Casal2 Contributors Guide

C. Marsh and S. Rasmussen

Contents

1. Introduction

The Contributors Guide provides an overview for those who wish to contribute to the Casal2 source code and compile the latest version of Casal2. Casal2 is open source and the Development Team encourages users to submit and contribute changes, bug fixes, or enhancements. This document is intended to provide a guide for those who wish to undertake the these tasks.

The Casal2 source code is hosted on GitHub, and can be found at `https://github.com/NIWAFisheriesModelling/CASAL2`.

A release bundle which includes a binary (both windows and Linux operating systems), manual, examples, **R** package and other help guides can be downloaded at `https://github.com/NIWAFisheriesModelling/CASAL2/releases`.

Casal2 release versions on the front page of the repository can lag behind the latest source code because uploading these is manual process. It is possible to get an executable for both windows and Linux (Ubuntu 20) using the GitHub actions artefacts see here. To obtain these, log into your GitHub account then click on the latest successfully checked commit (indicated by a green tick), scroll to the bottom and download either `Casal2-Linux-build` or `Casal2-Windows-build`. These files are only saved for 30 days so if there hasn't been a commit in a while, they may not be available.

To help maintain the quality of the code base, the development team has created this guide to assist contributors in understanding how to access the source code, how to compile and build, and the guidelines for submitting code changes to the Casal2 Development Team.

This guide covers basics such as setting up a GitHub profile, to using GitHub, and all the way to compiling and modifying source code.

As a suggestion to contributors, it may be helpful to contact the Development Team before you begin making the change. They can assist with ideas on how best to add functionality. Either contact them directly or open an issue at `https://github.com/NIWAFisheriesModelling/CASAL2/issues`.

If you have any questions or require more information, please contact the Casal2 Development Team at casal2@niwa.co.nz.

2. Creating a local repository

This section covers the following:

1. Registering a GitHub username

2. Download git software

3. Fork the master repository

2.1. Git username and profile

The first step is to create a username and profile on GitHub (if you do not already have one). Creating a username and profile on GitHub is free and easy to do.

Go to `https://www.github.com` to register a username and set up a profile if you do not already have one. See the help at GitHub for more information. Once you have set up a GitHub account, you need to download the git software (see next section) so you can push and pull changes between your local repository and the main online repository.

2.2. Git software

You will need to acquire a git client in order to clone the repository to your local machine.

Casal2 also requires a command line version of git in order to compile. The Casal2 build environment requires `git` in order to evaluate the version of the code used at compile time to include into the executable, manuals, etc. when being built.

You will need to download the git client from git software and you will need to make sure that it has been included into your system path. This can be checked by opening a terminal (powershell or command prompt and typing in `git -v`. Git is a command line program, you can also download a GUI interface which helps using git a lot more user-friendly. My personal preference is the github desktop application.

2.3. Cloning a repository

The publicly available Casal2 code is in the master repository. Only the Casal2 Development Team have permission to add, delete, or change code directly to the master repository.

To clone the master repository, navigate to `https://github.com/NIWAFisheriesModelling/CASAL2` and use the green code button, which is shown below to clone. There are multiple protocols (https, ssh etc.) for cloning highlighted in the red box. We recommend using the GitHub desktop method which is circled in Figure 2.1.
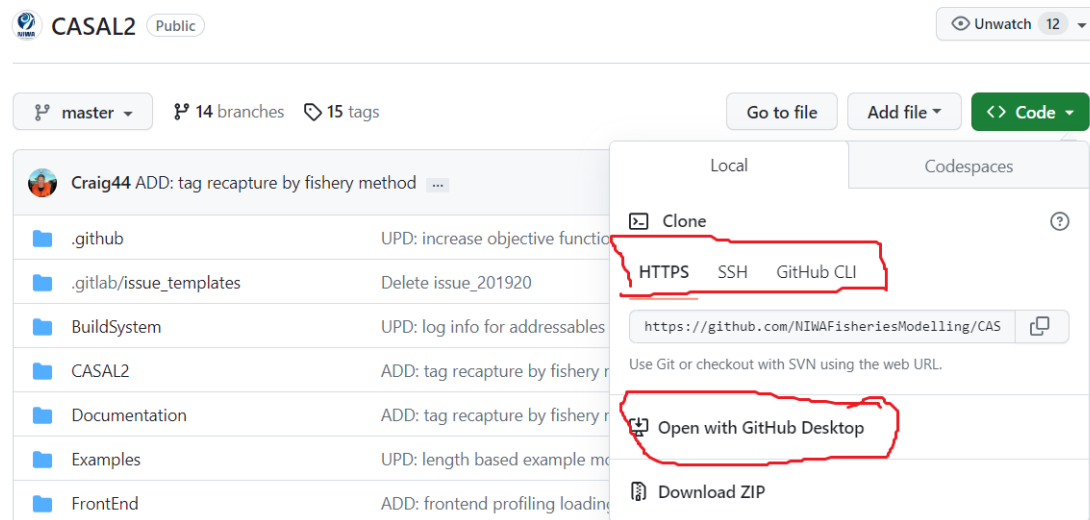
Figure 2.1: Cloning a repository

2.4.   Adding an SSH authentication key to your account on GitHub.com

One of the reasons contributors may have difficulty pushing and pulling changes to Casal2 is because they don't have a SSH authentication key linked to their GitHub account. The error message may look like that shown in Figure 2.2. Please see the information from this link for detailed instructions on how to create a public SSH key on your computer and then link it to your GitHub account.

Figure 2.2: Example of the error message when SSH authentication has not been enabled

3

3.   Create a branch to start making changes

This section covers how to create a branch and the importance of the naming convention that is used to track all the development.

If you are using git client from the command prompt then creating a branch is easy by using the following command `git -b <BranchLabel>`. The `BranchLabel` has a specific naming convention, which is `BranchLabel = simpledescription_YYYYMM`. For example `git -b SplineSelectivity_202307` which is adding or fixing the spline selectivity class which started in July of 2023. This makes it easier for the Development Team what is in the branch and how active it is.

Once you have created a branch you can switch between the master and branch using the `git checkout` command. To find out how to merge changes from the master into your branch just google "git merging master into feature branch" and you will find many useful resources.

Casal2 Once you do your first push to the branch it you will then be able to see it under the branches tab on the online master repository (See Figure 3.1). If you have trouble pushing the branch you may need one of the Casal2 developers to add you as a collaborator to the project. This would mean someone has to log into the GitHub account for the repository, go to settings tab, and add the user as a collaborator (left tab panel).

Figure 3.1: Example of choosing the branch in GitHub

4. Compiling Casal2

This section describes how to set up the environment on your local machine that will allow you to build and compile Casal2. The build environment can be on either Microsoft Windows or Linux systems. At present the Casal2 build system supports Microsoft Windows 10+ and Linux (with GCC/G++ 4.9.0+). Apple OSX or other platforms are not currently supported.

4.1. Overview

The build system is made up of a collection of Python scripts that do the various tasks. These are located in `CASAL2/BuildSystem/buildtools/classes/`. Each Python script has its own set of functionality and undertakes one set of actions.

The top level of the build system can be found at `CASAL2/BuildSystem/`. In this directory you can run `doBuild.bat help` from a command terminal in Microsoft Windows systems or `./doBuild.sh help` from a terminal in Linux systems.

The script will take one or two parameters depending on what style of build you want to undertake. These commands allow the building of various stand-alone binaries, shared libraries, and the documentation. Note that you will need additional software installed on your system in order to build Casal2. The software requirements are described below.

A summary of all of the doBuild arguments can be found using the command `doBuild help` in the BuildSystem directory.

The arguments to doBuild are:

Usage: doBuild <build_target> <argument>

 `help` Print out the doBuild help (this output)

 `check` Do a check of the build system

 `clean` Remove debug/release build files

 `clean_all` Remove all build files and ALL prebuilt binaries

 `version` Build the current version files for C++, R, and LaTeX

Build required libraries (DLLs/shared objects for Casal2)

 `thirdparty` Build the third party libraries

      <option> Optionally specify the target third party library to build, either adolc or betadiff (default is none)

Build development and test versions (for development builds only)

 `release` Build stand-alone release executable

      <option> Optionally specify the target third party library to build, either adolc or betadiff (default is none)

 `debug` Build stand-alone debug executable

<option> Optionally specify the target third party library to build, either adolc or betadiff (default is none)

`test` Build stand-alone unit tests executable

`unittests` Run the unit tests (requires that 'test' has been built)

`modelrunner` Run the test case models

Build the Casal2 end-user application

`library` Build shared library for use by front end application

<argument> Required argument to specify the target library to build: release, adolc, betadiff, or test

`frontend` Build Casal2 front end application

Create the archive, R Library, documentation, and the installers

`documentation` Build the Casal2 user manuals

`rlibrary` Create the R library

`archive` Create a zipped archive of the Casal2 application.

<true> if specified build skips everything but frontend

`installer` Create the Microsoft Windows installer package

`deb` Create Linux Debian installer

The outputs from the build system commands will be placed in sub-folders of `CASAL2/ BuildSystem/bin/<operatingsystem>/<build_type>`

For example:

`CASAL2/BuildSystem/windows/debug`

`CASAL2/BuildSystem/windows/library_release`

`CASAL2/BuildSystem/windows/thirdparty/`

`CASAL2/BuildSystem/linux/library_release`

The files `Casal2_build.bat` for Windows and `Casal2_build.sh` for Linux in the root folder contain all the calls in the correct order of `doBuild` required to successfully build Casal2, the documentation, the Windows installer (Windows) or the Debian installer (Linux), the R-Libraries, and run all the test cases and unit tests.

4.2. Building on Microsoft Windows

4.2.1. Prerequisite software

The building of Casal2 requires additional build tools and software, including Python, git version control, GCC compiler, LaTeX compiler, and a Windows package builder. Casal2 can require specific implementations of these packages and versions in order to build without modifying the build scripts.

C++ and Fortran compiler

Source: tdm-gcc (MinGW-w64) from `https://jmeubank.github.io/tdm-gcc/`.

Casal2 is designed to compile under GCC on Microsoft Windows and Linux. While it may be possible to build the package using different compilers, the Casal2 Development Team does provide any assistance or recommendations. We recommend using 64-bit TDM-GCC with a version of at least 10.3.0. Ensure you have the "fortran" and "openmp" options installed as a part of the "gcc" install option drop-down tick boxes as these are required. For example, from `https://jmeubank.github.io/tdm-gcc/articles/2021-05/10.3.0-release`, select the 64+32-bit MinGW-w64 edition, then select the Custom install and tick all boxes.

Note that a common error that can be made is having a different GCC compiler in your path when attempting to compile. For example, `rtools` includes a version of the GCC compiler. We recommend removing these from your path prior to compiling.

GIT version control

Source: Command line GIT from `https://www.git-scm.com/downloads`.

Casal2 automatically adds a version details to its files and any output based on the GIT version of the latest commit to its repository. This includes the name of source repository that was used. The command line version of GIT is used to generate the version details.

MiKTeX LaTeX Processor

Source: Portable version from `http://www.miktex.org/portable`.

The main user documentation for Casal2 is a PDF document generated from LaTeX. The LaTeX syntax sections of the documentation are generated, in part, directly from the code. In order to generate the user documentation, you will need the MiKTeX LaTeX compiler.

A number of additional LaTeX styles are used — these will usually be identified doing the doBuild process and can be installed as required.

7-Zip

Source: 7-Zip from `http://www.7-zip.org/download.html`.

The build system calls `7zip.exe` to unzip files in the build system; it is advised to have this in the path.

Python

Source: Python3 from `https://www.python.org/downloads/windows/`

Python is used to run the build scripts and set the required environment variables required to build Casal2.

Python modules

There are a number of Python3 modules that are required to build Casal2. These can be installed with `python -m pip install module-name`. For example, You may need to install `datetime`, `re`, and `distutils` Python modules.

7

Inno setup installer build (optional)

Source: Inno Setup 5 from `http://www.jrsoftware.org/isdl.php`

If you wish to build a Microsoft Windows compatible Installer for Casal2 (recommended) then you will need the 'Inno Setup 5' application installed on the machine. The installation path must be `C:\ProgramFiles(x86)\InnoSetup5\` in order for the build scripts to find and use it.

4.2.2.   Pre-build requirements

Prior to building Casal2 you will need to ensure you have both G++ and GIT in your path. You can check both of these by typing the following commands and checking that they return the correct version number:

```
g++ --version
```

```
git --version
```

This also allows you to check that there are no alternative versions of a GCC compiler that may confuse the Casal2 build. It's also worth checking to ensure GFortran has been installed with the G++ compiler by typing:

```
gfortran --version
```

If you wish to build the documentation, `bibtex` will also need to be in the path, e.g., to check, try:

```
bibtex --version
```

4.2.3.   Building Casal2

The build process is relatively straightforward. Before you start the build process, you can run `doBuild check` from the command prompt to check if your build environment is complete. Make sure that you are within `CASAL2/BuildSystem/` to run `doBuild`.

`doBuild check` will summarise Windows environment PATH as a part of its output, and this can be used to check that the paths for g++ and gfortran and the g++ point to where the correct version of GCC is installed.

The build process is as follows:

1. Download a clone of the code on your local machine

2. Navigate to the BuildSystem folder in `CASAL2/BuildSystem`

3. You need to build the third party libraries with the following commands from the command prompt:

    - `doBuild thirdparty`

4. You need to build the binary you want to use:

    - `doBuild release`

5. You can build the documentation if you want:

    - `doBuild documentation`

4.3. Building on Linux

This guide has been written against a fresh install of Ubuntu 20.04. With Ubuntu we use apt-get to install new packages. You'll need to be familiar with the package manager for your distribution to correctly install the required prerequisite software. For this you will require administrator level access.

4.3.1. Prerequisite software

G++ compiler

If gfortran is not installed, install this with: `sudo apt-get install gfortran`.

GIT version control

Git may not be installed by default and it can be installed with `sudo apt-get install git`

Casal2 automatically adds a version details to its files and any output based on the GIT version of the latest commit to its repository. This includes the name of source repository that was used. The command line version of GIT is used to generate the version details.

CMake

CMake is required to build multiple third-party libraries and the main code base. You can do this with `sudo apt-get install cmake`

Python

Python3 is used to run the build scripts and set the required environment variables required to build Casal2. This is usually installed by default on Linux systems, but if not, it can be installed using: `sudo apt-get install python3`

Python modules

There are a number of Python3 modules that are required to build Casal2. These can be installed with `sudo apt-get install module-name`. For example, You may need to install `datetime`, `re`, and `distutils` Python modules.

LaTeX

LaTeX on Linux is required, and the Texlive LaTeX Processor is recommended. This can be installed with:

`sudo apt-get install texlive-binaries` `sudo apt-get install texlive-latex-base` `sudo apt-get install texlive-latex-recommended`   `sudo apt-get install texlive-latex-extra`

Alternatively you can install the complete package with `sudo apt-get install texlive-full`

A number of additional LaTeX styles are used — these will usually be identified doing the doBuild process and can be installed as required.

4.3.2.   Building Casal2

The build process is relatively straightforward. You can run `./doBuild.sh check` to see if your build environment is ready.

1. Download a clone of the code on your local machine

2. Navigate to the BuildSystem folder in `CASAL2/BuildSystem`

3. You need to build the third party libraries with:

    * `./doBuild.sh thirdparty`

4. You need to build the binary you want to use:

    * `./doBuild.sh release`

5. You can build the documentation:

    * `./doBuild.sh documentation`

4.4.   Troubleshooting

4.4.1.   Third-party C++ libraries

It's possible that there will be build errors or issues building the C++ third-party libraries. If you encounter an error, then check the log files to locate the source of the problem. Each third-party build system stores a log of everything that was done. The files will be named

* casal2_unzip.log

* casal2_configure.log

* casal2_make.log

* casal2_build.log

* ... etc,.

Some of the third-party libraries require very specialised environments for compiling under GCC on Windows. These libraries are packaged with MSYS (MinGW Linux style shell system). The log files for these will be found in `ThirdParty/<libraryname>/msys/1.0/<libraryname>/`

e.g., `ThirdParty/adolc/msys/1.0/adolc/ADOL-C-2.5.2/casal2_make.log`
e.g., `ThirdParty/boost/boost_1_58_0/casal2_build.log`

A common issue when running doBuild thirdparty are Python error messages about missing modules, e.g., ModuleNotFoundError: No module named 'dateutil'. This type of error message indicates that a Python module (library) is missing and will need to be installed. For instance, to install the 'dateutil' module, type the following into a command prompt or terminal window: pip3 install python-dateutil.

4.4.2. Main code base

If the unmodified code base does not compile, the most likely cause is an issue with the third-party libraries not being built correctly. As updates and revisions are outside the control of the Development Team, problems can arise that may require the developers of the third party libraries to resolve first. However, versions of these libraries are included in the Casal2 source code and these should work. For any specific issues contact a local expert with regard to your specific system environment, or else the Casal2 Development Team for help.

5.   Casal2 build guidelines and validation

5.1.   Casal2 coding practice and style

Casal2 is written in C++ and uses the Google C++ style guide (see `https://google.github.io/styleguide/cppguide.html`).

In general when editing or writing code for Casal2

1. Using consistent indentations inside functions and loops, and descriptive and human readable variable or function names.

2. Use of the characters '_' on the end of class variables defined in the .h files.

3. Annotate and comment the code, especially where it would help another contributor understand the program logic and rationale.

4. Add descriptive log messages to aid in debugging and checking of the program logic flow.

5. Implement unit tests, internal models, and external models to test and validate the new or changed functionality.

6. Document the functionality in the Casal2 User Manual(s).

Casal2 allows printing of logging messages art runtime using the –loglevel command line argument. The levels of logging in Casal2 are:

- LOG_MEDIUM() usually reserved for iterative functionality (e.g. estimates during estimation phase)

- LOG_FINE() the level of reporting between an actual report and a fine scale detail that end users are not interested in (Developers)

- LOG_FINEST() Minor fine scale details within a function or routine.

- LOG_TRACE() put at the beginning of every function

e.g., to run Casal2 with logging, use

```
casal2 -r -loglevel finest > my_run.log 2> my_run.err
```

This will output all the logged information to `my_run.err`.

5.2.   Units tests and model validation

The Casal2 development places an emphasis on maintaining software integrity and reproducibility between revisions. Casal2 uses model validations and built in unit tests to validate and verify the code each time Casal2 is compiled and built.

There are three different validation approaches in Casal2. These are:

1. Mocking specific classes.

2. Implementing internal models (implemented in C++ source code with extension `.Test.cpp`) that have variable test cases for specific classes.

3. Implementing externally run models (found in the TestModel folder) that are validated to generate expected output.

To implement mocking of classes and internal models, Casal2 uses the Google testing framework and the Google mocking framework.

To implement testing of full models, input configuration files are run using the compiled Casal2 binaries, and the output compared with expected output using @assert commands.

5.2.1. Mocking specific classes

Classes are unit tested using unit tests that are a part of the source code. These are designed to check the components of the code to validate that functions provide expected output. These unit tests are run each time Casal2 is compiled.

When adding unit tests, they should to be developed and tested outside of Casal2. This gives confidence that the test does not contain any calculation errors.

Mocking specific classes is used to validate specific functionality and is encouraged because it is the easiest to isolate simple errors that may be introduced with code changes.

As examples, see (i) the file `VonBertalanffy.Test.cpp` mocks the von Bertalanffy age-length class and tests the mean length calculation, and (ii) the `Partition` class has the file `Partition.Test.cpp` that validates user inputs and model expectations.

5.2.2. Internal mocking of simple models

Mocking of simple models is done using a number of internal models. Most of the functionality for implementing these are in the source folder `/CASAL2/source/TestResources`.

These implements simple models and run test cases with differing class implementations by running an internal empty model and testing the output of classes from a model run.

As examples, see (i) the `LogNormal.Test.cpp` in the `Projects` class that test the lognormal distribution when used for projections, and (ii) the `TagByLength` process in `TagByLength.Test.cpp` that tests functionality of the tagging process.

5.2.3. External testing using test models

External tests are run following compilation using the Python modelrunner.py scripts (i.e., using the `DoBuild modelrunner` script in the BuildSystem folder). These models are used to test model runs, minimisation routines, and MCMC output.

The test model input configuration files are located in the `TestModel` folder and the command calls to run these are in the modelrunner.py script.

Contributors are encouraged to add additional models to the list of test models as these be used to validate the combined functionality of a range of interrelated commands and subcommands in Casal2.

5.3. Verification

After Casal2 executes Validate and Build it runs sanity checks in the verify state. These are business rules that can be checked across the entire system. This can be useful to suggest dependencies or configurations. For an example see the directory `Processes\Verification\` in the source code.

5.4. Reporting (optional)

Currently Casal2 has reports that are **R** compatible, i.e., all output reports produced by Casal2 can be read into **R** using the standard **CASAL2 R** package. If you create a new report or modify an old one, you most follow the standard so that the report is **R** compatible.

All reports must start with, `*label (type)` and end with, `*end`

Depending on what type of information you wish to report, will depend on the syntax you need to use. For example

{d} (Dataframe)

Report a dataframe

```
*estimates (estimate_value)
values {d}
process[Recruitment_BOP].r0 process[Recruitment_ENLD].r0
2e+006 8e+006
*end
```

{m} (Matrix)

Report a matrix

```
*covar (covariance_matrix)
Covariance_Matrix {m}
2.29729e+010 -742.276 -70160.5
-110126 -424507 -81300
-36283.4 955920 -52736.2
*end
```

{L} (List)

Report a List

```
*weight_one (partition_mean_weight)
year: 1900
ENLD.EN.notag {L}
mean_weights {L}
0.0476604 0.111575 0.199705
end {L}
age_lengths {L}
12.0314 16.2808 20.0135
end {L}
end {L}
*end
```

5.5.   Update the Casal2 User Manual(s)

Contributors will need to add or modify sections of the user manual(s) to document their changes. This includes the section that describes the methods and the section where the specific syntax is defined.

5.6.   Builds to pass before merging changes

Once you have made changes, you must run the following before your changes can be included in the master code.

- Build the unittest version. See Section 4 for how to build unittest depending on your system.

- Run the standard and new unit tests to check that they all pass. To do this first compile the test executable using the script `DoBuild test`. Then move to the directory with the location of the executable (`BuildSystem/bin/OS/test`) and run it (open a command terminal and run `casal2`) to check all the unit-tests pass.

- Test that the debug and release of Casal2 compiles and runs with `DoBuild debug`

- Run the second phase of unit tests (requires that the debug version is built). This runs the tests that comprise of complete model runs using `DoBuild modelrunner`

- Build the archive using `DoBuild archive` which builds all required libraries. There are small nuances between Double classes, especially when reporting the class that mean seemingly simple changes can sometimes cause a break in the full build.

6.   Adding functionality into the Casal2 C++ code

The following shows a sequence of figures and text of an example, where we demonstrate how to modify the Casal2 C++ code by adding a new child process called SurvivalConstantRate. Although we will be showing adding new process functionality for a process we will explain it in a generalised way so that this can be translated into adding a new selectivity, observation, likelihood, Projection, MCMC, minimiser, time varying class etc.

To add a new process you start by going into the `CASAL2\CASAL2\source\processes\`.

Figure 6.1: Example of how the main C++ process classes are structured in Casal2

This folder shown in Figure 6.1 contains a general layout of all main classes in Casal2. It contains a folder named `Age`, `Length`, `Common`, `Verification`, `Factory.cpp`, `Factory.h`, `Manager.cpp`, `Manager.h`, `Process.cpp`, `Process.h`. The folder `Age` contains all the current age-based processes implemented, `Length` contains all the current length-based processes and `Common` contains all processes that can be used in both age or length models. The `Factory` source code creates all the processes during runtime. The `Manager` source code manages the class; it can build pointers to specific processes to share information across classes. The `Process` source code contains base functionality that is inherited by all child classes (inheritance is a major concept in C++ and it is advised have some knowledge of the concept). Before creating a new child look at the parent (in this example `Process.cpp`, `Process.h`), to see the functionality you don't have to add in your child because it is already done at the parent level. The following figure shows the constructor in `Process.cpp`.

From Figure 6.2 the process parent class does not do much — it assigns a label and type for each child that will be created and a report subcommand. The point of this is to look at the parent class to reduce duplicating functionality in the child class.

Figure 6.2: Example of a C++ parent process class for SurvivalConstantRate

We will be returning to the factory source code later, but for now let's get adding the new process. Enter the `Children` folder and create C++ source code labelled `SurvivalConstantRate.cpp` and `SurvivalConstantRate.h` (See Figure 6.3). These files were should exist in the source code, but fell free to delete them and go through the process of adding the process to see how easy this process is. One way to start is by copying an existing process, renaming it, and using that as the basis for changes.

Figure 6.3: Example showing the C++ child process classes for SurvivalConstantRate

Once you have done that for each child class you may have to write code for the following functions: `DoValidate()`, `DoBuild()`, `PreExecute()`, `DoExecute()`, `PostExecute()`, `DoReset()`. To understand what these all do Figure 6.4 shows the state transition of Casal2.

Figure 6.4: Diagram of the state transitions of Casal2.

Now we describe the purpose of each transition and this will give you an idea of what you should be incorporating into each function.

StartUp

The model is in the blank start and the configuration system is loading the configuration files and parsing any extra inputs.

Tasks completed:

- Parse command line

- Parse configuration file

- Load plugins

- Load estimate values from input files

Validate

All user configurations have been loaded at this point. Now the model will go through every object that has been created and check that the parameters given to them.

This step will ensure every object in the model has sufficient parameters to be executed without causing a system fault or error in the model.

This state will not check the values to ensure they are logical in relation to an actual model. They will only test that they exist and meet minimum requirements to execute a model.

At the end of the validate stage each object should be internally consistent. No lookups or external references are allowed to be formed during this stage.

Build

The build phase is where the system will build relationships between objects that rely on each other. Because validation has been completed, each object in it's self-contained configuration is okay.

This phase generally assigns values to pointers for objects so they don't need to do lookups of objects during execution phases.

Verify

At this point pre-defined configurations are checked against the model's current configuration to verify if the model makes sense logically. These are business rules being applied to the model to help ensure the model is "sensible" e.g. you have an ageing and a mortality process in your annual cycle.

PreExecute

Pre-Execution happens at the beginning of a time step. This allows objects to calculate values based on the partition state before any of the other processes in the time step are executed.

Execute

This is the general work method of the model and where all of the processes will be run against the partition.

PostExecute

This is executed at the end of a time step after all of the processes and associated objects have been executed. This is typically used for things like reports and derived quantities.

IterationComplete

This is executed at the end of every model run. This is only useful when the model is in a multiple-iteration mode (e.g MCMC or estimation). After every model iteration this state is triggered.

Reset

If the model has to run multiple iterations then the reset state is used to reset everything back in to a state where the model can be re-executed without any legacy data remaining.

This state allows us to run multiple iterations of the model without having to re-process the configuration information or de-allocate/re-allocate large amounts of memory.

Finalise

Finalise will happen after all iterations of the model have been completed.

Coming back to our example, if we look in `SurvivalConstantRate.h` we will see the following setup, as shown in Figure 6.5.

We can see that we are implementing `DoValidate()`, `DoBuild()`, `DoExecute()`, `DoReset()`, which are public functions and we define our variables as private. This idea of structuring classes by

public, private and protected is known as encapsulation, and is another important C++ concept. Our variables will be stored in memory for the entire model run so you must make decisions on whether a variable needs to persist for the entire model run or can be temporarily created and destroyed at execution time. Note: the '_' at the end the variable names are used (by convention) to indicate a variable defined in the .h file.

```cpp
// namespaces
namespace niwa {
class Selectivity; // Forward declare Selectivity class

namespace processes {
namespace accessor = niwa::partition::accessors;
using utilities::OrderedMap;
/**
 * Class Definition
 */
class SurvivalConstantRate : public niwa::Process {
public:
  // Methods
  SurvivalConstantRate(Model* model);
  virtual                 ~SurvivalConstantRate() = default;
  void                    DoValidate() override final;
  void                    DoBuild() override final;
  void                    DoReset() override final;
  void                    DoExecute() override final;

private:
  // Members
  vector<string>          category_labels_;
  vector<Double>          s_input_;
  OrderedMap<string, Double>  s_;
  vector<Double>          ratios_;
  map<unsigned, Double>   time_step_ratios_;
  vector<vector<Double>>  survival_rates_;
  vector<string>          selectivity_names_;
  accessor::Categories    partition_;
  vector<Selectivity*>    selectivities_;
};
```

Figure 6.5: Example showing the definitions of the Casal2 standard functions in the class header file

Moving into SurvivalConstantRate.cpp, if we look at the constructor, we can see that this describes input parameters that are supplied by the user, and which variables are estimable. This is applied by the RegisterAsEstimable() function (See Figure 6.6).

```cpp
SurvivalConstantRate::SurvivalConstantRate(Model* model)
 : Process(model),
   partition_(model) {
  LOG_TRACE();
  process_type_ = ProcessType::kMortality;
  partition_structure_ = PartitionStructure::kAge;

  parameters_.Bind<string>(PARAM_CATEGORIES, &category_labels_, "List of categories", "");
  parameters_.Bind<Double>(PARAM_S, &s_input_, "Survival rates", "");
  parameters_.Bind<Double>(PARAM_TIME_STEP_RATIO, &ratios_, "Time step ratios for S", "", true);
  parameters_.Bind<string>(PARAM_SELECTIVITIES, &selectivity_names_, "Selectivity label", "");

  RegisterAsEstimable(PARAM_S, &s_);
}
```

Figure 6.6: Example showing the `RegisterAsEstimable()` function in the constructor

When you see keywords that begin with PARAM, for example PARAM_CATEGORIES, these will be defined in CASAL2\CASAL2\source\English_UK.h.

Moving on to the DoValidate() function in Figure 6.7. The purpose of this function is to validate the user's inputs and expand inputs to allow short hand syntax. You can see the annotation of what is happening in the function. This is considered one of the coding commandments **Thou shall annotate thy code**.

```cpp
/**
 * Validate our Survival Constant Rate process
 *
 * - Validate the required parameters
 * - Assign the label from the parameters
 * - Assign and validate remaining parameters
 * - Duplicate 's' and 'selectivities' if only 1 vale specified
 * - Check s is between 0.0 and 1.0
 * - Check the categories are real
 */
void SurvivalConstantRate::DoValidate() {
  category_labels_ = model_->categories()->ExpandLabels(category_labels_, parameters_.Get(PARAM_CATEGORIES));

  // If one S supplied expand for each category
  if (s_input_.size() == 1)
    s_input_.assign(category_labels_.size(), s_input_[0]);
  // Do the same for selectivity labels
  if (selectivity_names_.size() == 1)
    selectivity_names_.assign(category_labels_.size(), selectivity_names_[0]);
  //Check we have equal category labels as survival rates
  if (s_input_.size() != category_labels_.size()) {
    LOG_ERROR_P(PARAM_S)
        << ": Number of Ms provided is not the same as the number of categories provided. Expected: "
        << category_labels_.size()<< " but got " << s_input_.size();
  }
  //Check we have equal category labels to selectivity labels
  if (selectivity_names_.size() != category_labels_.size()) {
    LOG_ERROR_P(PARAM_SELECTIVITIES)
        << ": Number of selectivities provided is not the same as the number of categories provided. Expected: "
        << category_labels_.size()<< " but got " << selectivity_names_.size();
  }

  // Validate our S's are between 1.0 and 0.0
  for (Double s : s_input_) {
    if (s < 0.0 || s > 1.0)
      LOG_ERROR_P(PARAM_S) << ": s value " << AS_DOUBLE(s) << " must be between 0.0 and 1.0 (inclusive)";
  }
  // Assign survival rates to a map s_
  for (unsigned i = 0; i < s_input_.size(); ++i)
    s_[category_labels_[i]] = s_input_[i];

  // Check categories are real
  for (const string& label : category_labels_) {
    if (!model_->categories()->IsValid(label))
      LOG_ERROR_P(PARAM_CATEGORIES) << ": category " << label << " does not exist. Have you defined it?";
  }
}
```

Figure 6.7: Example of the validations in the `DoValidate()` function

Another nice inclusion to the Casal2 source code is it's error checking and messages to the users. The following list is of all the allowable types of errors that Casal2 can perform.

- `LOG_WARNING()` « `"warning message"`; if triggered print the warning message at the end of model run to, warn the user of a change or implication in the model.

- `LOG_ERROR()` « `"error message"`; After validate and build stop execution and print error message.

- `LOG_ERROR_P(PARAM_KEYWORD)` « `"error message"`; After validate and build stop execution and print error message with location of the parameter `PARAM_KEYWORD` in the configuration files.

- `LOG_FATAL()` if triggered halt execution and print error message.

- `LOG_FATAL_P(PARAM_KEYWORD)` « `"error message"`; If triggered stop execution and print error message with location of the `PARAM_KEYWORD` in the configuration files.

- `LOG_CODE_ERROR()` « `"error message"`; if triggered quit Casal2 with a message like contact the development team this is a bug in the software.

We encourage you to put these all throughout the validate and build functions, to help users correctly specify models and catch silly input sequences.

Moving on to the DoBuild() function. The purpose of the build function is to create pointers and relationships with other classes that will be required during execution. For example, in this process we will require access to the partition, time step and selectivity classes.

```cpp
/**
 * Build any runtime relationships
 * - Build the partition accessor
 * - Build our list of selectivities
 * - Build our ratios for the number of time steps
 */
void SurvivalConstantRate::DoBuild() {
  partition_.Init(category_labels_);

  for (string label : selectivity_names_) {
    Selectivity* selectivity = model_->managers().selectivity()->GetSelectivity(label);
    if (!selectivity)
      LOG_ERROR_P(PARAM_SELECTIVITIES) << ": selectivity " << label << " does not exist. Have you defined it?";

    selectivities_.push_back(selectivity);
  }

  /**
   * Organise our time step ratios. Each time step can
   * apply a different ratio of S so here we want to verify
   * we have enough and re-scale them to 1.0
   */
  vector<TimeStep*> time_steps = model_->managers().time_step()->ordered_time_steps();
  LOG_FINEST() << "time_steps.size(): " << time_steps.size();
  vector<unsigned> active_time_steps;
  for (unsigned i = 0; i < time_steps.size(); ++i) {
    if (time_steps[i]->HasProcess(label_))
      active_time_steps.push_back(i);
  }

  if (ratios_.size() == 0) {
    for (unsigned i : active_time_steps)
      time_step_ratios_[i] = 1.0;
  } else {
    if (ratios_.size() != active_time_steps.size())
      LOG_ERROR_P(PARAM_TIME_STEP_RATIO) << " length (" << ratios_.size()
        << ") does not match the number of time steps this process has been assigned to (" << active_time_steps.size() << ")";

    for (Double value : ratios_) {
      if (value <= 0.0 || value > 1.0)
        LOG_ERROR_P(PARAM_TIME_STEP_RATIO) << " value (" << value << ") must be between 0.0 (exclusive) and 1.0 (inclusive)";
    }

    for (unsigned i = 0; i < ratios_.size(); ++i)
      time_step_ratios_[active_time_steps[i]] = ratios_[i];
  }
}
```

Figure 6.8: Example of using the `DoBuild()` function

Moving on to the DoExecute() function. This is where the action happens and we apply a survival rate to our categories. The DoExecute function is usually called in each year and the time-steps specified in the user input configuration files when running a model.

Figure 6.9: Example of using the `DoExecute()` function

You will notice another set of logging, which is also encouraged to add all throughout the execution process. The levels of logging are described in Section 5.1

Adding a unit test

It is recommended to validate any new functionality using an independent method such as **R** or another stock assessment platform such as Stock Synthesis. For an example using **R** see the file `source\GrowthIncrements\Length\Basix.Test.cpp`. Figure 6.10 has an excerpt which shows the **R** code embedded as a comment within the unit-test. This **R** code will generate expected results that are used by the unit-test in Casal2 to validate functionality. This makes the unit-tests reproducible.

Figure 6.10: Example of R code embedded as a comment within a unit-test

7.   Adding reports

This section gives some additional points when adding reports to Casal2.

There are two variables that you must define when describing a report class, these are `model_state_` and `run_mode_`. We will try and explain what values you should give your report, but another avenue is to look at example report classes.

The `model_state_` parameters tells the model when to print the report `State::kExecute` will call the `DoExecute` after each year and `State::kIterationComplete` will print the report once the model has run. So obviously if the report prints annual information or doesn't cache information you will want to use `State::kExecute` e.g. `Partition` else use `State::kIterationComplete`.

Note: if you add a new report that has `model_state_` `State::kExecute` you will need to tweak the R-library function `extract.mpd()`. At line 50 of the of `extract.mpd.R` there is the following line of code.

```
1  multi_year_reports = c("partition", "PartitionBiomass", "PartitionMeanWeight")
```

You will need to add the report label type to this vector so that **R** will process the report correctly.

The other command `run_mode_` just tells Casal2 what run-mode should the report be generated in.

8. Merging code changes from a branch to the master

This section describes how to merge changes from a branch to the master repository. See the guidelines in Section 5 for more information on building Casal2.

Note: When merging changes, check that the new code doesn't break any of the existing unit-tests or auto-differentiation code. You will need to build using the commands listed below before merging changes.

1. `doBuild archive` this will build all the autodiff librarys.

2. `doBuild modelrunner`

3. Enter the directory `BuildSystem\Casal2\Casal2 - -unittest` to check all the unit-tests pass.

This section differs from Section 3 (where we pulled changes from the master repository into a branch). Here we are merging changes from a branch into the master, which will be incorporated into the master version of Casal2. From this example we can see from Figure 8.1 that our branch is 109 commit ahead and 844 commits behind (underlined in blue) that we want to incorporate into the master repository.

To incorporate these changes into the master you need to click on the "pull request" button. This will prompt a comparison of the changes that you are submitting for inclusion into the master.

Figure 8.1: Example showing differences between a fork and the master in GitHub

We can see that there are ten files changed from three commits. A the top left corner there is a green button with "create pull request" click that button. This will open a pull request on the master repository as shown in Figure 8.2
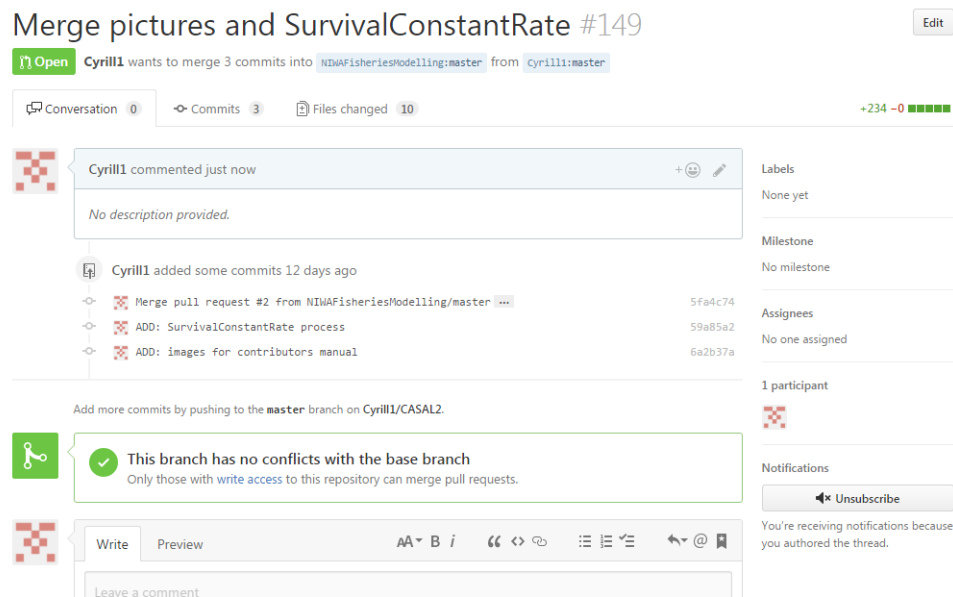
Figure 8.2: Example of a pull request in GitHub

This will notify the maintainers of the master repository that a contributor is requesting a merge of the forked code into the master. Maintainers are able to look through the proposed code changed and will discuss the changes with the contributor if they need clarification. But at this point we are just waiting for the maintainers to accept the changes which then get incorporated into Casal2.

9. Moderating code

When a pull request is received, the Development Team will check the functionality of the code, the syntax, documentation, and any optimisation code required to ensure it runs reasonably. The automated build system test system will report any broken unit-tests or test models.

When making a pull request, please ensure that the submitted changes:

1. Have documented code to explain the code

2. Use appropriate LOG messages to assist with debugging

3. Update the Casal2 User Manual(s), as required

4. Include a unit test for the changes or additional functionality

5. Include a test case model that demonstrates the changes or additional functionality

6. Pass all the unit tests and the test cases from `doBuild modelrunner`

10. Acknowledgements

11. References

B. Bull, R.I.C.C. Francis, A. Dunn, A. McKenzie, D.J. Gilbert, M.H. Smith, R. Bian, and D. Fu. CASAL C++ Algorithmic Stock Assessment Laboratory): CASAL user manual v2.30-2012/03/21. Technical Report 135, National Institute of Water and Atmospheric Research Ltd (NIWA), 2012.

A. Dunn, S. Rasmussen, and S. Mormede. Spatial population model user manual, spm v2.03-2021-06-03. Technical report, Ocean Environmental, 2021.