Session 3

Inheritance, Composition and Template Meta-Programming

Matt Denwood

2023-06-14

Recap

Material

Everything is on the public GitHub repo:

1. Use GitHub Desktop (https://desktop.github.com)
2. Choose "File" and "Clone Repository" and the "GitHub.com" tab
3. Enter ku-awdc/R6ModellingCourse2023 and click on Clone
4. Remember to pull changes before each day, as things will be added
5. Remember to copy/paste code from the HTML version, NOT the PDF version

Material

Everything is on the public GitHub repo:

1. Use GitHub Desktop (https://desktop.github.com)
2. Choose “File” and “Clone Repository” and the “GitHub.com” tab
3. Enter ku-awdc/R6ModellingCourse2023 and click on Clone
4. Remember to pull changes before each day, as things will be added
5. Remember to copy/paste code from the HTML version, NOT the PDF version

You only have write access, so you can’t push changes. It is best not to modify anything in the course files (notes etc) in case you get conflicts pulling changes later. . .

Programming Paradigms

Imperative / procedural - Fine for small tasks - Use encapsulation (procedural programming) where possible

Programming Paradigms

Imperative / procedural - Fine for small tasks - Use encapsulation (procedural programming) where possible

Functional - Great for data science! - Strong encapsulation (and abstraction) - R is a functional language at its core

Programming Paradigms

Imperative / procedural - Fine for small tasks - Use encapsulation (procedural programming) where possible

Functional - Great for data science! - Strong encapsulation (and abstraction) - R is a functional language at its core

Object-oriented (encapsulated OO) - Classes are the building blocks of code - Strict separation of usage and implementation - Core of C++ and R6

Inheritance

Basics of Inheritance

Consider the following:

Animal -> Mammal -> Dog -> Alsatian

Basics of Inheritance

Consider the following:

Animal -> Mammal -> Dog -> Alsatian

All alsatians have animal, mammal and dog properties, as well as some alsatian-specific properties.

Basics of Inheritance

Consider the following:

Animal -> Mammal -> Dog -> Alsatian

All alsatians have animal, mammal and dog properties, as well as some alsatian-specific properties.

Animal -> Mammal -> Cat -> Ragdoll

Ragdolls share some characteristics with alsatians

Inheritance in R6

Just use the “inherit” argument for R6Class - Specifies the “parent” or “super” class - A “child” or “derived” class can be a parent for another derived class - Methods can be “over-ridden” from the parent class

Inheritance in R6

Just use the “inherit” argument for R6Class - Specifies the “parent” or “super” class - A “child” or “derived” class can be a parent for another derived class - Methods can be “over-ridden” from the parent class

Special considerations:

- Use the keyword super to refer to the self of the super class
- It is possible (but not so easy) to refer to the super of the super (but super$super doesn’t work!)
- It is possible (but not so easy) to refer to the private environment of the super class
- [These things are both cleaner in C++, where we have public/protected/private inheritance]

```r
library("R6")
Animal <- R6Class("Animal",
    public = list(
      initialize = function(name){
        private$prv_name <- name
      }
    ),
    active = list(
      name = function() private$prv_name
    ),
    private = list(
      prv_name = character()
    )
)
geoff <- Animal$new("Geoff")
geoff$name
## [1] "Geoff"
```

```r
Dog <- R6Class("Dog",
    inherit = Animal,
    public = list(
      initialize = function(name){
        super$initialize(name)
      }
    ),
    active = list(
      legs = function() 4L
    )
)
bob <- Dog$new("Bob")
bob$legs
## [1] 4
bob$name
## [1] "Bob"
```

Inheritance: Example

We could have a base class “PigFarm” implementing a simple ASF model, with methods for getting the:

- Number of infected animals
- Number of infectious animals
- Number of dead animals

Inheritance: Example

We could have a base class “PigFarm” implementing a simple ASF model, with methods for getting the:

- Number of infected animals
- Number of infectious animals
- Number of dead animals

And derived classes “BreederUnit” and “FinisherUnit” that adds details specific to these types of farm

Virtual Base Classes

A virtual base class defines an interface only, i.e. what a class will do, but not how it will do it

Virtual Base Classes

A virtual base class defines an interface only, i.e. what a class will do, but not how it will do it

This is cleaner in C++ (virtual keyword) but also possible in R6:

```r
Animal$set("public", "speak",
  function() stop("The speak method is virtual and must be over-ridden")
)
```

Advantages of Inheritance

- Even more encapsulation
  - Additional features can easily be added to a complex parent class
  - This can also happen between R packages

Advantages of Inheritance

- Even more encapsulation
  - Additional features can easily be added to a complex parent class
  - This can also happen between R packages
- Virtual base classes provide a formal mechanism for defining usage first and implementation later
  - This can be “within coder” i.e. a design strategy
  - This can also be “between coders” i.e. a way of writing generic code

Composition

Basics of Composition

A class can contain many other classes!

e.g. an Animal has a:

- Nervous system
- Locomotive system
- Digestive system
- etc

Basics of Composition

A class can contain many other classes!

e.g. an Animal has a:

- Nervous system
- Locomotive system
- Digestive system
- etc

Remember:

- “is a” = inheritance
- “has a” = composition

Composition: Example

A Simulation class may contain:

1. One or more Population
2. One or more Spread mechanism
3. One or more Monitoring mechanism
4. A Logging mechanism

Composition: Example

A Simulation class may contain:

1. One or more Population
2. One or more Spread mechanism
3. One or more Monitoring mechanism
4. A Logging mechanism

The Population/Spread/Monitoring classes may be virtual base classes

- i.e. the Simulation class can be made to run with a highly flexible range of scenarios

Composition: Example

A Simulation class may contain:

1. One or more Population
2. One or more Spread mechanism
3. One or more Monitoring mechanism
4. A Logging mechanism

The Population/Spread/Monitoring classes may be virtual base classes

- i.e. the Simulation class can be made to run with a highly flexible range of scenarios
- This is what we do with BLOFELD

Composition in R6

Option 1: pass in components as arguments to initialize:

```r
library("R6")
Simulation <- R6Class("Simulation",
    public = list(
      initialize = function(population, spread, monitoring, logger){
        stopifnot(inherits(population, "Population"))
        private$population <- population
        stopifnot(inherits(spread, "Spread"))
        private$spread <- spread
        # etc
      }
    )
)
```

Option 2: generate components within initialize:

```r
library("R6")
Simulation <- R6Class("Simulation",
    public = list(
      initialize = function(population, spread, monitoring, logger){
        # population/spread as before
        private$monitoring <- Monitoring$new()
      }
    )
)
```

Option 3: fixed instance across the class:

```r
library("R6")
Simulation <- R6Class("Simulation",
    public = list(
      initialize = function(population, spread, monitoring, logger){
        # population/spread as before
        # monitoring as before
      }
    ),
    private = list(
      logger = Logger$new()
    )
)
```

Template Meta-Programming (TMP)

Template Meta-Programming

This is only relevant to compiled languages (i.e. it is a C++ thing) . . . but:

- It can be used instead of (or in addition to) composition/inheritance
- Generally results in faster code, as the optimiser is able to work more magic
- You will see it used for the C++ aspects of BLOFELD, but if you don't use C++ you can ignore it

Software Design

Approach

Think before you start coding! Try to group tasks into classes, and then think about how they will be used.

Approach

Think before you start coding! Try to group tasks into classes, and then think about how they will be used.

Work on small classes first, and test them.

Approach

Think before you start coding! Try to group tasks into classes, and then think about how they will be used.

Work on small classes first, and test them.

You can even write tests first and then implementation later (test-based development)

Exercise

Exercise

1. Create a virtual base class Farm with “stub methods” to:

- update
- return
- reset to the initial status

2. Create a derived class PigFarm that inherits from Farm but implements the methods:

- update to update an SIR model
- return to return the Time and current number of S, I and R animals (as a data frame)
- reset to reset the farm to the status at time 0

3. Create a simple “Simulation” class that takes a Farm class as input, and has the following methods:

- run (call reset of farm, then run for however many time points)
- extend (as for run, but without the reset)
- both should return the status per time point

4. Run the simulation

5. (Optional) Create a new derived class that implements an SEIR model, and use it with the Simulation class