CASAL2

THE ARCHITECT AND DEVELOPMENT OF A NEXT-GEN MODELING PLATFORM

WHO AM I

- Scott Rasmussen (scott@zaita.com)
- Security and Software Architect
- I designed and developed the code and framework for the Spatial Population Model (SPM)
- I designed and developed the code and framework for Casal2
- I do not know much about the technical mathematics or statistics, and I cannot do stock assessment science
- I do Software Development, C++, DevOps, DevSecOps, Agile development etc.

HOW THIS PRESENTATION IS LAID OUT

I'll be working through subjects related to the Focus Questions and how these are (or not) addressed by Casal2.

The opinions in these slides are my own only. I am not an employee, nor am I a representative of NIWA or their opinions.

I am not here for the rest of this week, if you would like to contact me for questions etc., please drop me an email to scott@zaita.com.

WHAT IS CASAL2?

- Not a modeling program
- A tool for building and running stock assessment models
- Has helper classes, utility functions and scaffolding for building stock assessment model platforms
- Is a state-machine (technical software term)
- Is not derived from CASAL, but uses some of its scientific principles and the coding approach is derived from the Spatial Population Model (SPM)

CODING PHILOSOPHIES & SOFTWARE STRUCTURE

- Multi-platform – Linux, Windows, WSL2. Nothing prevents OS X
- C++11, C++14 and experimental C++17
- Is built using the Google C++ Coding Style Guideline as a basis
- Is built using Continuous Integration/Deployment Tools
- Has 30 different automated builds for when code is changed (15 Windows, 15 Linux)
- Has >200 different unit test scenarios from single line validations to entire model runs
- Has a number of full models that are verified after each code change (and is easily extendible to add more as required)
- Uses the mingw/gcc compilers, experimental version compiles in Visual Studio 2019

SHOULD WE USE PROFESSIONAL DEVELOPERS?

Who sees a problem with the code below?

```
void main() {
  vector<double> v(1, 343);
  v.assign(10, v[0]);
}
```

SHOULD WE USE PROFESSIONAL DEVELOPERS?

Who sees a problem with the code below?

```
void main() {
  vector<double> v(1, 343);
  v.assign(10, v[0]);
}
```

What about now?

```
void vector::assign(size_t n, const T& val);
```

SHOULD WE USE PROFESSIONAL DEVELOPERS?

What about now?

```
void vector::assign(size_t n, const T& val);
```

This is undefined behaviour. Even though `val` is passed in by `const` reference, the underlying container holding it (the vector) will re-allocate its entire block of memory if `n` is greater than the currently amount of reserved space.

This code actually works fine in GCC <= 9.2.. Most of the time

This code works in clang++/llvm… if you're not using threads

This code sometimes works in Visual Studio, but never if you're using threads

SHOULD YOU USE PROFESSIONAL DEVELOPERS?

A professional developer is focussed on getting every last drop of performance from the underlying physical architecture. This includes understanding how different objects (CPU, RAM, Cache, HDD/SSDs) interact with each other and how best to write code for them.

With the release of consumer level 16 core / 32 thread CPUs and inexpensive 64 core / 128 thread server CPUs there is a much greater emphasis on being able to write fast, concurrent code. This is incredibly difficult.

GPUs and programming for them is a specialist skillset, only a few programmers have the skillset to do this. Libraries do exist to help (OpenMP, OpenAAC etc), but these still require study and knowledge of the underlying architectures.

Let scientists focus on the science, let programmers focus on the code. Focus on building interactions between the two that give the best outcomes for both parties.

WHAT CAN A PROGRAMMER DO?

Casal2 running a medium complexity test case through minimization

`Normal Numerical Differences: 53 seconds`

`Auto-Differentiation: 12 seconds`

`Threaded Normal Numerical Differences: 9 seconds (1st cut at optimization by me)`

Do we need auto-diff in 2019? It makes the code massively more complex.

MCMC:

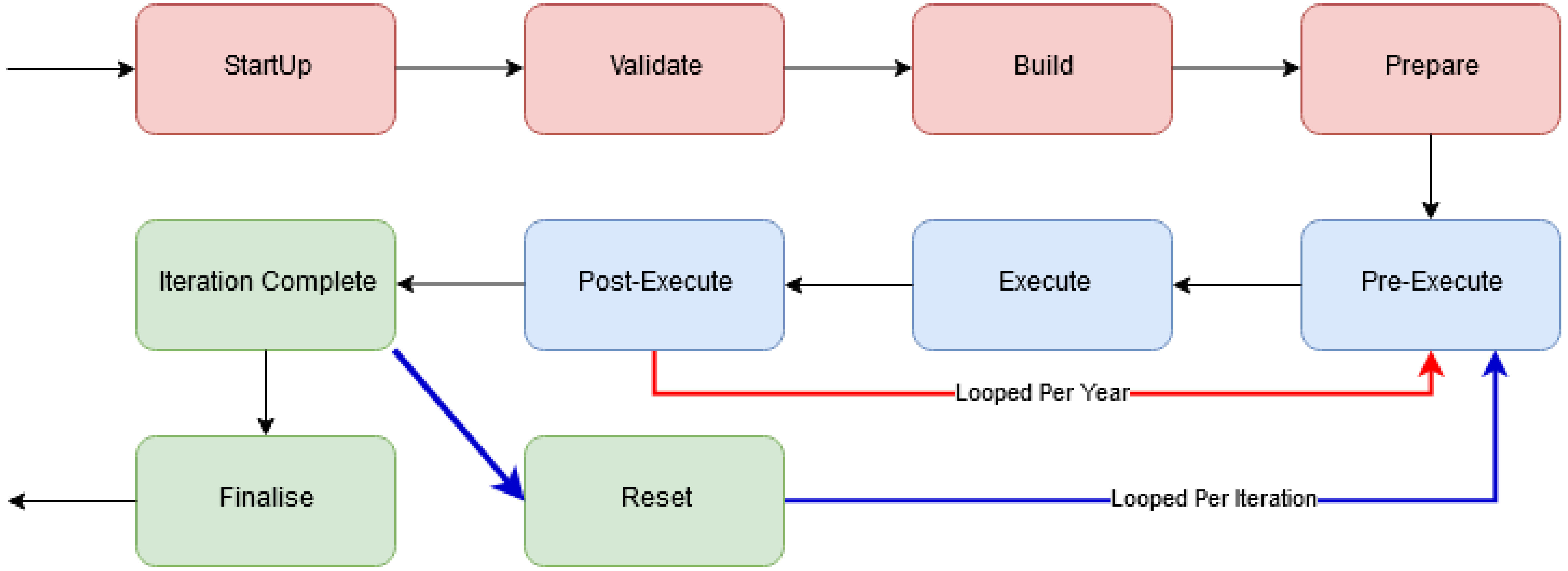
Threaded-speculation for Metrop Hastings can reduce total time by 50% with 3 threads and 67% with 7 threads.

MAINTAINING CODE EFFICIENCY / ADDITION OF NEW FEATURES

- Casal2 is a modular platform, code is only run when necessary
- Your model file is defined using a model definition language (MDL) and Casal2 compiles this into a runtime representation. This runtime representation is then executed.
- The code structure is designed to be easily modified and enhanced with new functionality
- By having a strong emphasis early on software architecture, you can save yourself many hundreds (thousands) of hours later with customizations

- A new process/observation class/selectivity can be added to Casal2 very quickly (for me, probably about 15 minutes). This would have no effect on the performance or complexity of any other parts of the code.

CASAL2 - THE STATE MACHINE

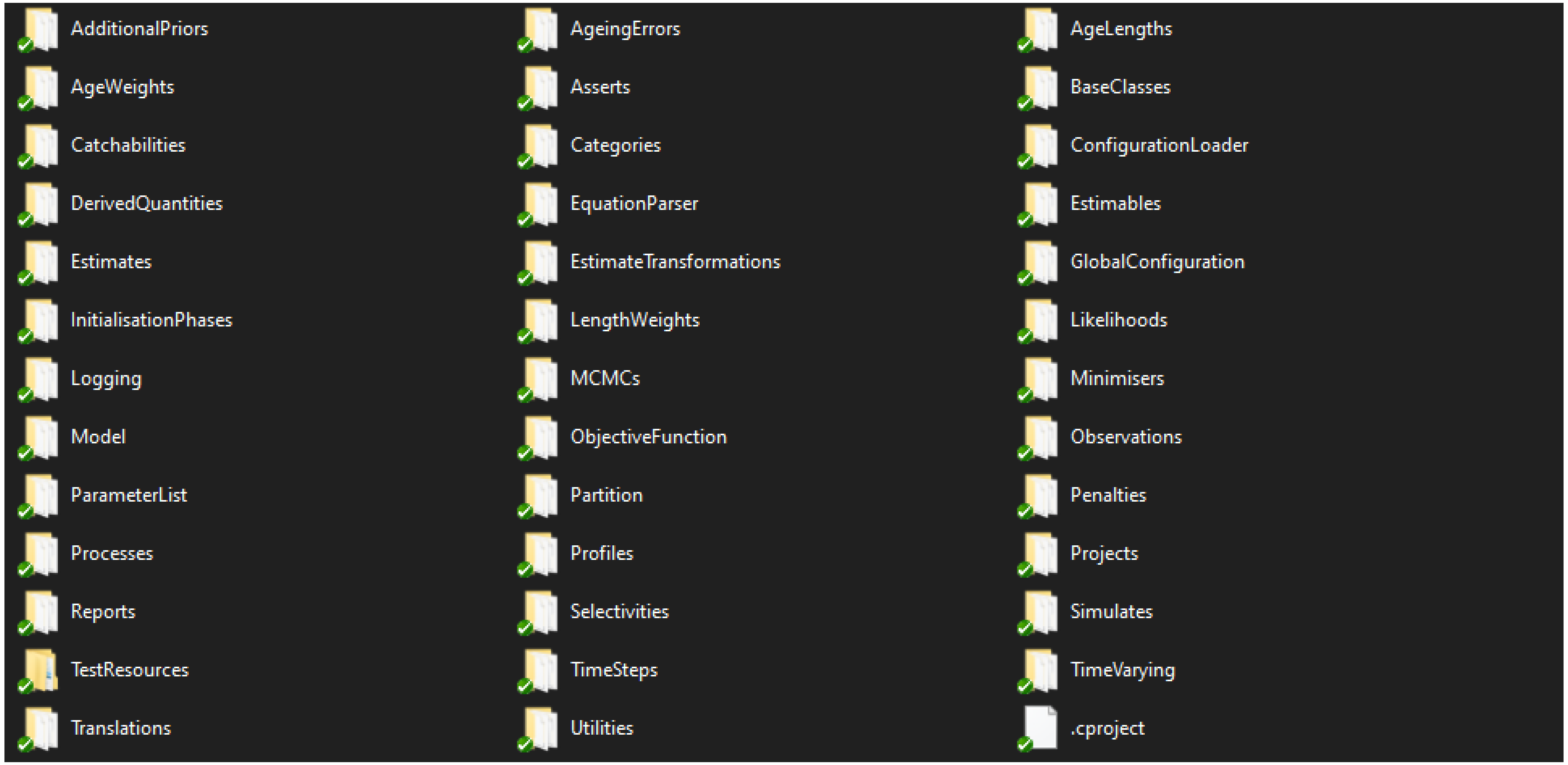


CASAL2 ALLOWS EASY EXTENSIBILITY

Casal2:

- The code base is split into logical areas (processes, reports, observations etc.)
- Uses standard design patterns (factory, façade, observer, strategy, state)
- Will build runtime objects for only things that it requires.
- Uses a model definition file to determine what model to build
- Allows us to build custom models for performance profiling and optimization
- Is a state-machine. The system moves through states in a pre-determined order so code can easily be placed into locations where it run only once, or many times.
- Is unit tested thoroughly to ensure changes to one part of the system do not cause significant performance issues across the whole thing

THE CASAL2 FILE SYSTEM



AdditionalPriors	AgeingErrors	AgeLengths
AgeWeights	Asserts	BaseClasses
Catchabilities	Categories	ConfigurationLoader
DerivedQuantities	EquationParser	Estimables
Estimates	EstimateTransformations	GlobalConfiguration
InitialisationPhases	LengthWeights	Likelihoods
Logging	MCMCs	Minimisers
Model	ObjectiveFunction	Observations
ParameterList	Partition	Penalties
Processes	Profiles	Projects
Reports	Selectivities	Simulates
TestResources	TimeSteps	TimeVarying
Translations	Utilities	.cproject

CASAL2 - ANATOMY OF AN OBJECT

```
class Process : public base::Object {
public:
  Process () = default;
  virtual ~Process() = default;
  void Validate();
  void Build();
  void Reset();
  void Execute();
}
```

WHAT LANGUAGE SHOULD WE USE?

- Casal2 was derived from the Spatial Population Model (SPM) codebase. SPM is written in C++
- C++ provides great interoperability with scientific libraries and existing code (Casal, CppAD, ADOL-C, Boost)
- C++ can be very fast, you're close to the metal
- There are great software tools for profiling, optimizing and analyzing the speed of your code
- C++ can integrate with pretty much every other language. It's trivial for us to add bindings for Python/R etc. This allows us to prototype code in Python/R, then build in C++ when functionality has been tested
- We can write the code to be cross-platform and will be guaranteed it will work for decades as the C++ standard and compilers maintain backwards compatibility. There are no breaking changes in the C++ standard, unlike the move from Python2 to Python3

CODE TESTING – INTEGRITY OF RESULTS

- The model system should use test driven development (TDD) (ish). Not pure TDD, but an emphasis should be on building functionality and integrity tests together.
- The code base must be configured in such a way as to easily allow the testing of discrete functions/methods and components. There is no excuse for not doing this in 2019.
- You should be using CI/CD (automated build systems triggered by code commits). These build systems should be the "source of truth" for integrity. Automated tests should be incorporated to your build system and run frequently. CPU time is cheap.
- The cost of finding and fixing a bug during development is 10% of that after the product has been released.

CASAL2 – MAINTAINING INTEGRITY

- Over 200 different tests that are run on every build for Windows and Linux
- 11 models of varying complexity executed, and the objective value is validated
- The core of Casal2 has near complete unit testing
- Processes/Observations are lacking at this point, but work is being done to improve this
- Work is being done to make adding tests to Casal2 even easier
- Casal2 uses the Google Test and Google Mock frameworks
- Casal2 has test harnesses and mocked classes to make integration testing easier
- We ship the unit tests with Casal2 so the users can verify them on their own machines. This is important as different CPUs, Operating Systems etc., can produce slightly different results due to inaccuracies in floating point numeric types.

SHOULD NEXT-GEN MODELS SUPPORT X?

- Build systems that are highly adaptable
- Yes, but Casal2 either already has it or can easily have it
- Casal2 is not fixed, not even the type of partition
- Casal2 supports key concepts like age, age-length, length, time-varying objects, category hierarchy (tagging), inter-species interactions

- Sub question: Should all possible features be added?
- Yes.

ERROR CHECKING IN MODELS

SPM was bad. We used C++ exceptions to capture the stack and throw errors. This was nice because it told us where in the code the error was and a description but `try…catch` came with a 100% performance overhead. We had to `#ifdef` these out in the release builds losing a good deal of error reporting.

Casal2 uses an in-built parameter system to track all the parameters and their origin (configuration file name and line). All parameters belong to an object and can be reported on. The parameter system handles automatic type conversion and low-level input validation (is numeric, is between values, is above X, is below Y, is one of X,Y,X). This is auto-documented through a set of python scripts to the User Manual.

In code we can do

LOG_ERROR_P(myParameterName) << "this parameter is bad";

ERROR CHECKING IN MODELS

Casal2 also supports the aggregation of errors. This allows us to parse a configuration file and accumulate many errors before displaying them to the user. This prevents the fix one error, re-run, get next error, fix one error, re-run loop.

Many of our error messages provide a hint as to what the user may have done wrong.

e.g:

```
At line 121 of population.csl2:

process Mortality cannot find the selectivity mySelecdtivity, did you define or
mis-spell it?
```

ERROR CHECKING IN MODELS

Casal2 supports different levels of errors. The lowest level is LOG_WARNING where the user is displayed a message, but execution continues, right up to LOG_FATAL that will quit immediately. All of these provide a stringstream object to push messages too.

e.g.

```
LOG_FATAL() << "when parsing configuration files, could not find included file " << file_name;
```

Casal2 also has a LOG_CODE_ERROR that we use in the code for displaying developer bugs. This is useful for catching things like nullptrs or incorrect input into methods that the user has no control over. A message to the user to contact the developer with an email address is given.

USE TOOLS. Compilers and Debuggers can help identify bad memory, thread race conditions etc automatically now.

RESOURCING NEXT-GEN MODELS

I'm very biased, but I think Casal2 is a great foundation.

Different organisations can take the code base and extend it based on their own use-cases. There is a lot of scaffolding and utility in Casal2 that will come for free.

Some of this includes:

- A generic model definition file format and parsing system
- A self-documenting parameter system with automatic type conversion and validation
- A huge amount of helper functionality for things like RNG, logging/errors, conversions, math
- Soon™ - Hopefully, A thread pool object that allows you to just pass in parameters and get results

PROJECT MANAGEMENT + SOFTWARE DEVELOPMENT METHODOLOGY

- Use tools and automation as much as possible.
- Major cloud providers (e.g. Microsoft) provide free build resources to Open Source projects
- Use DevOps tools (e.g. Azure DevOps) to manage builds, code, documentation, tickets, etc
- Find a good collaboration platform. Slack is currently the most popular tool used by developers and software teams. The tool should emphasis real-time collaboration.
- Automate, automate, automate – Web hooks, automated validation of commits (i.e., does this new class have unit tests or not). Prevent commits that don't have a ticket/feature description.
- Don't do things manually unless you absolutely must
- Learn from the wider software development industry, building software in a scientific context is not special

BONUS ROUND

If I am here, I’ve got some spare time to show off some of the things in Casal2 and how it works. Buckle up.

If not, please feel to contact me at scott@zaita.com

BUILDING MODELS IN CASAL2

Casal2 uses a custom model definition language (MDL) derived from that used by SPM. The model file is constructed from blocks, parameters and values.

The syntax is human readable, easily parsed and through simple scripting could be converted into other formats. The syntax is flexible enough to allow for the representation of complex models without becoming overwhelming.

The MDL uses `@blocks` to create separation of logical entities (processes, reports, observations). Each block has a collection of parameters and values. Casal2 supports automatic type conversions and checks of allowed values from pre-determined lists or ranges. The syntax parser in Casal2 is able to print the file, line and description of the error with virtually no extra work by the programmer.

EXAMPLE OF USING THE CONFIGURATION SYSTEM

In the Casal2 code, we have some sanity checks on ranges of values. For example, a proportion cannot be greater than1.0.

In the C++ header we have:

```
double proportion;
```

In the C++ constructor we have:

```
parameters_.Bind<unsigned>(PARAM_PROPORTION, &proportion_, “description of
parameter”)->set_range(0.0, 1.0, true, true);
```

Conversion from text to numeric, assignment to the proportion variable and checking the value is between 0.0 and 1.0 is now completed automatically by the Casal2 parameter system.

The programmer will have access to the populated variable `proportion_`.

EXAMPLE SYNTAX FROM REAL MODEL

```
@model
type age
min_age 1
max_age 17
age_plus true
base_weight_units tonnes
start_year 1972
final_year 2016
projection_final_year 2021
```

SOME OF THE COOL FEATURES OF CASAL2 - CATEGORIES

Casal2 has included the concept of category hierarchies.

By using a category hierarchy we're able to support some cool features with regards to defining and referencing categories.

We use a "format" keyword to specify the different parts of the category (e.g. format name.sex.stage). From this format you can short-hand syntax to specify a large number of categories.

e.g.,	name	sex	stage
	North	Female	Immature
	North	Male	Mature
	South	…	…

A short-hand syntax and lookup feature allows us to define and reference large numbers of categories quickly.

ACKNOWLEDGEMENTS

- Thank you to
  - NIWA and the Casal2 development team at NIWA
  - CAPAM
  - Others who have contributed to the code/knowledge/etc

THANK YOU

SCOTT@ZAITA.COM