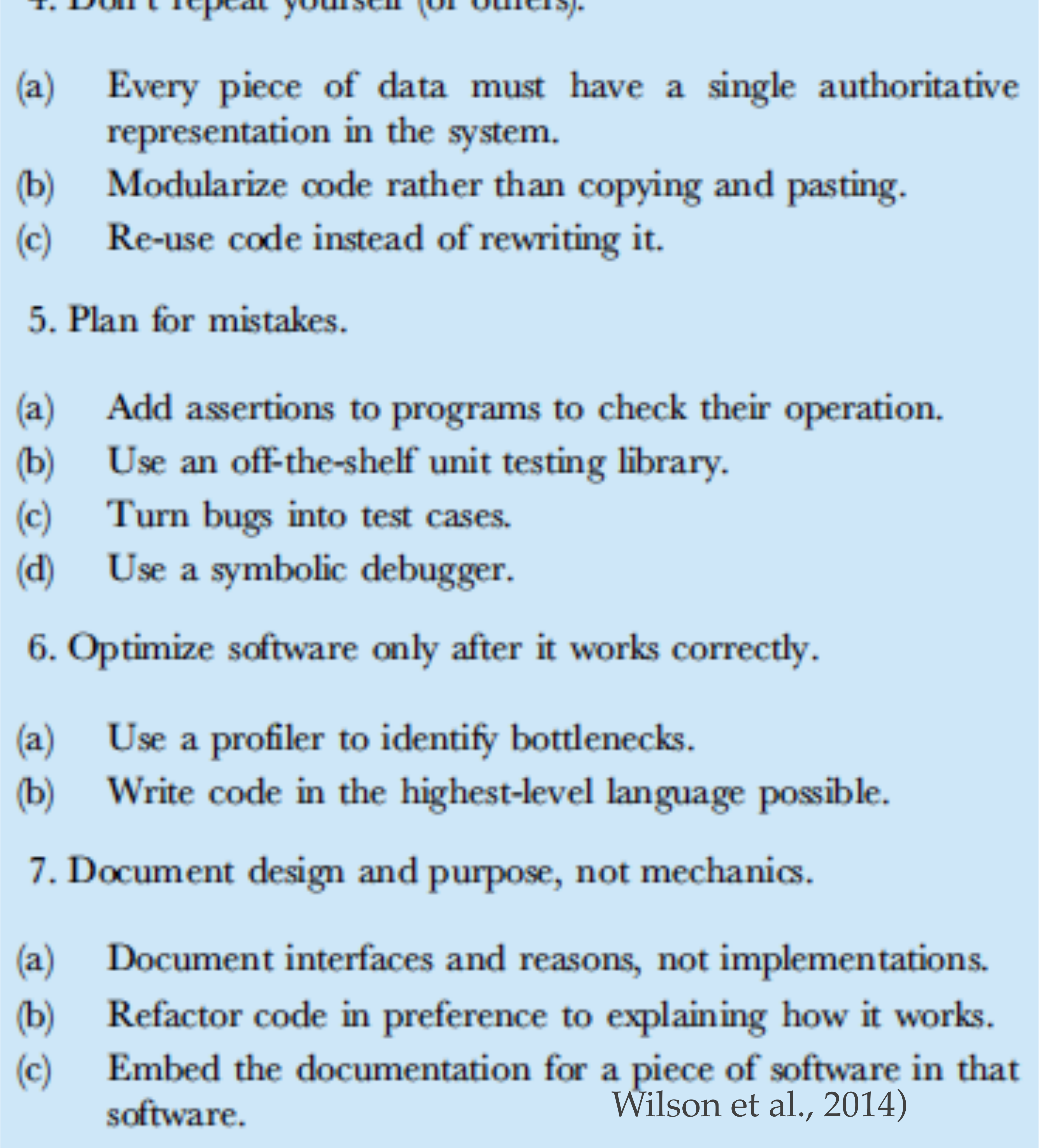
esting and



4. Don't repeat yourself (or others).

(a) Every piece of data must have a single authoritative representation in the system.
(b) Modularize code rather than copying and pasting.
(c) Re-use code instead of rewriting it.

5. Plan for mistakes.

(a) Add assertions to programs to check their operation.
(b) Use an off-the-shelf unit testing library.
(c) Turn bugs into test cases.
(d) Use a symbolic debugger.

6. Optimize software only after it works correctly.

(a) Use a profiler to identify bottlenecks.
(b) Write code in the highest-level language possible.

7. Document design and purpose, not mechanics.

(a) Document interfaces and reasons, not implementations.
(b) Refactor code in preference to explaining how it works.
(c) Embed the documentation for a piece of software in that software.

t
g
le that
d code

es:
s
ion

Wilson et al., 2014)

Testing

- Wikipedia reports that in 2002, NIST study found that software budge cost the US economy 59.5 billion annually

Testing

- Top 12 Reasons to Write Unit Tests - Burke and Coyner (Java programmers)
- http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html
- Tests reduce bugs in new features
- Tests reduce bugs in existing features
- Tests defend agains other programmers
- Testing forces you to slow down and think
- Testing makes development faster
- Tests reduce fear

Also notes their excuse list: “my code is too simple for tests”, “writing tests is too hard” , “I don’t have time”

Testing

- Types of testing
  - **DESIGN**
  - Does the code perform the functions that you want it to
  - Code specification - write out what you want the code to do - IN DETAIL
    - Flow charts

Testing

- Types of testing
  - **IMPLEMENTATION**
  - Does the code do what you think it does
  - Tricky to do this kind of testing, since if you knew the “correct” result of the code, you won’t need the model
  - Alternative?

Testing

- **IMPLEMENTATION**
  - Give functions/code inputs where you know what the answer should be
    - run your data clean up code on “fake code” where you know what to expect
  - Make sure that outputs conform to known expectations
    - conservation of mass, money, energy
    - positive/negative values
    - relative values

Testing

- **IMPLEMENTATION**
  - Developers now often use software to help them automate the testing process
  - Re-uses tests - makes it efficient to repeat many tests as you develop and modify the code
  - Particularly helpful when you have multiple modules (as in our mangrove example)
  - This type of software is available for R, Python, C etc.
  - In R, “testthat” library is my favorite

Error Checking

- A close cousin of testing is error checking
- Error checking are built-in features in functions/code that return a message to the user if something goes 'bad' -
    - often used to make sure the input data is in the format that the function requires
    - also used to return a message if something about the data gives an NA (e.g from a divide by zero)

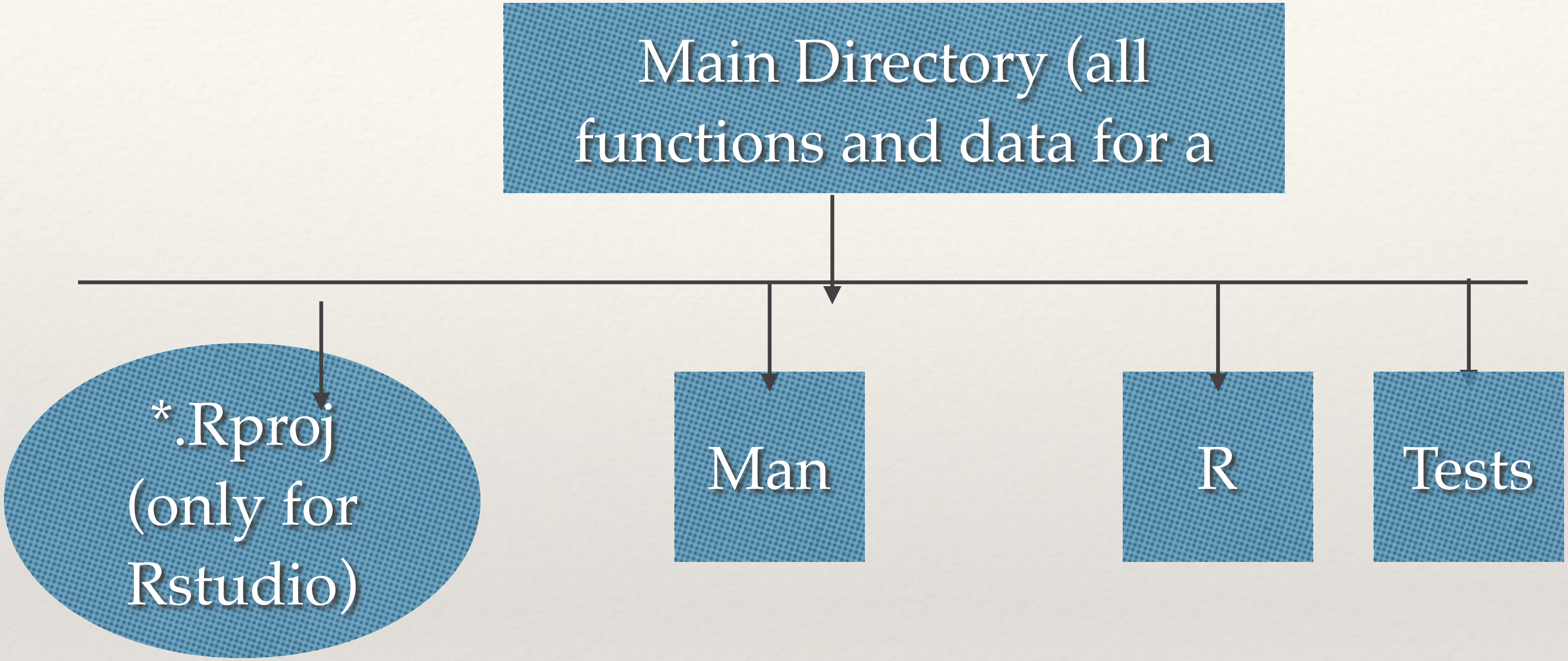
Testing

- There are both “formal” (coded) and “informal” just trying things out
- Automated “formal” testing workflow
  - Design your tests
  - Code them
  - Save in a format that can easily be repeated
  - Run the same set of tests every time you make a change
- In “R” there is a library called “*testhat*” which helps you to do this

Testing in R

- If you are in the working directory where you've stored the files for your project you can use
    - need *devtools* and "*testthat*" libraries
- l*oad_all*() :runs everything in "R" subdirectory)
- *document*() :creates documentation
- *test_dir*("name") :runs all tests in the "name" subdirectory (all files beginning with the word "test"
- *test_file*("name"): runs all the tests in a file called "name"

Building Models: Packages in R



Main Directory (all functions and data for a

*.Rproj (only for Rstudio)

Man

R

Tests

Testing in R

- In R, create a new project, you will give it a directory name;
- make sure you check “create a git repository”
- load the “testthat” and “devtools” libraries
- load your climate processing function

Testing

- **Expectation**
  - tests you can use to make sure your code is working the way you think it should be working
  - basically what you "expect" from your function given certain input parameters
  - often used to test extreme or "bad" values or 0

Testing

- **Test**
    - a single file with multiple expectations
    - one per sub-function; or section of a more complicated pieces of code
    - must start with the word “test”,
        - e.g “test_myfunction.R”

- **Context**
    - a project
    - multiple tests, stored in a directory called “tests”

```r
#' Summary information about spring climate
#'
#' computes summary information about spring temperature and precipitation
#' @param clim.data  data frame with columns tmax, tmin (C)
#' rain (precip in mm), year, month (integer), day
#' @param months (as integer) to include in spring; default 4,5,6
#' @return returns a list containing, mean spring temperature (mean.springT, (C))
#' year with lowest spring temperature (coldest.spring (year))
#' mean spring precipitation (mean.springP (mm))
#' spring (as year) with highest precip (wettest.spring (year))


spring.summary = function(clim.data, spring.months = c(4:6)) {

  spring = subset(clim.data, clim.data$month %in% spring.months)
  mean.springT = mean(c(spring$tmax, spring$tmin))
  lowyear = spring$year[which.min(spring$tmin)]
  spring.precip = as.data.frame(matrix(nrow=unique(spring$year), ncol=2))
  colnames(spring.precip)=c("precip","year")

  spring.precip = aggregate(spring$rain, by=list(spring$year), sum)


  colnames(spring.precip) = c("year","precip")
   mean.spring.precip = mean(spring.precip$precip)
  wettest.spring = spring.precip$year[which.max(spring.precip$precip)]

  return(list(mean.springT = mean.springT, coldest.spring=lowyear,
              mean.springP=mean.spring.precip,wettest.spring=wettest.spring ))
}
```

Testing

- **Expectation**

Functional

  - Output years should be within the range of initial years
  - If we give function, climate with all zeros, mean spring P will be zero

Physical

  - Mean spring P should be greater than zero
  - Temperatures should be between -50 and 50

Testing

- **Expectation (built in)**
    - *expect_that*(function, *equals*(value))
    - *expect_that*(function, *is_identical_to*(value))
  - difference between equals and is_identical_to is that equals included a tolerance (really really small difference OK)
    - *expect_that*(function, *matches*(value))
    - *expect_that*(function, *is_true*())
    - *expect_that*(function, *throws_error*(string))
  - You can also write your own

```
expect_that(4+7, equals(11))
expect_that(4+7 > 10, is_true())
expect_that(4+7 < 10, is_true())
expect_that(“animal”, matches(“lion”))

expect_that((-4)**2, throws_error())
expect_that(sqrt(-4), throws_error())
expect_that(sqrt(-4), gives_warning())
```

An example test

```
clim.data = as.data.frame(cbind(month=c(1:4), day=rep(1,
times=4), year=rep(1,times=4),
rain=rep(0, times=4), tmax=c(2,2,1,1), tmin=rep(0, times=4)))
```

Expectation Call to your function

```
expect_that(spring.summary(clim.data,
spring.months=4)$mean.springP, equals(0))
```

Tests that function works properly by giving it zero rainfall

An example test

```
test_that(“spring.summary.works” ,
{clim.data = as.data.frame(cbind(month=c(1:4), day=rep(1,
times=4), year=rep(1,times=4),
rain=rep(0, times=4), tmax=c(2,2,1,1), tmin=rep(0, times=4)))

expect_that(spring.summary(clim.data,
spring.months=4)$mean.springP, equals(0))
})
```

Put the expectations and test input data
into a single “test” with a name that says what it does
(because you may have multiple tests!)

An example test

```
test_that(“spring.summary.works” ,
{clim.data = as.data.frame(cbind(month=c(1:4), day=rep(1,
times=4), year=seq(from=1,to=4),
rain=rep(0, times=4), tmax=c(2,2,1,1), tmin=rep(0, times=4)))

expect_that(spring.summary(clim.data,
spring.months=4)$mean.springP, equals(0))
expect_that(spring.summary(clim.data,
spring.months=4)$mean.springT, equals(1))
expect_that(spring.summary(clim.data,
spring.months=1)$mean.springT, equals(0.5))
expect_true(spring.summary(clim.data,
spring.months=c(1:4)$coldest.spring > 2)

})
```

put multiple expectations in the test

```
test_that("spring.summary.works" ,
{
  clim.data = as.data.frame(cbind(month=c(1:4), day=rep(1, times=4),
year=rep(1,times=4),
                                  rain=rep(0, times=4), tmax=c(2,2,1,1), tmin=rep(0,
times=4)))

 expect_that(spring.summary(clim.data,
spring.months=4)$mean.springP, equals(0))
 expect_that(spring.summary(clim.data,
spring.months=4)$mean.springT, equals(0.5))
 expect_that(spring.summary(clim.data,
spring.months=1)$mean.springT, equals(1))
 expect_true(spring.summary(clim.data, spring.months=c(1:4))
$coldest.spring > 2)
})
Error: Test failed: 'spring.summary.works'
Not expected: spring.summary(clim.data, spring.months = c(1:4))
$coldest.spring > 2 isn't true.
```

Testing

```
test_dir("tests")
...1

1. Failure(@test.climate.processing.R#9): spring.summary.works
--------------------------------------
spring.summary(clim.data, spring.months = c(1:4))$coldest.spring > 2 isn't
true
test_file("tests/test.climate.processing.R")
...1

1. Failure(@test.climate.processing.R#9): spring.summary.works
--------------------------------------
spring.summary(clim.data, spring.months = c(1:4))$coldest.spring > 2 isn't
true
```

Multiple test in a file called “tests/test.climate.processing.R

The name of the test file must start with “test”
This way R will know that these are tests, and can run them
automatically, “test_dir” will run all the tests in a directory

Imagine you have multiple functions as part of your analysis

read.clim.data()
spring.summary()
pop.growth()
ecosystem.vulnerability()
main()

Why Testing

With multiple people working on the R, code making changes…automatic testing each time a change is made is helpful
different tests tied to differ functions so you know where the errors are

Testing workflow

- Develop your tests after you create each module
- Run them first by sourcing in R studio (to make sure original set up works)
- Save as a file in the tests subdirectory
- After you make any changes, run all your test, using *test_dir*

- This will also catch problems that arise when you make a change to one subroutine/submodel and it now now longer works with another one (e.g f you change compute_climatebased_surge so that output is in a different format, this routine might not fail, but adaptation_comparison will

Error Checking

- Other things to consider
  - building error checking into your sub models
    - check that input values are reasonable, if not return and error message
  - working in pairs, one person writes the code, the other tries to break it

Error checking

```
spring.summary = function(clim.data, spring.months = c(4:6)) {

  # check to make sure data is in required format
  requiredcols = c("tmax","tmin","year","month","rain")
  tmp = sapply(requiredcols, match, colnames(clim.data), nomatch=0)
  if (min(tmp)==0) {
       return("Error: Invalid Climate Input") }
  if (min(clim.data$rain < 0)) {}
     return("Error: Invalid Climate Input") }
  clim.data$tavg = (clim.data$tmin + clim.data$tmax)/2.0
  spring = subset(clim.data, clim.data$month %in% spring.months)
  mean.springT = mean(c(spring$tmax, spring$tmin))
  lowyear = spring$year[which.min(spring$tavg)]…..
```

We can also add this to our tests

Testing

- Testing levels
    - Unit testing (your individual subroutine)
    - Integration testing (testing the situations where one submodule call another)
    - Component interface or data passing testing (test format of outputs)
    - System testing (testing the whole model)
- Some of these can be done by “*testthat*” routines, but you can also have informal system testing; or write checks into your code for component interface testing