Building Confidently in Julia with Interface-Driven Design

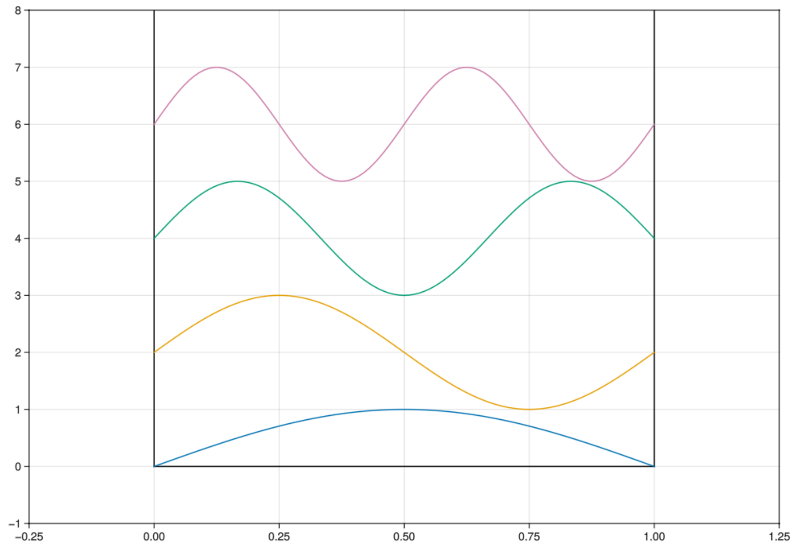
Sam Buercklin

G2I COMPUTING

https://github.com/SBuercklin/InterfaceDesignJuliacon2024

Using a Julia Project

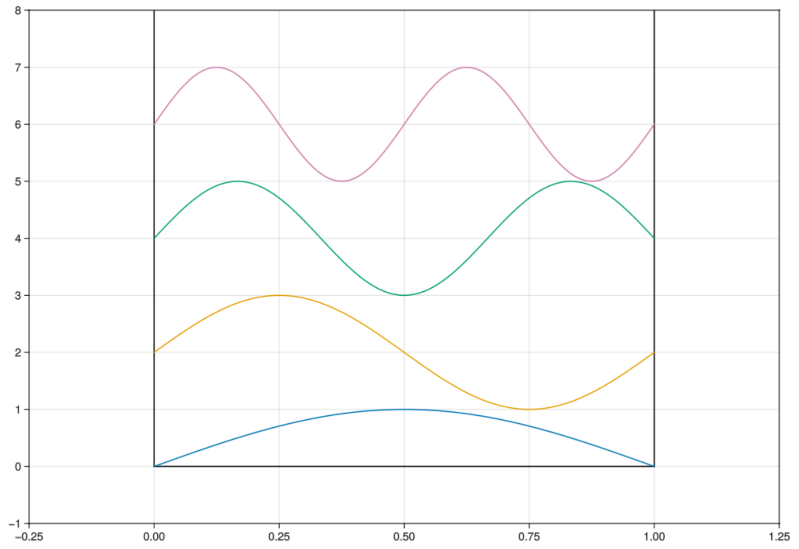
```julia
using SamsFunkySolver: InfiniteSquareWell, AnalyticSolver

isw = InfiniteSquareWell(; well_width = 5)
solver = AnalyticSolver()

solution = solve_problem(solver, isw)
```



✓

Using a Julia Project

```julia
using SamsFunkySolver: InfiniteSquareWell, AnalyticSolver

isw = InfiniteSquareWell(; well_width = 5)
solver = AnalyticSolver()

solution = solve_problem(solver, isw)
```



✓

```julia
fsw = FiniteSquareWell(; well_width = 5, V1 = 2, V3 = 8)
s_solver = ShootingMethod(; ψ0 = 1, dψ0 = 0.5)

solution = solve_problem(s_solver, fsw)
```

```
julia> solution = solve_problem(s_solver, fsw)
ERROR: type FiniteSquareWell has no field boundary_conditions
```

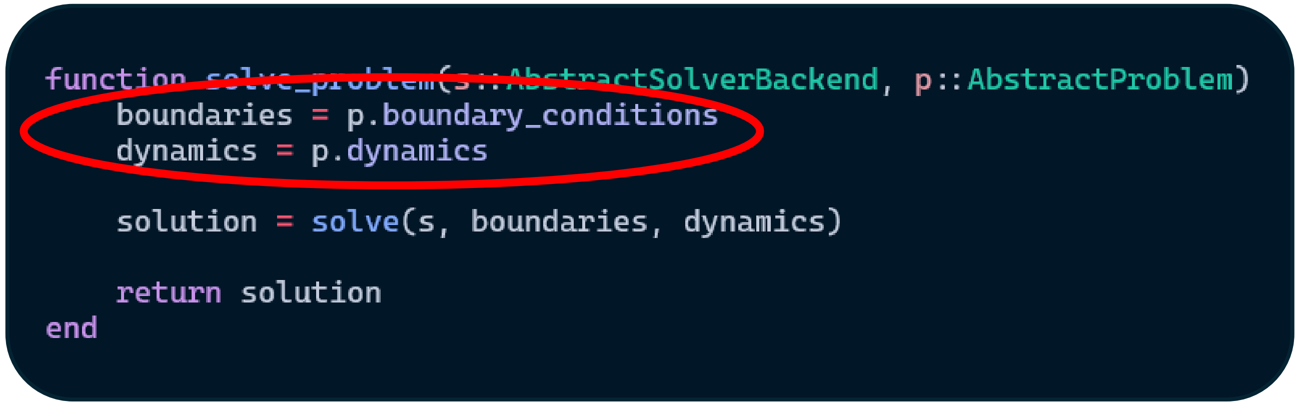
```
julia> solution = solve_problem(s_solver, fsw)
ERROR: MethodError: no method matching solve(::ShootingMethod{Float64},
::Vector{Float64}, ::Vector{Float64})
```

✗

What’s the problem?

```julia
function solve_problem(s::AbstractSolverBackend, p::AbstractProblem)
    boundaries = p.boundary_conditions
    dynamics = p.dynamics

    solution = solve(s, boundaries, dynamics)

    return solution
end
```

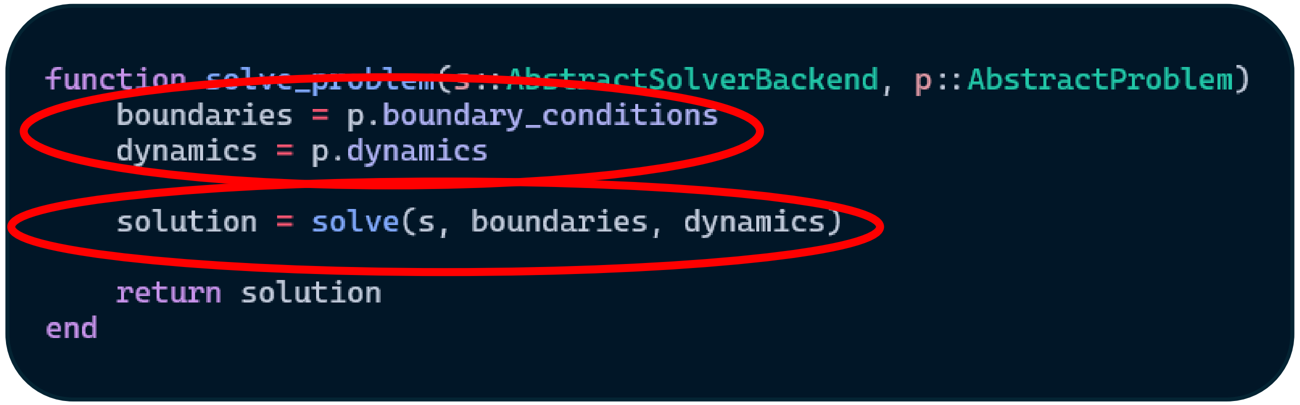
What’s the problem?



```julia
function solve_problem(s::AbstractSolverBackend, p::AbstractProblem)
    boundaries = p.boundary_conditions
    dynamics = p.dynamics

    solution = solve(s, boundaries, dynamics)

    return solution
end
```

- Hard coding required fields

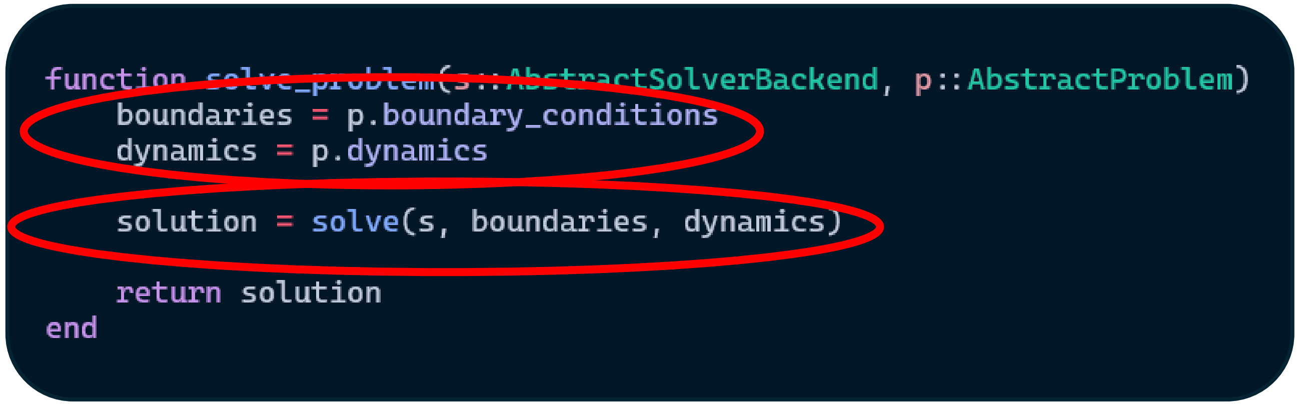
What’s the problem?



```
function solve_problem(s::AbstractSolverBackend, p::AbstractProblem)
    boundaries = p.boundary_conditions
    dynamics = p.dynamics

    solution = solve(s, boundaries, dynamics)

    return solution
end
```

- Hard coding required fields
- Solver needs a custom solve method

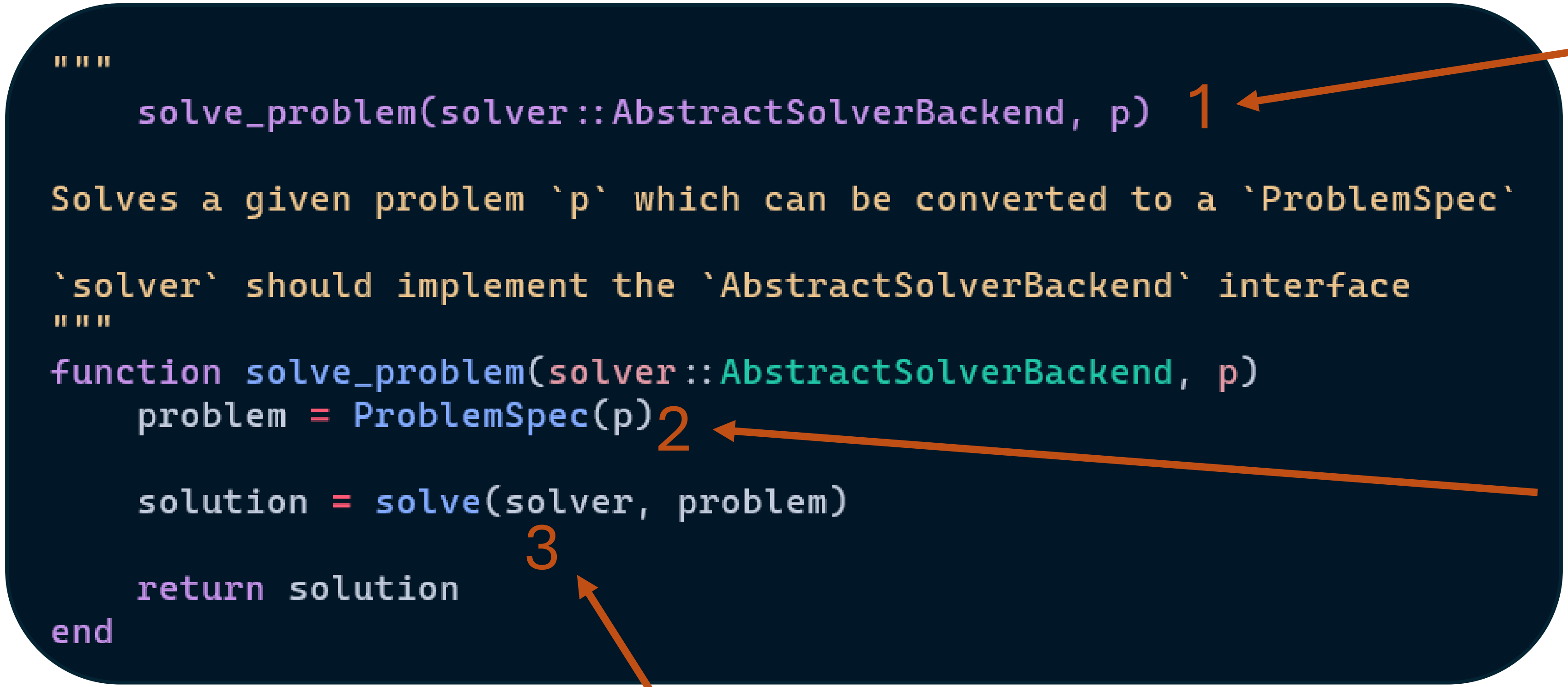
What’s the problem?



```
function solve_problem(s::AbstractSolverBackend, p::AbstractProblem)
    boundaries = p.boundary_conditions
    dynamics = p.dynamics

    solution = solve(s, boundaries, dynamics)

    return solution
end
```

- Hard coding required fields
- Solver needs a custom solve method
- How were you supposed to know?

What could be better?



```julia
"""
    solve_problem(solver::AbstractSolverBackend, p)

Solves a given problem `p` which can be converted to a `ProblemSpec`

`solver` should implement the `AbstractSolverBackend` interface
"""
function solve_problem(solver::AbstractSolverBackend, p)
    problem = ProblemSpec(p)

    solution = solve(solver, problem)

    return solution
end
```

Added a docstring

Standardized the problem representation

Formalized + generalized solve function

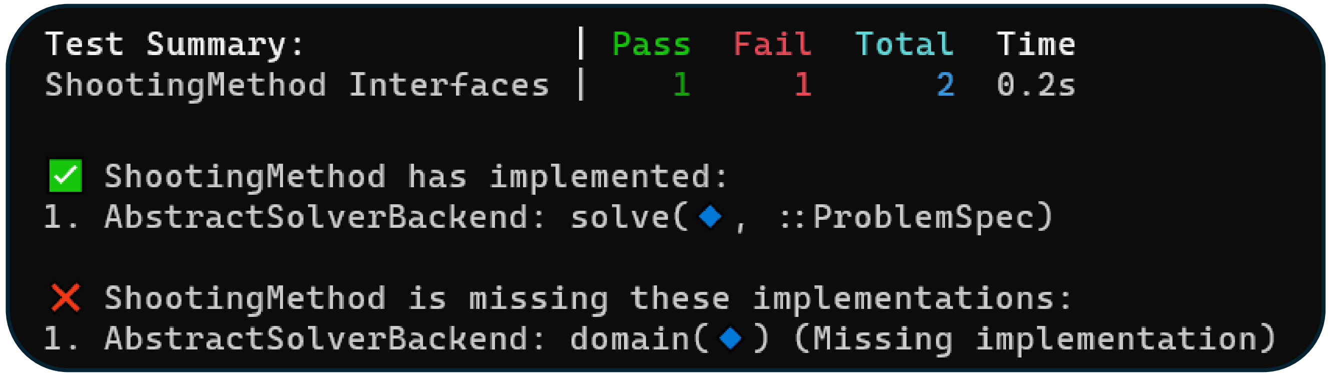
Is this good enough?

Is there anything my implemented missed?

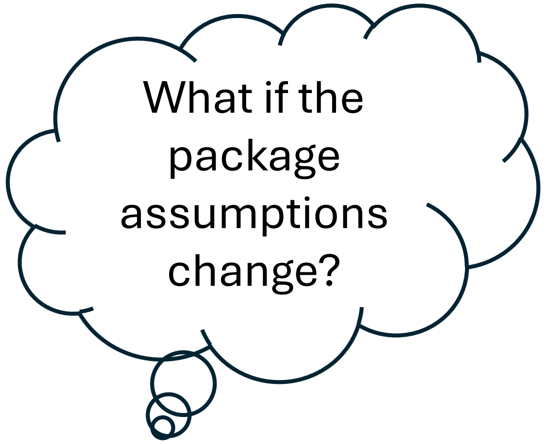
What if the package assumptions change?

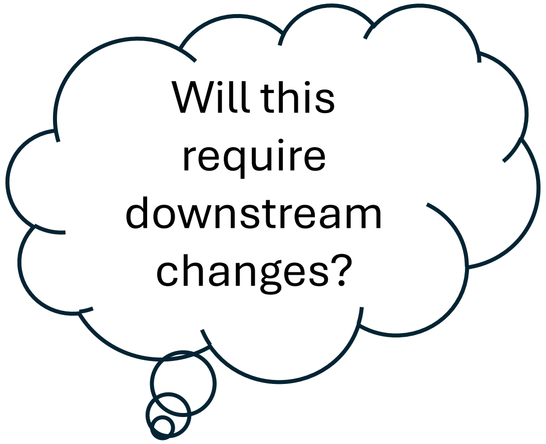
Will this require downstream changes?

Doing Better



```
Test Summary:           | Pass  Fail  Total  Time
ShootingMethod Interfaces |    1     1      2  0.2s

✅ ShootingMethod has implemented:
1. AbstractSolverBackend: solve(🔷, ::ProblemSpec)

❌ ShootingMethod is missing these implementations:
1. AbstractSolverBackend: domain(🔷) (Missing implementation)
```





But what is an Interface?

A set of methods which must be implemented…

But what is an Interface?

A set of methods which must be implemented…

with a particular signature…

But what is an Interface?

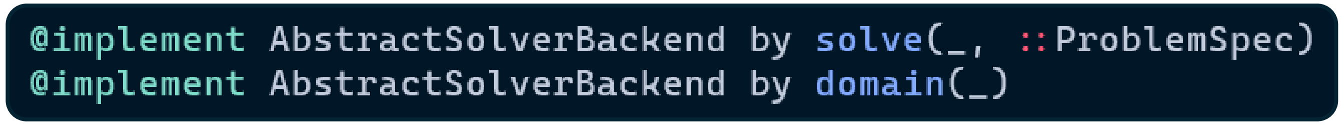
A set of methods which must be implemented…

with a particular signature…

to enable higher order behaviors for a type

But what is an Interface?

A set of methods which must be implemented...



```
@implement AbstractSolverBackend by solve(_, ::ProblemSpec)
@implement AbstractSolverBackend by domain(_)
```

with a particular signature...

to enable higher order behaviors for a type

But what is an Interface?

A set of methods which must be implemented...

```julia
@implement AbstractSolverBackend by solve(_, ::ProblemSpec)
@implement AbstractSolverBackend by domain(_)
```

with a particular signature...

```julia
function solve(s::ShootingMethod, p::ProblemSpec)
    ....
```

to enable higher order behaviors for a type

But what is an Interface?

A set of methods which must be implemented...

```julia
@implement AbstractSolverBackend by solve(_, ::ProblemSpec)
@implement AbstractSolverBackend by domain(_)
```

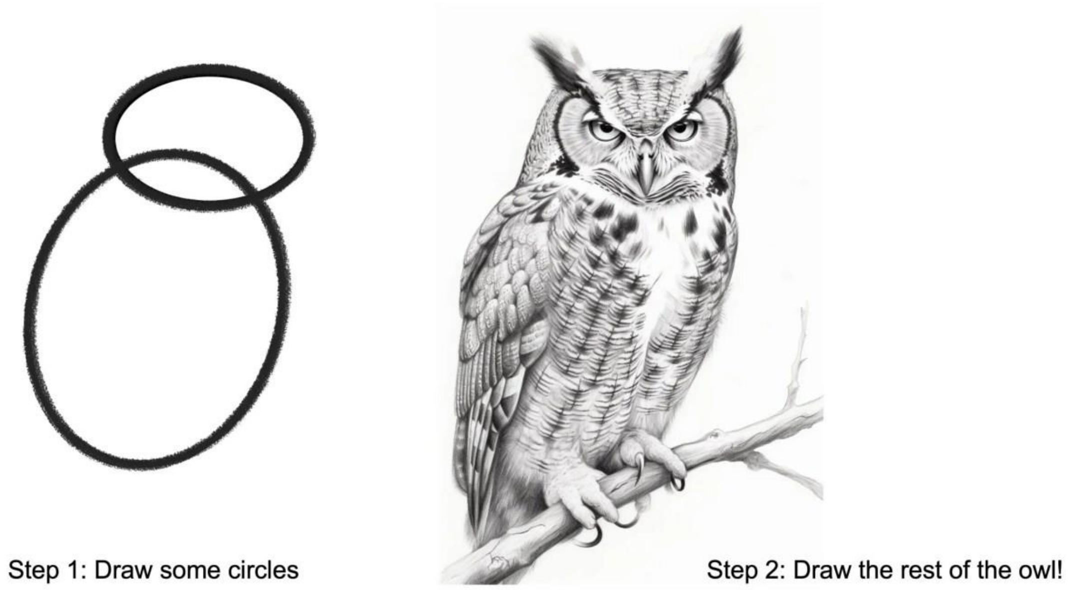
with a particular signature...

```julia
function solve(s::ShootingMethod, p::ProblemSpec)
    ....
```

to enable higher order behaviors for a type

```julia
function solve_problem(solver::AbstractSolverBackend, p)
    problem = ProblemSpec(p)

    solution = solve(solver, problem)

    return solution
end
```

Isn’t this just abstraction?



YES!

- ...with a focus on the actions, not just the subjects
- ...plus formalizing contracts of methods

Interface Definition via Packages

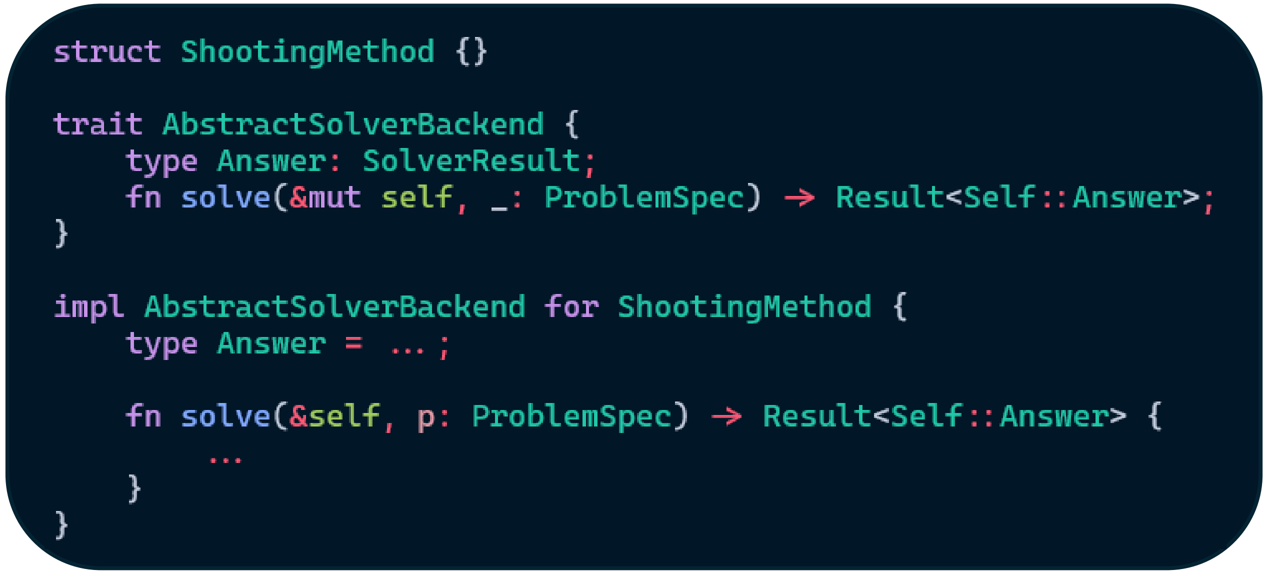
```julia
@implement AbstractSolverBackend by solve(_, ::ProblemSpec)
@implement AbstractSolverBackend by domain(_)
```

```julia
struct ShootingMethod end
@assign ShootingMethod with AbstractSolverBackend

function solve(s::ShootingMethod, p::ProblemSpec)
    ...
end
```

```
Test Summary:                | Pass  Fail  Total  Time
ShootingMethod Interfaces |    1     1      2  0.2s

✅ ShootingMethod has implemented:
1. AbstractSolverBackend: solve(◆, ::ProblemSpec)

❌ ShootingMethod is missing these implementations:
1. AbstractSolverBackend: domain(◆) (Missing implementation)
```

How do others solve this problem?

Traits + impl blocks, statically guaranteed, write methods explicitly in terms of interfaces



```rust
struct ShootingMethod {}

trait AbstractSolverBackend {
    type Answer: SolverResult;
    fn solve(&mut self, _: ProblemSpec) -> Result<Self::Answer>;
}

impl AbstractSolverBackend for ShootingMethod {
    type Answer = ... ;

    fn solve(&self, p: ProblemSpec) -> Result<Self::Answer> {
        ...
    }
}
```

How do others solve this problem?

Traits + impl blocks, statically guaranteed, write methods explicitly in terms of interfaces

```rust
struct ShootingMethod {}

trait AbstractSolverBackend {
    type Answer: SolverResult;
    fn solve(&mut self, _: ProblemSpec) -> Result<Self::Answer>;
}

impl AbstractSolverBackend for ShootingMethod {
    type Answer = ... ;

    fn solve(&self, p: ProblemSpec) -> Result<Self::Answer> {
        ...
    }
}
```

Abstract base classes, dynamic but class definitions fail without base class requirements

```python
class AbstractSolverBackend(ABC):
    @abstractmethod
    def solve(self, p):
        pass

class ShootingMethod(AbstractSolverBackend):
    # If we don't define this, we can't instantiate
    #   ShootingMethod
    def solve(self, p):
        ...
```

Tools for Interfaces

- RequiredInterfaces.jl + Supposition.jl
  - Interfaces + property based testing from Sukera
- Interfaces.jl
  - By Rafael Schouten, has a talk on this package later this week
- SimpleTraits.jl, WhereTraits.jl, BinaryTraits.jl, …
  - Many trait implementations solve similar interface problems

Interfaces vs Traits in Julia

Traits

- Often a dispatch tool, attempt at multiple inheritance
- Used with the Holy Traits pattern
- May assume an interface internally

Interfaces

- Just a collection of methods
- Not currently a dispatchable construct
- Interface inheritance could be:
  - Ad hoc
  - Abstract-type
  - Trait-based

Where do interfaces crop up?

Subject	Interface

DataToolkit.jl

Where do interfaces crop up?

Subject	Interface
CSV.jl writing to files	Tables.jl

Where do interfaces crop up?

Subject	Interface
CSV.jl writing to files	Tables.jl
S3Path, FTPPath, SystemPath	FilePathsBase.jl

DataToolkit.jl

Where do interfaces crop up?

Subject	Interface
CSV.jl writing to files	Tables.jl
S3Path, FTPPath, SystemPath	FilePathsBase.jl
DiffEq.jl solvers	SciMLBase.jl

DataToolkit.jl

Where do interfaces crop up?

Subject	Interface
CSV.jl writing to files	Tables.jl
S3Path, FTPPath, SystemPath	FilePathsBase.jl
DiffEq.jl solvers	SciMLBase.jl
Array, OffsetArray, SparseArray	Base.AbstractArray

DataToolkit.jl

Where do interfaces crop up?

Subject	Interface
CSV.jl writing to files	Tables.jl
S3Path, FTPPath, SystemPath	FilePathsBase.jl
DiffEq.jl solvers	SciMLBase.jl
Array, OffsetArray, SparseArray	Base.AbstractArray
Anything you mock	...is an implicit interface

DataToolkit.jl

Where do interfaces crop up?

Subject	Interface
CSV.jl writing to files	Tables.jl
S3Path, FTPPath, SystemPath	FilePathsBase.jl
DiffEq.jl solvers	SciMLBase.jl
Array, OffsetArray, SparseArray	Base.AbstractArray
Anything you mock	…is an implicit interface
For loops	Base.iterate interface

DataToolkit.jl

Advantages of Interfaces

Technical

- Write testers for interfaces
    - Implementation (methods exist)
    - Property based (correctness)
- Catch errors/fail faster than integration tests
- Mock your interfaces
    - Catch higher order errors

Advantages of Interfaces

Technical

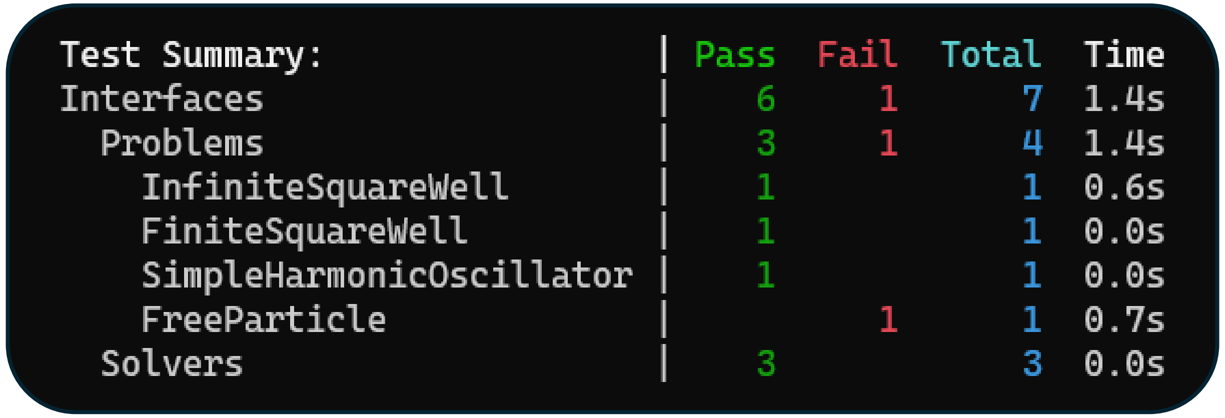
- Write testers for interfaces
  - Implementation (methods exist)
  - Property based (correctness)
- Catch errors/fail faster than integration tests
- Mock your interfaces
  - Catch higher order errors

Development

- Defines boundaries for new code
- Limits **where** logical errors can occur
- Standardizes “language” and assumptions around your code

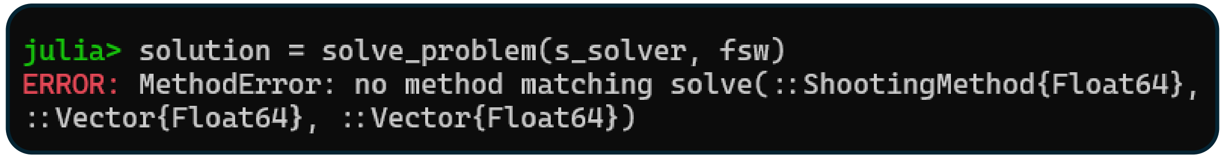
Building Confidently

- Stop `MethodErrors` missed by an incomplete test suite
  - Combinatorial explosion of multiple dispatch
- Helps isolate logical units to test, separate from integration tests
- SemVer is much easier to handle with interface testers



```
Test Summary:                | Pass  Fail  Total  Time
Interfaces                   |    6     1      7  1.4s
  Problems                   |    3     1      4  1.4s
    InfiniteSquareWell       |    1            1  0.6s
    FiniteSquareWell         |    1            1  0.0s
    SimpleHarmonicOscillator |    1            1  0.0s
    FreeParticle             |          1      1  0.7s
  Solvers                    |    3            3  0.0s
```

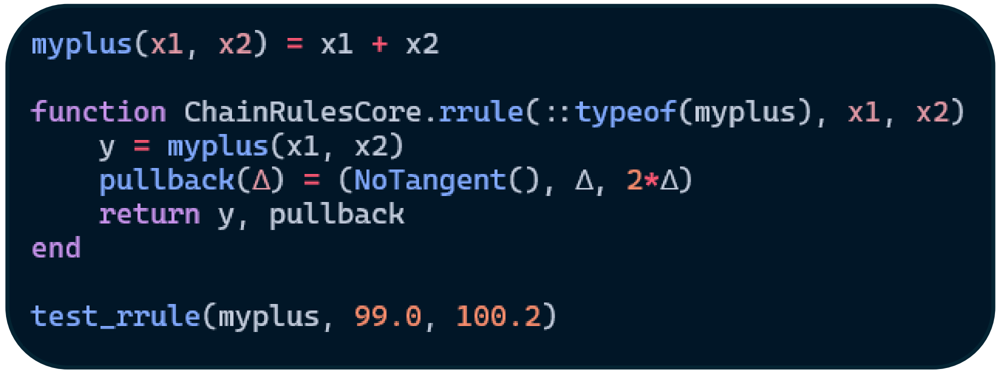
vs



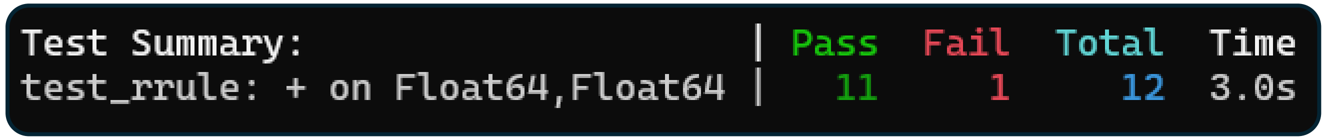
```
julia> solution = solve_problem(s_solver, fsw)
ERROR: MethodError: no method matching solve(::ShootingMethod{Float64},
::Vector{Float64}, ::Vector{Float64})
```

Testing Interfaces: ChainRules.jl

- Source-to-source autodiff uses libraries of differentiation rules
- This is an interface defined over functions!
- ChainRulesTestUtils.jl verifies interface **and** correctness



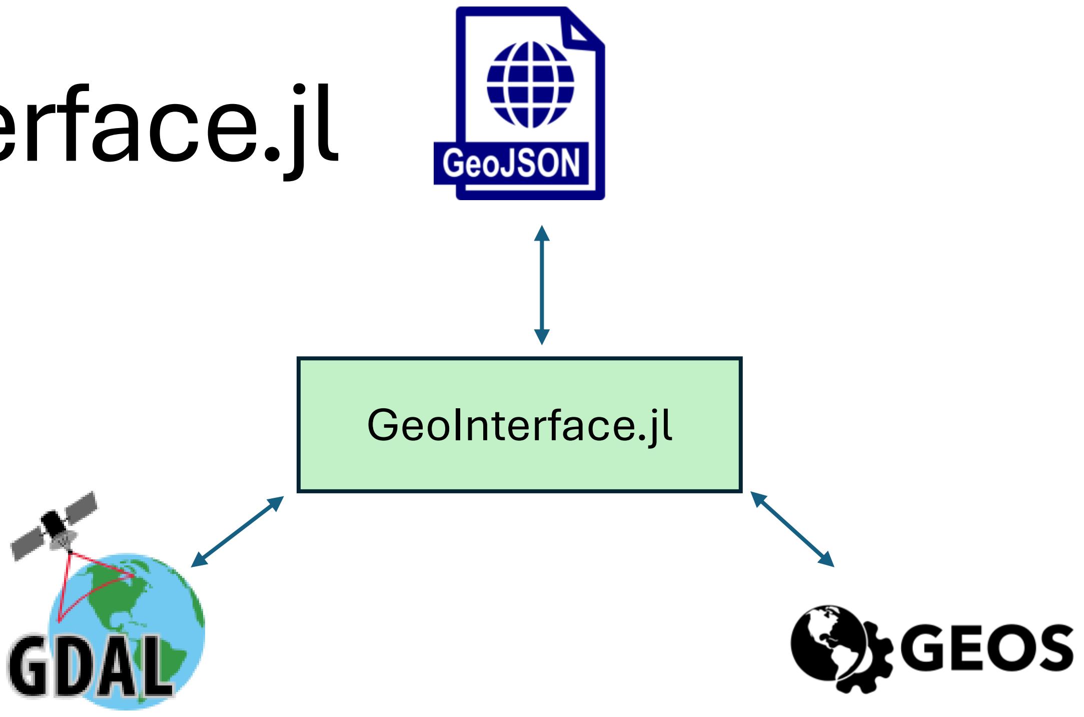
```julia
myplus(x1, x2) = x1 + x2

function ChainRulesCore.rrule(::typeof(myplus), x1, x2)
    y = myplus(x1, x2)
    pullback(Δ) = (NoTangent(), Δ, 2*Δ)
    return y, pullback
end

test_rrule(myplus, 99.0, 100.2)
```

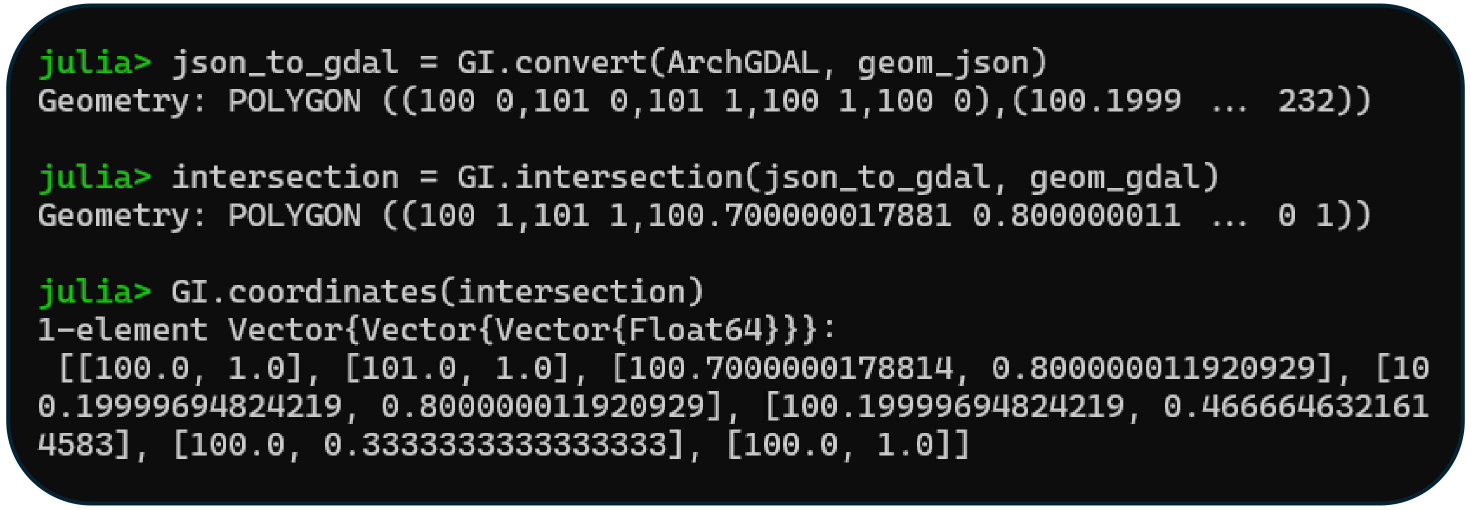


```
Test Summary:                    | Pass  Fail  Total  Time
test_rrule: + on Float64,Float64 |   11     1     12  3.0s
```

Uniting Packages: GeoInterface.jl

- Standard interface for geospatial data
- Trait system to represent geo features
- Facilitates inter-package conversions
- Package-agnostic algorithm implementations





```
julia> json_to_gdal = GI.convert(ArchGDAL, geom_json)
Geometry: POLYGON ((100 0,101 0,101 1,100 1,100 0),(100.1999  ...  232))

julia> intersection = GI.intersection(json_to_gdal, geom_gdal)
Geometry: POLYGON ((100 1,101 1,100.700000017881 0.800000011  ...  0 1))

julia> GI.coordinates(intersection)
1-element Vector{Vector{Vector{Float64}}}:
 [[100.0, 1.0], [101.0, 1.0], [100.7000000178814, 0.800000011920929], [10
0.19999694824219, 0.800000011920929], [100.19999694824219, 0.466664632161
4583], [100.0, 0.3333333333333333], [100.0, 1.0]]
```

Closing Thoughts

- Interface management is still awkward in Julia
  - No canonical “right” way to apply interfaces
- DuckDispatch.jl from Micah Rufsvold
  - Interfaces are ad-hoc but dispatchable
  - Can this be tightened up to improve usability?
- InterfaceSpecs.jl from Keno
  - Interface verification
  - Opens questions of “how do we handle incomplete interfaces”

Further Reading

[1] JuliaLang Issues 5 and 6975

- https://github.com/JuliaLang/julia/issues/6975

[2] Sukera’s writeup on RequiredInterfaces.jl

- https://github.com/Seelengrab/RequiredInterfaces.jl

[3] Jakob Nissen’s “What’s bad about Julia”

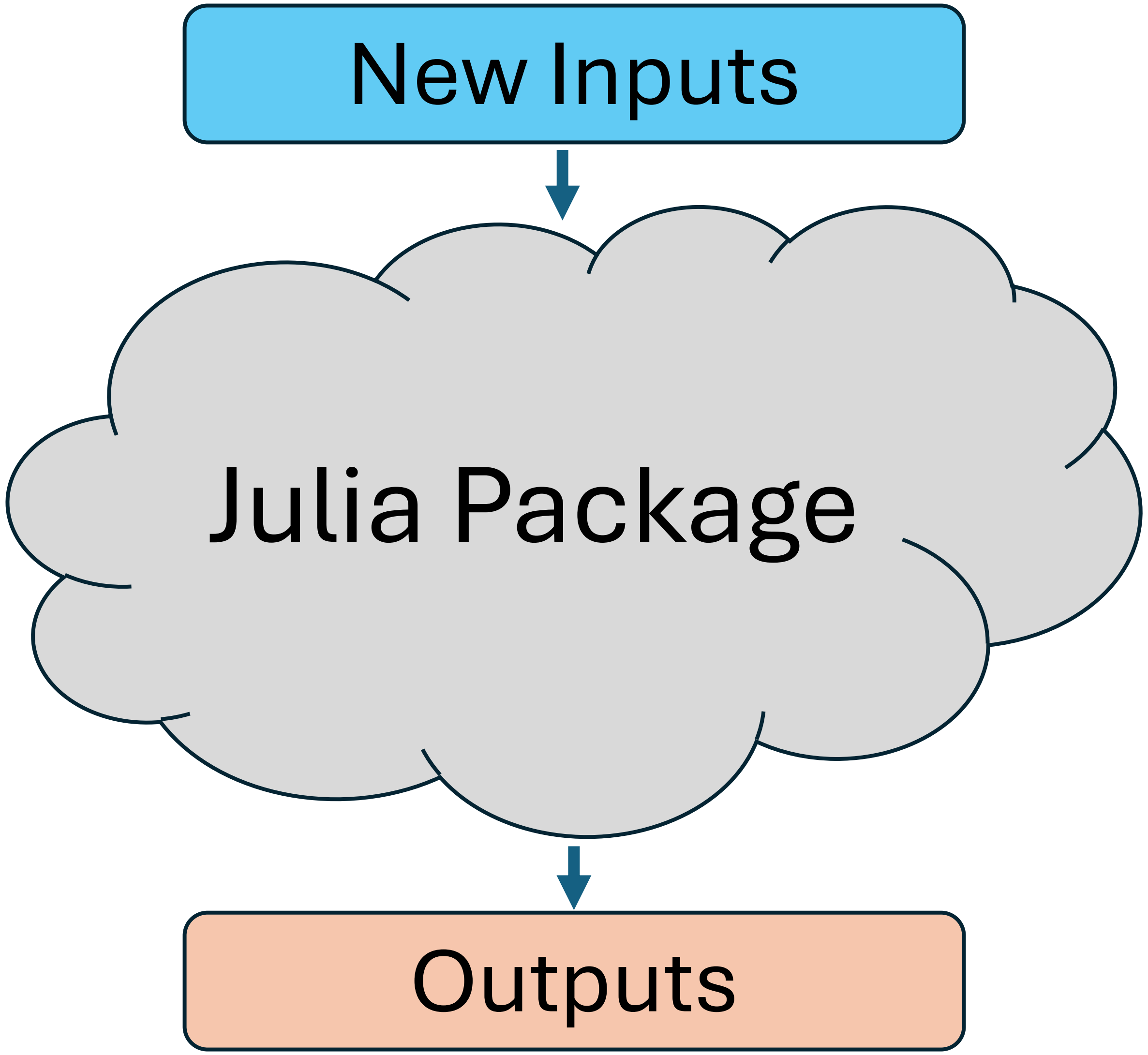
- https://viralinstruction.com/posts/badjulia/

[4] Keno Fischer’s InterfaceSpecs.jl

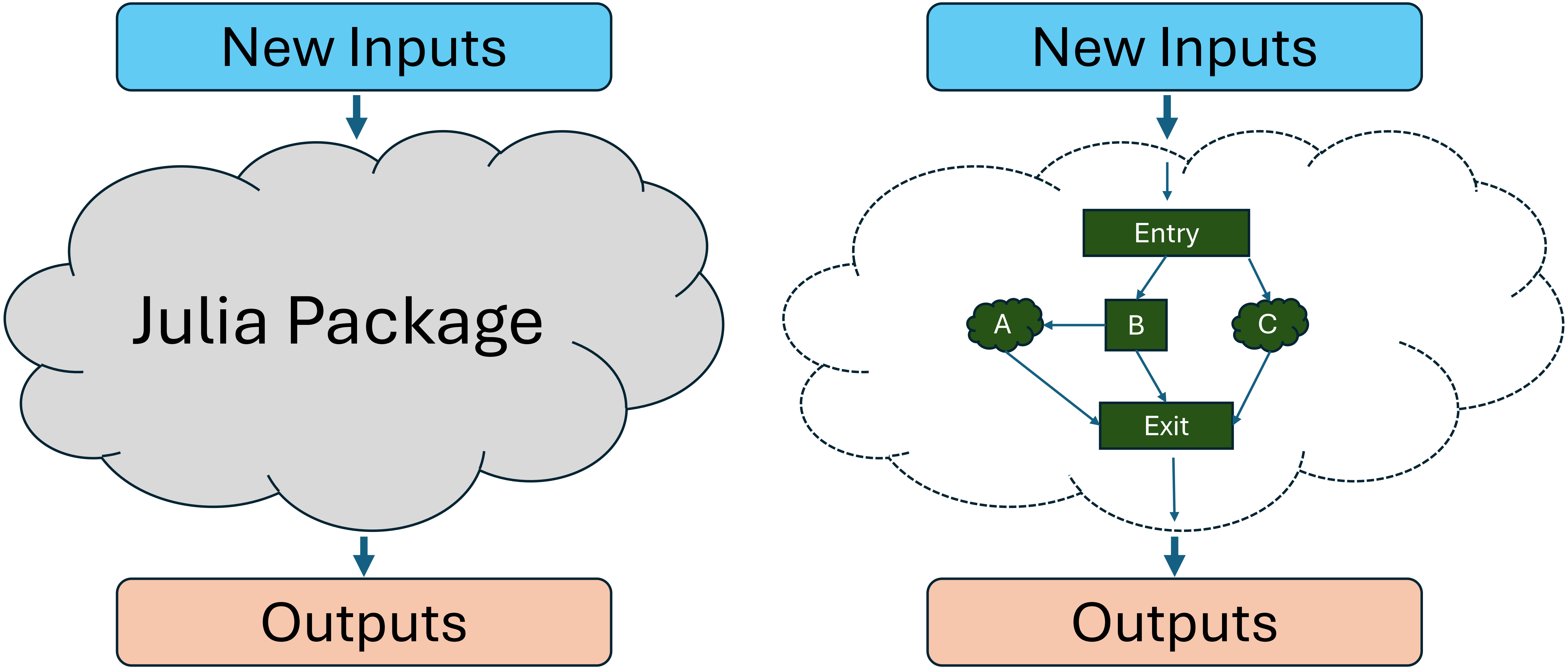
- https://github.com/Keno/InterfaceSpecs.jl

Thank You!

“Localizing” the Unknown



“Localizing” the Unknown


New Inputs
Julia Package
Outputs
New Inputs
Entry
A
B
C
Exit
Outputs