High Performance Computing in Julia

from the ground up.

Research Software Engineering

8/10

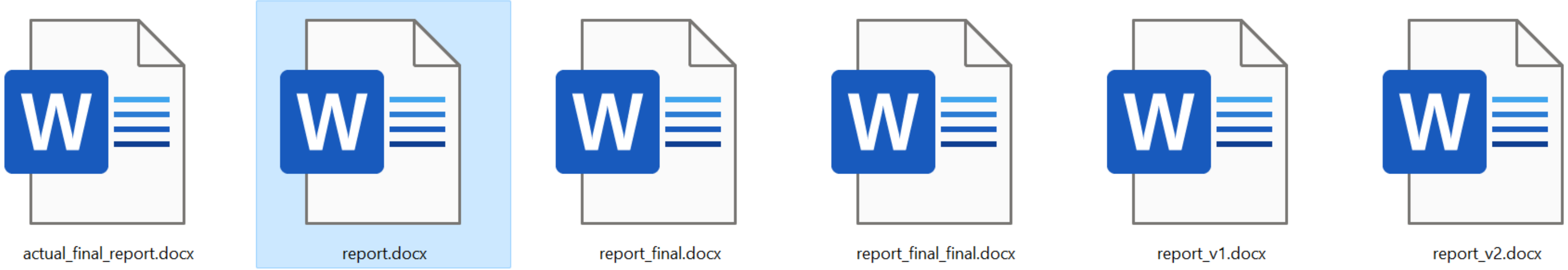
Aims

- To discuss key skills necessary for writing high quality, professional code
- Provide resources on how to develop your SE skills
- Discuss how to work with others & contribute to open source projects
- Show an example of writing a public package

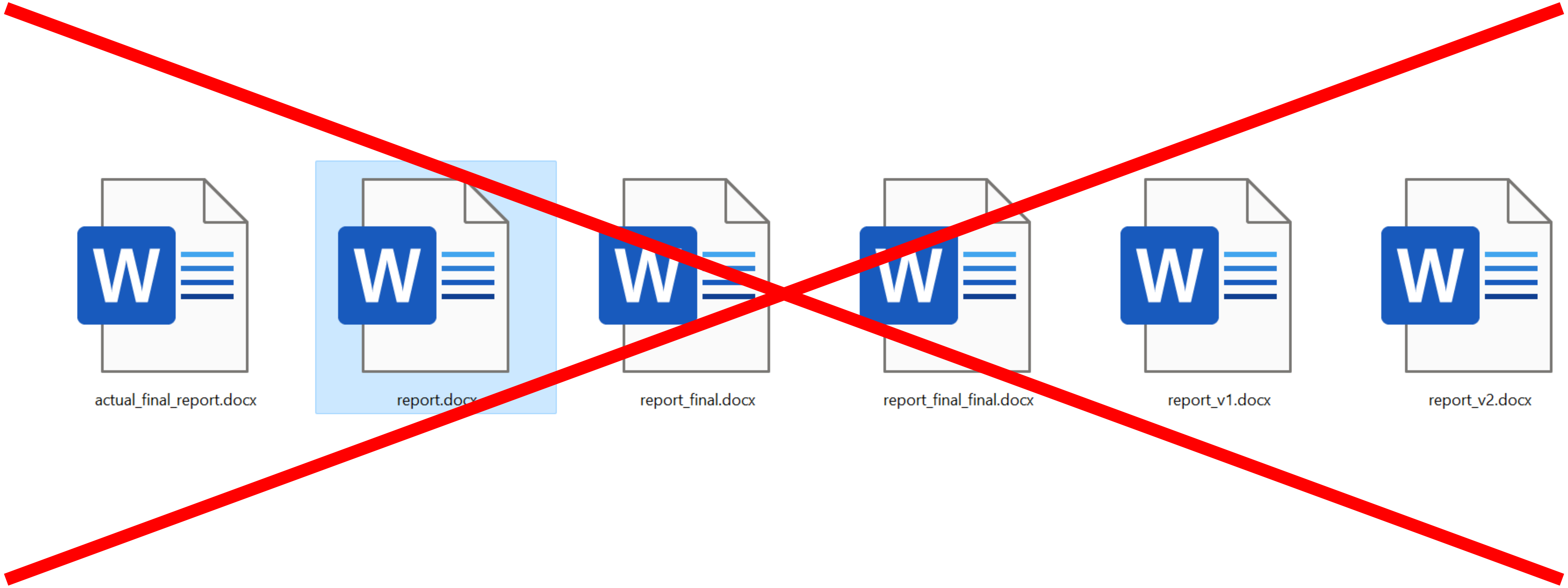
Software Engineering Skills

- Version Control
- Testing
- Documentation
- Reproducibility

Version Control



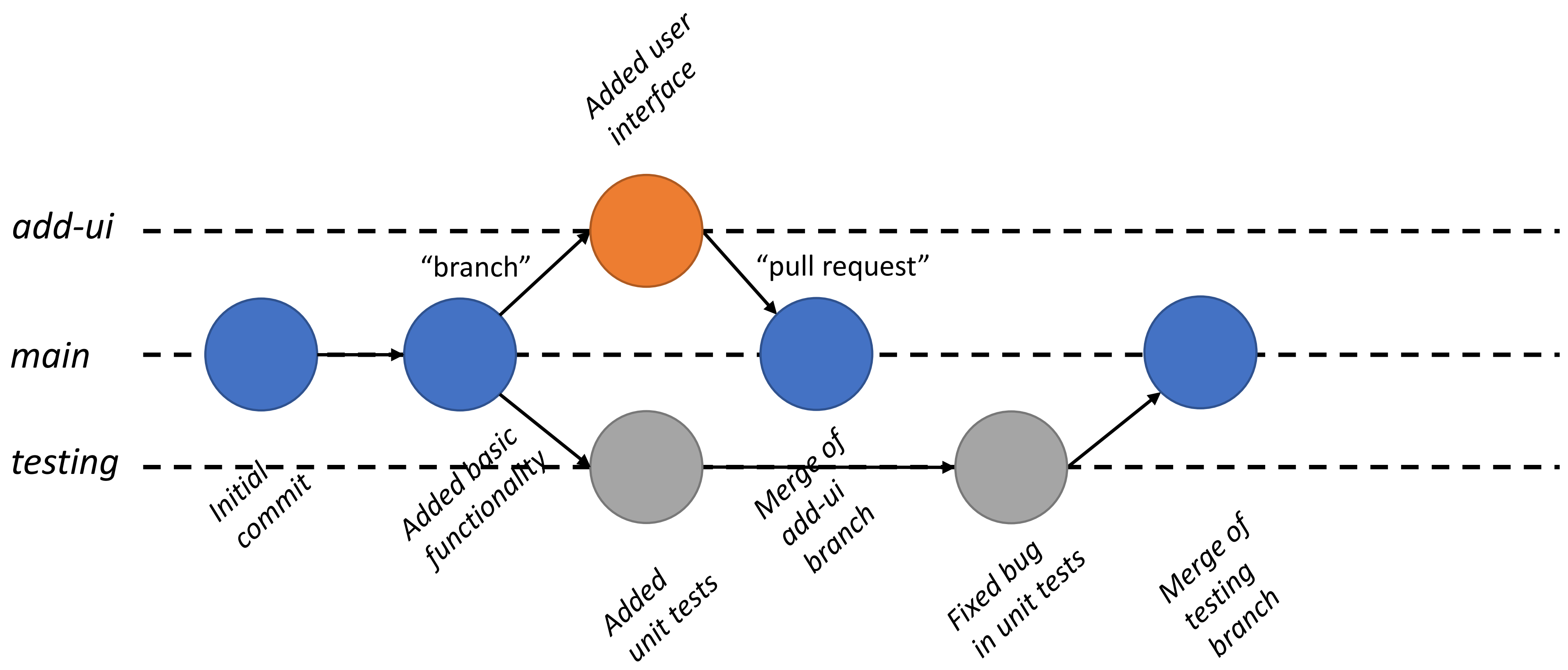
Version Control



Version Control

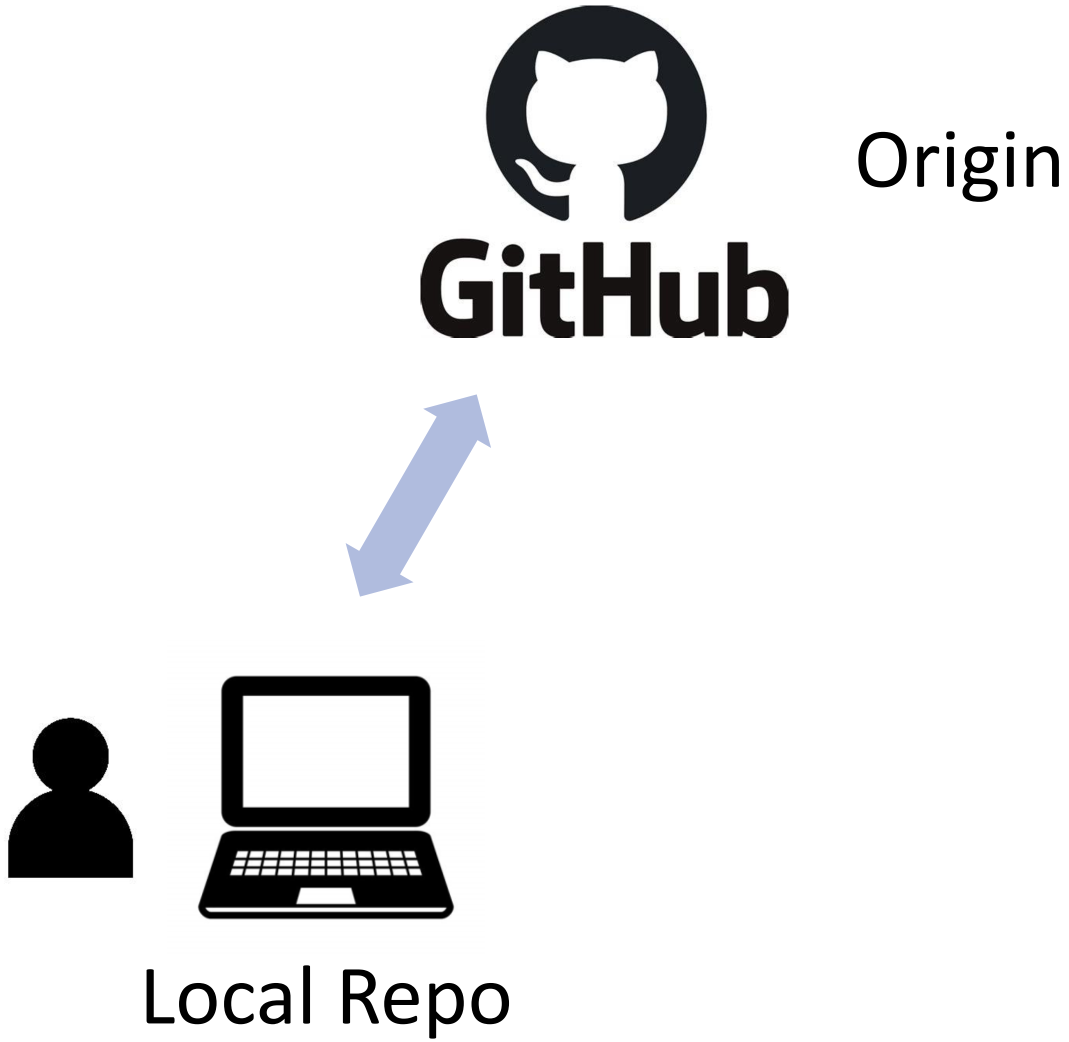
Version Control

Version Control

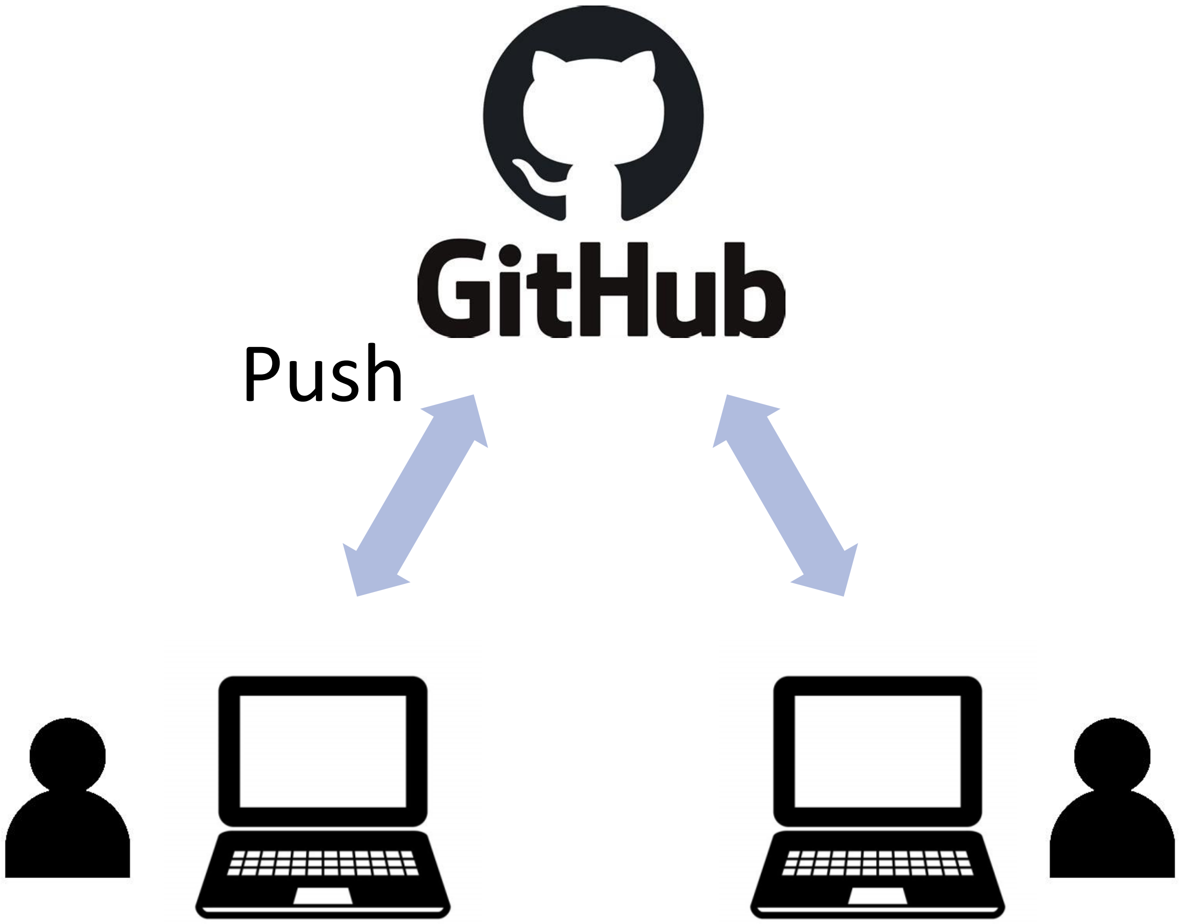


Git & GitHub

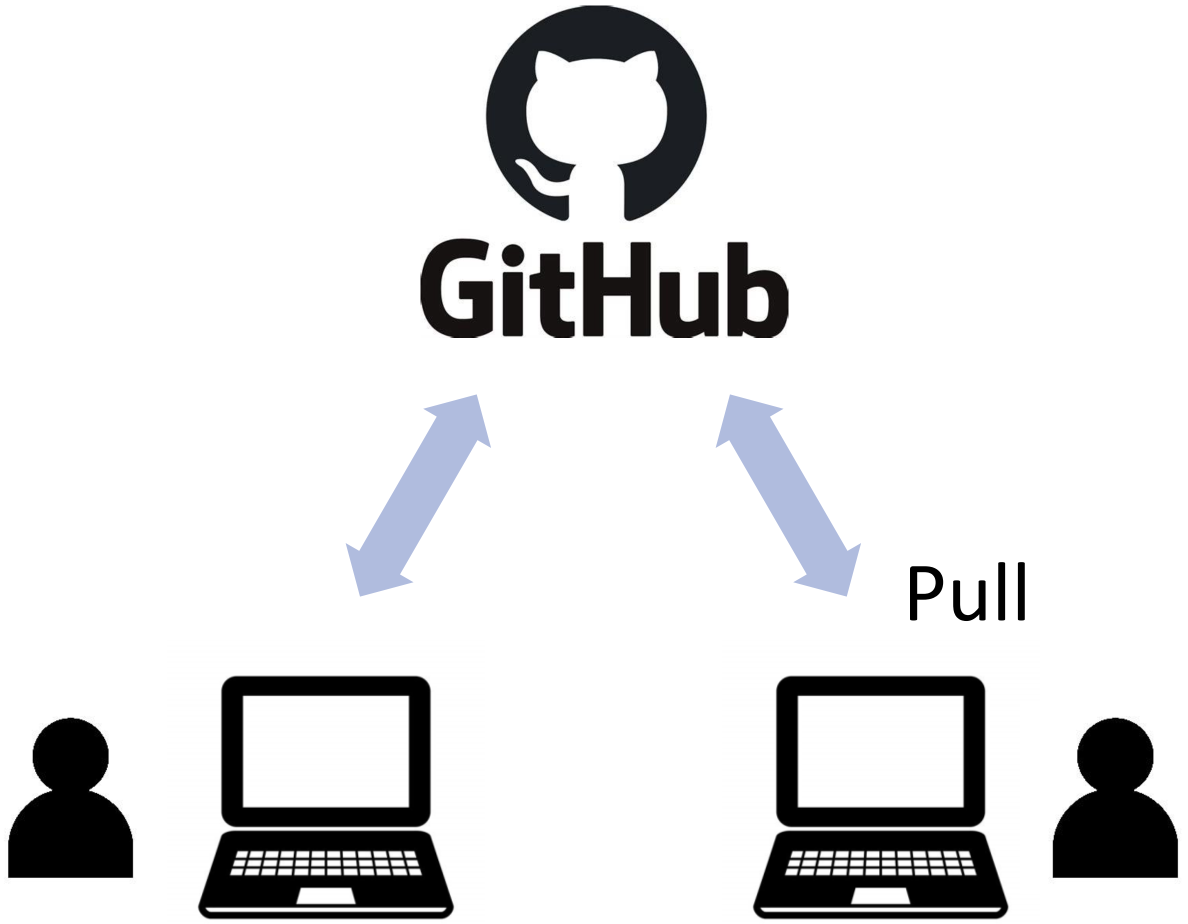
Git & GitHub



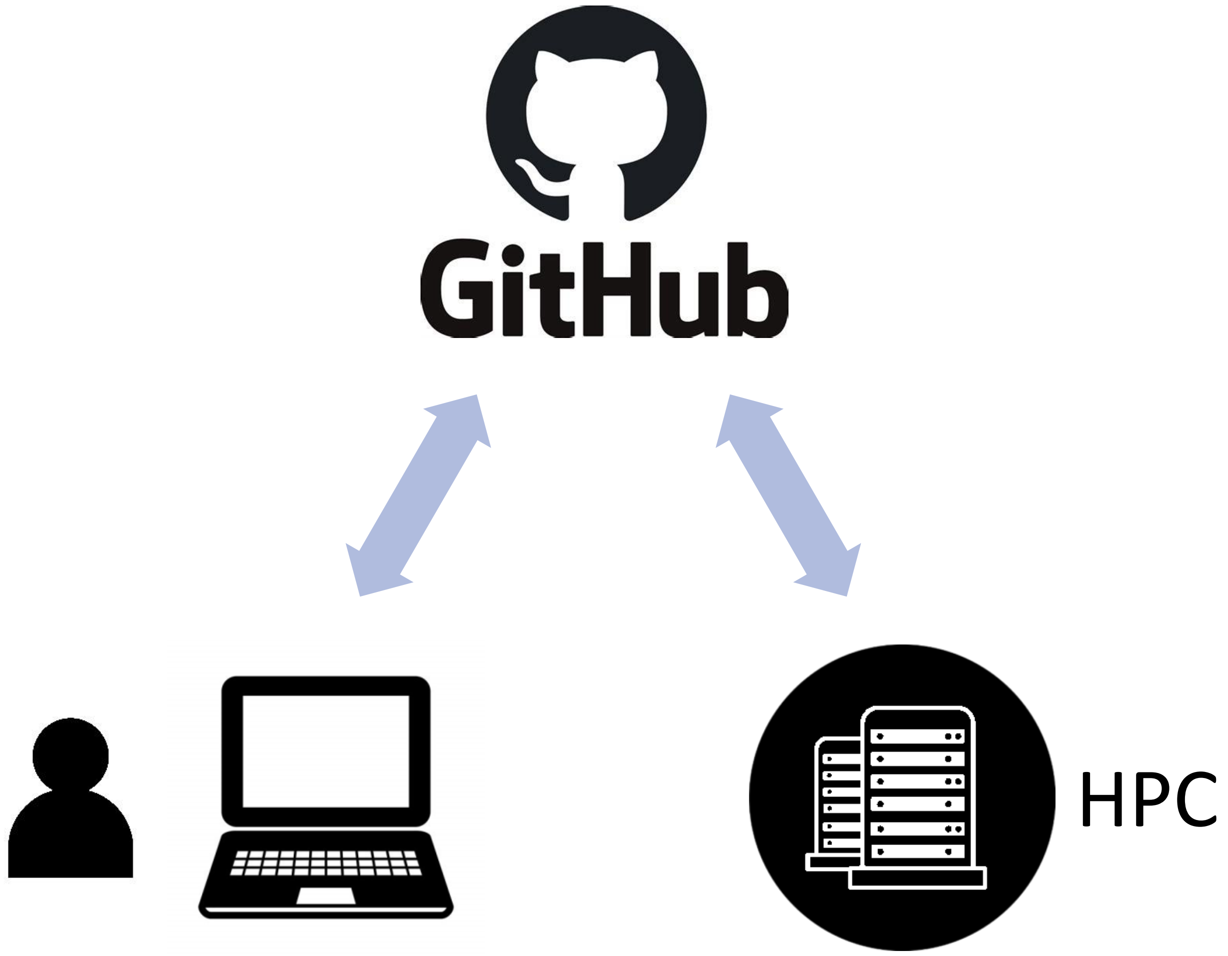
Git & GitHub



Git & GitHub



Git & GitHub



How to learn Git

- Avoid the command line
- Use a Git GUI tool, like **GitHub Desktop** or **GitKraken**
- Follow online guides (e.g. https://docs.github.com/en/get-started)
- Upload an existing project to GitHub and start using it for all your projects
- Write your papers using Git (integrates with Overleaf).

Documentation

How to write easily understandable code

Commenting is **not** documentation

- Comments are redundant, they can be inferred by the code
- Variable names are not descriptive, especially function names
- Comments are only necessary because of the poor variable names

```
"""
    d(a, b)

Calculates the distance between a and b,
where a and b are vectors.
"""
function d(a, b)
    # Check the make sure the vectors are the same length
    @assert length(a)==length(b)

    # Calculate the vector difference of a and b
    dlta = b .- a
    # Calculate sum of the squares of the difference
    l2 = sum(dlta.*dlta)
    # Calculate sqrt of the square sum to find dist
    l = sqrt(l2)

    return l
end
```

Commenting is **not** documentation

- Docstring disambiguates the type of distance, allowing a shorter function name
- The type restriction helps document how this function should be used
- Pythagoras is very standard, and does not need explaining

```
"""
    distance(a, b)

Calculates the Euclidean distance between a and b.
"""
distance(a<:AbstractVector, b<:AbstractVector) = sqrt(sum((a.-b).^2))
```

```julia
"""
    fit(x, y)

Fits a linear function y=mx+c using least squares method.
Returns (m, c).
"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return (m, c)
end
```

```julia
abstract type AbstractModel end
struct LinearModel{T}
    slope::T
    intercept::T
end
predict(model::LinearModel, x) = model.slope * x + model.intercept

"""
    fit(x, y)

Fits a linear function y=mx+c using least squares method.
Returns (m, c).
"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return (m, c)
end
```

```julia
abstract type AbstractModel end
struct LinearModel{T}
    slope::T
    intercept::T
end
predict(model::LinearModel, x) = model.slope * x + model.intercept

"""
    fit(x, y)

Fits a linear function y=mx+c using least squares method.
Returns (m, c).
"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return LinearModel(m, c)
end
```

```julia
"""
    fit(x, y)

Fits a linear function y=mx+c using least squares method.
Returns (m, c).

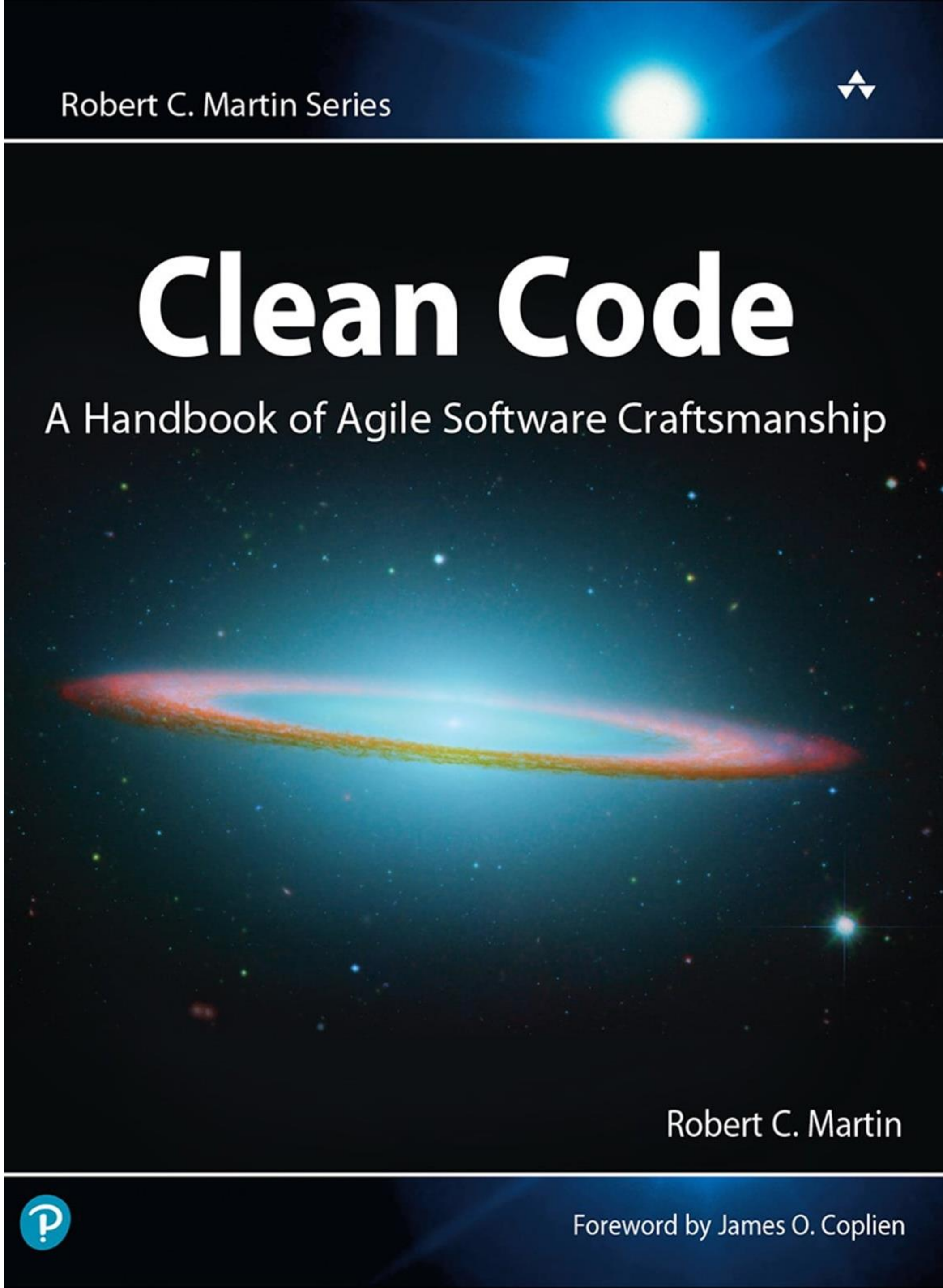
# Examples
```jldoctest
julia> x = [1, 5, 9];
julia> y = 3 .* x .- 7;
julia> linear_fit(x, y)
(3.0, -7.0)
```
"""
function linear_fit(x, y)
    n = length(x)
    @assert n==length(y)
    sum_xx = sum(x.*x)
    sum_xy = sum(x.*y)
    sum_x = sum(x)
    sum_y = sum(y)
    denominator = n * sum_xx - sum_x*sum_x
    m = (n*sum_xy - sum_x * sum_y) / denominator
    c = (sum_y*sum_xx - sum_x * sum_xy) / denominator

    return (m, c)
end
```

Documentation Overview

- Write good docstrings with examples (and possibly tests) for your public API
- Use good, descriptive variable names
- Comments can be used to reflect the *why* of the the code, instead of the *what*
- Code changes over time – make sure the comments are updated too!

- Read “**Clean Code**” by **Robert Martin**



Unit Testing

How to make sure your code is correct

Unit Testing

```julia
@testset "Database creation" begin
    db = create_db(in_memory=true);
    @testset "Customer table creation" begin
        @test has_table(db, "customer")
    end
    @testset "Product table creation" begin
        @test has_table(db, "product")
    end
    @testset "Supplier table creation" begin
        @test has_table(db, "supplier")
    end
    @testset "Inventory table creation" begin
        @test has_table(db, "inventory")
    end
    @testset "Staff table creation" begin
        @test has_table(db, "staff")
    end
end
```

- Unit tests are **small**, self-contained programs that test the **outputs** of a function
- Can be used to check for both logical & syntactic **errors**
- Ensures software works as expected
- Should test the edge cases especially
- Automates the process of testing the software after changes

```
(Experimenter) pkg> test

     Testing Running tests...
Test Summary:    | Pass  Total  Time
Experimenter.jl |   23     23  7.8s
      From worker 3:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 2:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
Test Summary: | Pass  Total   Time
Runner        |   24     24  26.3s
      From worker 5:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 4:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 4:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 5:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
Test Summary: | Pass  Total   Time
Snapshots     |   21     21  20.0s
      From worker 7:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 6:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 7:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
      From worker 6:      Activating project at `C:\Users\jamie\AppData\Local\Temp\jl_jEpmOD`
Test Summary:      | Pass  Total   Time
Restore from trial |   39     39  11.6s
     Testing Experimenter tests passed
```

Unit Testing

- Unit test the outward facing API of your code
- Don’t focus on testing every internal function, just the important ones
- Try out Test Driven Development (TDD)
- Integrate CI pipelines into your code – automatically run tests on pull requests
- Monitor code coverage over time

Reproducibility

Ensuring that you can get consistent results

Sharing code with others

Absolute Paths

```python
import numpy as np

def get_data():
    path = "D:\\Development\\University\\rledts\\data\\current.npy"

    return np.load(path)
```

Inject Folder

```python
import numpy as np
import os
def get_data(data_folder: str):
    path = os.path.join(data_folder, "current.npy")

    return np.load(path)
```

Sharing code with others

Absolute Paths

```python
import numpy as np

def get_data():
    path = "D:\\Development\\University\\rledts\\data\\current.npy"

    return np.load(path)
```

Relative Path from file

```python
import numpy as np
import os
def get_data():
    src_dir = os.path.dirname(__file__)
    data_dir = os.path.join(src_dir, os.pardir, "data")
    path = os.path.join(data_dir, "current.npy")

    return np.load(path)
```

Reproducible Environment

- Make sure you are always using an environment in Julia
- Keep track of packages installed in **Package.toml**
- Keep your global environment clean of packages to avoid conflicts and to ensure your dev environment has all the packages necessary
- Allows others to clone your code and run easily

Random Number Generation

- The majority of random numbers generated by a machine are **pseudo-random**
- You can ***seed*** a RNG to produce **predictable** and **reproducible** results
- Is useful for regenerating data or running a simulation for longer
- Can be useful for debugging if some errors only occur randomly

```
julia> Random.seed!(1234);

julia> sum(rand(10:99, 100_000))
5457782

julia> sum(rand(10:99, 100_000))
5442012

julia> Random.seed!(1234);

julia> sum(rand(10:99, 100_000))
5457782

julia> sum(rand(10:99, 100_000))
5442012
```

Contributing to Open Source Projects

Introduction to Open Source development (1/2)

Creating a Julia Package

Introduction to Open Source development (2/2)

Resources

- Chris Rackauckas – "Developing Julia Packages" - https://www.youtube.com/watch?v=QVmU29rCjaA
- PkgTemplates.jl - https://juliaci.github.io/PkgTemplates.jl
- Documenter.jl - https://documenter.juliadocs.org

- **Example:** https://github.com/JamieMair/Experimenter.jl