sonar sensor data, the second publish the colour sensor data, the third one subscribe to the sonar publisher and move until the data is lower than 10 centimeters.

First, I initialise the node with a connection to the robot.

```
EV3ROSPackage>>initialize
  super initialize.
  ev3 := Ev3Vehicle newIp: '192.168.1.4' daisyChain: #EV3.
```

Then, following there are the two scripts used to publish sensor data.

```
EV3ROSPackage>>scriptPublishColour
  | colourSensor colourPublisher |
  colourSensor := ev3 sensor3.
  colourPublisher := self node topicPublisher: '/ev3/colour' typedAs: 'std_msgs/Int32'.
```

```
  self paralellize looping publish: colourSensor at: colourPublisher.
```

```
EV3ROSPackage>>scriptPublishSonar
  | sonarSensor sonarPublisher |
  sonarSensor := ev3 sensor4.
  sonarPublisher := self node topicPublisher: '/ev3/sonar' typedAs: 'std_msgs/Float32'.

  self paralellize looping publish: sonarSensor at: sonarPublisher .
```

The usefull method used in the two previous method:

```
EV3ROSPackage>>publish: aSensor at: aPublisher.
  aPublisher send: [ : m |
    m data: aSensor read.
  ].
  0.5 hz wait.
```

The following methods allow the robot to move and to stop.

```
EV3ROSPackage>>letEv3Move
  ev3 motorB isRunning ifFalse: [ ev3 motorB startAtSpeed: 30 ].
  ev3 motorC isRunning ifFalse: [ ev3 motorC startAtSpeed: 30 ].
```

```
EV3ROSPackage>>stopEv3
  ev3 motorB stop.
  ev3 motorC stop.
```

Then this is the script that allows the robot to subscribe to the sonar sensor and move.

```
EV3ROSPackage>>scriptMoveMeanwhileYouCan
  (self node buildConnectionFor: '/ev3/sonar')
      typedAs: 'std_msgs/Float32';
      for: [ : m |  (m data < 10) ifTrue: [  self stopEv3.  ]  ifFalse: [ self letEv3Move ]];
      connect.
```

Here is the end of this tutorial.

Part II

Starting like our ancestors, by example

I strongly believe in directing teaching by example, and by usage. Then, before go to concepts, architecture and design we will just use what we have at hand.

Chapter 5

Topics: publishers and subscribers

5.1 Installing our first example with Pharos installation tool

To explain how to run a PhaROS program, first we need to have it. In order to figure this out, we will install an easy example available in the main PhaROS repositories.

Installing chatter

```
pharos@PhaROS:~$ pharos install chatter --location={path-to-workspace
} --ros-distribution={fuerte | groovy | hydro}
```

Installation output

```
--2014-02-03 11:11:44--  http://get.pharo.org/vm
Proxy request sent, awaiting response... 200 OK
Length: 5285 (5.2K) [text/html]
Saving to: `STDOUT'

100%[======================>] 5,285       --.-K/s   in 0.007s

2014-02-03 11:11:44 (772 KB/s) - written to stdout [5285/5285]

Downloading the latest pharoVM:
   http://files.pharo.org/vm/pharo/linux/stable.zip

--2014-02-03 11:11:46--  http://get.pharo.org/20
```

```
Connecting to 10.1.1.3:8080... connected.
Proxy request sent, awaiting response... 200 OK
Length: 2587 (2.5K) [text/html]
Saving to: `STDOUT'

 0% [ ] 0         --.-K/s              Downloading the latest 20 Image:
   http://files.pharo.org/image/20/latest.zip
100%[======================>] 2,587       --.-K/s   in 0.002s

2014-02-03 11:11:46 (1.08 MB/s) - written to stdout [2587/2587]

Pharo.image
Setting up variables, network and obtaining required code....
Running pre process...
Installing package...
Running post process...
Unloading installation code ...
Base path: /home/pharos/workspace
Source space: /home/pharos/workspace/src
Build space: /home/pharos/workspace/build
Devel space: /home/pharos/workspace/devel
Install space: /home/pharos/workspace/install
####
#### Running command: "cmake /home/pharos/workspace/src -
DCATKIN_DEVEL_PREFIX=/home/pharos/workspace/devel -
DCMAKE_INSTALL_PREFIX=/home/pharos/workspace/install" in "/home/pharos/
workspace/build"
####
-- Using CATKIN_DEVEL_PREFIX: /home/pharos/workspace/devel
-- Using CMAKE_PREFIX_PATH: /home/pharos/workspace/devel;/opt/ros/
groovy;/home/viki/ros/workspace/devel
-- This workspace overlays: /home/pharos/workspace/devel;/opt/ros/groovy
-- Using Debian Python package layout
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/pharos/workspace/build/
test_results
-- Found gtest sources under '/usr/src/gtest': gtests will be built
-- catkin 0.5.71
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 1 packages in topological order:
-- ~~  - chatter
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'chatter'
-- ==> add_subdirectory(chatter)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/pharos/workspace/build
```

```
####
#### Running command: "make -j1 -l1" in "/home/pharos/workspace/build"
####
```

The outline of this command is the installation of the chatter package. This chatter package includes two nodes:

- Talker – This program read the stander input (commonly keyboard) and send it to all the listener nodes.
- Listener – This program listen talkers and write the received information in the stander output.

Directory layout of our package ROS tools are based on a well known distribution of some folders and files in a proper place. This place is called workspace. Commonly since Groovy, a Catkin workspace. A catkin workspace is a folder with some configuration files distributed and self maintained in build and devel folders. There is also a third folder called src. In this folder we will find the packages, each folder is a package, a catkin package. Lets execute the command tree on the workspace and see what it show us.

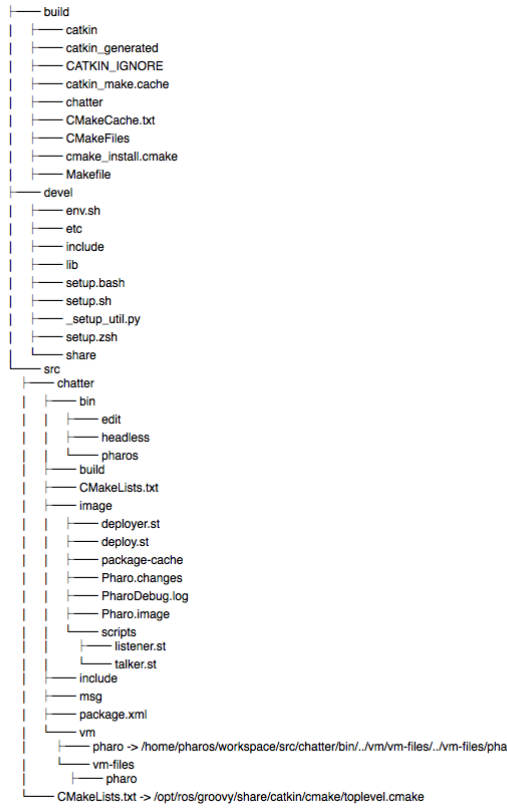


Figure 5.1: Output tree command

As we can see in the Figure 5.1, there are three main directories. The one we will focus is in the chatter one, inside the src folder.

This is the package that we just installed. Check Figure 5.2 for a file per file explanation of the chatter folder.

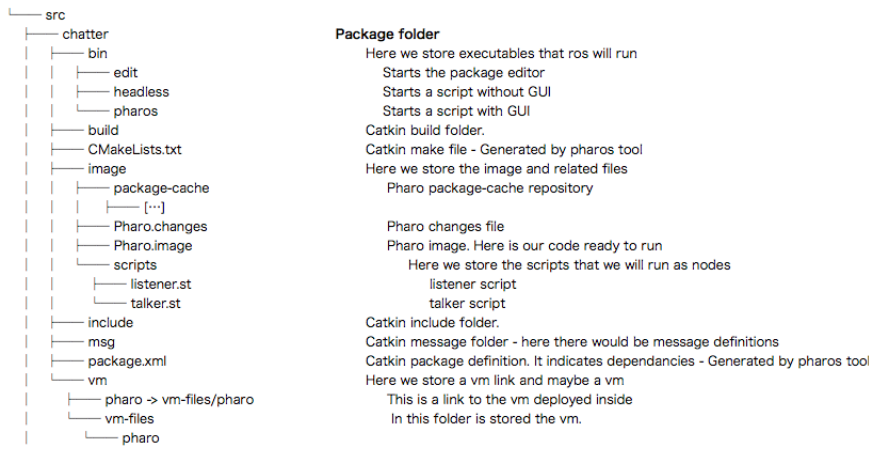


Figure 5.2: Output tree command

5.2 Executing Chatter

Open a terminal and startup ROS by executing

Starting up ROS

```
pharos@PhaROS:~$ roscore
```

This line will startup basic services from ROS needed to run this example. If you followed the basic installation tutorial http://wiki.ros.org/groovy/Installation/Ubuntu, the standard output of this command should be something like the following.

Starting up ROS - Output

```
... logging to /home/pharos/.ros/log/a3e8bd7e-8ea0-11e3-bfbd-0800275
a8d1d/roslaunch-PhaROS-12822.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:39207/
ros_comm version 1.9.47


SUMMARY
========

PARAMETERS
 * /rosdistro
 * /rosversion

NODES

auto-starting new master
process[master]: started with pid [12836]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to a3e8bd7e-8ea0-11e3-bfbd-0800275a8d1d
process[rosout-1]: started with pid [12849]
started core service [/rosout]
```

In one terminal (T1) execute the following command

Starting up listener

```
pharos/Pharos$~: rosrun chatter headless listener
```

The expected output (For other outputs, check the FAQ) of this command is:

Listener output

```
Starting up Listener
You will read here all the text written in talker nodes
```

In other terminal (T2) execute the following command

Starting up talker

```
pharos/Pharos$~: rosrun chatter headless talker
```

The expected output (For other outputs, check the FAQ) of this command is:

Talker output

```
Starting up Talker
Write your input and press enter
```

In the same console (The talker one) write something funny and press enter.

To see the following output

Talker output 2

```
Sending data to all subscribers
```

And to see in the listener console (T1), the following output.

Listener output 2

```
/PharoHandle−1391678151 said: {YOUR FUNNY COMMENT}
```

The node names in PhaROS, by default, are generated randomly, as PharoHandle-GeneratedNumber.

Now, if we try to add a new listener, repeating the process of the listener in a new console, you will see that this listener will also echo what ever you write in the talker console.

This is because in ROS architecture, the communications are stablished through named channels (Topics), all the processes that are connected to the same channel as publishers, talk to the same audience, all the processes that are connected as subscriber listen the same contents.

But, do not just believe me, lets see the code. In a terminal run the following command.

Inspecting the code

Pharo is a language that has it own IDE integrated, ready to use. With this line and the line that each package of PhaROS is one image, we will be able to have an editor able to edit each of our package installations.

For inspecting the code of the chatter we just need to execute the following line in a console (This execution don't need to have ROS platform running).

Editing

```
pharos/Pharos$~: rosrun chatter edit
```

This command will open a Pharo IDE.

Figure 5.3: Pharo IDE

Once in the IDE, lets browse the ChatterPackage class (shift+enter, write ChatterPackage into the spotlight. That will do the magic).

Figure 5.4: Pharo IDE

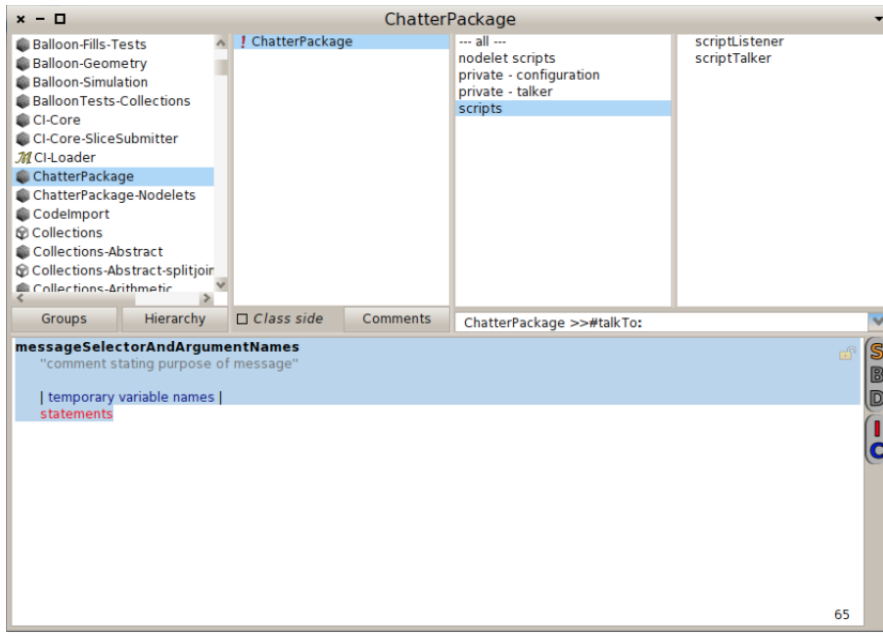


Figure 5.5: Chatter

In this class we encounter two method categories in the instance side. "scripts" and "private". For keeping simple the things we will browse just scripts now. Here we can find the methods scriptTalker and scriptListener.

Lets browse them.

Listener

```
self log: 'Starting up Listener '.
(self node buildConnectionFor:  '/chat/channel')
        typedAs: 'std_msgs/String' ;
        for: [ : aMsg :aChannel |
            self stdout nextPutAll: aChannel owner name, ' said:'.
            self stdout  nextPutAll: aMsg data.
            self stdout  nextPutAll: String lf.
        ];
        connect.

self log: 'You will read here all the text written in talker nodes'.
```

Before break the code in pieces, is important to understand the semantic of the operator ;. In Pharo ; means that this message will be sensed to the object that received the previous message. We can rewrite the code as

```
| builder |
self log: 'Starting up Listener '.
builder := (self node buildConnectionFor:  '/chat/channel').
builder typedAs: 'std_msgs/String'.
builder for: [ : aMsg :aChannel |
            self stdout nextPutAll: aChannel owner name, ' said:'.
            self stdout  nextPutAll: aMsg data.
            self stdout  nextPutAll: String lf.
        ].
builder connect.
self log: 'You will read here all the text written in talker nodes'.
```

```
| builder |
self log: 'Starting up Listener '.
builder := (self node buildConnectionFor:  '/chat/channel').
```

This lines define a variable builder, log to the stdout and ask the related internal node (Which is already builded and configured and represents our gateway to ROS) for a builder to connect to the topic named '/chat/channel'.

Typed as the ROS type std_msgs/String. What means that the subscription resultant of this building will receive this kind of data.

```
            self stdout nextPutAll: aChannel owner name, ' said:'.
            self stdout  nextPutAll: aMsg data.
            self stdout  nextPutAll: String lf.
        ].
```

For the specified callback. This callback is very easy, it just write to the console 'The owner of the channel said ... ' And at the end it adds a lf character.

```
    self log: 'You will read here all the text written in talker nodes'.
```

Finally, the message connect will make all the magic to happen. From now on, all communications that arrive in name of '/chat/channel' will be managed by the given block of code.

Talker

Given the need of a thread for this script, in order to make it easy to read we have splitted it into two methods.

```
    | publisher |
    self log: 'Starting up Talker'.
    publisher := self node topicPublisher: '/chat/channel' typedAs: 'std_msgs/String
'.
    self paralellize looping talkTo: publisher.
    self log: 'Write your input and press enter'.

ChatterPackage>>#talkTo: publisher
    | token |
    publisher send: [ : m |
        token :=self stdin upTo: Character lf.
        self log: 'Sending data to all subscribers'.
        m data: token.
    ].
```

Let split this code into understandable pieces.

```
    | publisher |
    self log: 'Starting up Talker'.
    publisher := self node topicPublisher: '/chat/channel' typedAs: 'std_msgs/String
'.
```

This lines define the variable publisher, log a welcome message and ask to related internal node (Which is already builded and configured and represents our gateway to ROS) for publisher for the topic called '/chat/channel', typed as std_msgs/String for assign it to the variable publisher. This means that after this line we will have in our publisher variable a publisher object that allow us to send messages to this topic, in the proper typed way.

```
self log: 'Write your input and press enter'.
```

From the following two lines, the second one is the easiest, it just log a need-for-action message to the user. But the first one is tricky. Thanks to TaskIT framework, this means paralellize inside an infinite loop the following message send. Which is talkTo: publisher.

This means that from now on, there will be a thread running inside a loop the following code

What lead us to the next part of the code. The #talkTo: method .

```
| token |
publisher send: [ : m |
    token := self stdin upTo: Character lf.
    self log: 'Sending data to all subscribers'.
    m data: token.
].
```

This method do not need to be breached, because the code is quite easy to understand in general. The only strange message is #send:

A publisher object in PhaROS understand the message #send: and receives a block of code. This block of code should receive the message to be sent in order to configure it. So you don't need to worry about how to build this type, or even if this type exists actually in the image. What you know is that the type will exist when the code is being executed. And that it will be managed as a typical DTO, trough setters and getters.

Because of this, we know that m will have the type std_msgs/String. Which is an object with a field called data, of type string. Inside the block we read the standard input, log a working-message to the user and set up the message object.

In the big picture, this means that we have set up a thread running in a loop asking reading what the user introduce and sending it to the '/chat/channel' topic.

5.3 What we learned

From console

- pharos install chatter –location=your-workspace
  Install chatter package in you ROS workspace
- rosrun chatter headless listener
  Starts up a listener process
- rosrun chatter headless talker
  Starts up a talker process
- rosrun chatter edit
  Starts up the editor on the chatter package

From a PhaROS package

- self node topicPublisher:TopicName typedAs:TopicType
  Construct a publisher for the given topic
- (self node buildConnectionFor:TopicName) typedAs:TopicType;for:[:aMsg |ACallBack];connect.
  Construct a subscription to the given topic and callback.

Chapter 6

Services: call and be called

6.1 Installing our second example with Pharos installation tool

For the next set of tools to use we will use a new example. Also installable.

Installing donatello

```
pharos@PhaROS:~$ pharos install donatello --location={path-to-workspace}
```

Since is a bit more complex than the previous one, the output of the installation will be a bit more complex as well.

Installing donatello - Output

```
--2014-02-14 15:36:56--  http://get.pharo.org/vm
Connecting to 10.1.1.3:8080... connected.
Proxy request sent, awaiting response... 200 OK
Length: 5285 (5.2K) [text/html]
Saving to: `STDOUT'

100%[=============================>] 5,285       --.-K/s   in 0.005s

2014-02-14 15:36:56 (1021 KB/s) - written to stdout [5285/5285]

Downloading the latest pharoVM:
   http://files.pharo.org/vm/pharo/linux/stable.zip
mkdir: cannot create directory `pharo-vm': File exists
--2014-02-14 15:36:59--  http://get.pharo.org/20
Connecting to 10.1.1.3:8080... connected.
Proxy request sent, awaiting response... 200 OK
```

```
        Length: 2587 (2.5K) [text/html]
        Saving to: `STDOUT'

        Downloading the latest 20 Image:              ] 0         --.-K/s
        100%[============================>] 2,587       --.-K/s   in 0.003s

        2014-02-14 15:36:59 (814 KB/s) - written to stdout [2587/2587]

           http://files.pharo.org/image/20/latest.zip
        Pharo.image
        Setting up variables, network and obtaining required code....
        Running pre process...
        Installing package...
        Running post process...
        Base path: /home/pharos/ros/workspace
        Source space: /home/pharos/ros/workspace/src
        Build space: /home/pharos/ros/workspace/build
        Devel space: /home/pharos/ros/workspace/devel
        Install space: /home/pharos/ros/workspace/install
        ####
        #### Running command: "make cmake_check_build_system" in "/home/
pharos/ros/workspace/build"
        ####
        -- Using CATKIN_DEVEL_PREFIX: /home/pharos/ros/workspace/devel
        -- Using CMAKE_PREFIX_PATH: /home/pharos/ros/workspace/devel;/opt/
ros/groovy
        -- This workspace overlays: /home/pharos/ros/workspace/devel;/opt/ros/
groovy
        -- Using Debian Python package layout
        -- Using CATKIN_ENABLE_TESTING: ON
        -- Call enable_testing()
        -- Using CATKIN_TEST_RESULTS_DIR: /home/pharos/ros/workspace/build
/test_results
        -- Found gtest sources under '/usr/src/gtest': gtests will be built
        -- catkin 0.5.77
        Unloading installation code ...
        -- BUILD_SHARED_LIBS is on
        -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        -- ~~  traversing 1 packages in topological order:
        -- ~~  - donatello
        -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        -- +++ processing catkin package: 'donatello'
        -- ==> add_subdirectory(donatello)
        -- donatello: 0 messages, 1 services
        -- Configuring done
        -- Generating done
        -- Build files have been written to: /home/pharos/ros/workspace/build
        ####
```

```
#### Running command: "make -j1 -l1" in "/home/pharos/ros/workspace/
build"
####
[ 25%] Generating Python code from SRV donatello/Switch
[ 50%] Generating Python srv __init__.py for donatello
[ 50%] Built target donatello_generate_messages_py
[ 75%] Generating C++ code from donatello/Switch.srv
[ 75%] Built target donatello_generate_messages_cpp
[100%] Generating Lisp code from donatello/Switch.srv
[100%] Built target donatello_generate_messages_lisp
[100%] Built target donatello_generate_messages
```

6.2 Executing Donatello

Donatello is a package with some uses for the well known ROS example of the turtle. It implements different algorithms for managing automatically a turtle, and allow us to change that algorithm for each managed turtle.

Since this example is quite complex, we will use it and analyse code alternatively several times. Each one of the implemented algorithms will serve us as entry point for a new knowledge.

Lets start then. In order to have the minimum code running for the example you need to run in different consoles each of the next lines:

Starting up ROS

```
pharos@PhaROS:~$ roscore
```

With the same output as before:

Starting up ROS - Output

```
... logging to /home/pharos/.ros/log/a3e8bd7e-8ea0-11e3-bfbd-0800275
a8d1d/roslaunch-PhaROS-12822.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:39207/
ros_comm version 1.9.47


SUMMARY
========
```

```
PARAMETERS
 * /rosdistro
 * /rosversion

NODES

auto-starting new master
process[master]: started with pid [12836]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to a3e8bd7e-8ea0-11e3-bfbd-0800275a8d1d
process[rosout-1]: started with pid [12849]
started core service [/rosout]
```

Starting up TurtleSIM

```
pharos@PhaROS:~$ rosrun turtlesim turtlesim_node
```

The TurtleSIM is a node that let us spawn turtles, as manageable agents that have both sensors and actuators on board. Each simulated turtle is related with an exclusive set of topics for managing movement and knowing information about the turtle (we will see better this when we analyse the code).

As output of the TurtleSIM startup we have the information of the turtle that is spawned by default.

Starting up TurtleSIM - Output

```
[ INFO] [1392390503.860630887]: Starting turtlesim with node name /turtlesim
[ INFO] [1392390503.865011573]: Spawning turtle [turtle1] at x=[5.544445], y
    =[5.544445], theta=[0.000000]
```

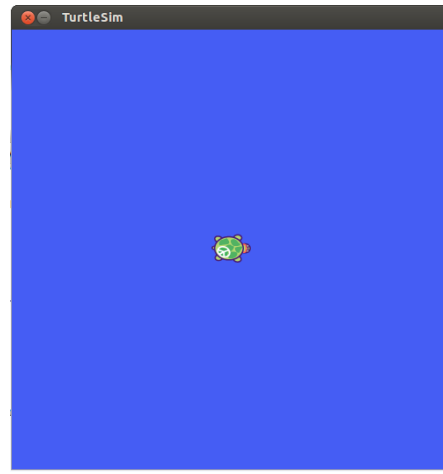
This run command, also will spawn immediately one screen like the following one.

This is the simulator. On this screen is where we will see the effect of each control algorithm.

Having the TurtleSIM running, now we just need to have the controller node from Donatello package running.

Starting up Donatello/AlgorithmSwitcher

```
pharos@PhaROS:~$ rosrun donatello headless algorithm_switcher
```



TurtleSIM Screen

Starting up Donatello/AlgorithmSwitcher - Output

```
Starting up default turtle(turtle1) handler...
Starting up switch service...
Done.
```

Taking advantage of one of the several debugging tools from ROS, we will see the node graph just to understand the current processing layout.

Analysing graph topology

```
pharos@PhaROS:~$ rqt_graph
```

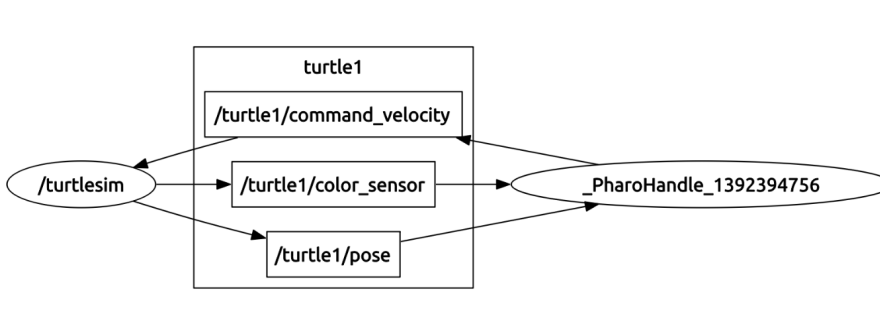


Figure 6.1: TurtleSIM - AlgorithmSwitcher Graph

The AlgorithmSwitcher node is the one called _Pharo_Handle_1392394756, since we did not specified a node name,

this one is generated. (We will learn how to do this later)

We can see, then, that we have two nodes (processes). The one that represents a turtle expose sensor information (colour and pose), and consume commands of velocity. The other one, that controls, read the information from the sensors and give directives.

But well, if we go back to the TurtleSIM screen, we will realise that there is not change. What means that there is no really a flow of movement directives.

In order to do this, AlgorithmSwitcher ask us for set an algorithm that manage this directives. In order to achieve this, we will need to execute a new command. A command that sends a petition of change to the AlgorithmSwitcher. This command will prompt us for a turtle name and an algorithm name.

Starting up Donatello/SwitchRequest

```
pharos@PhaROS:~$ rosrun donatello headless switch_request
```

Starting up Donatello/SwitchRequest - Interactive Output

```
Please, write the name of turtle you want to switch the algorithm. (The default
one is called turtle1)
turtle1

Please, write the name of one of the following algorithms  #('random' 'quiet' '
pursuiter' 'circular' 'pharo')
random

Done.
```

As we have been informed before, the turtle that is spawned by default is named turtle1. As response for the first prompting the command asked for, we wrote down **turtle1**. Then, as algorithm we choose **random**

As we can think, and actually see in the TurtleSIM screen, random means a random behaviour. And that is what we have.

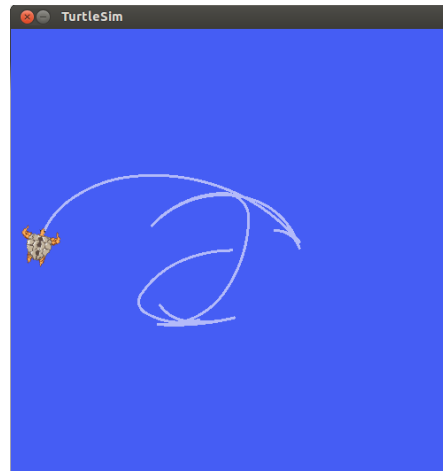
If we repeat the command with the same turtle and a different algorithm name

Starting up Donatello/SwitchRequest

```
pharos@PhaROS:~$ rosrun donatello headless switch_request
```

Starting up Donatello/SwitchRequest - Interactive Output

```
Please, write the name of turtle you want to switch the algorithm. (The default
one is called turtle1)
```



Random TurtleSIM

```
turtle1

Please, write the name of one of the following algorithms  #('random' 'quiet' '
pursuiter' 'circular' 'pharo')
circular

Done.
```

We will see how the turtle change completely the current behaviour for a new one.

So, the behaviours of the Turtle simulator are dynamic. They can change as our wishes.

How does it happens? Why the execution of a bizarre command make so big changes in a node? This doubts have their answers, all in the code.

6.3 Inspecting Donatello nodes

Inspecting AlgorithmSwitcher

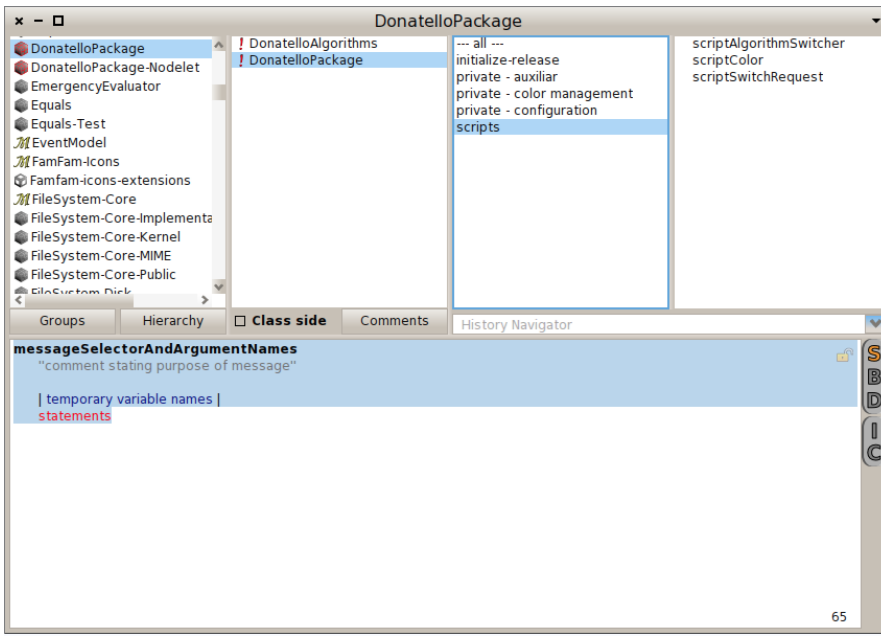
So, let's browse it!

Editing

```
pharos@PhaROS:~$ rosrun donatello edit
```

Once in the Pharo IDE, as show Figure 5.4, press control+enter for accessing spotlight. Type DonatelloPackage and enter.

As in the Chatter package, we will go before any other place to the script category.



Donatello package

We can see here as scripts two node definitions. #scriptAlgorithmSwitcher and #scriptSwitchRequest. Let analyse first the what does #scriptAlgorithmSwitcher:

```
DonatelloPackage>>#scriptAlgorithmSwitcher

    self log: 'Starting up default turtle(turtle1) handler... '.
    self registerDefaultHandler.
    self log: 'Starting up switch service... '.
    self controller node serve: [ :req :rsp |
        self log: ('{1} will behave as {2}' format: {req turtleID . req command} ).
        (self getOrCreateHandleFor: req turtleID)  algorithm: (DonatelloAlgorithms new
         algorithmFor: req command).
    ] at: '/donatello/switch' typedAs:'donatello/Switch'.
    self log: 'Done.'.
```

Ok, it looks like there are some new concepts here. We were up-to-date with publishing and subscribing topics, but, this is not a topic, or is a really weird way to write them (breath quietly and relaxed, this is not a topic). This is how we write a service in PhaROS.

Services.

This code is quite more complex than what it appear to be. (If we remember the graph of the Figure 6.1, we should remember that there are a lot of connections that are not shown in this piece of code). So let's clean up of logs ...

```
DonatelloPackage>>#scriptAlgorithmSwitcher

    self registerDefaultHandler.
    self controller node serve: [ :req :rsp |
        (self getOrCreateHandleFor: req turtleID)  algorithm: (DonatelloAlgorithms new
        algorithmFor: req command).
    ] at: '/donatello/switch' typedAs:'donatello/Switch'.
```

...and breakdown this code in pieces. (There is a design view of this package in the first appendix Appendix A).

```
scriptAlgorithmSwitcher

    self registerDefaultHandler.
```

This first line registers a handler object for the default Turtle (named turtle1). Since is a turtle spawned by default, the cycle for having it working, is different. Then, in order to have it working, it needs a particular way to be configured.

```
        (self getOrCreateHandleFor: req turtleID)  algorithm: (DonatelloAlgorithms new
        algorithmFor: req command).
    ] at: '/donatello/switch' typedAs:'donatello/Switch'.
```

This part is more complex, so i will split it in a snippet of service definition, and in the inner code.

In order to set up a service we need to give a block that will act as a callback, and it will receive two parameters: request and response. This callback is responsible for filling up the content of the response object, and it should rely on the request information.

In order to understand better, we need to browse a service type. A service type has two parts in it definition: the request and the response. Let's analyse our donatello/Switch type, to understand what is happening here.

Browsing service types

```
pharos@PhaROS:~$ rossrv show donatello/Switch
```

donatello/Switch definition

```
string random='random'
string quiet='quiet'
string pharo='pharo'
string pursuiter='pursuiter'
string circular='circular'
string command
string turtleID
---
# empty
```

The first part of the listing (until the - - - splitter), is the definition of the request type. Is an object that has two fields and several constants for making easy to choose the command value from code. Before that is extended the response type. In this example, as we can see, there is no response definition. What means that the response is not important, and that the service will give not feedback.

So, lets go again to the code we were analysing

```
    (self getOrCreateHandleFor: req turtleID)  algorithm: (DonatelloAlgorithms new
    algorithmFor: req command).
] at: '/donatello/switch' typedAs:'donatello/Switch'.
```

Now we know that the request object has the following information #turtleID and #command. We know also that there is not information needed by the response.

So, we can rewrite, in terms of making easy the example like

```
(self getOrCreateHandleFor: turtleID)  algorithm: (DonatelloAlgorithms new
    algorithmFor: anAlgorithmName).
```

Quite easy. It get the related handler for a given turtle name (or create it if needed). Then, it choose an algorithm by name and set it to the handle.

Ok, until here is a very simple method, and that is the reason why we can make our self several questions about the domain like

- What is a handle?
- Why when i create one handle it appears a new turtle in the screen?
- What is an 'algorithm'? Why it can be set up?

And also some nasty questions about the lifetime of a package, like

- The Switch type belongs to our package?
- How do i define a type?

In order to close the example domain first, we will browse the involved methods

```
turtles at: 'turtle1' put: self nodelets turtlesim defaultTurtleHandler.
```

Ok, against what we supposed, this code does not solve many doubts, but generate others instead. In this line we access a local dictionary of 'turtles' (handles) and set as 'turtle1' (the name of the default turtle) the return value of

```

```

What the hell does it means?

First approach to the Nodelets

We will say that a nodelet is a piece of code that is reusable, (not like a package and a script/node, which are quite coupled with the behaviour we want to implement). A Nodelet is an instance of a subclass of the PhaROSNodelet class.

A Nodelet has a defined lifecycle and it can be injected into a Nodelet container inside our default controller.

If we browse the #buildController method in our DonatelloPackage we will find

```
^ self nodeletInjectionExample: super buildController.

DonatelloPackage>>#nodeletInjectionExample: aController
```

```
  aController nodelets use: TurtlesimNodelet as:#turtlesim.
  aController nodelets turtlesim isKindOf: TurtlesimNodelet.
  ^ aController.
```

The #nodeletInjectionExample: method is a method that set up a configuration for our package. In our example, the only requirement is the TurtlesimNodelet. And we define it to be accessible under the name of #turtlesim.

So, after this, we know that

```

```

will give us something related with TurtlesimNodelet class. For now, in order to not go into deep implementation details, i ask you to believe me, that the response is an instance of TurtlesimNodelet, also that this instance is always the same, and that it is stored in the nodelet container.

Finally i ask you also to believe that inside a package

```

```

will give us a nodelet container.

Going back to the code analysis, now we know where go to find the meaning of

```

```

We need to browse #defaultTurtleHandler in the class TurtlesimNodelet.

```
  ^ self turtleHandleFor: 'turtle1'.

TurtlesimNodelet>>#turtleHandleFor: aTurtleName
  ^ TurtleHandle new initializeWith: aTurtleName and: self; yourself.
```

Then, here we find some easy code. The default handler method ask for a handler for a turtle named 'turtle1' (that we know already that is the name of the default turtle). And then, a turtle handle is an instance of TurtleHandle, initialised with a name, and a TurtlesimNodelet.

Finally we reach what is it. So, lets peek the #initializeWith:and: at the TurtleHandle class.

```
TurtleHandle>>#initializeWith: aName and: aTurtlesimNodelet
  name:= aName.
  turtlesimNodelet := aTurtlesimNodelet.
```

```
velocityOut := turtlesimNodelet rosnode topicPublisher: '/', aName, '/
    command_velocity' typedAs: 'turtlesim/Velocity'.

(turtlesimNodelet rosnode buildConnectionFor: '/',aName,'/color_sensor')
        typedAs: 'turtlesim/Color';
        for: [ : aColor | self currentColor: aColor ];
        connect.

(turtlesimNodelet rosnode buildConnectionFor: '/',aName,'/pose')
        typedAs: 'turtlesim/Pose';
        for: [ : aPose | self currentPose: aPose ];
        connect.
```

Even when the code is longer, and at first view it mixes several meanings, we should feel better, because now we are watching something that is already familiar.

Yes, actually this is just configuration of connections with ros. Since we already made a detailed analysis of this kind of code in the Chatter example at section 5.2, we will just make a quick understanding.

```
TurtleHandle>>#initializeWith: aName and: aTurtlesimNodelet
   name:= aName.
   turtlesimNodelet := aTurtlesimNodelet.

   velocityOut := turtlesimNodelet rosnode topicPublisher: '/', aName, '/
      command_velocity' typedAs: 'turtlesim/Velocity'.
```

First it stores references to the name and nodelet. Also register a publisher for /{turtle-name}/command-velocity. Typed as turtlesim/Velocity. This publisher will let us print a velocity to the turtle we are handling.

```
        typedAs: 'turtlesim/Color';
        for: [ : aColor | self currentColor: aColor ];
        connect.
```

Then it subscribes to /{turtle-name}/color-sensor, with a setter as callback. This means that our handler have a reference to the last detected colour .

```
        typedAs: 'turtlesim/Pose';
        for: [ : aPose | self currentPose: aPose ];
        connect.
```

Finally, it subscribes to /{turtle-name}/pose, with an other setter as callback. What again means that our handler have a reference to the last detected pose .

So, as we supposed, #defaultTurtleHandler returns a handle object configured to listen the sensors and control the movements of the default turtle.

We can go back to analyse in deep the rest of the Algorithm Switcher script node.

```
DonatelloPackage>>#scriptAlgorithmSwitcher

  self registerDefaultHandler.
  self controller node serve: [ :req :rsp |
     (self getOrCreateHandleFor: req turtleID)  algorithm: (DonatelloAlgorithms new
      algorithmFor: req command).
  ] at: '/donatello/switch' typedAs:'donatello/Switch'.
```

We already understand what happens with #registerDefaultHandler. We can go now for #getOrCreateHandleFor:.

```
  ^ turtles at: aTurtleID ifAbsentPut: [ self nodelets turtlesim spawnTurtle: aTurtleID ].
```

As well as #registerDefaultHandler, it access the handle dictionary. But if there is not a handle, does not ask for a handle but for spawning a turtle. This is because if the turtle is not in the dictionary is because it does not exist, and it needs to be created.

```
  ^ self spawnTurtle: aName at: 10@10.


TurtlesimNodelet>>#spawnTurtle: aName at: aPoint
  (self controller node service: '/spawn' ) call: [ : rqst |
     rqst x: aPoint x.
     rqst y: aPoint y.
     rqst name: aName
  ] .
  ^ self turtleHandleFor: aName.
```

In order to create new turtles inside the TurtleSim, the Turtle simulator expose a service called /spawn. This service is typed as turtlesim/Spawn .

We can use rossrv to check the definition

Browsing service types

```
pharos@PhaROS:~$ rossrv show turtlesim/Spawn
```

turtlesim/Spawn definition

```
float32 x
float32 y
float32 theta
string name
---
string name
```

The spawn serve then takes as request type the initial pose of the turtle to create (x, y, theta) and the name of it. And its response the name of the turtle.

Let's analyse the service call as a snippet.

```
response := (self controller node service: '/service/name' ) call: [ : request |
    "make up the request"
] .
```

In order to call a service, we need then to ask for the service object, looking up through the built in node. Since the services are supposed to have unique name, we don't really need to check the type for avoiding name collision. So there is not need of specify. A service understands #call:,method that receives a block for making up a request. The returning of this message is the response of the service request.

Going back to the last code that we were analysing

```
(self controller node service: '/spawn' ) call: [ : rqst |
    rqst x: aPoint x.
    rqst y: aPoint y.
    rqst name: aName
] .
^ self turtleHandleFor: aName.
```

We now know that #spawnTurtle:at, calls /spawn service, and returns a handle for the new turtle.

So going back to the script for Algorithm Switcher, we know that #getOrCreateHandleFor: turtleName, get or create handles for turtles, where create a handle also means to create the new turtle.

In our analysed code

After resolving the handler, we set an algorithm, fetched by name from a DonatelloAlgorithms object.

Let's check then how does it look an algorithm, specially the one we just used: 'random'

```
  ^ algorithms at: aCommand asLowercase  ifAbsent: [
     self idleHandler.
   ]

DonatelloAlgorithms>>#initialize
  algorithms := Dictionary new.

  algorithms at: 'random' put:self randomHandler.
  algorithms at: 'quiet' put:self idleHandler.
  algorithms at: 'circular' put:self circularHandler.
  algorithms at: 'pharo' put:self pharoHandler.
  algorithms at: 'pursuiter' put:self pursuiterHandler.
```

Since the resolver of an algorithm based it self in a dictionary, i browsed also the initialise method of the same object, and we can find a configuration that put 'names' to different 'algorithms'.

For the random algorithm then we just need to browse #randomHandler and check what it returns.

```
  ^ [ : handle |
     | linear angular random sign|

     random := (Random seed: DateAndTime now asUnixTime).
     sign := 1.

     [ true ] whileTrue: [
        (random nextInt: 20) even ifTrue: [
           random := (Random seed: DateAndTime now asUnixTime).
        ] ifFalse: [
           sign := sign * -1.
         ].


        linear := (random next; next; nextInt:20 )+((random nextInt:9)) / 10.
        angular := (random next;next; next; nextInt:9) + (random  next; next; nextInt:9) /
      10.
```

```
        linear := linear * sign.
        angular := angular * sign * -1.

        handle moveAt: linear and: angular.
        ( (Random seed: DateAndTime now asUnixTime) next; nextInt: 3) seconds
      asDelay wait.
     ].
  ].
```

Well, then we know what is an algorithm now. At least one example of it. Is a block that receives a handler by parameter, and that it defines a behaviour.

I let you the detailed analysis of this code, is quite easy. In general terms it randomise a linear amount of velocity and an angular amount of velocity, it print this velocity in the handle sending #moveAt:and: and wait a randomised amount of seconds between 1 and 3 for the next iteration.

We can see also that all the behavioural code is inside an infinite loop.

Finally, we need to check what happens when we assign this block to the turtle handle through #algorithm: message.

```
processHandler ifNotNil:[
   processHandler finalize.
].

 processHandler := [aBlock cull: self cull: turtlesimNodelet ] shootIt asStickyReference.
```

Well, this setter is a bit more complex as a normal one, but some how, it was expected, if we remember that this code is executed by a service call and that the algorithm block has an infinite loop inside.

This setter, first check if there is a previous algorithm running, if there is, it finalise it. Then configures a block that executes the given block with two optional parameters, the handle (as we realise analysing the random algorithm), and the nodelet. It executes the configured block with a #shootIt message, that means that it will be executed in a different thread, and that returns a future. Finally, it sends asStickyReference to that future, what means that if the processHandler variable change it reference by other one (like by example nil, or other future), it will stop the related process. (For more information about this, check the TaskIT reference).

Making a quick brief

AlgorithmSwitcher setup a handle for the default turtle, without any algorithm set. Then it setup a service, that receives as request a turtle name

and a algorithm name. When this service is called it check if it has a handler, if there i no handler for the cited turtle, it spawn a turtle with that name and creates a handler for it. Whatever if the handle exists or is created, it sets the algorithm that is in the callback. If the algorithm does not exist, it installs a default algorithm.

This means that if we try the example again executing the Switch request command with other turtle name, we should have other turtle in the TurtleSim screen! Lets try it!

6.4 Executing switch request with several turtle names

For trying our new example, we need to startup all the system again.

Starting up ROS

```
pharos@PhaROS:~$ roscore
```

With the same output as before:

Starting up ROS - Output

```
... logging to /home/pharos/.ros/log/a3e8bd7e-8ea0-11e3-bfbd-0800275
a8d1d/roslaunch-PhaROS-12822.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:39207/
ros_comm version 1.9.47


SUMMARY
========

PARAMETERS
 * /rosdistro
 * /rosversion

NODES

auto-starting new master
process[master]: started with pid [12836]
ROS_MASTER_URI=http://localhost:11311/
```

```
setting /run_id to a3e8bd7e-8ea0-11e3-bfbd-0800275a8d1d
process[rosout-1]: started with pid [12849]
started core service [/rosout]
```

Starting up TurtleSIM

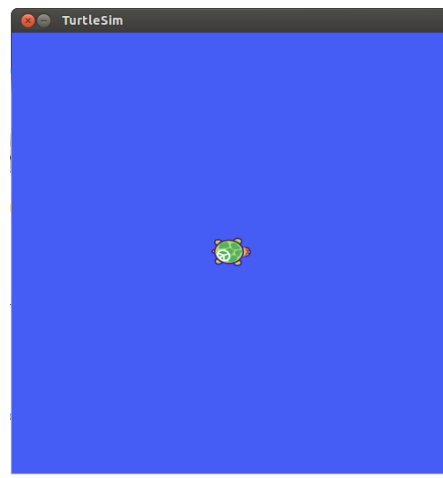
```
pharos@PhaROS:~$ rosrun turtlesim turtlesim_node
```

As output of the TurtleSIM startup we have the information of the turtle that is spawned by default.

Starting up TurtleSIM - Output

```
[ INFO] [1392390503.860630887]: Starting turtlesim with node name /turtlesim
[ INFO] [1392390503.865011573]: Spawning turtle [turtle1] at x=[5.544445], y
    =[5.544445], theta=[0.000000]
```

This run command, also will spawn immediately one screen like the following one.



TurtleSIM Screen

Starting up Donatello/AlgorithmSwitcher

```
pharos@PhaROS:~$ rosrun donatello headless algorithm_switcher
```

Starting up Donatello/AlgorithmSwitcher - Output

```
Starting up default turtle(turtle1) handler...
Starting up switch service...
Done.
```

So now we will execute a different the same command as in the previous example, but with turtle2 as turtle name.

Starting up Donatello/SwitchRequest

```
pharos@PhaROS:~$ rosrun donatello headless switch_request
```

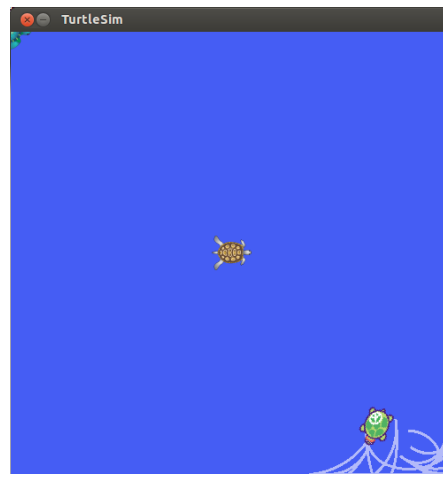
Starting up Donatello/SwitchRequest - Interactive Output

```
Please, write the name of turtle you want to switch the algorithm. (The default
one is called turtle1)
turtle2

Please, write the name of one of the following algorithms  #('random' 'quiet' '
pursuiter' 'circular' 'pharo')
random

Done.
```

Let's go to the screen to see what happens.



TurtleSIM Screen

Taking in care the code analysis, at this point, the AlgorithmSwitcher node, should have two handles: turtle1 and turtle2. Each of them connected with the respective turtles.

Since handle objects are invisible for ROS, lets check what have to show the rqt-graph command

Analysing graph topology

```
pharos@PhaROS:~$ rqt_graph
```

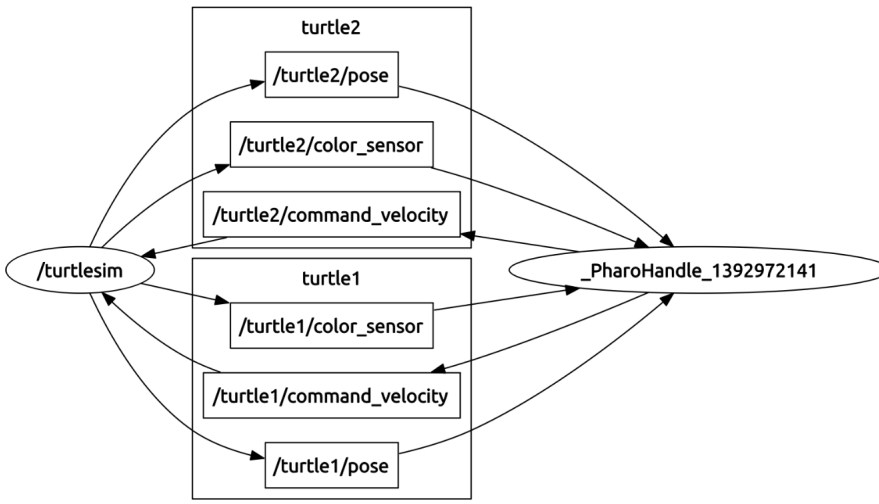


Figure 6.2: TurtleSIM - AlgorithmSwitcher Graph

We can see how the turtle1 and the turtle2 are both connected to our node, just as expected.

Now, if we execute again the command and we send and algorithm for the default turtle, we should have the two turtles moving

Starting up Donatello/SwitchRequest

```
pharos@PhaROS:~$ rosrun donatello headless switch_request
```

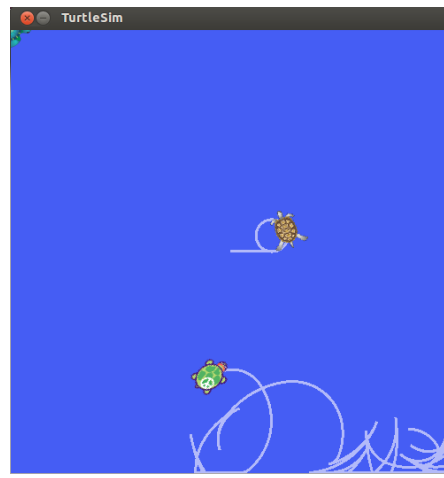
Starting up Donatello/SwitchRequest - Interactive Output

```
Please, write the name of turtle you want to switch the algorithm. (The default
one is called turtle1)
turtle1

Please, write the name of one of the following algorithms  #('random' 'quiet' '
pursuiter' 'circular' 'pharo')
circular

Done.
```

Let's go to the screen to see what happens.



TurtleSIM Screen

I leave to you the inspection of the other node (the one called SwitchRequest), because it has nothing new to care about. So you are able to do it your self.

6.5 What we learned

We learned several things with this example. (and we will use it again for other two concepts)

From the console

- rossrv show serviceType/Name
  It shows the definition of a type related with a service

From a PhaROS package

- self controller node serve: [:rqt :rsp | "Executes something with request and fills up the response"] at: '/service/name' typedAs:'service/Type'.
  Register the given block as a service accessible as '/service/name' typed as 'service/Type'
- response := (self controller node service: '/service/name') call: [: request | "make up the request"] .
  Calls the service served at '/service/name' configuring the request with the given block.
- self nodelets use: Nodelet as: #name
  Registers a nodelet to be accessed as #name. (self nodelets name).

Conceptually

- Packages are for group scripts
- Packages define a set of behaviours related by domain.
- Nodelets define reusable pieces of robotic software.
- Is cool to design our own domain abstractions

Chapter 7

Types: Topics and services

Answering the unanswered questions

During the analysis of the Donatello example there were some questions that appeared but we did not respond during the analysis. This questions are:

- The Switch type belongs to our package?
- How do i define a type?

This questions a fair questions, but need to answer an other question that it maybe appeared maybe not, but is a bit more abstract. What is a type?

7.1 ROS Types

In the first time that it appeared a type i just told 'this is the kind of data that the topic will send'

And it is indeed. A type is an structure that defines which information will be sent and received, and also it defines how it will be marshalled and unmarshalled to travel in the ROS channels.

Since is a structure for data sending, and is some how shared with all the rest of ROS nodes that are interested in, this types are not more than a **structure**. This means that this types have no related behaviour.

In a ROS system we have three kind of types.

- Basic types (which can be used **only** for defining the other two kinds of types)

  - int8, int16, int32, int64, uint8, uint16, uint32, uint64
  - float32, float64
  - string
  - boolean
  - time, duration
  - array and fixed array ([] [N])
- Topic types (Or message type)
- Service types

The first classifications are the basic tool that we have for defining topic and services types. This are all the types defined also in the protocol of serialisation. Topic and service types just have a way to deal with the structure, but in the end is the kind of inlining of all the basic types.

Then the main difference in between a service type and a topic type is that the service one has two parts (as we saw during the previous analysis): request and response.

7.2 Topic or Message type

In ROS world, Topic types are defined in a .msg file in the msg folder of the respective package.

A common Message type is called std-msgs/Header. If we execute

Browsing Topic types

```
pharos@PhaROS:~$ rosmsg show std_msgs/Header
```

std-msgs/Header definition

```
uint32 seq
time stamp
string frame_id
```

If we want to find the type definition, we just need to execute

Browsing Topic types

```
pharos@PhaROS:~$ roscd std_msgs/msg
pharos@PhaROS:~$ ls -l Header.msg
-rw-r--r-- 1 root root 500 Sep  6 00:42 Header.msg
```

But this is just the definition of the type, is not the type it self. The life time of a ROS package involves c/c++/python code generation, and compilation as well. We will no go deep in this topic, which is extended in the ROS reference.[1]

7.3 Service type

As we saw before, a service type is browsed with a similar command as rosmsg, which is called rossrv. Lets browse the only standard type for services std-srvs/Empty

Browsing service types

```
pharos@PhaROS:~$ rossrv show std_srvs/Empty
```

std-srvs/Empty definition

```
---
```

Ok is not very cool definition, but is the only one i can be sure that is there to be browsed. The service type definitions are stored in a folder named srv, in the related package, as a .srv file. So, we can do

Browsing Service types

```
pharos@PhaROS:~$ roscd std_srvs/srv
pharos@PhaROS:~$ ls -l Empty.srv
-rw-r--r-- 1 root root 3 Oct  4 10:13 Empty.srv
```

Just as the topic type definition, the service types are also generated and compiled, and pretty well explained in the ROS Reference.

Automatising types with Pharos installation tooll

If you want to make things manually, you should go to the cites pointed before. And take in care the fact that the Pharo code life cycle is not cool dealing with files.

In order to make easy our lives in several aspects, Pharos installation tool provides a way to automatise all the lifecycle of both topic and service types. Including dependency analysis, and ros files generation (makefiles, xml, and .msg/.srv).

[1] **?**, .

Is quite important remember that in order to have this feature available, you **must** install/create packages with Pharos installation tool.

Lets go back to donatello then. And go to the package side class, to browse the method #types.

This method returns all the types that this package define.

```
^ {
    self switchCommandServiceType.
 }
```

```
^
    ROSType service named: #Switch package: #donatello request:{
        String constant:#random value: 'random'.
        String constant:#quiet value: 'quiet'.
        String constant:#pharo value: 'pharo'.
        String constant:#pursuiter value: 'pursuiter'.
        String constant:#circular value: 'circular'.
        String named: #command.
        String named: #turtleID.
    } response: {
    }.
```

If we compare this code with the definition printed by rossrv

donatello/Switch definition

```
string random='random'
string quiet='quiet'
string pharo='pharo'
string pursuiter='pursuiter'
string circular='circular'
string command
string turtleID
---
# empty
```

We can see that is almost the same.

Generalising in a snippet of code, we know now that for defining a service type we need to write:

```
        {Type} constant:#constantName value: constantValue.
        {OtherType} named: #variableName
    } response: {
```

```
    {Type} constant:#constantName value: constantValue.
    {OtherType} named: #variableName
}.
```

The valid types to use inside a type definition are:

```
Uint8, Uint16, Uint32, Uint64
String, Time, Duration
Float32. Float64
FixedArray, Array
ROSType
```

For the cases of FixedArray and Array, that are complex types (types that are typed inside), the way to define a field is a bit different:

```
    FixedArray named: #name sized: 20 ofType: Int8 .
} response: {
    Array named: #name ofType: String
}.
```

Finally if you want to use a type defined in ROS, you should use ROSType. That is used like

```
    ROSType definedBy: 'std_msgs/String' named: #string .
} response: {
    ROSType definedBy: 'std_msgs/Header' named: #header .
}.
```

Note that FixedArray, Array and ROSType does no have constant example. This is because is not supported by ROS.

If the type you are looking to create is not a service type, but a topic one, the code that you need to write is quite similar.

```
    {Type} constant:#constantName value: constantValue.
    {OtherType} named: #variableName
}
```

The only difference is that it has not two parts definition but just one. The types you can use inside the definition are the same as for the service: all the basic types and any of the message types defined in ROS. (Service type are not allowed for definition)

During the life cycle of the package, only when is installed/created by

Pharos installation tool, each time you change the type method, all your types will be generated. They will be generated also during the installation.

Other thing that it will be generated are the scripts. Every method which it selector starts with the word 'script' inside the package related with the image installation, it will have related a file with the same name (without the script part) with the ROS convention, available the entry point to execute each script with rosrun command.

7.4 What we learned

From console

- rosmsg show typeId
  Shows the definition of a message/topic type
- roscd package/msg
  Lead us to the folder where the message types are defined
- roscd package/srv
  Lead us to the folder where the service types are defined
- pharos install package / pharos create package
  Both commands leave residual resident code for managing the life cycle of our package.

From image

- For defining a topic/message type i need to write

```
    {Type} constant:#constantName value: constantValue.
    {OtherType} named: #variableName
}
```

- For defining a service type i need to write

```
    {Type} constant:#constantName value: constantValue.
    {OtherType} named: #variableName
} response: {
    {Type} constant:#constantName value: constantValue.
    {OtherType} named: #variableName
}.
```

- For letting pharos resident code to manage my types i need to define them in #types class method of my package.

Conceptually, since PhaROS Framework don't need to run inside an environment with ROS installed:

- PhaROS Framework is not responsible for generating scripts
- PhaROS Framework is not responsible for generating types
- PhaROS Framework is not responsible for generating package.xml and CMakeLists.txt

By other hand, Pharos installation tool provides a way to install a package into a catkin workspace and deal with the implications of being in a ROS installation:

- Pharos installation tool will generate all your scripts
- Pharos installation tool generate all the specified types
- Pharos installation tool generate both package.xml and CMakeLists.txt

Chapter 8

Parameters

Donatello example has other nodes but the ones we have used. It have also some algorithms that we will analyse in the future (Because of it complexity)

Our new example then is already there ready to be executed (If you need to check how to install Donatello example, please go to section 6.1)

8.1 The Color node example

We will do as in the previous example, a first instantiations of the basics of the example:

Starting up ROS

```
pharos@PhaROS:~$ roscore
```

With the same output as before:

Starting up ROS - Output

```
... logging to /home/pharos/.ros/log/a3e8bd7e-8ea0-11e3-bfbd-0800275
a8d1d/roslaunch-PhaROS-12822.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:39207/
ros_comm version 1.9.47


SUMMARY
```

```
========

PARAMETERS
 * /rosdistro
 * /rosversion

NODES

auto-starting new master
process[master]: started with pid [12836]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to a3e8bd7e-8ea0-11e3-bfbd-0800275a8d1d
process[rosout-1]: started with pid [12849]
started core service [/rosout]
```

Starting up TurtleSIM

```
pharos@PhaROS:~$ rosrun turtlesim turtlesim_node
```

The TurtleSIM is a node that let us spawn turtles, as manageable agents that have both sensors and actuators on board. Each simulated turtle is related with an exclusive set of topics for managing movement and knowing information about the turtle (we will see better this when we analyse the code).

As output of the TurtleSIM startup we have the information of the turtle that is spawned by default.

Starting up TurtleSIM - Output

```
[ INFO] [1392390503.860630887]: Starting turtlesim with node name /turtlesim
[ INFO] [1392390503.865011573]: Spawning turtle [turtle1] at x=[5.544445], y
    =[5.544445], theta=[0.000000]
```

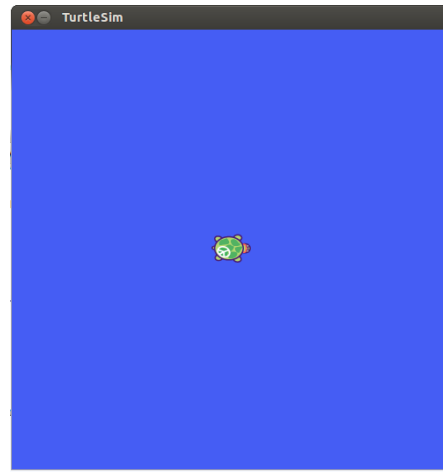
This run command, also will spawn immediately one screen like the following one.

This is the simulator. On this screen is where we will see the effect of each control algorithm.

Having the TurtleSIM running, now we just need to have the controller node from Donatello package running.

Starting up Donatello/AlgorithmSwitcher

```
pharos@PhaROS:~$ rosrun donatello headless algorithm_switcher
```



TurtleSIM Screen

Starting up Donatello/AlgorithmSwitcher - Output

```
Starting up default turtle(turtle1) handler...
Starting up switch service...
Done.
```

Starting up Donatello/SwitchRequest

```
pharos@PhaROS:~$ rosrun donatello headless switch_request
```

Starting up Donatello/SwitchRequest - Interactive Output

```
Please, write the name of turtle you want to switch the algorithm. (The default
one is called turtle1)
turtle1

Please, write the name of one of the following algorithms  #('random' 'quiet' '
pursuiter' 'circular' 'pharo')
random

Done.
```

Now our system is about a TurtleSIM that has related a turtle handle for the default turtle, and is moving it randomly inside it cage.

Now we will add then our new node to the system, which is named color.

Starting up Donatello/Color

```
pharos@PhaROS:~$ rosrun donatello headless color
```

Starting up Donatello/Color - Output

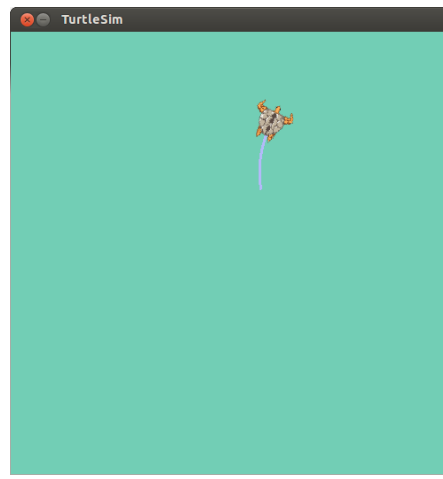
```
 Running Donatello color node at ROS 1.9.50 (groovy)
    Donatello color node is a node that change the background color of the
running TurtleSIM node with the behaviour of a turlte
    Which turtle you want to define the background color? — Remember default
turtle's name is turtle1

    turtle1

    Color node running at: 0.5 hz
```

In the output will be prompted as well for a turtle name, in this case is to 'define the background colour'. After giving this information we are informed that the node is running at 0.5 hz.

If we go to the TurtleSIM screen we will see something like the following screenshots

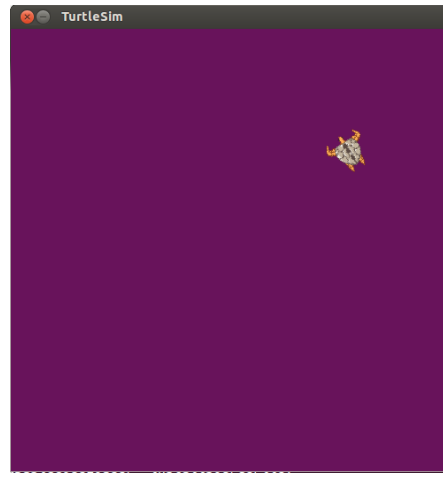


TurtleSIM Screen - Color 1

So, now the background colour of the TurtleSIM it seems to change according to the movement of the turtle (Is easy to realise that there is not change in the colour meanwhile the turtle is static)

Somehow, this new node is getting information from the given turtle and changing inner information in the TurtleSIM.

As usual, we have no much more things to do than speculate or browse the code :).



TurtleSIM Screen - Color 2

8.2 Inspecting the Color node from Donatello package

Editing

```
pharos@PhaROS:~$ rosrun donatello edit
```

Once in the Pharo IDE, as show Figure 5.4, press control+enter for accessing spotlight. Type DonatelloPackage and enter.

As in the Chatter package, we will go before any other place to the script category.

Here then we go to the #scriptColor method, to encounter the following code

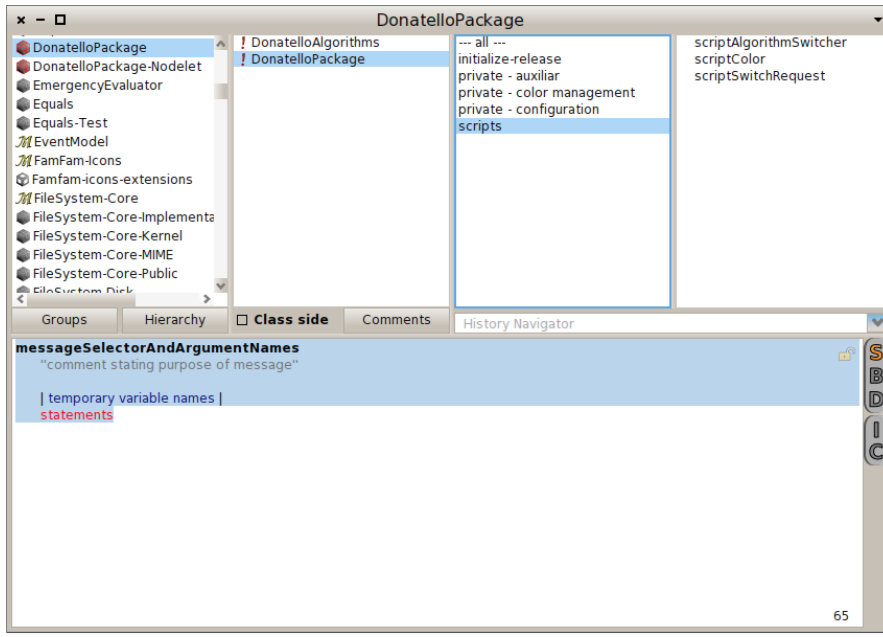
```
| version distro red green blue  rate handle |

self registerDefaultHandler.

version := self node parameter:'/rosversion' initialized:''.
distro := self node parameter: '/rosdistro' initialized:''.

self log: (' Running Donatello color node at ROS {1} ({2})' format: { version get . distro
    get }).

red :=  self nodelets turtlesim bgcRed.
green :=  self nodelets turtlesim bgcGreen.
blue :=  self nodelets turtlesim bgcBlue.
```



Donatello package

```
rate := self rate.

self log: 'Donatello color node is a node that change the background color of the
    running TurtleSIM node with the behaviour of a turtle'.
handle := self getOrCreateHandleFor: (self askUserFor: 'Which turtle you want to
    define the background color? — Remember default turtle"s name is turtle1 ').

self paralellize looping
            bindRed: red
            green: green
            blue: blue
            with: handle
            rate: rate.

self log: ('Color node running at: {1} hz' format: { rate get }).

rate onChange: [ :newRate :oldRate |
   self log: ('Changeing color rate update from{1} to {2}' format: { oldRate . newRate }).
 ].
```

Well, this method looks quite long, but, if we put out all the log and the things related just with logging, is not really long, but, in terms to follow the idea of the example (showing several ways to use parameters), let's analyse it with logs included.

```
| version distro red green blue  rate handle |

self registerDefaultHandler.
```

The first two lines are already well known, the first is the variable declaration, the second one was analysed in the previous chapter. It register a handle in a dictionary for the turtle named 'turtle1', which is the default turtle name.

```
distro := self node parameter: '/rosdistro' initialized:''.

self log: (' Running Donatello color node at ROS {1} ({2})' format: { version get . distro
    get }).
```

This lines are new. The first two lines are quite similar. This lines are the needed code to ask for a parameter.

What is a Parameter? well, a parameter, or as i call them, public parameter, is a named value that can be accessed (for overriding it or reading it) for any one that knows it name, in any moment of the life of the system. This way we can have dynamically configured nodes.

Lets analyse it as a generic snippet.

```
```

In order to get a binder object (an object that references to the value and that is always up-to-date with the system) that points a parameter, we ask to the related node for a parameter initialised with a value.

The initialisation value to define a value if the parameter is not yet defined. It also will define implicitly a type restriction (String in our example). Since we will use this values for specific mechanism and that the value comes from outside, we want to be sure that is suitable to do the work.

Then lastly we have the log line. If we check again the output of the node, the first log line is: ' Running Donatello color node at ROS 1.9.50 (groovy)'

So, we know now that '/rosversion' and '/rosdistro' are common parameters that indicates this information. And we know also that to a bind object, we can ask for 'get' in order to get its value.

Let's follow with the code.

```
red := self nodelets turtlesim bgcRed.
green := self nodelets turtlesim bgcGreen.
blue := self nodelets turtlesim bgcBlue.
```

As we already saw in the previous chapter, turtlesim is the accessor for TurtleSim nodelet in Donatello package, then, we will browse the methods related in this class.

Here we ask for bgcRed, bgcGreen and bgcBlue. BGC is a common acronym for Background colour.

```
bgcRed
^ self controller node parameter:'/background_r' initialized: 0.
bgcGreen
^ self controller node parameter:'/background_g' initialized: 0.
bgcBlue
^ self controller node parameter:'/background_b' initialized: 0.
```

Taking in care the analysis of the snippet of parameters, we can quickly say that this methods return a bind to each of the colour of an RGB configuration.

Sadly the names of this parameters are not directly attached to TurtleSIM. We encourage you a lot to put names like '/turtlesim/background-r', so is easy to know which is the node that is being feed with this information.

So, in the end of the execution of this lines, we have three binded parameters, one for each part of the RGB configuration of the background colour of the TurtleSIM.

Then we ask for the rate.

```
^ self node parameter:'/color/rate' initialized: 0.5
```

Which is other parameter. But in this case the variable (since it start with /color) it will be related with the behaviour of our node. The type of this parameter is float. So it always should be float.

```
handle := self getOrCreateHandleFor: (self askUserFor: 'Which turtle you want to
    define the background color? — Remember default turtle"s name is turtle1 ').
```

```
self paralellize looping
        bindRed: red
        green: green
        blue: blue
        with: handle
        rate: rate.
```

Then we just log some info about the node previously to prompt the user (as already saw in our previous chapter) for a turtle name, to configure an appropriate handle object.

When we have already the colour handler, the turtle handle and rate, we spawn a thread that will execute, in an infinite loop. the message

#bindRed:green:blue:with:rate:.

```
bindRed: aRedColorBind green: aGreenColorBind  blue: aBlueColorBind with:
    aTurtleHandle  rate: aRate
  aRate get hz wait.
  aRedColorBind set: ((aTurtleHandle pose position x * 100) % 255) asInteger.
  aGreenColorBind set: ((aTurtleHandle pose position y* 100) % 255) asInteger.
  aBlueColorBind set: 255 − ((aTurtleHandle pose orientation z* 100) % 255) asInteger.
  self nodelets turtlesim clear.
```

This code is quite easy. We need to remember that #get obtains the value of a binder object, and learn that #set: sets it up.

The expected value of rate, as we saw in the previous lines, is a float. This float will be used as hz rate to regulate the execution of the method.

The red, green and blue colours setting, is related with the turtle pose. And finally theres is an execution of something called #clear.

```
^  (self controller node service: '/clear') call.
```

In the specification of the turtle sim, it says that for changing the colour of the turtle you need not just to set the parameters, but also the call the clear service.

So we do it. (This is why the drawn path of the turtle it disappear suddenly, in a constant interval of time).

We probable have noticed that the call to the service was without parameters. This is because /clear service do not receive anything as request, and give no real response. In this cases so can invoke #call instead of #call:

```
rate onChange: [ :newRate :oldRate |
    self log: ('Changeing color rate update from {1}hz to {2}hz' format: { oldRate .
    newRate }).
].
```

Finally, we have a final log, that shows the configured rate for our node, and an #onChange: message send.

#onChange: method will register a callback that will be executed when there is a change in the parameter, exposing (both optional) the new and the old value of the parameter.

In this case then, if we set the parameter from outside, we should have a log in the node console, and of course, we should have a different rate of update of the background colour of the TurtleSIM.

For dealing with parameters, ROS give us a console command named 'rosparam'. This command let us list the parameters in general, and set/get a particular one.

The following command will show us the current value of a parameter

rosparam - get

```
pharos@PhaROS:~$ rosparam get /color/rate
```

In this case, as we already expected from the code analysis, the output is

rosparam - get - output

```
5.0
```

Then, for changing the value we just run

rosparam - set

```
pharos@PhaROS:~$ rosparam set /color/rate 10.0
```

This parameter has not output, but it will have an impact in the color node console.

Donatello/Color - Output

```
Changing color rate update from 0.5hz to 10.0hz
```

If we go to the TurtleSIM screen, we will see that the change of the background colour is more fluent, as we expected.

8.3 What we learned

From image side.

- parameter:= self node parameter: '/parameter/name' initialised: defaultValue
  returns a binded parameter.
- parameter set: aValidValue
  change the value of a parameter
- parameter get
  get the value of a parameter
- parameter onChange: [:new :old |"callback"]
  adds a callback that will be called when the value has changed.
- parameters names should express the ownership.
  by example '/node-name/parameter'
- service call
  calls a service with null request configuration

From console side

- rosparam list
  lists the parameters of the whole system.
- rosparam set /parameter/name value
  sets a parameter with the given value
- rosparam get /parameter/name
  gets a parameter's value

In general

- Parameters allow us to have dynamic configurations
- The get message will always give us the last registered value, what is helpful for basic values in algorithms.
- The onChange: callback registration, give us a mechanism that is more powerful, since it runs orthogonally

Chapter 9

Launch configurations

9.1 ROS Launch

9.2 PhaROS Launch

-should implement

Part III

When the boy becomes a man

At this point, we should have already the base to start to fly a bit without central examples, in order to go a bit faster in conceptual and technical meanings.

This whole part is about writing your own robotic solutions, from the first image, to the code distribution.

Chapter 10

Creating our own package

The minimum to have something running in a ROS installation is a package. In the lasts distributions, a catkin package. Since is the minimum and is the unit we choose for automations, the first thing you need to learn is how to create one.

There are actually two ways to create a package.

- Pharos installation tool
- From scratch

Since this is a text that aims to be agile, we will focus in the Pharos installation tool way. The second one, From scratch, is widely treated in the Appendix B.

10.1 Getting started with Pharos installation tool for creating new packages

To create a catkin package in a ROS installations means have a package folder in a catkin workspace with the proper distribution of things. Means to have an image installed in this package that has the implementation of this package and is suited up with the PhaROS.

In the end, it means to have something similar to the packages we had installed for the examples, something similar to the directory layout explained at paragraph 5.1

For achieving this noble goal, we have the Pharos installation tool.

Package creation

```
pharos@PhaROS:~$ pharos create −−help
```

Package creation help

```
usage: pharos create PACKAGE [OPTIONS]

Options:
−−help          Shows this text
−−location      Absolute path to a valid catkin workspace (not source folder. The
    workspace. By example /home/user/workspace ). Default value is the current
    directory.
−−silent    { true | false }. If silent is false you will be able to see the installation of the
    output image. Default value is true.
−−version       { 1.4 | 2.0 | 3.0 }. It indicates the version of pharo to download. It will
    not count if you are in a not silent session.  Default value is 2.0
−−ros−distro    Indicates the distribution of ROS that you want to use. By default is
    groovy { groovy | hydro | fuerte }
−−archetype     Indicates the name of the archetype to use for creating things. Default
    is basic−archetype.
−−author        Indicates the name of the author
−−maintainer    Indicates the name of the maintainer Default value is the indicated in
    author

−−author−email      Indicates the email  of the author
−−maintainer−email  Indicates the email of the maintainer. Default value is the
    indicated in author−email
−−description       Indicates the description of the package
−−pharo−user        Indicates the user name for the result image.
−−force−new     DELETE the package if it exists in the pointed location.
```

Then, the easiest way to go is by stepping into the workspace folder and executing

Package creation

```
pharos@PhaROS:~$ pharos create my−package−name
```

But before execute any command, lets make an analysis of the options we have.

location Location simply indicates the location of the workspace. If you have stepped on it before, this parameter is not needed.

silent The silent flag indicates if the installation should be silent. If it is false it will open the graphic interface of the image that is being installed. This means that you will be able to debug problems related with your configuration of. But, there are dragons there. The code that loads your code is orthogonal to PhaROS. In order to debug that you will need to get used to an other framework that is about deployment.

version Version indicates the pharo version, it could be 1.4, 2.0, 3.0. We still supporting pharo 1.4 because there are some architectures that needs old vm, and for running on old vms you may need to run an older pharo. A good example of this is raspberry py. For the rest, the tool actually support just this versions.

ros-distro The relation in between PhaROS and the current ROS installation is quite light, but is needed, because the commands to execute for some things are located in different places. The actual supported distributions are the listed in the help: fuerte | groovy | hydro

archetype The archetype is the base definition of the dependancies. We will extend this definition widely in the code distribution chapter.

author, author-email, maintainer,maintainer-email This information is needed by the catkin meta data file. Is quite important that you provide well formed email addresses, because if not, the tool will fail after doing all the work, taking much time for having acknowledge of the error. Just check twice this data.

description Describe your project here. This is for human reading, so you can explain what ever you want. If you want to change information later you just need to execute the related package.xml file.

pharo-user During the execution of the installation scripts and even after, when usage of the package begins, you always need a user set in your image. If you have not configured it, strange things can happen (like scripts to not work in headless mode). So, we encourage you to use this flag for setting it. It is also the name that will be signing the committed changes.

force-new Some times you just mistake a configuration, or had an error. Or just want to start again. In this cases when you already have a package in your workspace with the same name, and you want to delete it, without

needing to delete it your self, add this flag as true. Take in care that this command makes no backup, you will not able to recover what is deleted.

10.2 Making some package creations

I will expose here the most complex things to understand for configuring the creation, this is the not-silent mode of creation and the chosen archetype. We make also some permutations of values meanwhile we change this two, to have some expected feedback knowledge.

The default creation

The default creation is quite good. It is based on the basic-archetype. What means that it download the PhaROS basics, some examples. It generates also a basic package and it gives some cheats. Remember that since you can have several distribution in your machine, is your responsibility to define which distribution you want to use. The default installation uses Pharo 2.0 in a groovy installation. This is our most widely tested bundle.

Lets test it.

Default package creation

```
pharos@PhaROS:~$ cd ~/ros/workspace
pharos@PhaROS:~$ pharos create basicPackage.
```

Default package creation - output

```
−−2014−02−25 11:12:14−−  http://get.pharo.org/vm
Connecting to 10.1.1.3:8080... connected.
Proxy request sent, awaiting response... 200 OK
Length: 5285 (5.2K) [text/html]
Saving to: `STDOUT'

100%[=======================================================================================
     5,285      −−.−K/s   in 0.006s

2014−02−25 11:12:14 (876 KB/s) − written to stdout [5285/5285]

Downloading the latest pharoVM:
  http://files.pharo.org/vm/pharo/linux/stable.zip
mkdir: cannot create directory `pharo−vm': File exists
−−2014−02−25 11:12:17−−  http://get.pharo.org/20
Connecting to 10.1.1.3:8080... connected.
Proxy request sent, awaiting response... 200 OK
Length: 2587 (2.5K) [text/html]
```

```
Saving to: `STDOUT'

100%[=======================================================================
     2,587       --.-K/s   in 0.001s

2014-02-25 11:12:17 (1.79 MB/s) - written to stdout [2587/2587]

Downloading the latest 20 Image:
  http://files.pharo.org/image/20/latest.zip
Pharo.image
Setting up variables, network and obtaining required code....
Running pre process...
Installing package...
Running post process...
Base path: /home/pharos/ros/workspace
Source space: /home/pharos/ros/workspace/src
Build space: /home/pharos/ros/workspace/build
Devel space: /home/pharos/ros/workspace/devel
Install space: /home/pharos/ros/workspace/install
WARNING: Package name "basicPackage" does not follow the naming conventions. It
     should start with a lower case letter and only contain lower case letters, digits and
     underscores.
####
#### Running command: "cmake /home/pharos/ros/workspace/src -
     DCATKIN_DEVEL_PREFIX=/home/pharos/ros/workspace/devel -
     DCMAKE_INSTALL_PREFIX=/home/pharos/ros/workspace/install" in "/home/
     pharos/ros/workspace/build"
####
-- Using CATKIN_DEVEL_PREFIX: /home/pharos/ros/workspace/devel
-- Using CMAKE_PREFIX_PATH: /home/pharos/ros/workspace/devel;/opt/ros/groovy
-- This workspace overlays: /home/pharos/ros/workspace/devel;/opt/ros/groovy
-- Using Debian Python package layout
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/pharos/ros/workspace/build/
     test_results
-- Found gtest sources under '/usr/src/gtest': gtests will be built
-- catkin 0.5.77
Unloading installation code ...
-- BUILD_SHARED_LIBS is on
WARNING: Package name "basicPackage" does not follow the naming conventions. It
     should start with a lower case letter and only contain lower case letters, digits and
     underscores.
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 2 packages in topological order:
-- ~~  - basicPackage
-- ~~  - donatello
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'basicPackage'
```

```
−− ==> add_subdirectory(basicPackage)
WARNING: Package name "basicPackage" does not follow the naming conventions. It
     should start with a lower case letter and only contain lower case letters, digits and
     underscores.
−− +++ processing catkin package: 'donatello'
−− ==> add_subdirectory(donatello)
−− donatello: 0 messages, 1 services
−− Configuring done
−− Generating done
−− Build files have been written to: /home/pharos/ros/workspace/build
####
#### Running command: "make −j1 −l1" in "/home/pharos/ros/workspace/build"
####
[ 25%] Built target donatello_generate_messages_lisp
[ 75%] Built target donatello_generate_messages_py
[100%] Built target donatello_generate_messages_cpp
[100%] Built target donatello_generate_messages
```

If we check in this installation, we can see that there is a warning because of the name of our package. ROS adopt snake case for writing names. So, in order to respect the standard our package should be named something like `basic_package`.

But is just a warning. Take it in care in case of bizarre errors.

Lets browse our image and understand what the hell is waiting for us there

Default package editing

```
pharos@PhaROS:~$ rosrun basicPackage edit.
```

The first thing that receives us is a workspace titled as PhaROS - Cheat Sheet

In this workspace you will find quick how to deal with your new package, and with packages in general. It exposes also some snippets that we have already analysed in the previous part.

The idea of this workspace is to be something you can have at hand. There is a copy of this cheat sheet in the Appendix D of this text.

Then we have al so the following window opened, but in second plane

The creation cycle it generates a package with the given name and our PhaROS conventions (Which are more related with pharo than with ROS, since we use Pharo to work)

This class have several scripts that also shows how to deal with the basics

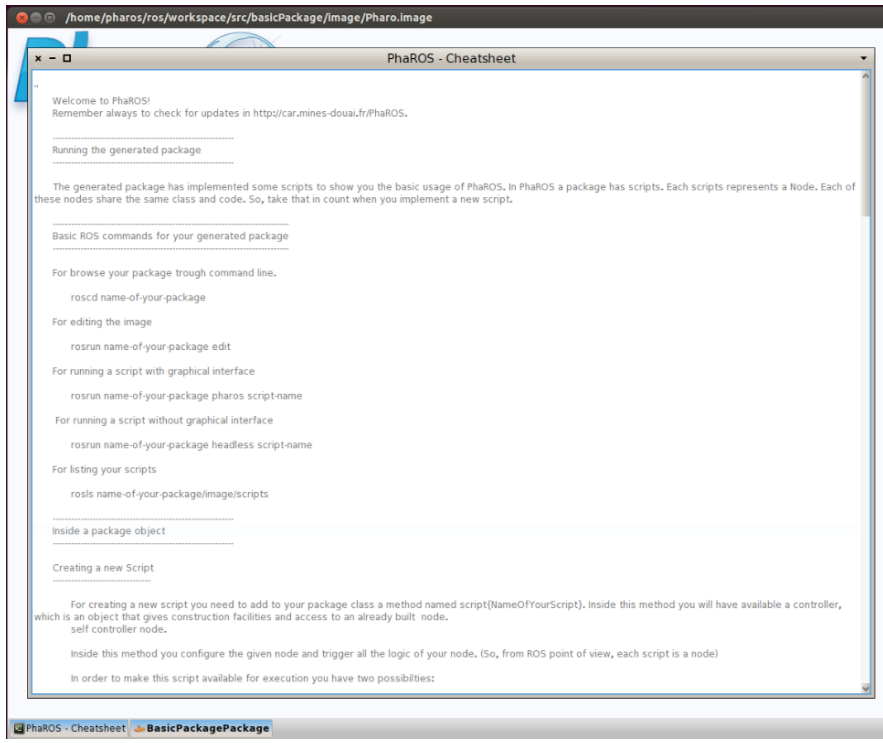


Figure 10.1: Basic Package - Cheat Sheet

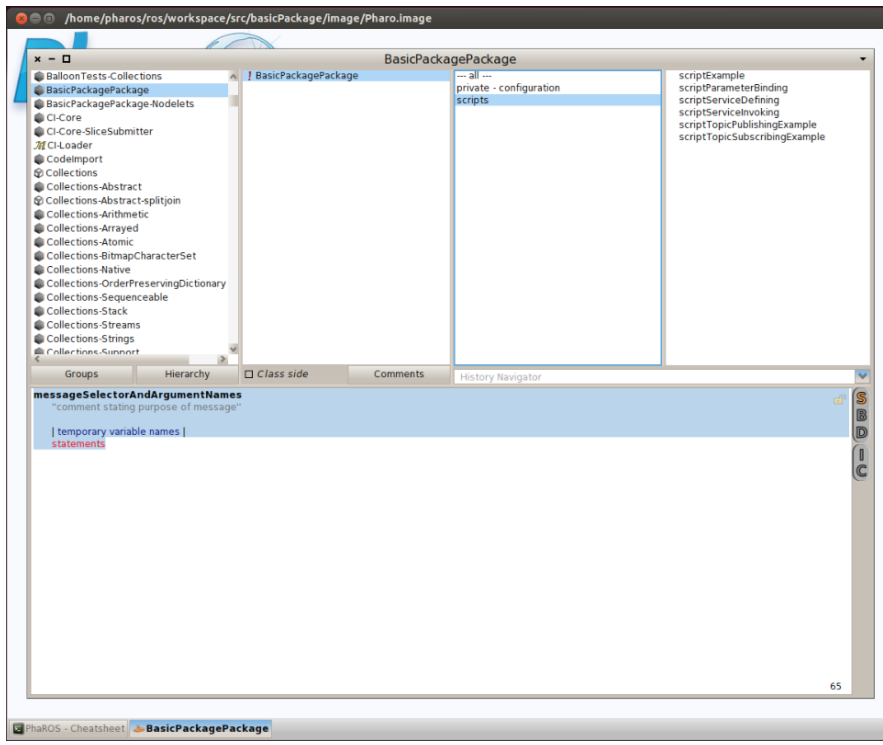


Figure 10.2: Basic Package - Generated package class

It shows you what an script is, it shows you also how to deal with topics , services and parameters in an easy example that is not fully functional. It also show you how to inject a nodelet and for that reason it generates a nodelet

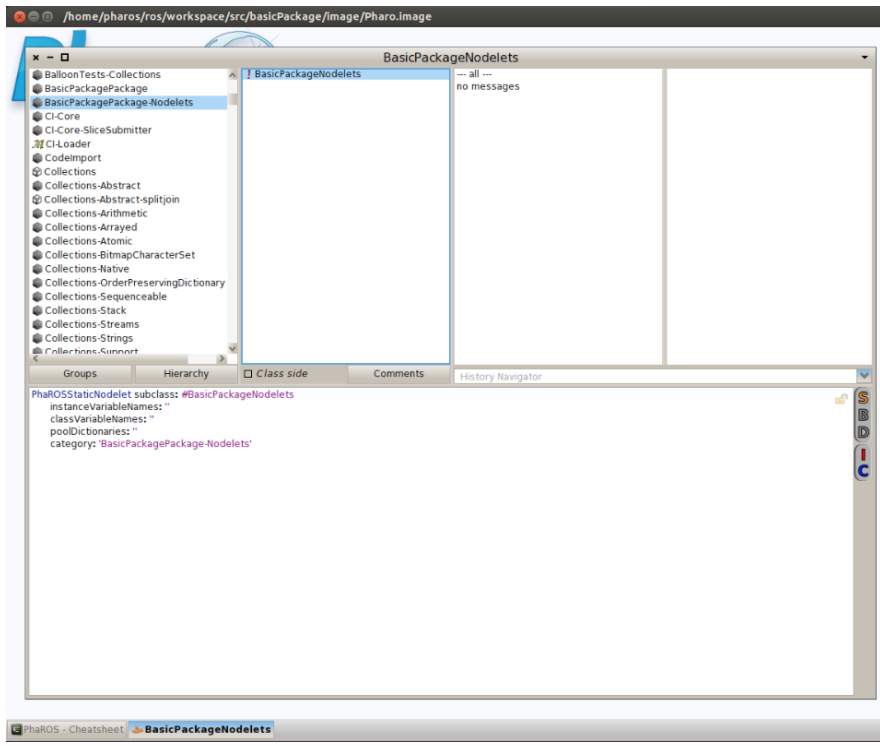


Figure 10.3: Basic Package - Generated nodelet class

Which is, like we see, empty.

The rest of the examples will be almost the same, they will change just a bit about what is downloaded from with the framework, which distribution is setter and which pharo version is used.

Watching our installation

There are cases when a creation should be supervised, by example, if the configuration chosen is not compatible with the pharo version. For this cases, we have the flag named 'silent'. This flag let us ask for showing the graphic interface of the image that is being built.

So this example will be focused mainly in this flag. we will use also the author, maintainer specification.

So let's execute the following lines

Unsilent package creation

```
pharos@PhaROS:~$ cd ~/ros/workspace
pharos@PhaROS:~$ pharos create basicPackage −−author="
MyUserName" −−author−email="user@mail.com" −−maintainer="
OtherUserName" −−silent=false
```

Since the package named basicPackage already exists, we will have an output like this

Unsilent package creation -Error

```
The package already exists in the source folder. Add −−force−new for
deleting the existant package
```

So lets add, as proposed, the –force-user

Unsilent package creation

```
pharos@PhaROS:~$ cd ~/ros/workspace
pharos@PhaROS:~$ pharos create basicPackage −−author="
MyUserName" −−author−email="user@mail.com" −−maintainer="
OtherUserName" −−silent=false −−force−new
```

It will show the same output as in the basic installation on the console, but this time we will see something like

After the installation is done, the pharo IDE will close by it self.

lately, if we step into the created package and we read the package.xml file

roscd - cat

```
pharos@PhaROS:~$ roscd basicPackage
pharos@PhaROS:~$ cat package.xml
```

we will find something like

package.xml

```
<?xml version="1.0"?>
<package>
  <name>basicPackage</name>
  <version>0.1.0</version>
  <description>A PhaROS package</description>
```

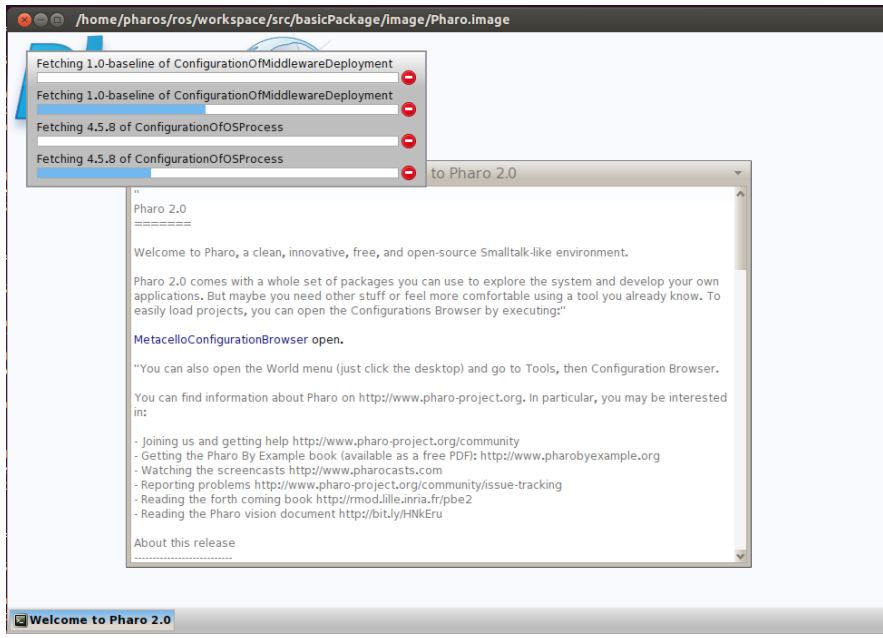


Figure 10.4: Basic Package - Unsilent installation

```
    <maintainer email="user@mail.com">MyUserName</maintainer>
      <license>MIT</license>
      <author email="user@mail.com">OtherUserName</author>
      <buildtool_depend>catkin</buildtool_depend>
</package>
```

Here we will find the usage of the user information. We can see here that the email is both times the same (Because we just specify one) meanwhile the user is different, because we specified it both times.

If we analyse the created package with the edit command, we will see that the installation is completely the same, even the generated code, because our new package has the same name.

Customising the dependencies

There are by default three archetypes in the default repository. And each repository can define it owns archetype, as we will see in chapter 13

- basic-archetype
- core-archetype

- full-archetype

The basic-archetype downloads as dependencies the PhaROS core, with the TF (transformation) nodelet, and the TurtlesimNodelet as example of usage of PhaROSTransformationNodelet. (We will explain this nodelet in the next part of the book)

The core-archetype downloads as dependencies the PhaROS core, with nothing else but package support. This is the lighter configuration you can have with the Pharos installation tool.

The full-archetype downloads as dependencies the PhaROS core, with all the nodelets available in the default repository. This way is not recommended in any but the real case that you need all the nodelets, or that you need to understand the effect of a refactor on the whole system

For our example we will choose the core-archetype. We also will use the pharo user specification and the description parameters.

Unsilent package creation

```
pharos@PhaROS:~$ cd ~/ros/workspace
pharos@PhaROS:~$ pharos create basicPackage --archetype=core-archetype --pharo-user=AUserForPharo --description="This is an example for the PhaROS package creation chapter" --force-new
```

nist

It will show the same output as in the basic installation. As much, we will find the installation faster.

The important details of this example is that you will find not other subclasses of nodelet but the one generated for the package, that if you evaluate in a workspace the following

```
```

it will print 'AUserForPharo'

And finally, if we cat the package.xml file,

roscd - cat

```
pharos@PhaROS:~$ roscd basicPackage
pharos@PhaROS:~$ cat package.xml
```

we will read the following

package.xml

```
<?xml version="1.0"?>
<package>
    <name>basicPackage</name>
    <version>0.1.0</version>
    <description>This</description>
     <maintainer email="author@mail.com">author name</maintainer>
     <license>MIT</license>
     <author email="author@mail.com">author name</author>
     <buildtool_depend>catkin</buildtool_depend>
     <export>
     </export>
```

All the rest of the installation will remains the same.

10.3 Brief

- pharos create package
  Creating a default package is easy as this one line code
- pharos create package –silent=false
  Use –silent=false for checking the image installation time.
- pharos create package –archetype=– an archetype –
  Check for suitable archetypes to make easier the setup of your new package
- Almost information is for set up the catkin package as needed. So you can give the information now, or setup the package.xml file later.

Chapter 11

Advanced topics subscriptions

11.1 Adapting incoming data

Topic connections are typed, what is obviously needed for high-performance transference, but it makes to repeat a lot of code, just because types just not match.

A common example is to have a code that works with a Pose type:

rosmsg - show

```
pharos@PhaROS:~$ rosmsg show geometry_msgs/Pose

geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

But we are subscribing to a PoseStamped type:

rosmsg - show

```
pharos@PhaROS:~$ rosmsg show geometry_msgs/PoseStamped

std_msgs/Header header
```

```
        uint32 seq
        time stamp
        string frame_id
    geometry_msgs/Pose pose
        geometry_msgs/Point position
            float64 x
            float64 y
            float64 z
        geometry_msgs/Quaternion orientation
            float64 x
            float64 y
            float64 z
            float64 w
```

so, we will need to write a different callback for each case, just because the type is not the same.

Or even better, we may have a piece of software that uses instead of a ROS type, a domain type, which is always mapped the same way from a given ROS type.

For solving this kind of problems we have a concept called Adaption.

Navigating data

A common case then, is to need by example, the pose, instead the pose stamped. For this case, what we need is to adapt the incoming data, in order to execute the callback with the information available in the pose field of the object.

```
typedAs: 'geometry_msgs/PoseStamped';
adaption: #pose;
for: [ : pose |  Transcript show: pose ];
connect .
```

This means that the result subscription will have an adaption object that will send the pose message to the given object, and send the result to the callback.

What happen if we need to get the position of the pose, and not the pose.

Not big deal, we just need to:

```
typedAs: 'geometry_msgs/PoseStamped';
adaption: #pose \> #position;
for: [ : position |  Transcript show: position ];
```

```
        connect .
```

the

```
            \>
```

operator is an operator that defines a chain of transformations. This means that, after send the message pose, it will send the message position, and use that as input of our callback.

Like this, we can navigate any kind of object. The main cons is that we cannot receive any but the value of one field. (we cannot yet choose more than one field). What means that we will lose the rest of the data if we use this.

Transforming data

Other of the common problems are the reifications. ROS Types are just for sending information, but they are not domain objects. They are generated on demand, i order to make light weight the image. This means that the object i will receive in my callback is always a dumb object that cannot do but set and get attributes.

There are, however, some objects that are commonly transformed to domain objects, that can be usually mapped from one type to other. By example, PhaROS comes with a class named PhaROSPose. This class has implemented for it objects several methods for conversion an measures. In other words, when we are working with poses, in almost cases we will want to deal with this object and not with the generated one.

For this can of cases, we can add to our chain of adaption a transformation of types

Lets picture that we have a topic typed as Pose, we can connect it as

```
        typedAs: 'geometry_msgs/Pose';
        adaption: PhaROSPose;
        for: [ : pose |  Transcript show: pose ];
        connect .
```

The received pose, it will be of type PhaROSPose.

Again, if we need the position instead of the pose, we can write it in two ways

```
typedAs: 'geometry_msgs/Pose';
adaption: PhaROSPose \> position;
for: [ : position |  Transcript show: pose ];
connect .
```

This will convert the pose to a PhaROSPose and then ask for the position. Or, for making less computations

```
typedAs: 'geometry_msgs/Pose';
adaption: #position \> PhaROSPosition;
for: [ : position |  Transcript show: pose ];
connect .
```

This will first navigate to the position field, for converting it content to a PhaROSPosition type. Which is the type of the position component of a PhaROSPose.

How this bizarre mechanism works? Because is not really easy to map two types, it cannot be free, and, actually is not.

This notation allow us to have the transformation of a type coded in just one place, and even hide the name of the conversion.

For having this conversion available, the class that is wanted to be created should have a constructor named fromClassNameOfTheROSType).

If we browse the class side of PhaROSPose and PhaROSPosition we will find

```
  ^ self position: (PhaROSPosition from: aPose position) orientation: (
     PhaROSQuaternion from: aPose orientation)

PhaROSPosition>>#fromGeometry_msgsPosition: aPosition
  ^ PhaROSPosition from: aPosition
```

If you are asking why the PhaROSPosition constructor is so generic, it is because it construct it self from any object that understands the messages #x #y and #z.

This mechanism have also the same problem, there is information that we will lost and do not be able to recover.

11.2 Conditionals

Usually what happens is that we need some part of the information for conditions, or even that the acceptance condition of a packet of data is complex enough to want it out of the callback logic.

For this usage we have the message #when: that can be sent to the connection builder.

The usage is quite simple

Picture that we want the pose object, but we don't want all the poses, just the poses that are as older 1 second. This is a quite common restriction, since we want data to be fresh as possible (even 1 second is chosen to make the example easier, but is maybe a lot of time)

```
        typedAs: 'geometry_msgs/Pose';
        adaption: #position \> PhaROSPosition;
        when:[ : message | DateAndTime now − (message header stamp) < = 1
second ];
        for: [ : position |  Transcript show: pose ];
        connect .
```

So, the callback will be executed only when the lifetime of the pose stamped is less than 1 second. If is more, it will be rejected. Is important to see that the conditional block will receive the raw information, so, when you want to make some conditions related to information that you don't need in your callback, you just use a conditional block. Then the callback can remains easy and domain oriented.

11.3 Time stamp

One of the problems of all this machinery is that we may still needing the timestamp for, maybe, synchronising data, or for other kind of calculations. For this cases, where the only thing we need from a data packet, but one concrete field, is the time stamp related with the header, or the packet it self, we can rely not the optional parameters of a callback

And we can rely in this parameter always, when the first type has a header, the stamp will be the defined in it

```
        typedAs: 'geometry_msgs/PoseStamped';
        adaption: #pose \> PhaROSPose;
        when:[ : message | DateAndTime now − (message header stamp) < = 1
second ];
```

```
for: [ :position :channel :stamp |  Transcript show: stamp  ];
connect .
```

And when we have no header

```
typedAs: 'geometry_msgs/Pose';
adaption: PhaROSPose;
for: [ :position :channel :stamp |  Transcript show: stamp  ];
connect .
```

The stamp received will be the stamp of the reception of the packet

The restriction of this mechanism are

- The algorithm check for a header in the first type, it does not do it in inner types. So, if the header is in the an inner type it will not be minded
- The stamp mechanism is only available in the callback scope, not in the conditional scope. Thats why our second example of usage has not when: message.

11.4 Callbacks

There are times where we need to configure our connection in one part and set the callback in other place, in a place that you don't want to use a builder, but the real topic subscription. In this cases you just need define your connection as always but do not use the for: message during the configuration.

The builder will set a default callback that writes in the transcript this legend 'Warning! This topic conection it has no callback configured! ', in order to do not let you forget the fact that you have a thread running, receiving information that is not being used.

Then, where you receive the connection, you can send to the topic the messages #for: and #callback: Both do the same, but express different semantics.

```
typedAs: 'geometry_msgs/Pose';
adaption: PhaROSPose;
connect .

connection for: [ :position :channel :stamp |  Transcript show: stamp  ].
```

If you don't mind to use a builder object, and have the connect message in the receiver context, we recommend to use the builder, so you are sure that there is always a useful callback setter.

```
                typedAs: 'geometry_msgs/Pose';
                adaption: PhaROSPose.

        connection := connectionBuilder for: [ :position :channel :stamp |
    Transcript show: stamp  ];  connect.
```

11.5 Tip about performance

One implementation detail that is important is the fact that a topic is connected one time per node. Since we are always working with the same node inside the package definition, all the times that we connect to a topic, we are using always the same connection, but defining different objects for callback management.

This is an important trade off. We can have then several callbacks for a topic, each one with it own full configuration. This is flexble, and reduces the IO work. In exchange, the same thread that does the IO work, is working on the callbacks. So, the more callbacks you define, the more work have the thread in between input and other, making the available messages to read quite bigger, and losing efficiency in the reading, what have impact in the how-fresh is a data.

So, take this in care when you are deciding to have more than one connection to the same topic from the same instance.

11.6 Brief

- Adapting incoming data
  - Add the message #adaption: to the connection building chain of messages to shape the message that it will be received by your callback.
  - Use symbols to adapt sending messages
  - Use class names to make conversions (Reminding to implement the needed class methods)
  - Combine your adaptions with the \> operator.

- Add the message #when: to define a callback based on raw data to accept or reject a received packet
- Let your callbacks rely on the third optional parameter to have the time stamp of the packet reception.
- Divide your connection building from the callback setting to be able to split the name and type of a topic from your behaviour code.
- Mind the thread architecture when you do strange stuff in your callbacks.

Chapter 12

The SMA (SRMA) pattern

I always use to say that a design or even an architectural pattern has to patterns inside. The one that we use to see, the solution pattern, and the one we use to avoid, the problem pattern.

A pattern then is a guide, when you see something that looks like 'problem-pattern' you can end up doing something like 'solution-pattern'. For architecture patterns, the 'something like' part of the sentence, is a bit more strict, because once you defined the architecture layout, you want to be as tight as you can from respecting it, and what we propose here is an sub architectural one.

I call it sub architectural because we are already in the ROS architecture, the main general lines are already drawn. What we are drawing here is the architecture of the node or process we want to program.

12.1 The Problem Pattern

Until here we have seen several examples about how to make our node to be communicated and how to make it part of a larger system, but in technology terms. Aiming to learn how to use the basics of ROS and PhaROS.

We tried use some cool abstractions to respect our own philosophy, but some times, in order to keep it simple we did not put much emphasis, so the connections of ROS are easy to understand in our domain.

But, it happens that we don't want to make it that way. We don't want our code to be obviously connected with ROS, because we want to scope the connections and ROS knowledge as much as we can.

ROS Topics are cool, and organic, but really hard to manage and track. Since the connection is done by name, you **really** need to guarantee that

there will not be collisions of names.

By other side, the needed of adaption of a code to work with one or other type is quite common. If your code is tightly related to the type definition of a topic, you will need to write it again each time you need to deal with different nodes, or add infinite adapter nodes in the middle, generating even more noise in the definition of what the system is.

Also if we analyse a common node has three parts: a part when it consumes sensors, and synchronise them if needed. A part where it make calculations about the sensed data, and the part where it stimulates other nodes (or hardware) based on the sensed information.

Al this three parts use to be mixed in the same code.

This has a lot of dirty implications

- You **hardly can reuse** your model with other sensors
- You need to define the sensor logic **each time** you use the same sensor
- You need to write down the name of the sensor topics each time you define a connection **repeating yourself** an annoying amount of times
- Your sensor logic is just **testable if the whole system is working**.
- Your sensor logic is **not unitary testable**.
- Your solution code is **coupled with ROS**.
- Your solution code is **coupled with the distribution of ROS**.
- Your solution code is **coupled with nodes** you use through the names and types defined by this nodes.
- Your solution code is **not unitary testable**.
- Your solution code is **not testable for requirement achievement**.
- Your solution code needs a **whole ROS installation to be tested**.
- You **hardly can reuse** your model with other actuators
- You need to define the way to interact with the defined actuators **each time** you use them
- You need to write down the name of the actuator topics each time you define a connection **repeating yourself** an annoying amount of times
- Your actuator logic is **not unitary testable**.
- Your actuator logic is just **testable if the whole system is working**.
- Your experiments are really **hard to reproduce**.

12.2 The solution pattern

Inspired in MVC pattern, adapted for robotics, we defined the SMA pattern, or SRMA, acronyms of **Sensor - (Reactive) Model - Actuator pattern**.

A first view of the code outline with this pattern

```
MyPackage>>#configureModel
    " Reactive Model configuration "
    map := Map new.
    map onRobotOutOfRangeDo: [ : pose |
        self actuator pathplanner scheduleGoal: (map closestBoundTo: pose).
    ] cooldown: 20 seconds.


MyPackage>>#scriptBoundController

    self configureModel.

    self clock tickAt: 2 hz on: ({
        self sensor pose.
    } asSensorBundle tolerance: 1 second);
      for: [
     : robotPose |
        " Model feeding "
        map noteRobotAt: aRobotPose.
    ]
```

But before go to the big picture, lets presents the concepts

Sensors

We define as sensor, an object that act as that. Any input that give us information about our domain is treated as a sensor. Some of them will be actual sensors, others can be processed data from a sensor, (by example, pose distilled from and odometry sensor), or high-level sensors, like when some logic state of our system changes, usually because of an actual sensor.

Our classification then can be briefed as

- Environment perception
  - Raw data: Laser, odometry, telemeters, etc.
  - processed data: Pose, occupancy, etc

- Self perception: battery state, laser state rate, etc.
- System(Logic) perception: Objects encountered, Critic battery state reached, Forbidden area reached.

Actuators

We define as actuator any object susceptible to be controlled. Any thing that needs an input to change it state is treated as an actuator. Is an object that will give us the chance to make it do something by given the proper data. Some of this actuators will be actual actuators, some of them will have a real impact over the environment (like legs, differential drive, arms, etc), others will just have impact over the robot (start/stop sensors), others will have impact over our system, by example, setting a goal to a path planner.

Our classification then can be briefed as

- Phyisical modification
  - Environment modification: Differential drive, arms usage, legs usage, lightsaber, etc.
  - Self modification: Start/stop devices.
- System (Logic) modification: Inform object recognition, inform a point to reach.

Reactive Model

Ok, we have read about models in a lot of places. Is about make up the concepts needed to make a model of what ever we need to deal with. By example, an array of bits can be the way to model an image.

We want to tight our selves to Domain driven design as much as we can. So, we want our models to be highly coupled to the domain.

This domain model, in order to match with our SMA pattern, need to take in care also the following facts

- Punctual ways to feed the model
- Punctual ways to give feedback

This means, being able to have a nice way to interact with sensors that will give information from different threads and actuators, that we want to not have them referenced inside our model.

We define as reactive model a model that reacts somehow to the changes proposed by the sensors caption. For this part, we chosen to model the reaction with events.

Lets analyse the code of the beginning.

```
MyPackage>>#configureModel
    " Reactive Model configuration "
    map := Map new.
    map onRobotOutOfRangeDo: [ : pose |
        self actuator pathplanner scheduleGoal: (map closestBoundTo: pose).
    ] cooldown: 20 seconds.


MyPackage>>#scriptBoundController

    self configureModel.

    self clock tickAt: 2 hz on: ({
        self sensor pose.
    } asSensorBundle tolerance: 1 second);
      for: [
     : robotPose |
        " Model feeding "
         map noteRobotAt: aRobotPose.
    ],
```

We have two methods. On that make up the model. This example is illustrative, you should have the initialisation of the model probably in the package, and in the node configuration just the specific configuration of the node.

The model related is a map. This map has the characteristic that it knows the places that are forbidden for the robot.

In order to react to this, when the robot is not in range, it has a callback for event handler, that is set with #onRobotOutOfRangeDo:.

In this case the callback will ask to the actuator to schedule as goal the closest point inside an allowed area. This action will have a cool down of 20 seconds (This means that the model will not execute again this callback before 20 seconds).

Lets analyse now the main node code.

```
            robot sensor pose.
        } asSensorBundle tolerance: 1 second);
```

```
        for: [ : pose | callback ]
```

This code spawn thread 'clock' that runs at 2hz ticking (sending the message tick) to a sensor bundle. A sensor bundle is a collection of sensors that should be synchronised. For this, the bundle has a tolerance of 1 second (a message with more than 1 second is considered obsolete and dropped). Each time the tick has a synchronised response, it executes the given callback with all the sensors data.

```
        " Model feeding "
        map noteRobotAt: aRobotPose.
    ],
```

Our synchronised sensor callback has the responsibility of feeding the model with the proper information.

Then in de end, the model will respond to this information, rising an event when the robot is out of range.

12.3 How does this pattern help us??

Expressivity The first thing to get is the fact that the lines needed for setting up the node are quite understandable. We have an specific place for placing to actuators configuration, one specific place for the sensor perception.

We reach a programming

We have nothing related with topic naming or typing in the code, actually we have no idea if any of our code is related with topics. Neither if is related with ROS.

Since both, sensor and actuator are points of injection, replace one source for other, is trivial.

For testing your domain, you can rely on the working system, or mock one or more of the related artefacts and inject them during setup.

You can mock the sensors, relating it with known easy data, or with files of sensor dump. You can easily mock actuators and check messages sent, what is the most important part from our side of testing.

Since the sensor encapsulate what is related with each punctual sensor, you can reuse this logic all the times you need it, and the same happens with the actuators.

In the cases that you need your node to use other sensors, or other ac-

tuators, you can write a SensorAdapter, or ActuatorAdapter, that makes a sensor look like the expected sensor.

If the sensor is quite different, probably you don't want to use the same node.

This model lead us to have all the configuration of sensors and actuators of all the system in just one really coherent place. Meaning that the experiments will be easier to reproduce and far easier to adapt.

And in the end, for testing requirement meetings, you just need to test event rules. If the event rules are ok, and the actuators mock receive the expected messages with the expected sensor entry, the requirements are meet.

12.4 What we learned

Using SMA pattern for defining almost possible nodes in this architecture is a good idea. The effort required for making it happen benefit back with high quality through expressivity, reusability and testability.

Use sensor bundles to have synchronisation based on specified tolerances . Use the rate to make your system performant enough. Make your model to be reactive, is the smarter choice to model a behaviour (what is by nature reactive).

Note: We will have a real complete example of usage and good for understanding soon :) .

Chapter 13

Distributing our code

Ok, i have done my code, and i want it to be installed with Pharos installation tool, because is so cool to do not need to do anything but type one line.

Distributing your own packages is quite easy. The first thing you need to have is a metacello repository (or compatible), and a ConfigurationOfYOUR-REALM that understands how to load your package, responding to

(ConfigurationOfYOUR-REALM project version: #WeWillSupplyThisParameter) load: 'YourPackageName'.

Once you have that done, proceed to the next stage!

13.1 Creating a new repository

A repository of code that have packages for PhaROS needs to have a class called directory. This directory will have metadata of the available packages in this repository.

For doing this, the Pharos installation tool has a command that will generate code that you can install in your image, customise and commit into your repository.

Repository creation

```
pharos@PhaROS:~$ pharos create−repository −−help.
```

Repository creation

```
usage
  pharos create−repository PACKAGENAME [ OPTIONS ]
```

```
                PACKAGENAME is the name of the project that this directory will
represent. By Example PhaROS.

                    --user name of the user for generation by default is 'Generated'
                    --output path to where the script will be dumped. By default is the
standar output.
```

Well there are not much options to take in care

package name This name will be the hooked up, You can also put here the name of our your solution. By example we could make a directory for Donatello package it self, but we decided to include it in PhaROS directory. In the end, is a repository. Just try to keep coherence about the fact that is stored in the same repository. This name will be the name of the generated class also, so mind the conventions.

user The output of this command will be generated code, ready to be installed. Like all the code in pharo, it should have a related user. Setup this parameter if you are not happy with the user 'Generated'

output The output is a file to be installed in an image. You can setup the name of the file where you want the output. Since the default output is the standard output, you can use the operator \> of the operative system to redirect it to a file you want.

Let's execute this and see what happen

Repository creation

```
pharos@PhaROS:~$ pharos create-repository example > example.st
```

Repository creation - Error

```
example is not a valid name for a package. It must start with uppercase.
ThisIsACorrectPackageName
```

In pharos, we rely on the conventions of Pharo. This is why, the command had an error. Lets try now with other name

Repository creation

```
pharos@PhaROS:~$ pharos create-repository Example > example.st
```

Now we have a file named example.st in our location

Lets open then our image, a browser for this folder and drag and drop the file on the image

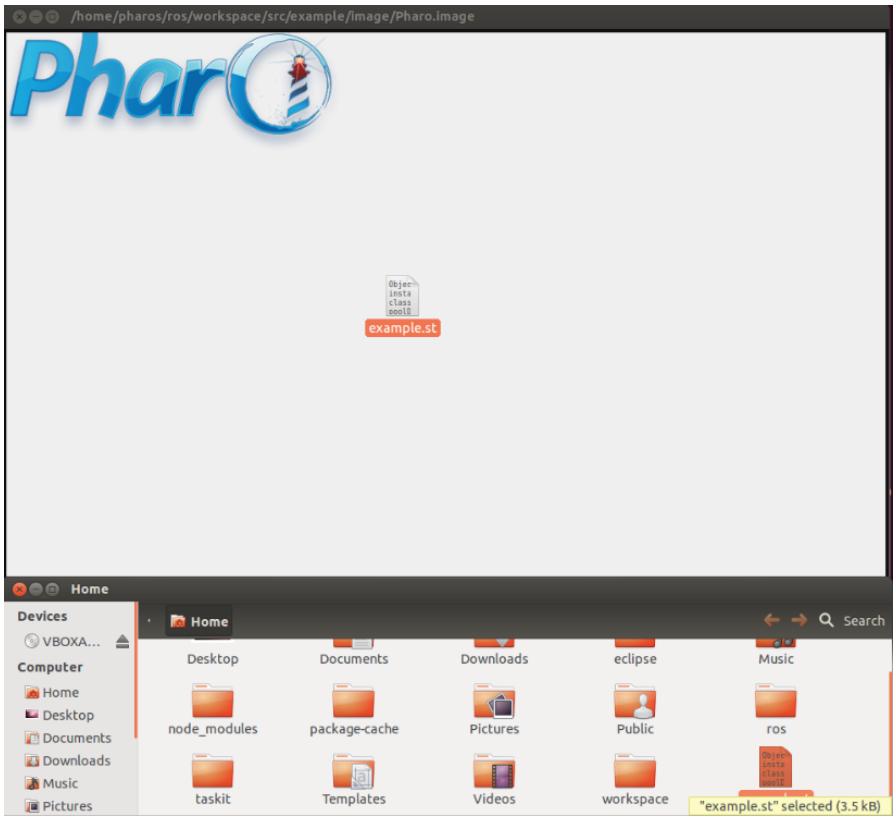


Figure 13.1: Installing generated file

And click the 'filein entire file' option

Finally, browse the class side part of the class named ExampleDirectory

You will find something like

13.2 Defining packages

Well, we have then our ExampleDirectory. This class have several class methods. But there are almost examples or auxiliary methods we don't need to watch. (Even when they are all quite obvious)

We can see the categories: archetypes, packages, and until.

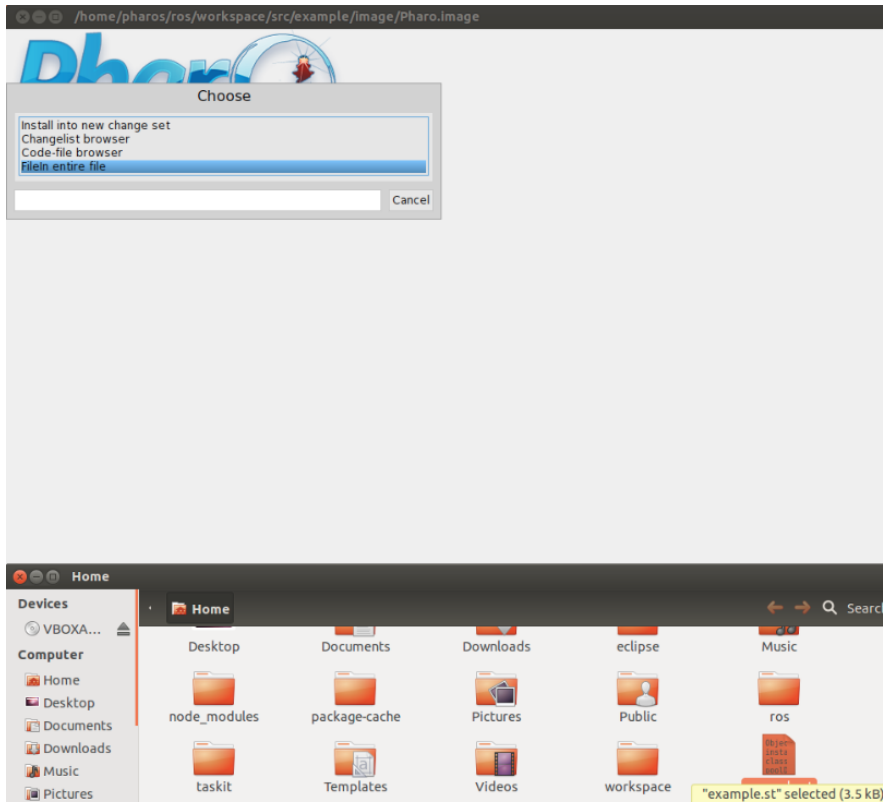


Figure 13.2: Installing generated file

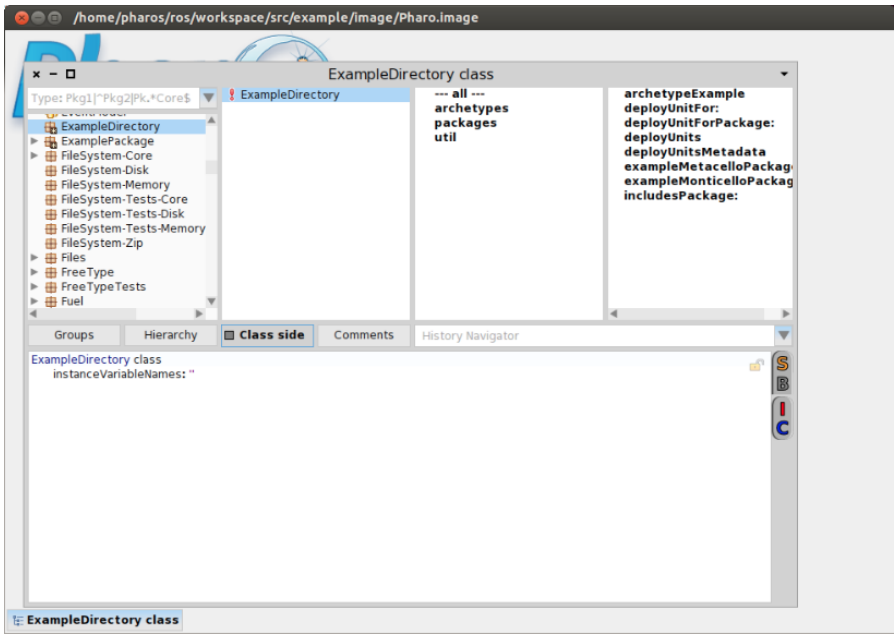


Figure 13.3: Installing generated file

In archetypes and packages we have methods that define both, archetypes and packages.

So, if we want to have archetypes for creating nodes, we need to set them up, if we want just to register our own PhaROS packages, we just need to set up a new package method.

Lets analyse both, that are quite similar.

```
^ {
    #name -> 'example' .
    #description -> 'Example package' .
    #license -> 'MIT' .
    #version -> '0.1.0' .
    #maintainer -> ({
      #name-> 'Generated' .
      #email -> 'Generated@mail.com'
    } asDictionary) .
    #author -> ({
      #name-> 'Generated' .
      #email -> 'Generated@mail.com'
    } asDictionary) .
    #metacello -> ({
      #url -> 'http://smalltalkhub.com/mc/Generated/Example/main' .
```

```
        #configurationOf -> 'ConfigurationOfExample' .
        #package -> 'Example'
      } asDictionary)
    } asDictionary
```

This is a metacello based package. We will just learn to use this flavour, because is the only really needed.

If you check the only thing we need to do is to change the things generated for things related with our package.

Name, description, license, version, maintainer, author, are all information required by ROS for making up the package installation. Then we have the metacello configuration. Here we need to setup the url of the repository, the name of the ConfigurationOf we made for our repository, and finally, the name that we will use to get the package code.

If your repository needs credentials even to download the code, add the following to the metacello definition:

```
        #url -> 'http://smalltalkhub.com/mc/Generated/Example/main' .
        #configurationOf -> 'ConfigurationOfExample' .
        #package -> 'Example'  .
        #needCredentials -> true
      } asDictionary)
```

This flag will make the installation tool to prompt the user for a user/-password during the installation

BEWARE!

You cannot have more than one subclass of PhaROSPackage. This will be the one hooked for installation.

Then, for archetypes the configuration is almost the same, it just change the fact that there is not name attribute, but archetype.

```
^ {
    #archetype -> 'Example-archetype' .
    #description -> 'description' .
    #license -> 'MIT'  .
    #version -> '0.1.0' .
    #maintainer -> ({
      #name-> 'arechetype-default-user' .
      #email -> 'arechetype-default-user@email.com'
    } asDictionary) .
    #author -> ({
      #name-> 'arechetype-default-user' .
```

```
        #email −> 'arechetype−default−user@email.com'
      } asDictionary) .
      #metacello −> ({
        #url −> 'http://smalltalkhub.com/mc/Generated/Example/main' .
        #configurationOf −> 'ConfigurationOfExample' .
        #package −> 'Example−archetype'
      } asDictionary)
    } asDictionary
```

The expected thing to fetch from the metacello repository are the dependencies needed by this kind of package.

We do not have yet hooks for asking the archetype definition or the configuration of to generate any kind of code.

Just as with the packages, if your repository needs credentials even to download the code, add the following to the metacello definition:

```
        #url −> 'http://smalltalkhub.com/mc/Generated/Example/main' .
        #configurationOf −> 'ConfigurationOfExample' .
        #package −> 'Example−archetype'  .
        #needCredentials −> true
      } asDictionary)
```

This flag will make the installation tool to prompt the user for a user/-password during the creation

After define your package and archetypes methods, we need to modify a method in the util category of our directory class.

```
^ {

    self exampleMetacelloPackage.
    self exampleMonticelloPackage.
    self archetypeExample.

 }
```

This method will be called by the installation tool to fetch all the packages and archetypes. So, modify how the array is construct by your own package and archetype methods.

Once this is set up, you just need to commit it in to your repository.

13.3 Registering repositories

Ok, now that we have our repository set up we need to make the tool to know about this endpoint.

For doing this we have a proper command.

Repository registration

```
pharos@PhaROS:~$ pharos register-repository --help
```

Repository creation

```
usage: pharos register-repository --url=anUrl --package=aPackage [
OPTIONS ]
Options:

--url [ MANDATORY ] url of a monticello repository
--package [ MANDATORY ] monticello package name.
--directory Name of the class that works as directory This class
must implement
- #includesPackage:
- #deployUnitForPackage:
Default value - the value given for package.
--user Username for the monticello repository
Default value - empty
--password Password for the monticello repository
Default value - empty


The directory class needs to implement

- #includesPackage: aPackageName
this method receive a name of package and return a boolean indicating if
package is included by this directory

- #deployUnitForPackage: aPackageName
this methods receive a name of package and return a kind of
MDPharoDeployUnit.

For more information checkout the package PhaROSDeploymentDirectory
from the squeaksource repository located at:
http://car.mines-douai.fr/squeaksource/PhaROS
```

The options

url of the repository. In this repository i should be able to locate the named package where the directory is located.

package where the directory is located. Usually the name of the package is the same as the directory.

directory this is the directory class that we just generate and modified before. As it is indicate in the help, the important fact of the directory is to be polimorfic with the classical directory class. If you need an special implementation instead of the generated one, just mind the cited methods.

user/password If there is need of user / password for accessing the directory class, you need to give them here. but, take in care that your password will be stored in a plain text file.

If the Directorty cannot be resolved (because the directory does not exist, or the url is wrong)

Repository registration

```
pharos@PhaROS:~$ pharos register−repository −−url=http://fake.url.
com −−package=Directory
```

You will have this error:

Repository registration

```
Error installing repository: Unable to resolve Directory
```

In this case, check any typo in this or the previous step.

For having a successful example, lets try to do it with PureROS project.

Repository registration

```
pharos@PhaROS:~$ pharos register−repository −−url=http://car.mines−
douai.fr/squeaksource/Pure −−package=PureROSDirectory
```

Repository registration

```
repository installed correctly.
```

Finally, once you have this lecture. You should be able to install and create packages with your own archetypes and package definition :).

13.4 Brief

From the console

- pharos create-repository YourPackageName > example.st
  execute this in the console for generating a directory
- pharos register-repository –url=http://fake.url.com –package=Directory
  execute this in the console for registering an existing repository. Yours of from anyone.

From the image

- Define archetypes and packages metadata for allowing your users to be agile
- Define #needCredentials -> true in your metacello definitions of the metadata to ask for user/password during installation.

Conceptual part

- Set up your repository.
- Use Pharos installation tool for support your development.
- Distribute your nodes quoting the line to register your repository, packages and archetypes available.

Part IV

Appendix

Appendix A

Donatello Design

A.1 AlgorithmSwitcher - Class diagram

As first approach we can analyse this UML graph

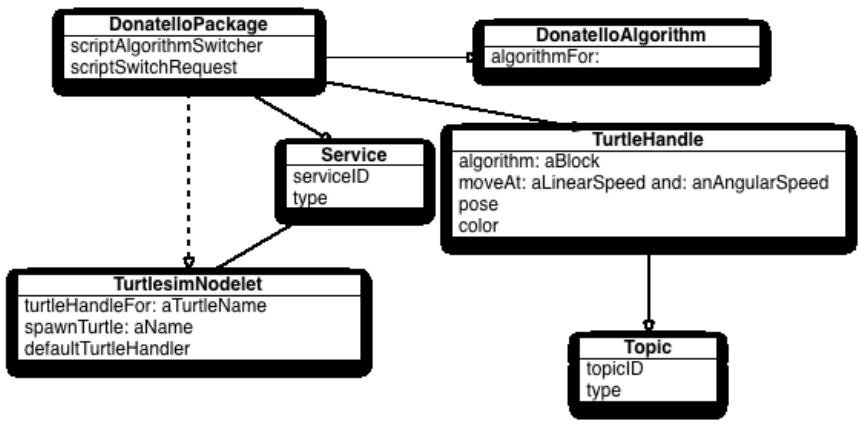


Figure A.1: Donatello - Class Diagram

Since we analysed the code method per method, the UML diagram should be easy to understand. I used the dashed line arrow that goes from the package to the nodelet to make understand that is not a hard reference.

A.2 AlgorithmSwitcher - Object diagram

And to understand deeper the relations, we have two object diagrams, that shows the relation of reference in between objects.

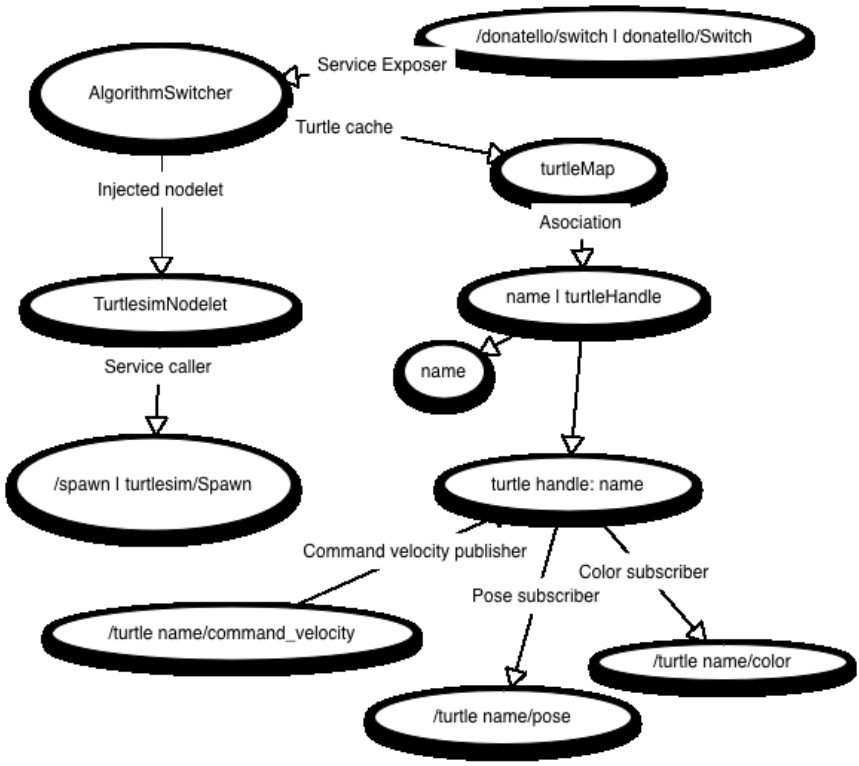


Figure A.2: AlgorithmSwitcher - Object Diagram

In this case the arrows are drawn in the sense of usage. By example, the Service exposer, once configured, it will call the code related to the service, which is in the package definition.

A.3 SwitchRequest - Object diagram

This diagram is quite simple, because as you saw in the code, the only thing that happens is that the node ask the user to insert two parameters and then call a service with that data.

Figure A.3: SwitchRequest - Object Diagram

Appendix B

Building an image for using PhaROS from scratch

B.1 Obtaining a new image

First we will need a new fresh image. For doing this we will rely on zero conf scripts provided in get.pharo.org

Create a new directory for your installation, step into and execute a zero conf script. By example:

Downloading an Pharo 2.0 image

```
pharos@PhaROS:~$ mkdir myPackage
pharos@PhaROS:~$ cd myPackage
pharos@PhaROS:~/myPackage$ wget −O− get.pharo.org/20 | bash
```

Downloading an image - output

```
−−2014−02−21 16:53:39−−  http://get.pharo.org/20
Length: 2587 (2,5K) [text/html]
Saving to: ï£¡STDOUTï£¡
100%[======================================>] 2 587
−−.−K/s  in 0,004s
2014−02−21 16:53:39 (704 KB/s) − written to stdout [2587/2587]
Downloading the latest 20 Image:
    http://files.pharo.org/image/20/latest.zip
Pharo.image
```

Remember that for downloading other versions of pharo you just need to change the last part of the url (20) for the version you want: 14, 20, 30. There

is no support PhaROS to run at a Pharo 1.3.

We will use the Pharo 2.0 image for our example.

Downloading an image - output

```
pharos@PhaROS:~/myPackage$ pharo−vm−x Pharo
```

Figure B.1: Pharo IDE

B.2 Downloading the PhaROS framework

We will copy then the following code into the workspace

```
    package: 'ConfigurationOfPhaROS';
    load.

"If you want to load just the basic stuff, run "
(Smalltalk at: #ConfigurationOfPhaROS) load: 'default'.
```

```
"If you want to load the tests as well, run instead "
(Smalltalk at: #ConfigurationOfPhaROS) load: 'default+tests'.
```

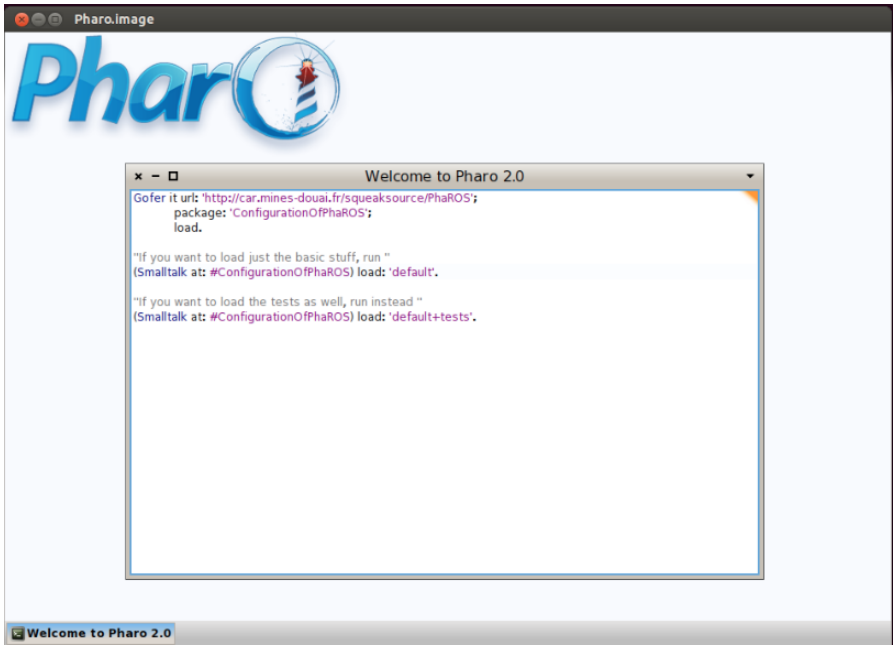


Figure B.2: Gofer code

And execute to load the default package.

Once all the code is loaded we just need to create our own both pharo and pharos packages.

B.3 Making up our package

For doing this we need to subclass the class called PhaROSPackage executing the following code

```
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: '[[MyCrazyProjectName]]'
```

Changing, of course, [[MyAmazingPhaROSPackage]] and [[MyCrazyProjectName]] by the names you want.

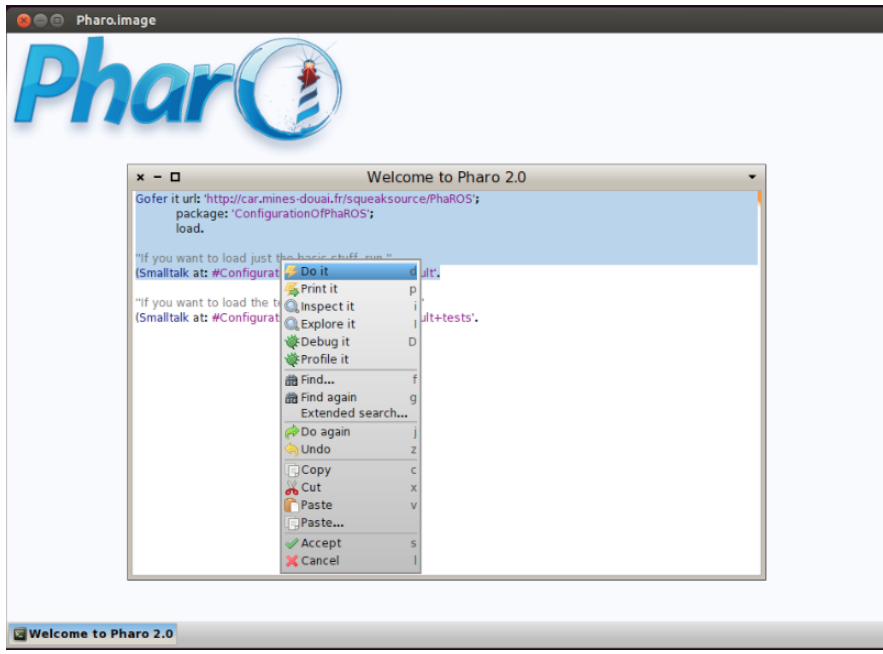


Figure B.3: Gofer code

Save the image, You are done. Now you can start to write your scripts as seen before.

Remember however that automatic script and type generation is not available in this mode.

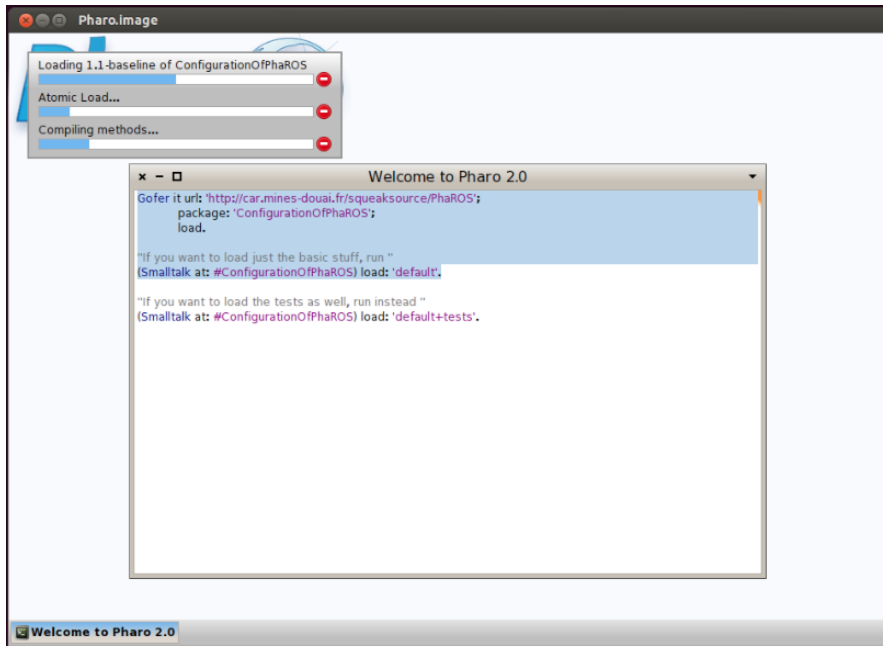


Figure B.4: Gofer code

Appendix C

Pharos installation tool

C.1 Installing Pharos installation tool

Installing Pharos installation tool

```
pharos@PhaROS:~$ wget http://car.mines-douai.fr/wp-content/
uploads/2014/01/pharos.deb .
pharos@PhaROS:~$ sudo dpkg -i pharos.deb
```

Pharos installation tool help output

```
pharos usage
  pharos install              - Installs a package
  pharos create               - Creates a new package based on an
archetype
  pharos create-repository    - Creates a smalltalk script for
creating your own repository
  pharos register-repository  - Register a new package repositry
  pharos list-repositories    - List registered repostiries
  pharos update-tool          - Check if there is any update of the
tool, and it update it.
  pharos ros-install          - Installs a ros
```

C.2 Installing ROS with Pharos installation tool

Pharos installation tool for ROS installation

```
pharos@PhaROS:~$ pharos ros-install --help
```

Pharos installation tool for ROS installation

```
usage: pharos ros−install
It installs a ROS in the machine. (Valid just for ubuntu).

Options:

−−version          version of ROS. { fuerte | groovy | hydro }. Default: groovy.
−−type−installation Type of ros installation { desktop | desktop−full | ros−base }.
   Default: desktop.
−−configure     Indicates if the installation should be configured (modifiy .bashrc file )
   .{true|false} Default: true.
−−create−workspace  Indicates if a workspace should be created. It will create a
   workspace at ~/ros/worskpace. { true|false }. Default: true.
```

The tool propose us several configurations, and an easy way to install by default (which is quite good by default)

Version: Specify the distribution of ROS we want to install (It should be compatible with our Ubuntu installation). By default it will try with Groovy.

Type-installation: Specify what do we want to load. Ros-base will load just the architecture things, desktop will download several graphic tools and some stacks, desktop-full even more tools and stacks. (Check in the wiki site of each distribution to know what will be downloaded if you need more information) By default it will download desktop.

Configure: Is a boolean value. If you are beginner, just left it with the default. It will modify .bashrc file of the current user, in order to have ROS always available.

Create-Workspace: Is a boolean value. If you are beginner, just left it with the default. In order to develop we need to have a place to place our code. This place is called workspace. And if this value is true, the tool will make up a new one in the standard location.

This command is safe for installing because it will tell you if the ROS that you want to install is compatible with your Ubuntu installation.

Running in an ubuntu 13.04 (Raring) , lets try to install a Fuerte distribution.

Pharos installation tool Installing incompatible version

```
pharos@PhaROS:~$ pharos ros−install −−version=fuerte
```

Pharos installation tool Installing incompatible version - Output

```
fuerte version is not available for: raring
```

In order to have a rich example, i will install an hydro version in my raring ubuntu installation.

Pharos installation tool Installing Hydro in Ubuntu: Raring

```
pharos@PhaROS:~$ pharos ros−install −−version=hydro
```

The output of this script is shown at section 3.3

Appendix D

PhaROS Cheat sheet

Welcome to PhaROS!
Remember always to check for updates in http://car.mines-douai.fr/squeaksource/PhaROS.html.

Running the generated package

The generated package has implemented some scripts to show you the basic usage of PhaROS. In PhaROS a package has scripts. Each scripts represents a Node. Each of these nodes share the same class and code. So, take that in count when you implement a new script.

Basic ROS commands for your generated package

For browse your package trough command line.

roscd name-of-your-package

For editing the image

rosrun name-of-your-package edit

For running a script with graphical interface

rosrun name-of-your-package pharos script-name

For running a script without graphical interface

rosrun name-of-your-package headless script-name

For listing your scripts

rosls name-of-your-package/image/scripts

D.1 Inside a package object

Creating a new Script
For creating a new script you need to add to your package class a method named scriptNameOfYourScript. Inside this method you will have available a controller, which is an object that gives construction facilities and access to an already built node. self controller node.

Inside this method you configure the given node and trigger all the logic of your node. (So, from ROS point of view, each script is a node)

In order to make this script available for execution you have two possibilities:

- write a text file that executes this method (PackageName new scriptNameOfYourScript.) and save it into packageFolder/image/scripts
- commit all your code and use the pharos tool to install it back, this will generate all the needed files. (we are working for making this step easier)

For examples of what an script is, browse the generated example package and match names with the names available in the script folder. (roscd package-name/image/scripts).

Publish topic

```
publisher := self controller node
                topicPublisher: '/example/string'
                typedAs: 'std−msgs/String'.

publisher send:[ : string | string data: 'this is an example' ].
```

Subscribe topic

```
        typedAs: 'std−msgs/String';
```

```
        for: [ : string |  Transcript show: string data ];
        connect .
```

Call service

```
service := self controller node service: '/rosout/get−loggers'.
service call.
\end{code }


\paragraph{Define service \newline}

\beign{code}

self controller node serve: [ :req :rsp |
    Transcript show: 'Service has been called.'; cr.
] at: '/pharos/service' typedAs:'roscpp/Empty'.
```

Inject/install a nodelet

Specifying controller configuration

In the package object implement the message #buildController. Build controller has the responsibility to build the controller and return it.

For building your own controller

```
    ^ MyController build

buildController
    \^ self myControllerConfigurationMethod: super buildController.

myControllerConfigurationMethod: aController
    << Make here your configurations >>
    \^ aController
```

Define a new type

Define as class method a method with a cool name, as myCoolTypeDefinition

```
^ PhaROSCompositeType named: 'anStandarROS/TypeName' definedBy: {
    #header −> (PhaROSTypeBrowser instance definition: 'std−msgs/Header').
    #auint8 −> (PhaROSUInt8Type new).
    #auint16 −> (PhaROSUInt16Type new).
    #aint32 −> (PhaROSInt32Type new).
    #afloat32 −> (PhaROSFloat32Type new).
    #afloat64 −> (PhaROSFloat64Type new).
    #astring −>( PhaROSStringType new ) .
    #atime −> (PhaROSTimeType new ).
}  withConstants: {
    #CONSTANT −> ASimpleObjectValue
}.
```

As shown in the definition you give an array of associations with (#nameOfTheField -> Type new). For checking all the available types, just browse any of this classes to go to the package. Or check the reference.

Constants values cannot be complex. Just numbers, strings, booleans.

Register a type
Define in class side of your package the method #types

```
^ super types, { #YourTypeName −> self myCoolTypeDefinition }.
```

In order to deploy the type into ROS you will need to commit all your work and install it through the pharos command (as shown in the shell commands section). We are working to enhance this step.

D.2 Shell commands

Install PhaROS based Package
pharos install PACKAGE [OPTIONS]

Example
pharos install esug –location=/home/user/ros/workspace –version=2.0
Help pharos install –help

Create PhaROS based Package
pharos create PACKAGE [OPTIONS]

Example
pharos create –location=/home/user/ros/workspace –version=2.0 –author=YourName –author-email=YourEmail
Tip: Be sure the email is a correct one. If is not a correctly spelled one you will notice during last step.
Help
pharos create –help

Register Repository of packages
pharos register-repository –url=anUrl –package=aPackage [OPTIONS]

Example
pharos register-repository –url=http://smalltalkhub.com/mc/user/YourProject/main –package=YourProjectDirectory –directory=YourProjectDirectory
Tip: If your repository requires user/password for reading add –user=User –password=Password to the example.
Disclaimer: User/Password will be stored in a text file without any security.
Help pharos register-repository –help

Listing registered repositories
pharos list-repositories

Creating a directory for your own project repository
pharos create-repository PACKAGENAME [OPTIONS]

Example
pharos create-repository example –user=UserName > directory.st
pharos create-repository example –user=UserName –output= directory.st

Help
pharos create-repository –help