S3 Classes for CCMPP Input Data Structures

# Introduction

This document presents a general framework for a set of S3 classes that will be used for objects that are inputs and outputs to CCMPP functions.

## Context

CCMPP requires as input a collection of data by age, sex, and period:

1. Population counts at the baseline year (or “jump-off” population).
2. Period(?) fertility rates in each period of the projection.
3. Period(?) sex ratios at birth in each period of the projection.
4. Cohort(?) survival ratios in each period of the projection.
5. Period(?) net migration in each period of the projection, preferably as counts.

There may also be

1. Population counts at some years in the interval of projection.

CCMPP produces as output

* Projected population counts at each period, according to the step size.

Other quantities of interest that can be derived from the inputs and outputs include (all possibly age- sex- period-specific):

* Birth counts in each period of the projection.
* (Period, Cohort ?) Death counts in each period of the projection.
* Period(?) mortality rates in each period of the projection.
* Other?

Other systems will be interacting with the *R* code performing CCMPP operations, such as systems that prepare the input and systems that read and process the output.

## Approach

*R*’s S3 object oriented programming system will be used to formally encode input and output object structures as *classes*. Once such classes are defined one can write functions that validate an object as a member of the class by checking that it’s structure matches the class definition. For example, a class for fertility rates might specify that they be non-negative. Objects passed into the CCMPP functions can be validated (once) at the top of the function to ensure that the input object matches the specification the function expects. This approach reduces the chance of failures later on in the function and of undetected errors in computations.

For further reading on *R*’s object-oriented programming system, including S3, see (Wickham ([2019](#ref-wickham_advanced_2019)), Part III).

## Justification

Having specifications for valid data structures makes development and maintenance of R code easier, and makes the code less susceptible to errors and failures. It makes it easier for users to generate inputs since they know exactly what is required, and for them to work with the outputs since they know exactly what the outputs will look like. Complete specification of input and output objects are particularly important for interoperability between systems where one system may generate inputs and another process the outputs.

S3 programming promotes transparency and rigor as the specifications of the objects are defined in *R* code. Each object of a given class has attached to it meta data describing its specification. These meta data are stored in the objects’ *attributes*.

Creating object classes abstracts the object type from the details. A user-interface can be developed to perform key operations on objects of a class independently of the class’s implementation. It is not relevant to the user, nor should it affect the visible behaviour of the function, whether the object uses data frames, matrices, or lists to store the information. These underlying details can be changed without affecting the user interface and, thereby, breaking existing user code.

Functions that operate on objects with classes can be written such that different *methods* are available for different classes of objects. This means that the same function call when applied to objects of different classes can produce different outputs. This is another useful form of abstraction; the user interface is separated from the details of the implementation. In *R* it also allows new behaviours for new classes to be quickly added to existing functions.

This system is also efficient in terms of programming effort, and easy to extend over time, because of *inheritance*. A series of nested class definitions can be defined such that objects inherit the attributes of their ‘parents’. This means that specifications of ‘child’ classes need only include the additional attributes not held by their parents. Moreover, functions with a method defined for a parent will automatically apply that method to objects of a child class, unless a specific method for the child class is defined.

# S3 Classes Defined

The classes defined and their inheritance relations are shown in Figure XXX.

Hierarchy of S3 classes for CCMPP input and output objects

# Base Class: demog\_change\_component\_df

The base class is a data frame with meta data (known as “attributes”) called demog\_change\_component\_df[[1]](#footnote-27).

The class was designed with reference to

data(wpp\_input\_example)  
head(wpp\_input\_example$mig\_net\_count\_age\_sex)  
#> time\_start time\_span sex age\_start age\_span value  
#> 1 1950 1 male 0 1 -1906.5913  
#> 2 1950 1 male 1 1 -772.8935  
#> 3 1950 1 male 2 1 52.2876  
#> 4 1950 1 male 3 1 765.6268  
#> 5 1950 1 male 4 1 1291.9252  
#> 6 1950 1 male 5 1 1651.7646

which is a list data frames of components of demographic change down by time, age, and sex.

The full set of CCMPP inputs consists of a list of data frames with all or some of these dimensions. For example, the sex ratio at birth component has only the time dimension, the fertility rate component has only the time and age dimensions. These will be handled as subclasses with different validity condidions. *This is in progress* so please read the below with the understanding that I still need to make more specific classes for the various subtypes contained in the input list. These should be straightforward to code once the base type is finalized.

I noticed that the input list also contains a life table. I think we should make a life table class because it will be useful anyway.

* **We need to decide** if we want the full life table to be part of the official input spec or only one or two of the columns.

## Purpose

The purpose of this S3 object class is to specify the structure a set of inputs to CCMPP functions requires to be valid. It is not intended to be a class for a data base or any other use-case. In order to minimize errors, especially invisible errors, the inputs to functions should be checked to ensure they contain valid values. To achieve this using classes:

1. A complete description of the object class encoded in a series of *R* functions that *create* and *validate* objects as members of the class.
2. Functions that take objects of the class as their primary input should be *generics* with *methods* for the class. This provides only a *minimal* amount of safety but makes a clear statement to the user that the object expected should conform to expectations.
3. Functions *validate* the input object before doing any calculations. This should prevent any invalid inputs making their way into the function calculations potentially causing undetected errors. *Subsequent* calculation errors due to the function’s own operations are the responsibility of the function, not the class.

## General Format

The class shall consist of a data.frame holding the data and a set of attributes holding meta data. What goes where is a design decision; we can discuss and modify as we go through the development phase.

Currently, the data frame component has columns

* age\_start
* sex
* time\_start
* value

The attributes are

* age\_span
* time\_span

## Definition

The definition of a valid object this class is as follows:

Data frame component

* The data frame contains columns with exact names:
  + age\_start
  + sex
  + time\_start
  + value
* The data frame has exactly one finite and non-missing value per age\_startsextime\_start combination because this is what CCMPP requires (this is not as class for a data base of many possible inputs).
* For every unique time\_start there is an age\_start that equals 0; i.e., the age breakdown must start at age 0.
* Both time\_start and age\_start must be complete sequences in steps of time\_span and age\_span, respectively. That is, if age\_span == 1 then, within each time\_spansex block, age\_start must be 0, 1, 2, … .
* The only values for sex are both, female, or male.

Attributes component

* age\_span and time\_span are finite non-missing scalars.

The definition of the class is contained in the documentation of the function: ?demog\_change\_component\_df (the documentation file is *to be completed*).

## Construction and Validation

### Main Constructor Function

A user-level constructor is provided which takes as input the base data frame object plus arguments specifying the age and time spans:

mig\_vr\_df <- demog\_change\_component\_df(x = wpp\_input\_example$mig\_net\_count\_age\_sex,  
 age\_span = 1,  
 time\_span = 1)  
#> Argument 'dimensions' is 'NULL'; setting 'dimensions' to 'time, age, sex' based on column names of 'x'.  
#> Argument 'value\_type' is 'NULL'; setting 'value\_type' to 'real'.

The age and time spans may be part of the data frame, in which case they will be read in automatically as long as they are valid:

mig\_vr\_df <- demog\_change\_component\_df(x = wpp\_input\_example$mig\_net\_count\_age\_sex)  
#> Argument 'dimensions' is 'NULL'; setting 'dimensions' to 'time, age, sex' based on column names of 'x'.  
#> Argument 'time\_span' is 'NULL'; taking 'time\_span' from 'x$time\_span'.  
#> Argument 'age\_span' is 'NULL'; taking 'age\_span' from 'x$age\_span'.  
#> Argument 'value\_type' is 'NULL'; setting 'value\_type' to 'real'.

The class of the object is contained in the “class” attribute accessed with base *R* function class()

class(mig\_vr\_df)  
#> [1] "demog\_change\_component\_df" "data.frame"

Note that the data.frame is the second element of the class. This indicates that objects of class demog\_change\_component\_df explicitly *inherit* from the base class data.frame. Inheritance is part of *R*’s S3 object oriented programming system. In effect it means that when in doubt *R* will act as if demog\_change\_component\_dfs are data.frames, which makes the most sense since that is the class they are based upon. Inheritance is relevant for the creation of generic functions and class-specific methods which will be demonstrated below.

### Validation

The function validate\_ccmpp\_object() is available to validate an object as a member of the class. As per convention, if the object is valid it is returned; if not, an error is signalled:

x <- validate\_ccmpp\_object(x = mig\_vr\_df)  
class(x)  
#> [1] "demog\_change\_component\_df" "data.frame"  
  
## Not run:  
## y <- validate\_ccmpp\_object(x = wpp\_input\_example$mig\_net\_count\_age\_sex)  
## > Error in validate\_ccmpp\_object(x = wpp\_input\_example$mig\_net\_count\_age\_sex) :   
## 'x' must have attributes 'class', 'names', 'row.names', 'age\_span', 'time\_span'; some are missing.

### Construction Conveniences

demog\_change\_component\_df() can be designed to provide various conveniences to the user. Currently:

* The user can supply time\_span and age\_span as columns in the input data frame. demog\_change\_component\_df will extract and check them, and set them as attributes in the constructed object (see above).
* Pre-sorting is not required; demog\_change\_component\_df will sort the input data frame properly according to the specification.
* z <- wpp\_input\_example$mig\_net\_count\_age\_sex  
  z[, "age\_start"] <- rev(z$age\_start)  
  z <- demog\_change\_component\_df(x = z, age\_span = 1, time\_span = 1)  
  #> Argument 'dimensions' is 'NULL'; setting 'dimensions' to 'time, age, sex' based on column names of 'x'.  
  #> Argument 'value\_type' is 'NULL'; setting 'value\_type' to 'real'.  
  class(z)  
  #> [1] "demog\_change\_component\_df" "data.frame"

### Restrictions

* No extra columns or rows are allowed; the object is intended to be the minimal required to run CCMPP correctly, not a databse type object with unnecessary information. These will be silently dropped:
* x <- wpp\_input\_example$mig\_net\_count\_age\_sex  
  z <- data.frame(x, source = "census")  
  head(z)  
  #> time\_start time\_span sex age\_start age\_span value source  
  #> 1 1950 1 male 0 1 -1906.5913 census  
  #> 2 1950 1 male 1 1 -772.8935 census  
  #> 3 1950 1 male 2 1 52.2876 census  
  #> 4 1950 1 male 3 1 765.6268 census  
  #> 5 1950 1 male 4 1 1291.9252 census  
  #> 6 1950 1 male 5 1 1651.7646 census  
    
  z <- demog\_change\_component\_df(x = z)  
  #> Argument 'dimensions' is 'NULL'; setting 'dimensions' to 'time, age, sex' based on column names of 'x'.  
  #> Argument 'time\_span' is 'NULL'; taking 'time\_span' from 'x$time\_span'.  
  #> Argument 'age\_span' is 'NULL'; taking 'age\_span' from 'x$age\_span'.  
  #> Argument 'value\_type' is 'NULL'; setting 'value\_type' to 'real'.  
  head(z)  
  #> time\_start age\_start sex value  
  #> 1 1950 0 male -1906.5913  
  #> 2 1950 1 male -772.8935  
  #> 3 1950 2 male 52.2876  
  #> 4 1950 3 male 765.6268  
  #> 5 1950 4 male 1291.9252  
  #> 6 1950 5 male 1651.7646

Expectations of the user include:

* Exactly one row per age\_startsextime\_start combination is expected; demog\_change\_component\_df() will not attempt to figure out which from several combinations the user wants in the finished object. Use dplyr or similar before passing to demog\_change\_component\_df().
* The sequencing of age\_start and time\_start must be correct and must match age\_span and time\_span.
* No missing values are allowed; CCMPP won’t work with them.
* sex must use the values both, female, and male (or some subset). Eg., “Both”, “MALE”, etc. won’t work and won’t be corrected.

We can modify some of the conveniences but I recommend including as few as possible for clarity of purpose and ease of maintenance[[2]](#footnote-36).

# Methods

The main reason for creating demog\_change\_component\_df is to have a definition and validation system for inputs and outputs of the CCMPP functions. However, the object oriented programming system is really designed to allow specific behaviour of functions based on the class of the objects passed as inputs. It makes sense to define some basic methods for demog\_change\_component\_df.

## Subsetting

*R*’s basic subset and replacement functions have been given methods for the class. If a demog\_change\_component\_df object is subsetted in any way the result looses the class as it cannot be gauranteed the required criteria will still be met after extraction. A warning will be issued if used interactively:

## Extract one column:  
z <- mig\_vr\_df[, "age\_start"]  
#> Warning in `[.demog\_change\_component\_df`(mig\_vr\_df, , "age\_start"): Subsetting  
#> a 'demog\_change\_component\_df' will not preserve the class or attributes. See '?  
#> subset\_time' and friends for an alternative approach.  
str(z)  
#> num [1:14140] 0 1 2 3 4 5 6 7 8 9 ...

## Extract a sub-data frame  
z <- mig\_vr\_df[mig\_vr\_df$time\_start == 1950, ]  
#> Warning in `[.demog\_change\_component\_df`(mig\_vr\_df, mig\_vr\_df$time\_start  
#> == : Subsetting a 'demog\_change\_component\_df' will not preserve the class or  
#> attributes. See '?subset\_time' and friends for an alternative approach.  
class(z)  
#> [1] "data.frame"  
head(z)  
#> time\_start age\_start sex value  
#> 1 1950 0 male -1906.5913  
#> 2 1950 1 male -772.8935  
#> 3 1950 2 male 52.2876  
#> 4 1950 3 male 765.6268  
#> 5 1950 4 male 1291.9252  
#> 6 1950 5 male 1651.7646

The warning is only issued if “[” (or “$”, or “[[”) are called at the top level. E.g.,

## Same effect as `[` above but no warning  
z <- subset(mig\_vr\_df, time\_start == 1950)  
class(z)  
#> [1] "data.frame"

To create a valid subset, re-make the object as a member of the class (suppressWarnings() is optional):

z <- demog\_change\_component\_df(suppressWarnings(mig\_vr\_df[mig\_vr\_df$time\_start == 1950, ]),  
 age\_span = attr(mig\_vr\_df, "age\_span"),  
 time\_span = attr(mig\_vr\_df, "time\_span"))  
class(z)  
#> [1] "demog\_change\_component\_df" "data.frame"  
head(z)   
#> time\_start age\_start sex value  
#> 1 1950 0 male -1906.5913  
#> 2 1950 1 male -772.8935  
#> 3 1950 2 male 52.2876  
#> 4 1950 3 male 765.6268  
#> 5 1950 4 male 1291.9252  
#> 6 1950 5 male 1651.7646

We could define some functions to do this if we think it will be a common operation. E.g., a method for subset could be written or the existing method for [ could be enhanced to automatically keep the class if the result is a valid member. *However* this would then mean that subsetting could have multiple different outcomes depending on the validity of the subsetted data frame as a member of the class. This might be confusing and difficult to use. An advantage of the current design is that the user always knows subsetting will remove the class and that they need to explicitly re-create (and, hence, validate) it if they want the class.

## Replacement

The user can replace a column in a demog\_change\_component\_df and if it is valid the class will be retained. If not an error is signalled.

z <- mig\_vr\_df  
z[, "age\_start"] <- z$age\_start #trivially replace the age column but note the class is retained  
#> Warning in `[<-.demog\_change\_component\_df`(`\*tmp\*`, , "age\_start", value =  
#> c(0, : Replacing elements in a 'demog\_change\_component\_df' will not preserve the  
#> class or attributes.  
class(z)  
#> [1] "data.frame"

## print and summary

print and summary are two common functions for examining objects. Special methods have been written to make these useful for the demog\_change\_component\_df class. The print method takes inspiration from the method for tibbles:

mig\_vr\_df   
#> # A 'demog\_change\_component\_df' with 14,140 rows.  
#> # dimensions = 'time, age, sex', age\_span = '1', time\_span = '1', value\_type = 'real'.  
#> # 'sex' has levels: female, male.  
#> time\_start age\_start sex value  
#> 1950 0 male -1906.5913  
#> 1950 1 male -772.8935  
#> 1950 2 male 52.2876  
#> 1950 3 male 765.6268  
#> 1950 4 male 1291.9252  
#> 1950 5 male 1651.7646  
#> # ... etc.

summary is based on the data.frame method with a few extra bits of information:

summary(mig\_vr\_df)  
#> dimensions: time, age, sex  
#> time: range = [1950, 2019] span = 1  
#> age: range = [0, 100] span = 1  
#> sex: levels = female, male  
#> value\_type: real  
#> ------------------------------------------------------------  
#> table:  
#> time\_start age\_start sex value   
#> Min. :1950 Min. : 0 Length:14140 Min. :-1918.91   
#> 1st Qu.:1967 1st Qu.: 25 Class :character 1st Qu.: 40.51   
#> Median :1984 Median : 50 Mode :character Median : 410.62   
#> Mean :1984 Mean : 50 Mean : 744.91   
#> 3rd Qu.:2002 3rd Qu.: 75 3rd Qu.: 1201.08   
#> Max. :2019 Max. :100 Max. : 6661.00

# What Next?

1. The next essential step is to accommodate the different sub-types present in the example CCMPP input list, namely vital rate data frames broken down only by age, or time, or sex, or a combination of two. I plan to do this by adding an attribute to the current class marking its subtype. The validation function will then allow for, e.g., the sex column to be missing, or the age\_start to be missing.
2. Following (1), I will create some superficial classes for the specific vital rates (fertility, migration, sex ratio at birth) etc. These will be a very thin layer on top of the appropriate base class. Where appropriate I will add the requirement of non-negative values or values between 0 and 1.
3. I will add a life table class to hold life tables. *Alternatively* we can search for an existing class of which I’m sure there are several already on *CRAN*.

* **Question:** does *DemoTools* already have a life table class?

1. I will create a ccmpp\_input\_list class that is a list with the required data frames as elements. This will be able to use all the existing creation and validation functions as the elements will be of class demog\_change\_component\_df. This will ensure that the inputs passed to the CCMPP functions are complete and valid.

# References

Wickham, Hadley. 2019. *Advanced R*. 2nd ed. Boca Raton, Florida: CRC Press. <https://adv-r.hadley.nz/>.

1. ‘df’ for ‘data frame’. [↑](#footnote-ref-27)
2. Allowing case insensitive values for sex in the input might be OK. [↑](#footnote-ref-36)