Writing Scientific Software

Lessons from the Software Engineering Community

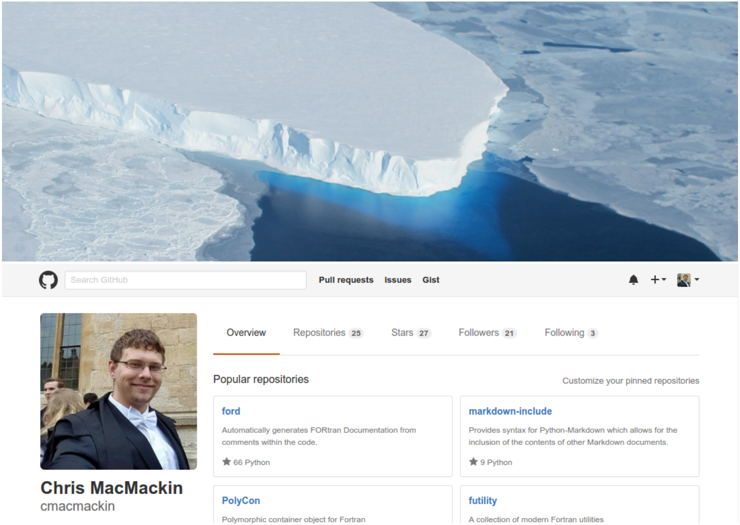
Christopher MacMackin

Atmospheric, Oceanic, and Planetary Physics
University of Oxford
Oxford, UK

christopher.macmackin@physics.ox.ac.uk
cmacmackin.github.io

4 November 2016

Glaciologist by day. . .



Scientists are not Programmers

Problem:

- Science relies on increasingly large codes
- Large codes become complex, difficult to maintain
- Improving or altering codes becomes more time consuming

These problems can be mitigated by good design. **But,**

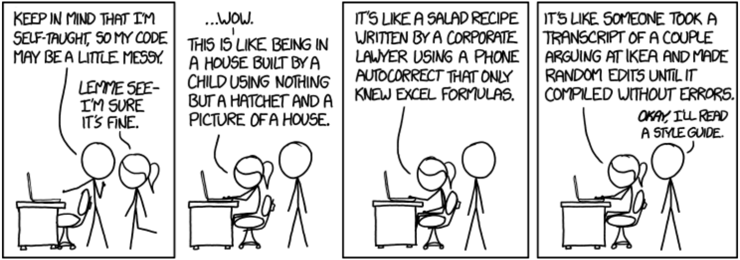
- Scientists not taught software design
- Code often written quickly, is unmaintainable
- Can end up costing more time in the long run

(Real) Example

A magneto-hydrodynamics code:

- 130 thousand lines long
- All in one file
- No procedure has name longer than 8 characters
- All important variables global

Only maintainable if you have worked with it for years.



https://xkcd.com/1513/

Outline

Object Oriented Programming

OOP is now the dominant programming paradigm.

- Define "classes", like new data-types
- Classes contain data, have associated procedures
- Variables of these types are "objects"

Object oriented languages feature:

Encapsulation controlling what data can be seen by different procedures

Inheritance "Is-a-type-of" relationship, giving subclasses access to parent's methods

Polymorphism Ability to run code without knowing exact type of an object

Why OOP?

Object oriented programming has following advantages:

- Useful way to organise data
- Can use one object as argument, rather than many
- Increases code-reuse via inheritance
- Conceptually convenient way to partition code
- Define an interface, independent of implementation

However (e.g. arithmetic), some things best accomplished through other paradigms.

Program to an Interface

Changing one part of the code, shouldn't force you to change other parts. Can do this is by defining an API.

Definition

API: Application Programming Interface

1. Create procedures stubs provide desired functionality
2. Prevent direct access to any non-constant variables
3. Write procedure bodies
4. If making changes, don't alter names, arguments, return values of publicly accessible procedures

But I Really Need to Change It!

Would you like it if a new version of PETSc changed so it no longer worked with your code?

Strategies:

- Add optional arguments
- Add new procedures, overload old ones
- Design it well in the first place

(With apologies to Randall Munroe.) https://xkcd.com/292/

Describing Your Software

When designing code, can be useful to have way to easily represent its structure, relationships.

Unified Modelling Language (UML): standardised collection of graphical elements to do that.

Examples of free software to produce UML:

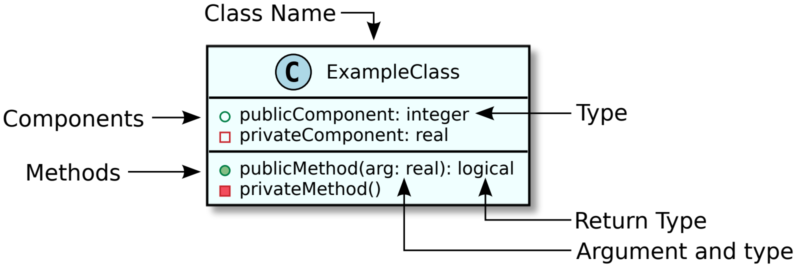
Dia (GUI)

PlantUML (language)

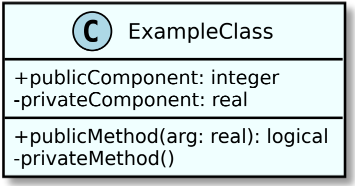
Will use UML in this presentation. (Pay attention!)

UML Class Diagrams

Can describe contents of a class/derived type:

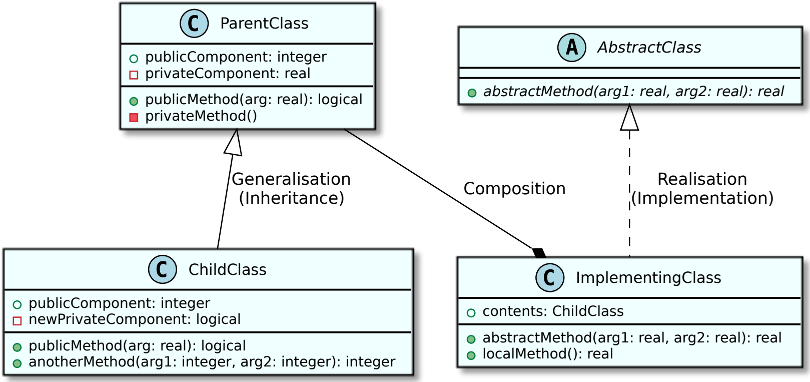


Note that `public`/`private` also specified with “+”/“-”:



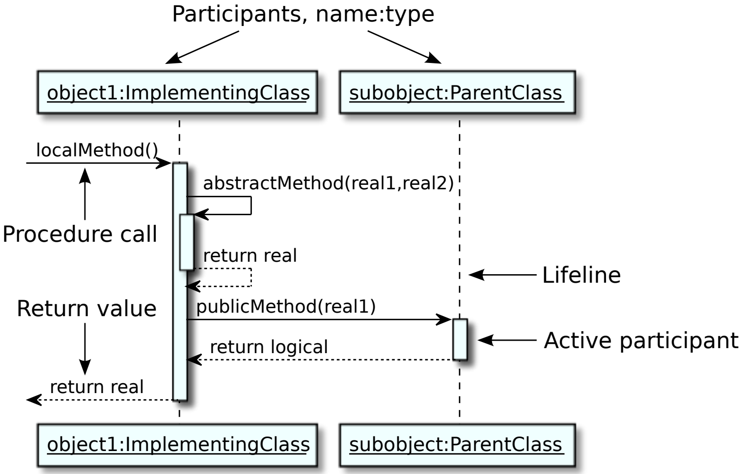
UML Class Relationships

Can also show how classes inherit from or contain others.



UML Sequence Diagrams

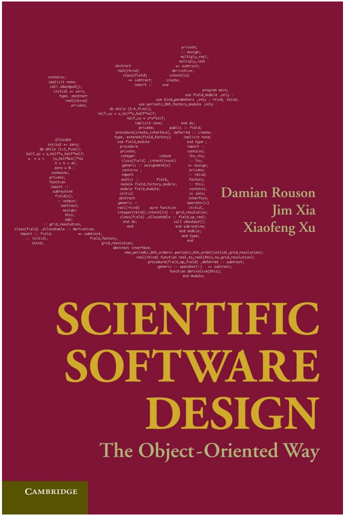
UML diagrams can also describe call sequence.



Design Patterns

Writing software, common problems frequently reoccur. Canonical solutions have been developed called *design patterns*.

Many general-purpose patterns can be relevant to scientific software. Will focus here on domain-specific patterns introduced by Rouson, Xia, and Xu (2011).



The Abstract Calculus Pattern

The Problem

Physics and mathematics use high level representations of concepts which can not be simply expressed in code.

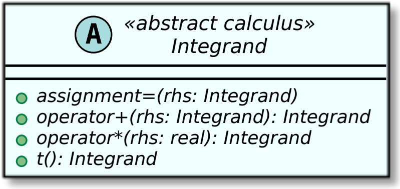
Consider integration using the forward Euler method,

$$T(x, t + \Delta t) = T(x, t) + \Delta t \frac{\partial T(x, t)}{\partial t}.$$

But code will look quite different and much messier than this. Also somewhat difficult to reuse. (Even worse for coupled equations.)

Abstract Calculus to the Rescue!

We can take advantage of operator overloading to provide a nice syntax:




```
subroutine integrate(T, dt)
class(integrand), intent(inout) :: T
real, intent(in) :: dt
! Forward Euler integration:
! T^{n+1} = T^n + T_t Δt
T = T + T%t()*dt
end subroutine integrate
```

The Why and Wherefore

Using an abstract `integrand` type facilitates reuse of integration code, which is agnostic towards:

- Type of problem
- Storage pattern of data
- Precision of data
- Size of computational domain

Elegance of syntax useful for higher-order methods. Can also apply pattern to, e.g., vector calculus:

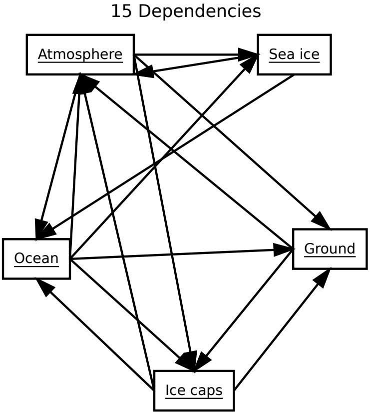
$$\mathbf{g} = -\nabla\phi \quad \rightarrow \quad \texttt{g} = - \texttt{.grad. phi}$$

The Puppeteer Pattern

Often it is impractical to package entire program together. E.g., "multiphysics" problems have many components.

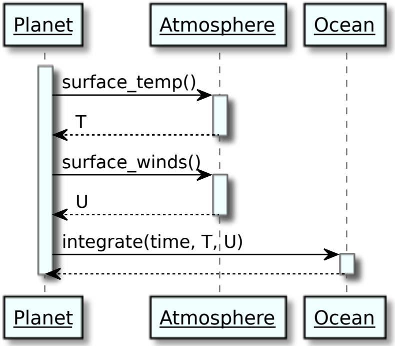
N components $\Rightarrow N(N+1)$ interdependencies.

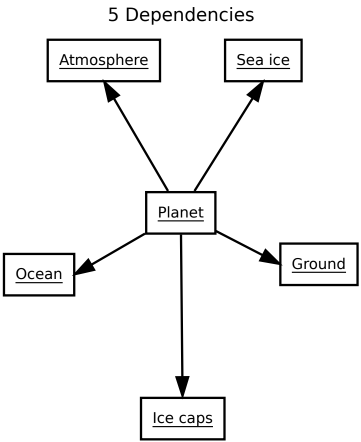
Complexity/fragility grows quickly. Also becomes harder to maintain data-hiding.



Puppeteers to the Rescue!

Can use a single class to manage interactions between all others. Only the one class needs to know the interfaces of the others.





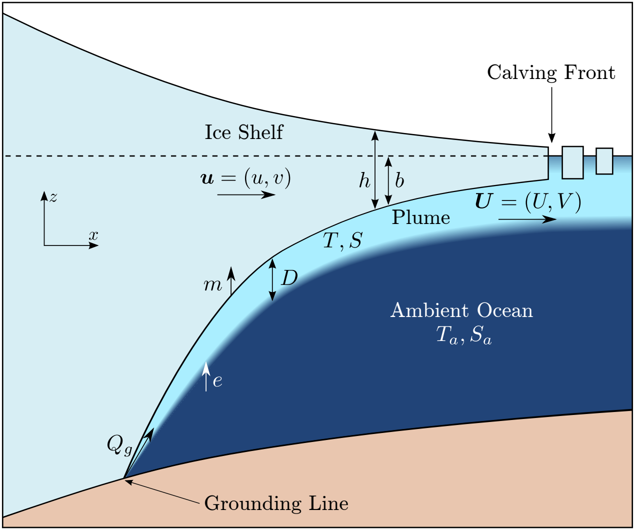
The Why and Wherefore

This provides many advantages:

- Changing component's API affects only puppeteer
- Avoids circular dependencies
- Each component designed independently from others
- Can easily turn components off, decoupling system
- Place to assemble global data (e.g. a Jacobian)

A puppeteer may be a concrete implementation of abstract calculus.

Case Study: Ice Shelf



Equations

Ice Shelf

$$h_t + \nabla \cdot (h\vec{u}) = -\lambda m,$$
$$(4\eta h u_x)_x - \chi\left(h^2\right)_x = 0.$$

Plume

$$\nabla \cdot \left(D\vec{U}\right) = e + m,$$

$$\nabla \cdot \left(D\vec{U}U\right) = D(\rho_a - \rho)\left(b_x - \delta D_x\right) + \nu \nabla \cdot (D \nabla U) - \mu|\vec{U}|U + \frac{\delta D^2}{2}\rho_x,$$

$$\nabla \cdot \left(D\vec{U}S\right) = eS_a + \nu \nabla \cdot (D\nabla S) + mS_m - \gamma_S(S - S_m),$$

$$\nabla \cdot \left(D\vec{U}T\right) = eT_a + \nu \nabla \cdot (D\nabla T) + mT_m - \gamma_T(T - T_m).$$

Breaking Down the Problem

Choice of numerics:

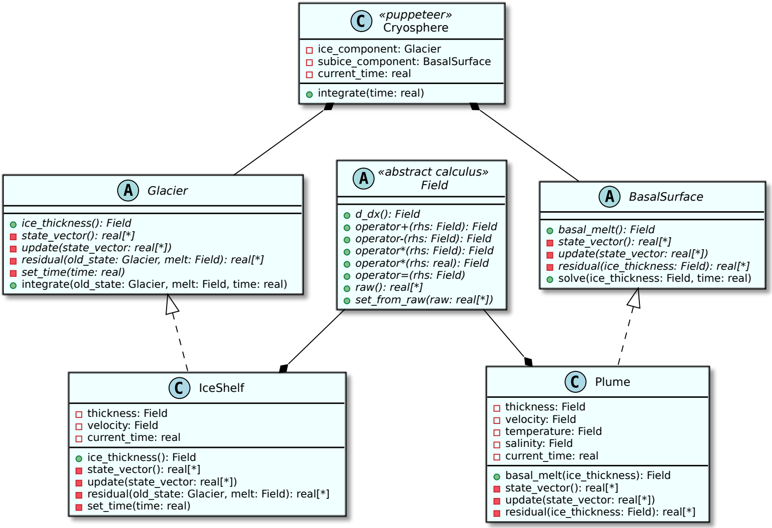
- Chebyshev pseudo-spectral spatial discretisation
- Backwards-Euler time integration

Let $S(t)$ be state of ice shelf at time t, with $\dot{S}(S, m)$. The melt rate is $m(S, t)$. Backwards-Euler requres

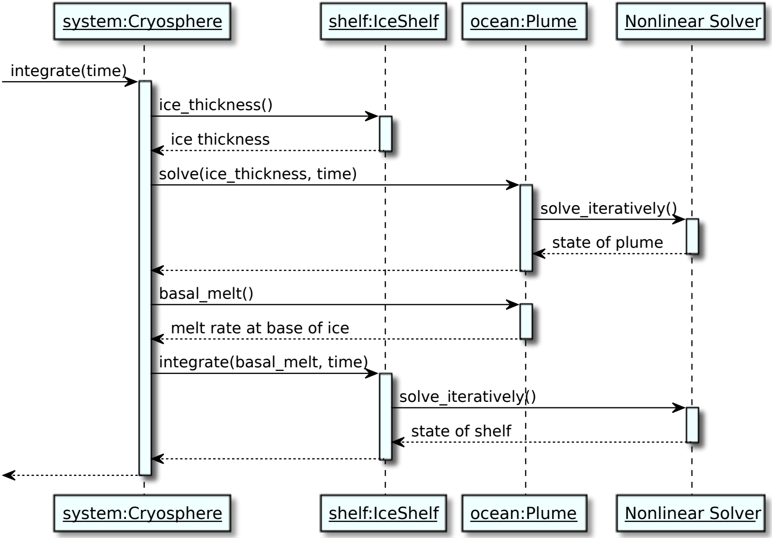
$$S(t + \Delta t) = S(t) + \dot{S}[S(t + \Delta t), m]\Delta t.$$

This requires iterative nonlinear solver. To reduce cost of this, evaluate $m[S(t), t]$—semi-implicit method.

Collating our Classes



Solution Sequence



Parameterisation Choices

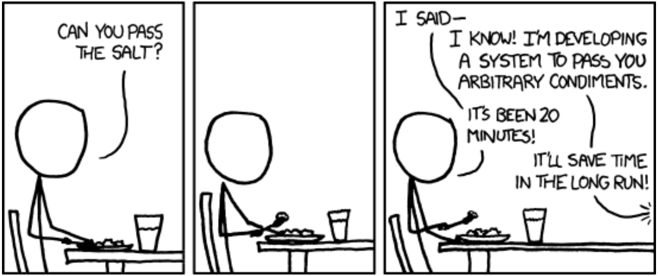
Will be useful to compare results for different parameterisations of, e.g.:

- viscosity
- melting
- entrainment

To make it easy to switch these, use the “strategy” pattern.

Is it worth it?

"I find that when someone's taking time to do something right in the present, they're a perfectionist with no ability to prioritize, whereas when someone took time to do something right in the past, they're a master artisan of great foresight."



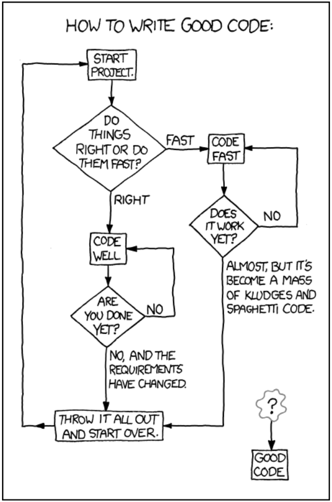
https://xkcd.com/974/

Summary

Science is increasingly reliant on large pieces of software.

- Think before coding!
- Plan your software (e.g. with UML)
- Program to an interface
- Leverage language features for natural syntax
- Minimise inter-dependency

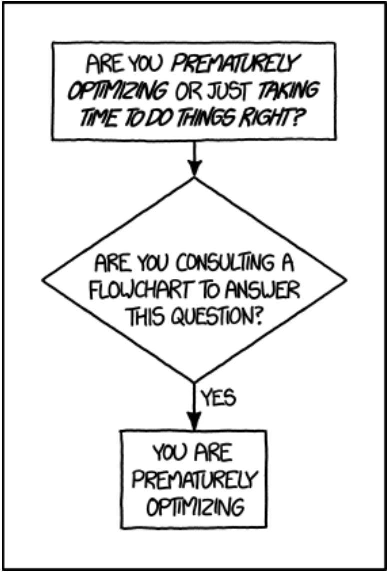
Time invested up front *can* save time later.



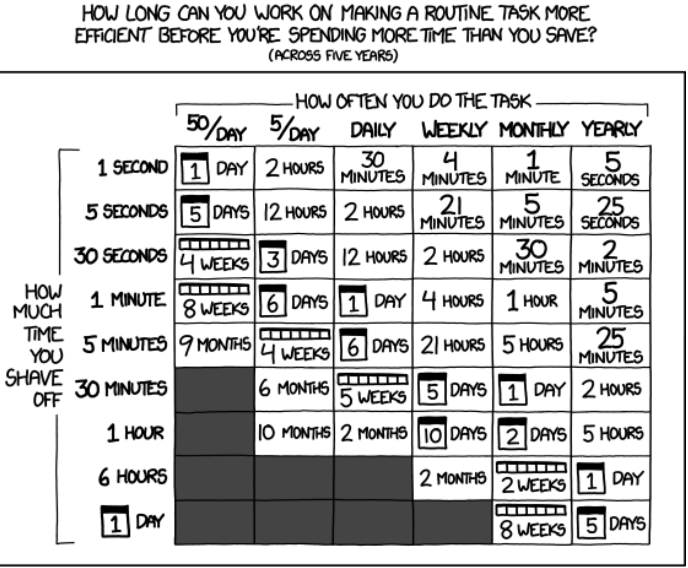
https://xkcd.com/844/

Thank You

Questions?



https://xkcd.com/1691/



https://xkcd.com/1205/