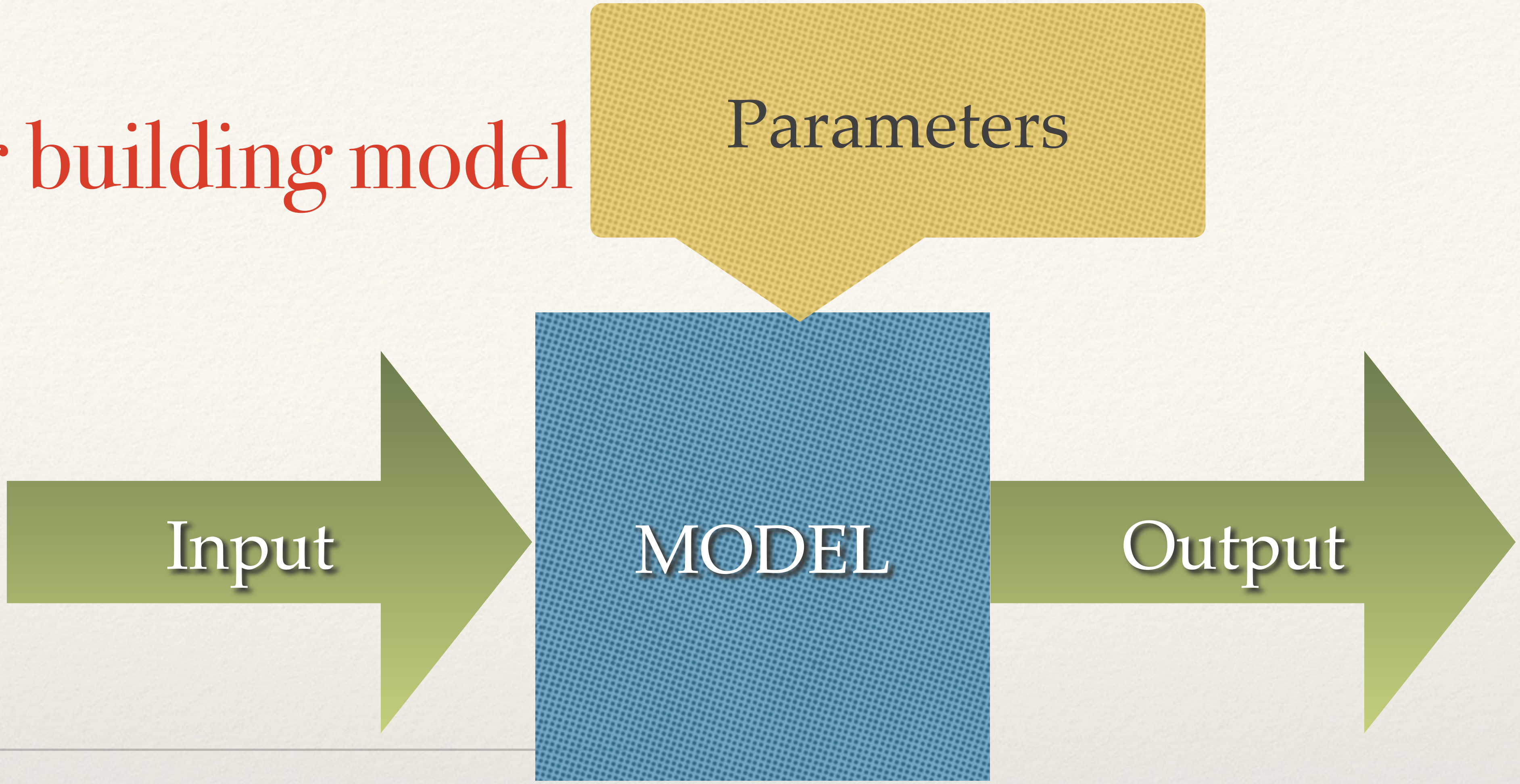
Steps for building model



1. Design conceptual model
2. Translate conceptual model into inputs/output/parms and set of discrete tasks (transfer function)
3. Choose programming language
4. Define inputs (data type, units)
5. Define output (data type, units)
6. Define model structure
7. Write model
8. Document the model (meta data)
9. Test model

Design/Selecting Models

- What are your inputs-outputs
- What's in the box (the model itself) that gives you a relationship between outputs and inputs
  - Transfer function
- Parameters, values that influences how the model relationships work

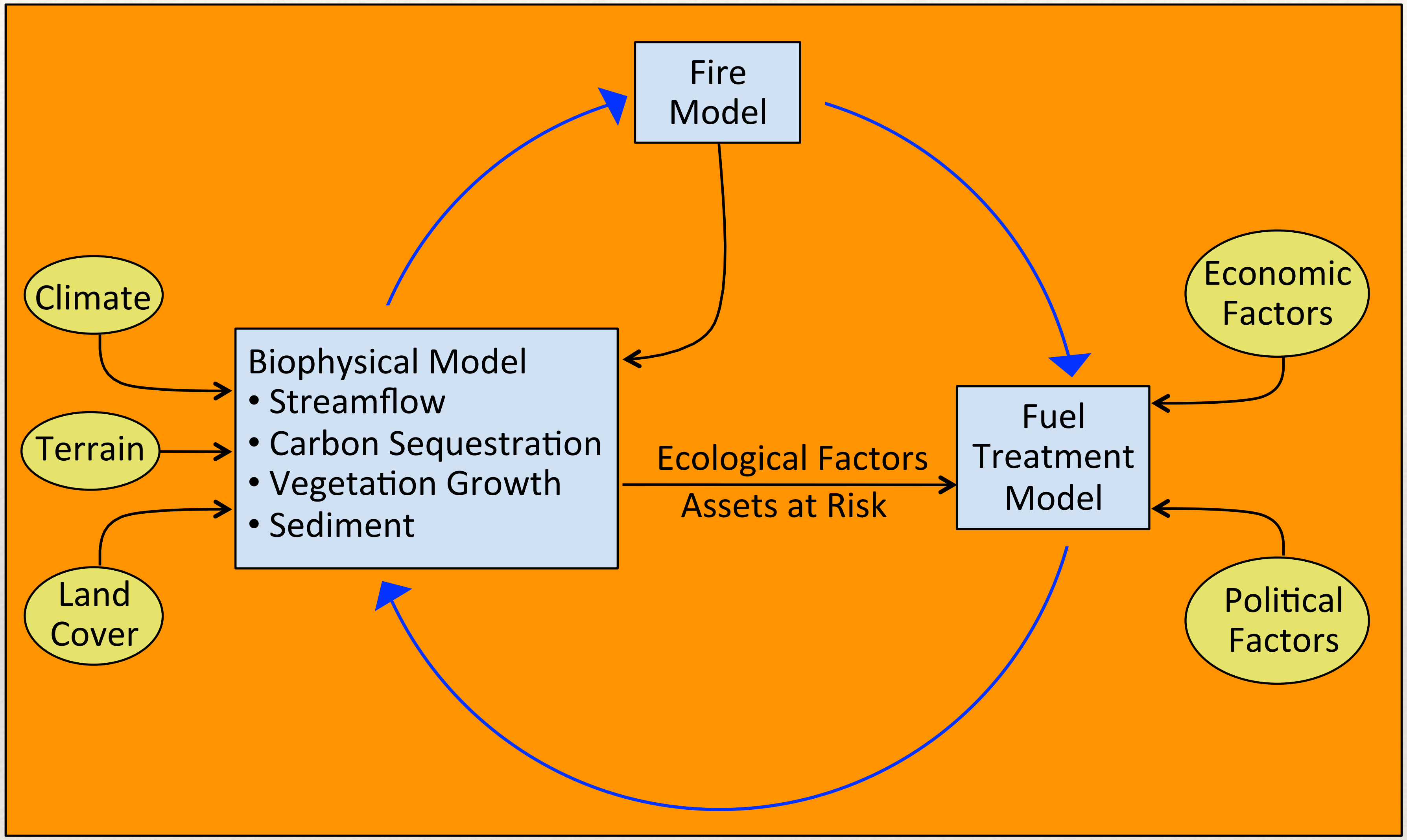
Best practices for model (software) development

- Common problems
  - Unreadable code (hard to understand, easy to forget how it works, hard to find errors, hard to expand)
  - Overly complex, disorganized code (hard to find errors; hard to modify-expand)
  - Insufficient testing (both during development and after)
  - Not tracking code changes (multiple versions, which is correct?)

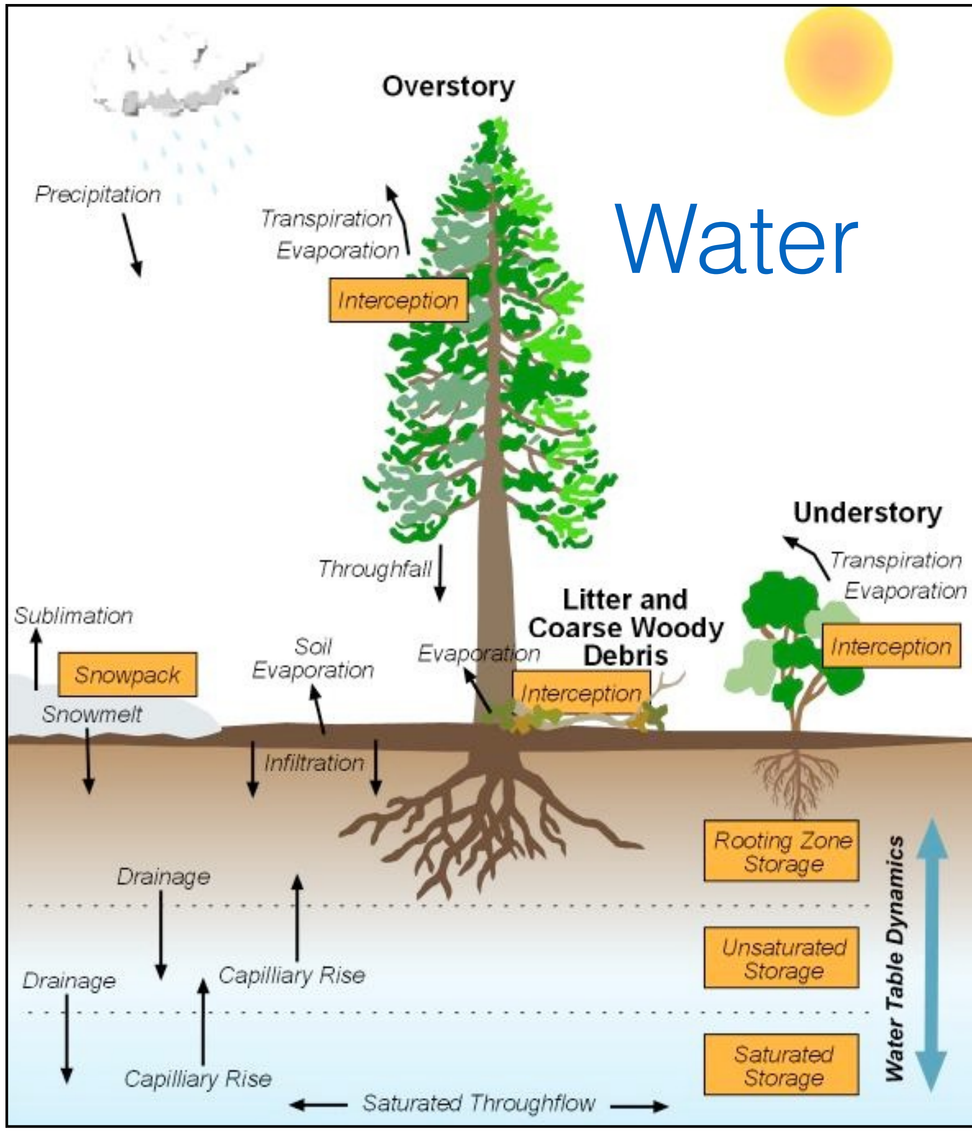
Best practices for model (software) development

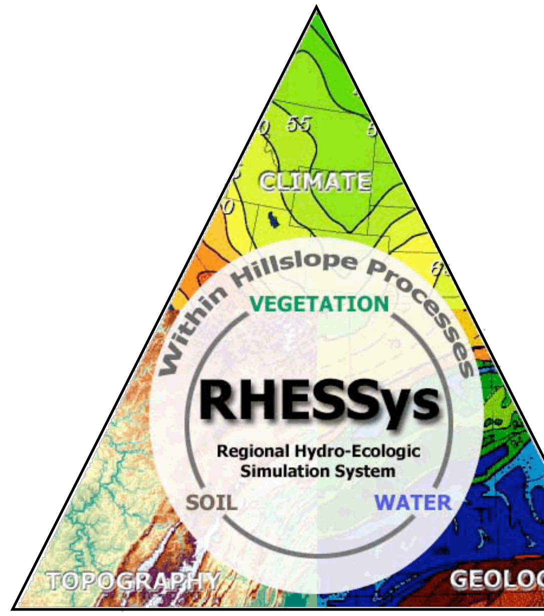
- Basic Solution
  - Structured practices that ensures
    - clear, readable code
    - modularity (organized "independent" building blocks)
    - testing as you go and after
    - code evolution is documented

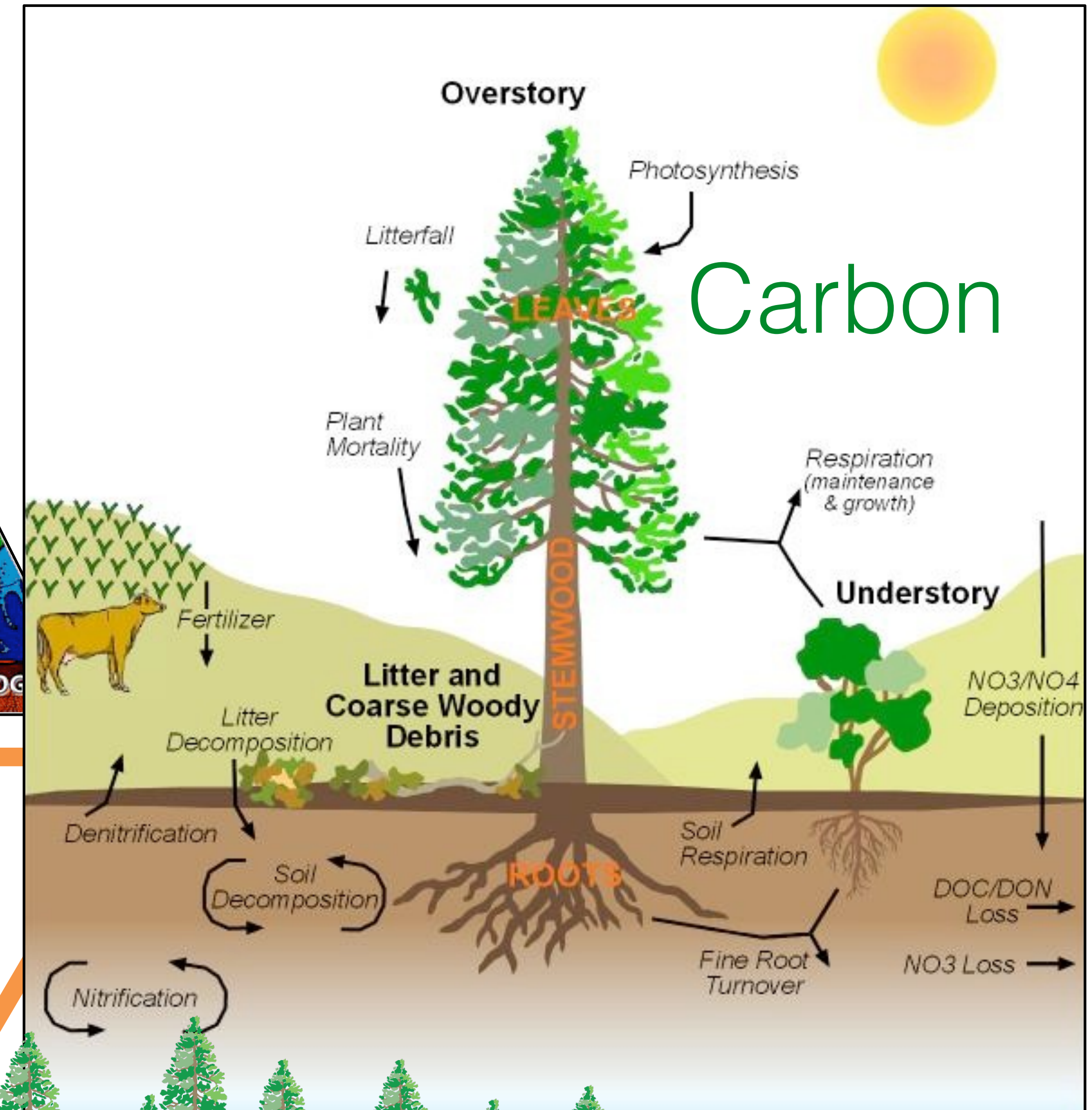
Conceptual Models can be Hierarchical (submodels)



Example: RHESSys-FIRE

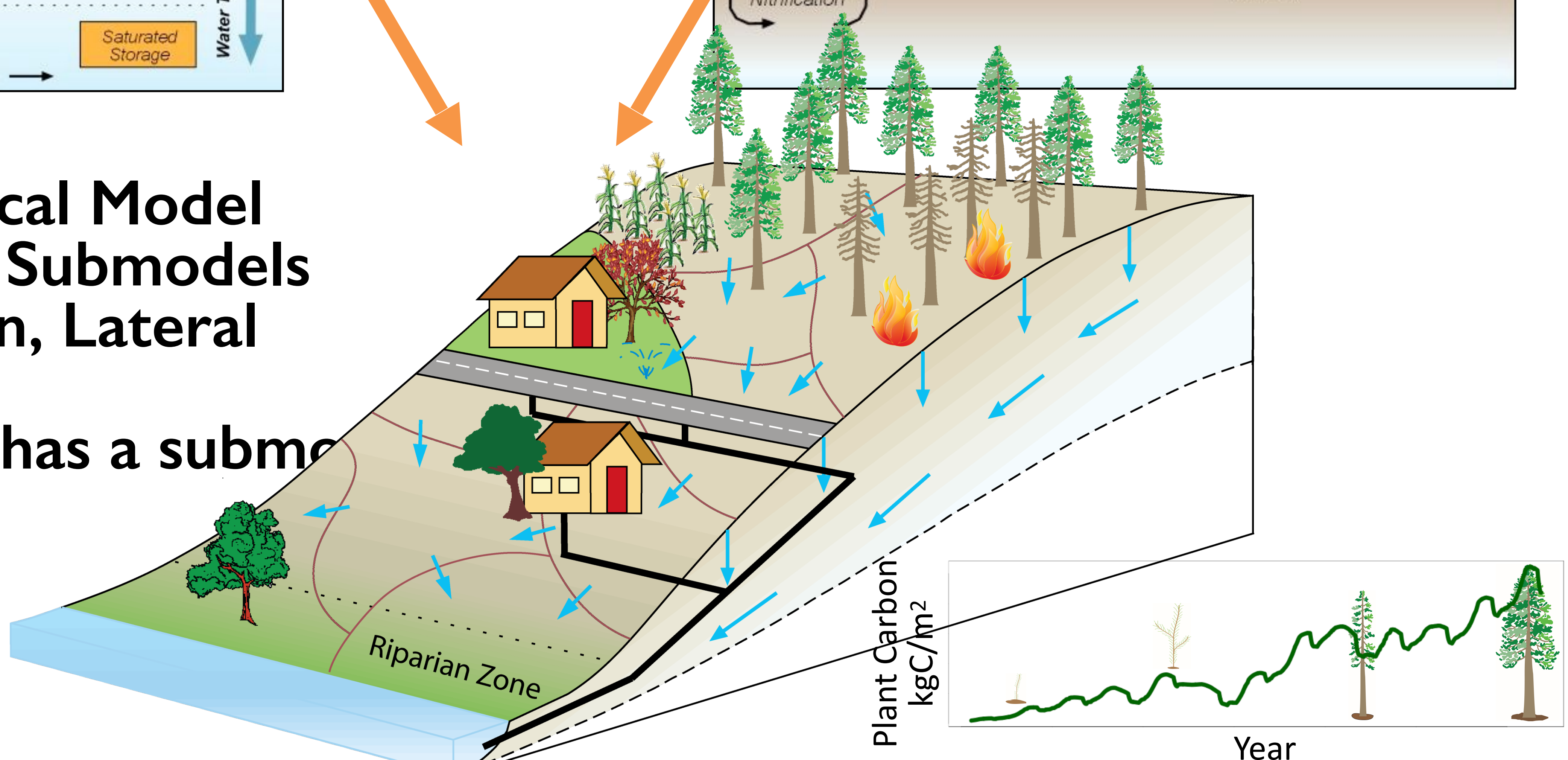






Biophysical Model Itself with Submodels

- Water, Carbon, Lateral Diffusion
- Each of those has a subm



Hierarchical Conceptual Model - Discrete Submodels

- Hierarchical in level of details
- Big chunks (coarse detail) -> progressively finer
- Note the ‘tasks’ that need to be repeated
- All tasks should have inputs and outputs
- Many models are made up of linked sub-models

Building Models

- Functions!
- The basic building blocks of models
- Functions can be written in all languages; in many languages (object-oriented) like C++, Python, functions are also objects
- Functions are the "boxes" of the model - the transfer function that takes inputs and returns outputs
- More complex models - made up of multiple functions; and nested functions (functions that call/user other functions)

Functions

- Write down:
  - all inputs and parameters
  - all outputs
- Decide what there data types, units, names should be
  - use descriptive names
  - use data types that will allow you to apply your function in many different cases

Functions - Contracts

- START with a “contract” - agreement about what the function does
- Write down what the function/submodel will do - given different inputs and parameters
- By communicating this you can a) see if it will meet the goal and b) share it with others (or link with a multi-component model)

Functions - Contracts

- START with a "contract" - agreement about what the function does
- Simple example
  - input (air temperature every day); output (growth rate on that day)
- more complex example
  - inputs (daily temperature for at least one year, organism type, parameters that determine a threshold minimum and maximum temperature for growth, temperature-growth/respiration curve parameters)
  - output (if animal, total annual respiration; if plant, total annual growth)

- Write a contract for a function to estimate the impact of pollution concentration on microbial biomass

Implementing Functions in R

- Format for a basic function in R

#' documentation that describes inputs, outputs and what the function does

FUNCTION NAME = function(inputs, parameters) {

body of the function (manipulation of inputs)

return

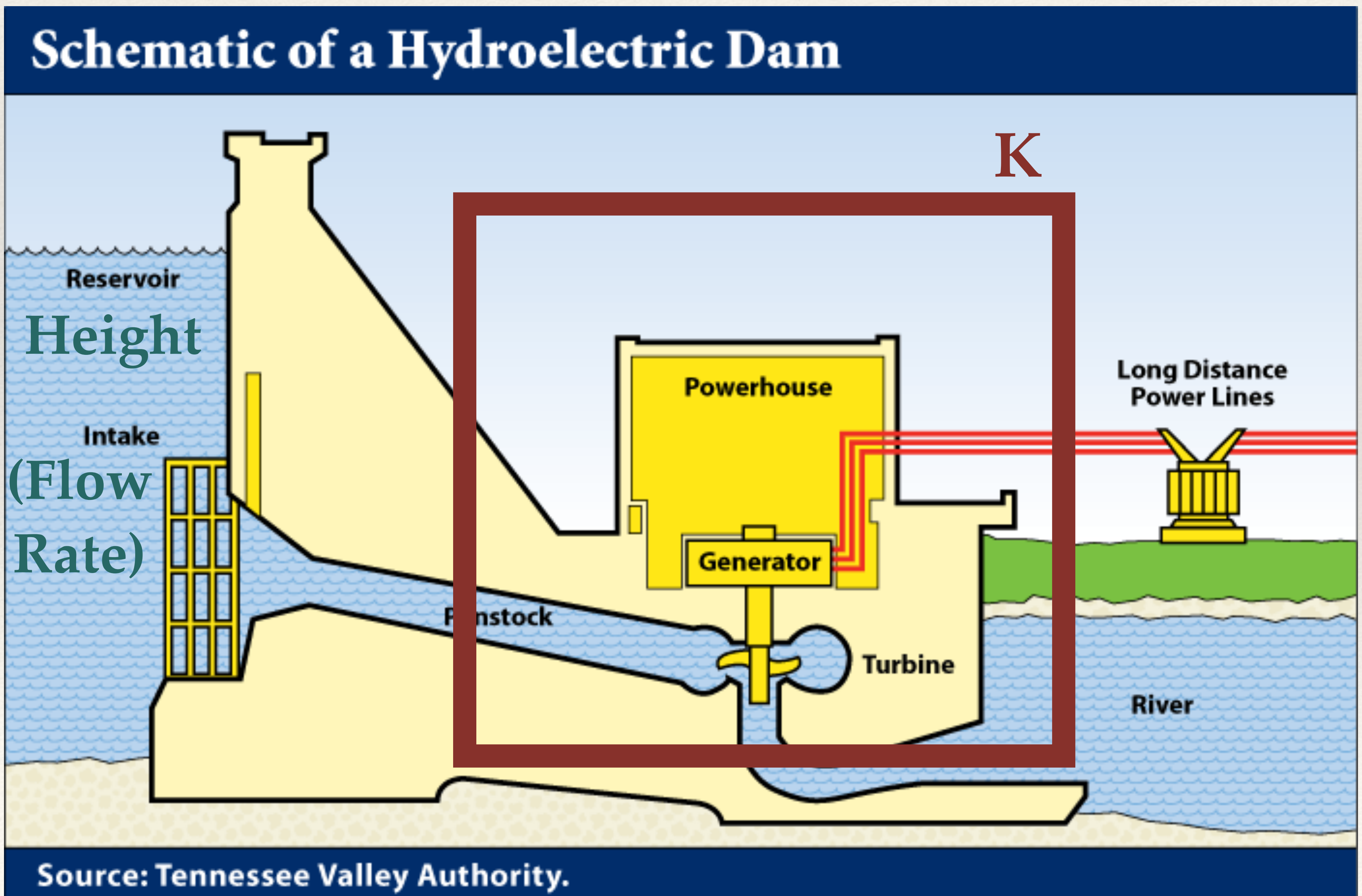
}

In R, inputs and parameters are treated the same; but it is useful to think about them separately in designing the model - collectively they are sometimes referred to as arguments

ALWAYS USE Meaningful names for your function, its parameters and variables calculated within the function

Types of models: Example

- Input: Reservoir height and flow rate
- Output: Instantaneous power generation (W/s)
- Parameter: Reservoir Efficiency
- Conceptual model



Function "Contract"

- Given Input: Reservoir height and flow rate
- Output: Instantaneous power generation (W/s) when reservoir has that height and flow rate
- Parameters: $K_{Efficiency}$, ϱ (density of water), g (acceleration due to gravity)

$P = \varrho * h * r * g * K_{Efficiency};$

P is Power in watts, ϱ is the density of water (~1000 kg/m3), h is height in meters, r is flow rate in cubic meters per second, g is acceleration due to gravity of 9.8 m/s2, $K_{Efficiency}$ is a coefficient of efficiency ranging from 0 to 1.

This is a static (one point in time), deterministic, lumped (one place) model; its more or less physically based

Building Models (see Rmarkdown)

- Inputs height, flow,
- Parameters are rho, g, and K
- For parameters, we provide default values by assigning them a value (e.g Keff = 0.8), but we can overwrite these
- Body is the equations between { and }
- *return* tells R what the output is

```
power_gen = function(height, flow, rho=1000, g=9.8, Keff=0.8) {

result = rho * height * flow * g * Keff
return(result)
}
```

Building Models

- write your function in a text editor and then copy into R or in R studio create a new R script
- By convention we name files with functions in them
  - the name of the function.R
  - so power_gen.R
- you can also have R read a text file by *source("power_gen.R")* - make sure you are in the right working directory
- Eventually we may want our function to be part of a package (a library of many functions) - to create a package you must use this convention (name.R)

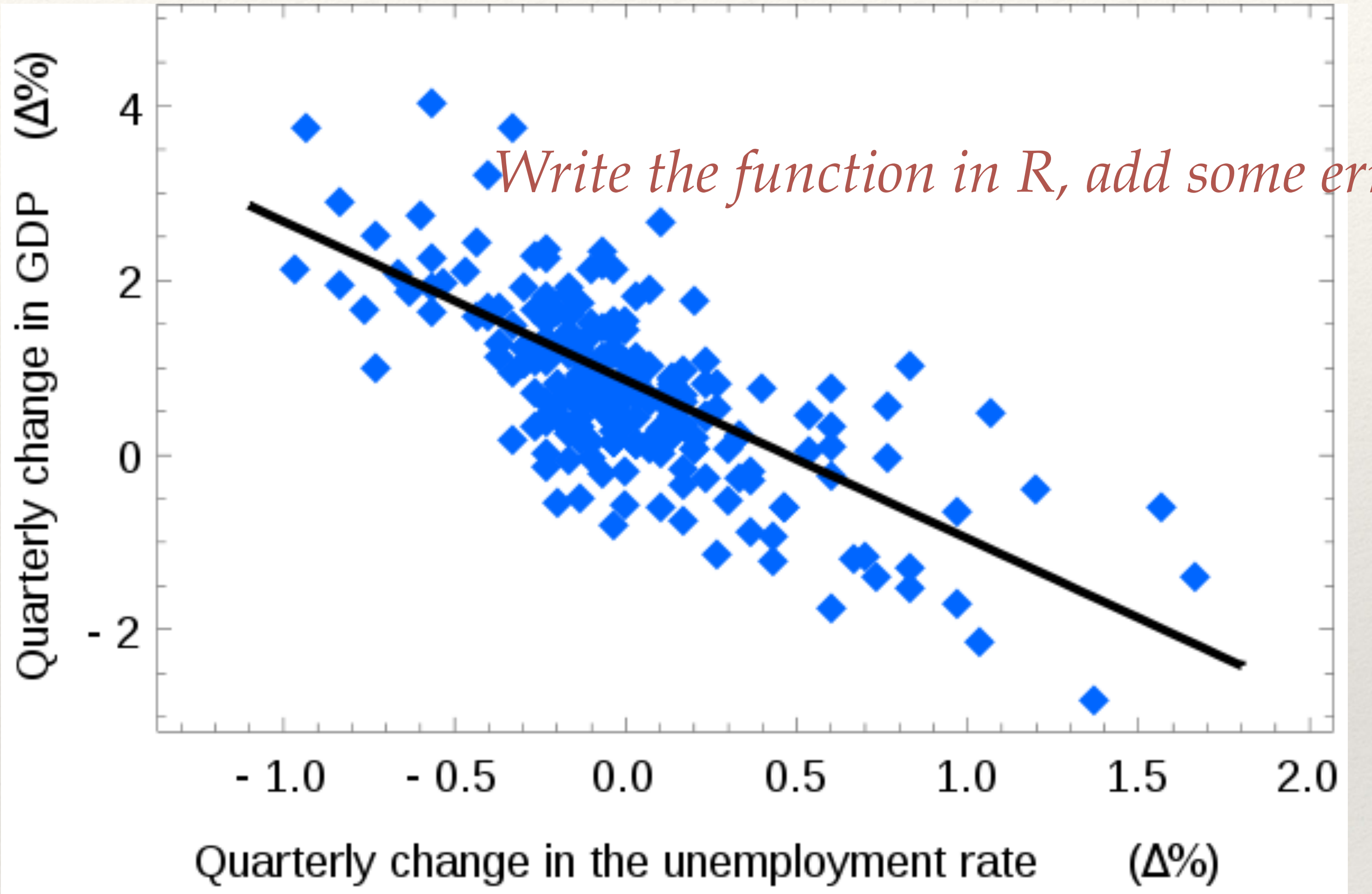
Work flow*

- Keep your function definitions "clean"
  - in their own file with ONLY the function definition and documentation (not its application!)
  - Ideally in a subdirectory called "R"
- Use a different document (R markdown is a good choice) to keep track of scripts that you use to execute your function with different datasets
  - For this course I'll put these files in directory called Rmarkdown
- WHY?
  - Increases reusability and sharing of the function
  - Allows you to add to a package if you are creating a multi-component model

Building Models

- Another example: Okuns Law (conceptual, abstract model)

%Change GNP = .856 - 1.827*(Change Unemployment Rate).



Write the function in R, add some error checking

Graph of US quarterly data (not annualized) from 1947 through 2002 estimates a form of the difference version of Okun's law:
%Change GNP = .856 - 1.827*(Change Unemployment Rate).
R^2 of .504.

http://en.wikipedia.org/wiki/Okun%27s_law

```
#' Okuns Law
#'
#' function uses Okuns Law to estimate the quarterly change in GDP,
#’ from the quarterly change in unemployment
#' @param delta.unemploy Change in Unemployment Rate as percent
#' @param slope Slope of linear relationship Default is -1.827
#' @param intercept Intercep of linear relationship Default is 0.856
#' @examples
#' okun(delta.unemploy=3)
#' @references
#' Okun, Arthur, M, Potential GNP, its measurement and significance (1962).
Cowles Foundation, Yale University.
#' \url{http://cowles.econ.yale.edu/P/cp/p01b/p0190.pdf}


okun = function(deltaunemploy, intercep=0.856) {

deltaGDP = deltaunemploy*slope+intercep
return(deltaGDP)
}
```

What’s wrong with this model?

```r
#' Okuns Law
#'
#' function uses Okuns Law to estimate the quarterly change in GDP,
#' from the quarterly change in unemployment
#' @param deltaunemploy Change in Unemployment Rate as percent
#' @param slope Slope of linear relationship Default is -1.827
#' @param intercept Intercep of linear relationship Default is 0.856
#' @examples
#' okun(deltaunemploy=3)
#' @references
#' Okun, Arthur, M, Potential GNP, its measurement and significance (1962).
Cowles Foundation, Yale University.
#' \url{http://cowles.econ.yale.edu/P/cp/p01b/p0190.pdf}


okun = function(deltaunemploy, slope=-1.827, intercep=0.856) {

deltaGDP = deltaunemploy*slope+intercep

return(deltaGDP)
}
```

Usage

```r
> okun(2)
[1] -2.798
> okun(-2)
[1] 4.51
> okun(-2, slope=-1.9)
[1] 4.656
```