NHR4CES NHR for Computational Engineering Science

TECHNISCHE UNIVERSITÄT DARMSTADT

Design Patterns and Programming Paradigms in Open Source CFD Software

Mohammed Elwardi Fadeli[1,2], Holger Marschall[1] and Christian Hasse[2]

April, 2024

[1] Mathematical Modeling and Analysis (MMA)
[2] Simulation of Reactive Thermo Fluid Systems (STFS)
Energy Conversion Group, NHR4CES - TU Darmstadt

30mins Journey through C++ Programming Paradigms

- Programming paradigms for CFD/C++ developers
- Principles for better software design
- Most common design pattern in Open Source C++ CFD software
- Applications: Design patterns for common mechanisms
- Closing remarks

Programming pardigms i

It's all about how to manage **the program's state**

Procedural Programming

State mutated externally to code.

Mutate a variable

```cpp
// Symptoms: pass-by-reference, void return type of free functions
void inc(int &x) { x++; }
int x = 0;
inc(x);
```

"Give a man a state, and he will have a bug one day. Teach him to mutate state everywhere, and he will have bugs for a lifetime" — Mutated Joshoa Bloch's saying

Object-Oriented Programming

State and code logic are coupled.

Object classes

```cpp
// Symptoms: class data members, member methods
class Counter {
    int x = 0;
public:
    void inc() { x++;}
};
Counter i;
i.inc();
```

Check Tomislav's Object-oriented crash course for OpenFOAM devs

Programming pardigms iii

Functional Programming

State, huh? There shall be no state,
only immutable variables and zero-side-effects functions

No-capture pass-by-value lambdas

```cpp
// Symptoms: pure functions, immutability everywhere
// math-correctness; i.e. cannot write x=x+1
auto inc = [](int x) { return x + 1; };
const int x = 0;
const int xPlusOne = inc(x);
```

No scientific computing in Haskell, or F# - non-existent user base.

Programming pardigms iv

Declarative Programming

State management is abstracted away. Facts/Rules/Queries as seen in logic programming and Database Systems (SQL ... etc)

C++ ranges but it's c++23

```cpp
// Symptoms: Queries, rules and facts
// Transform is a view rule, lazy evaluated! but do we care?
// as long as it does what it's supposed to
auto inc = [](int x) { return x + 1; };
vector<int> v = {1, 2, 3};
auto result = v | ranges::views::transform(inc);
```

Popular in Database Systems and web stuff; not so much in scientific computing.

Parallel Programming

State can be shared, and must be carefully managed.

Fear of race conditions and deadlocks gives rise to locks, mutexes, and atomic operations.

Execution policies since c++17

```cpp
// Symptoms: multithreading, MPI, GPU offloading
auto inc = [](int x) { return x + 1; };
vector<int> v = {1, 2, 3, 4};
std::transform(std::execution::par, v.begin(), v.end(), v.begin(), inc);
```

Check my Workshop: Parallel programming in OpenFOAM

Generic Programming

State is abstracted away, and code logic is type-agnostic.

Concepts and templates

```
// Symptoms: meta-programming, templating and compile-time programming
template<typename T>
concept Incrementable = std::is_move_constructible<T>::value
    && requires(T x) { { x + 1 } -> std::convertible_to<T>; };
template<Incrementable T> T inc(T x) { return x + 1; }
int x = 0;
if constexpr (Incrementable<decltype(x)>) {
    x = inc(x);
} else {
    // Something else
}
```

Principles for better software design i

- Program for interfaces
  - Inheritence (Composition?) and Polymorphism in OOD are your best friends.
  - Relating to the Facade and Strategy patterns.
  - Example of API design: Three Levels of API calls in OpenFOAM-SmartSim (Service, Developer, and Generic interfaces)
  - Benefits: Modularity, Flexibility, Dependency Injection, and Testability.
  - Enhanced with generic programming (specifically concepts).

Principles for better software design ii

- Seperation for concerns
  - OpenFOAM is built mostly as a set of dynamic libraries linked to a binary.
  - You can write code for your concerns (new BC? new model?) and load it at runtime.
  - Relating to the Factory, Strategy and Registry patterns.
  - Example of implementing Load-balanced adaptive mesh refinement by hooking to the dynamic mesh library and extending its classes: blastAMR
  - Benefits: Modularity, Parallel Development and easier Maintability.

Principles for better software design iii

- Composition Over Inheritance
  - OpenFOAM deviates from this principle significantly.
  - The overhead of dynamic dispatch and indirection introduced by composition make it so the benefits from composition are not worth it! Inheritence plays better with polymorphism
  - Relating to the Strategy, Decorator, and dependency injection patterns.
  - Example 1: SU2's fluid model class has viscosity, diffusivity, and thermal conductivity as data members instead of inheriting from them.
  - Example 2: schemesLookup class from OpenFOAM is composed of different scheme kinds (interpolation, div, grad, ..., etc) instead of inheriting from their base classes.
  - Benefits: Reduced coupling, no fragile base classes. Also, easier on the unit tests.

Principles for better software design iv

- **Principle of Least Astonishment**
  - Aim for interfaces to be intuitive and surprise-free.
  - Examples:

Transport equations

```cpp
// How hard it is to figure out the terms?
fvm::ddt(T) + fvm::div(phi, T) == fvm::laplacian(DT, T);
```

Ways to find a Max

```cpp
int a, b; Foam::volScalarField c;
Foam::max(a, b);
c.max(); max(c); Foam::gMax(c);
Foam::volScalarField d = c.max(1.0);
```

  - Benefits: Less of a learning curve and less missunderstanding bugs.

Creational

Builder

Construct complex objects step by step.

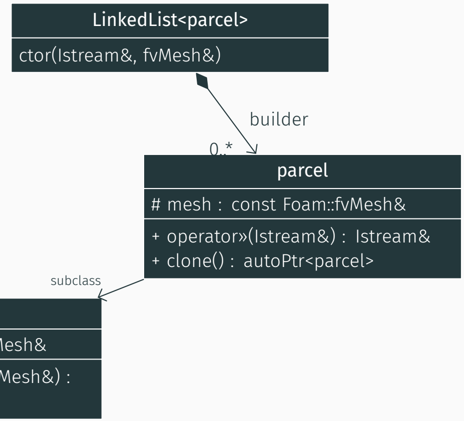


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

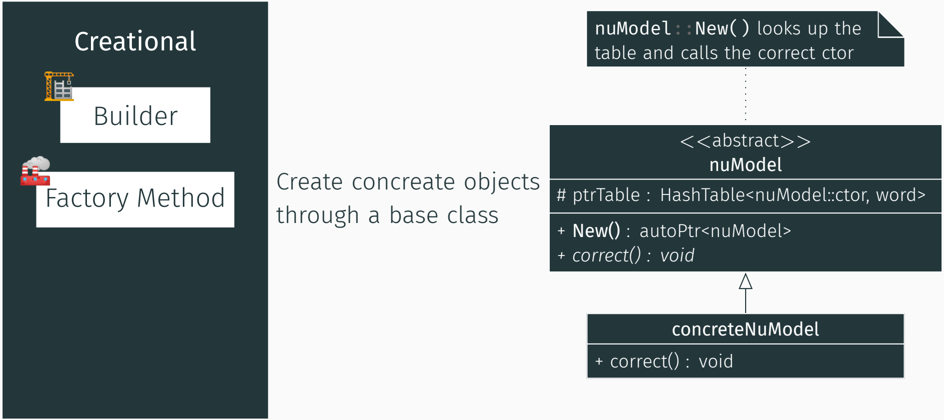


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

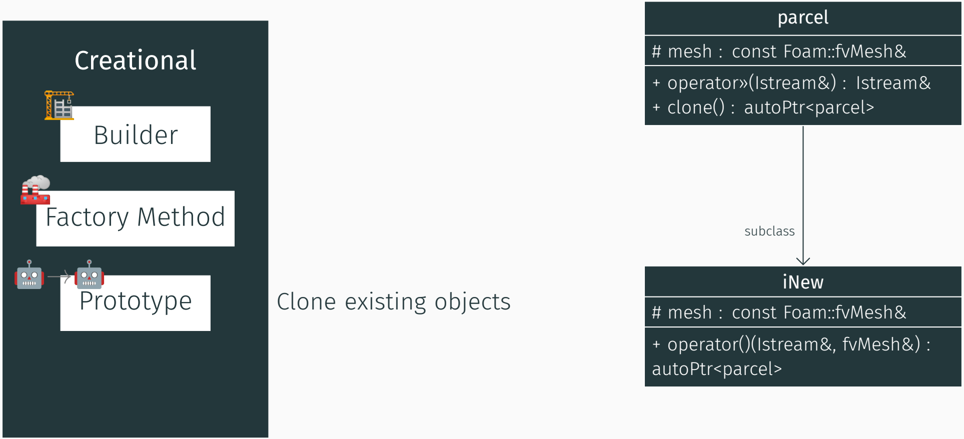


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

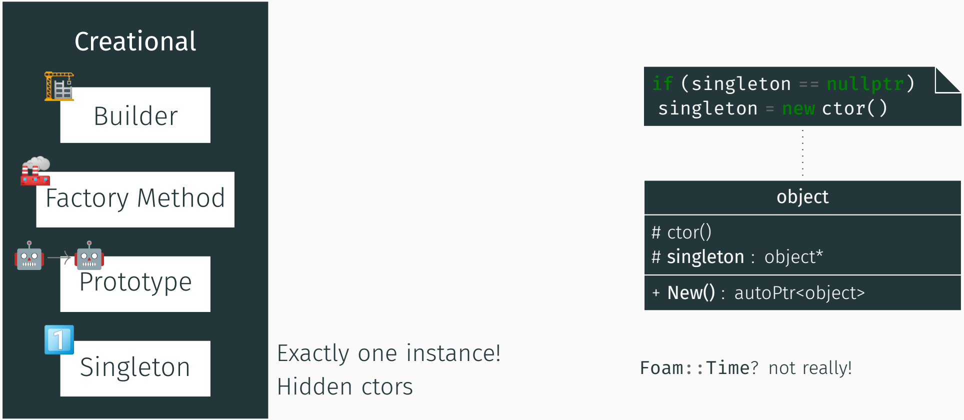


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

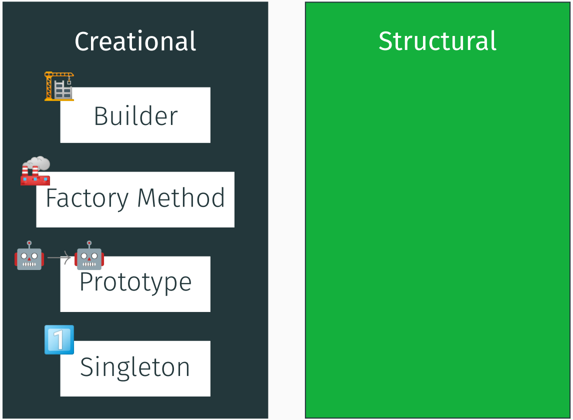
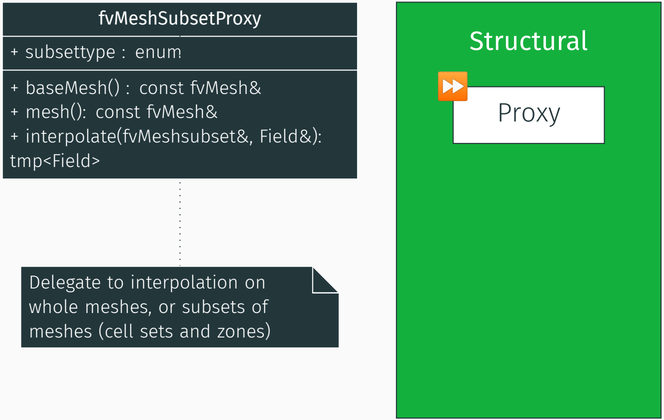


Figure 1: Design pattern examples from OpenFOAM

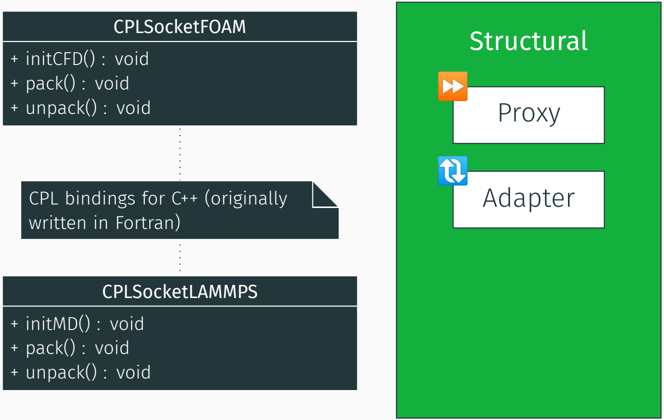
Design patterns: A quick run-down



Provide a substitue or placeholder

Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down



Collaboration of incompatible interfaces

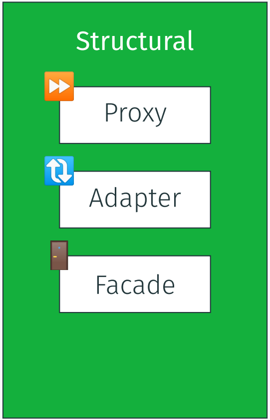
Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

smartRedisClient

mesh : const fvMesh&
redisDB : tmp<smartRedisAdapter>

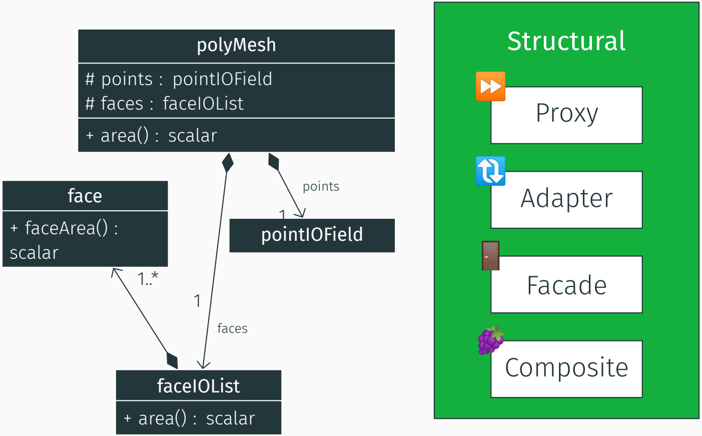
updateNamingConvention() : void
+ sendGeometricFields (const wordList&) : void
+ packFields<T> (DataSet&, const wordList&) : void
+ sendList<Type> (List<Type>&, const word&) : void



Simplified interface to libraries

Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down



Compose objects into a tree structure

Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

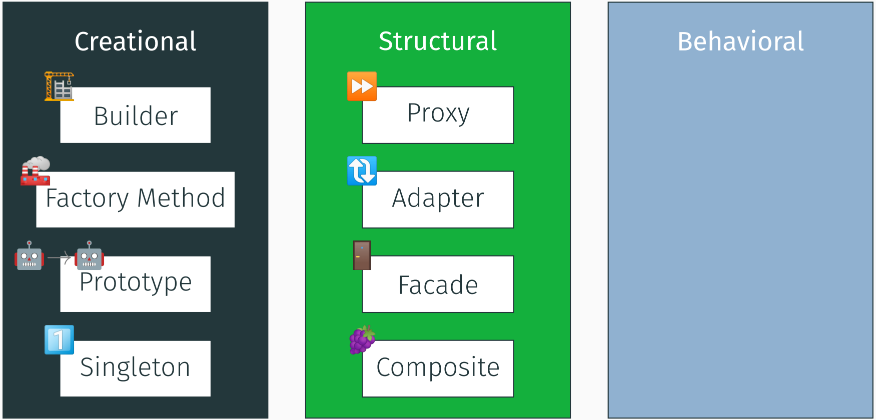


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

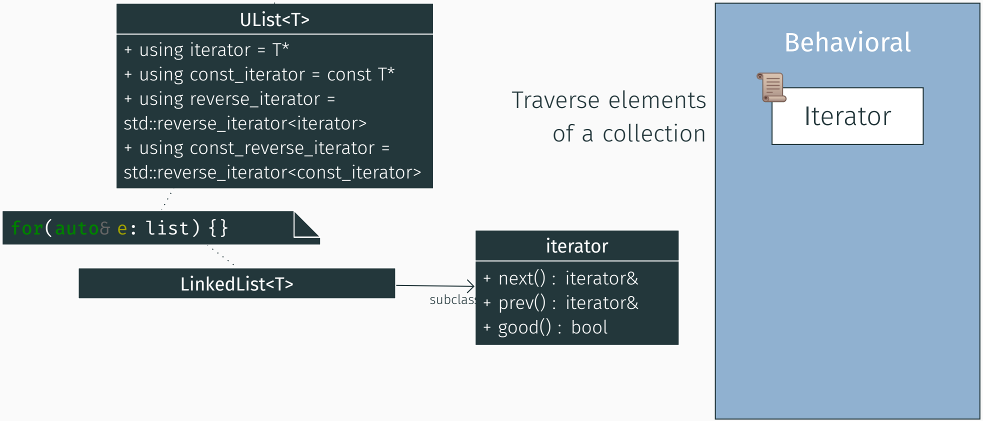


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

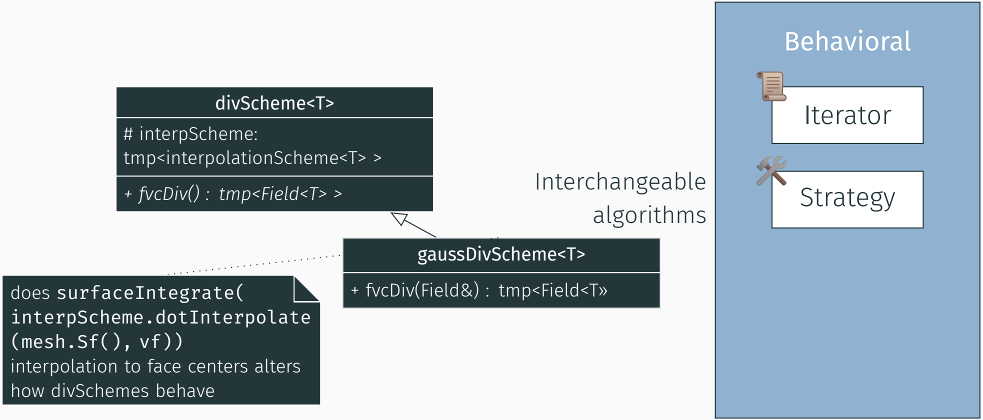


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

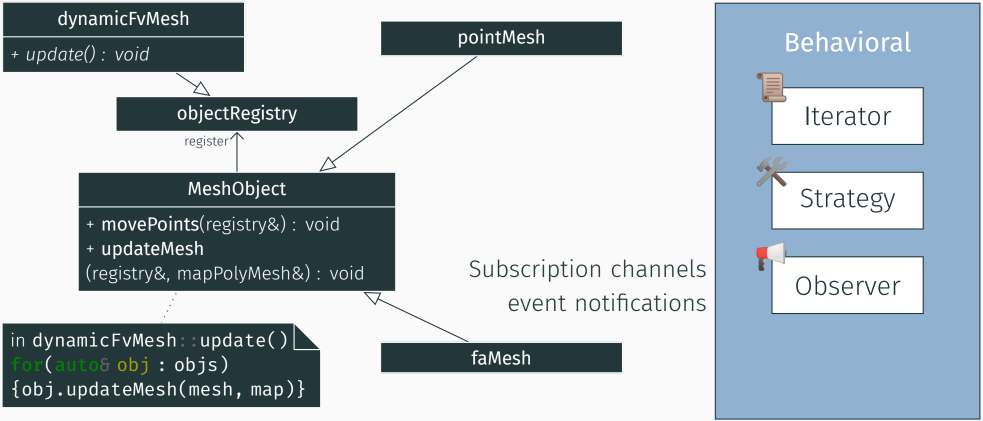


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

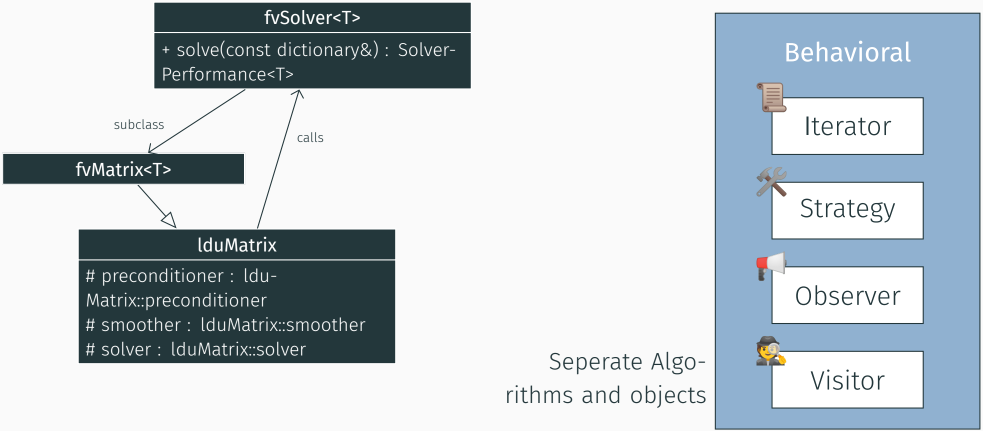


Figure 1: Design pattern examples from OpenFOAM

Design patterns: A quick run-down

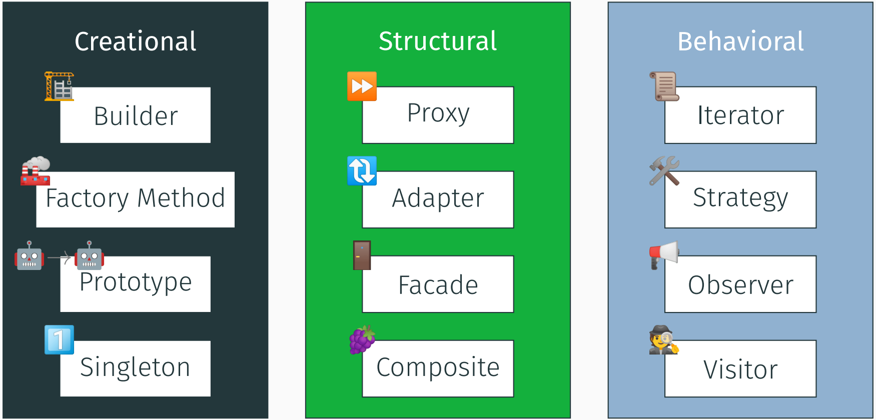


Figure 1: Design pattern examples from OpenFOAM

Design patterns for common mechanisms - RTS

Scenario 1

You want users to select a model for a particular concern **at runtime**.
These models are implemented as children of a base (template) class.
You also want them to add new models without altering your code.

Scenario 2

You have a legacy code base, that you want to unit-test.
But writing a binary for each test is cumbersome.
So the unit tests should be selectable **at runtime** too.

Design patterns for common mechanisms - RTS

Scenario 1

You want users to select a model for a particular concern **at runtime**.
These models are implemented as children of a base (template) class.
You also want them to add new models without altering your code.

Scenario 2

You have a legacy code base, that you want to unit-test.
But writing a binary for each test is cumbersome.
So the unit tests should be selectable **at runtime** too.

Solution

A runtime type selection table (RTS) that maps names to objects.
Think of it as a manual `vtable`.
Relying mainly on **Global (or static) variables** and a bit of metaprogramming.

Here is a little experiment with OpenFOAM's RTS:

Exploring memory layouts with GDB

```
git clone https://github.com/FoamScience/foamUT
cd foamUT
source /usr/lib/openfoam/openfoam2112/etc/bashrc # Or any version
# Compile with debug symbols
sed -i 's/14/14 -g -ggdb -O0/g' tests/exampleTests/Make/options
./Alltest --no-parallel
gdb ./tests/exampleTests/testDriver
    (gdb) b main
    (gdb) r --- -case cases/cavity
    (gdb) ptype 'Foam::Function1<double>'
    # In particular, we are interested in:
    (gdb) ptype 'Foam::Function1<double>::dictionaryConstructorTableType'
    (gdb) ptype 'Foam::Function1<double>::dictionaryConstructorPtr'
    # See what's available through:
    (gdb) ptype 'Foam::Function1Types::CSV<double>::'
    # Here is the metaprogramming part:
    (gdb) ptype 'Foam::Function1<double>::adddictionaryConstructorToTable<*>::'
```

Design patterns for common mechanisms - RTS i

Discoveries:

- **Base class** keeps a **static pointer** for a `HashTable` of **function pointers** to ctors of 'derived' classes.
  1. **static** so it gets initialized before main, after dynamic library loading.
  2. **pointer** because order of initialization of global variables is **not guaranteed**.
  3. Technically; ctors don't have addresses, so cannot have function pointers to them. Instead we store pointers to little helper construction functions.
- **The memory** pointed to by the table pointer is managed manually
  1. `ConstructorTablePtr_construct(bool)` called by the base class ctor.
  2. Flexible enough to have multiple ways to construct objects (from dictionary, from Istream, etc)

Design patterns for common mechanisms - RTS ii

- Instatiation of `Base::addConstructorToTable<Derived>` will cause Derived's ctor to be added to the table.
    1. Which is an effect of how template subclasses work.
    2. and with some macros, the boilerplate code is burried.
- The factory pattern comes into play:

  ```
  autoPtr<Base> obj = Base::New(ctor_args);
  ```

  which will forward the args to the selected ctor depending on a type name from user configs

  No need to `#include "Derived.H"` and if Derived implements some pure virtual functions from Base, they will be called instead.

Design patterns for common mechanisms - objectRegistry

Scenario 1

You have a nice RTS for viscosity models; implementing all kinds of fluids.
Base ctor now takes 10 args so you cover evey desirable field to calculate ν
Next guy comes by and wants to use custom fields... and now your interface needs to change!

Scenario 2

Your ModelA depends on a ModelB from another library, but ModelB also depends on ModelA.
Flat-out bad design, but it's too late to refactor.

Design patterns for common mechanisms - objectRegistry

Scenario 1

You have a nice RTS for viscosity models; implementing all kinds of fluids.
Base ctor now takes 10 args so you cover evey desirable field to calculate ν
Next guy comes by and wants to use custom fields... and now your interface needs to change!

Scenario 2

Your ModelA depends on a ModelB from another library, but ModelB also depends on ModelA.
Flat-out bad design, but it's too late to refactor.

Solution

Promise to write class ctors with **nothing more than (a config + a mesh)** as arguments.
Have an Object Registry with 1 rule: Only takes global object in.

Design patterns for common mechanisms - objectRegistry i

Say you want a function object (a UDF) to send fields to a Database for some ML/AI processing:

```
// In system/controlDict
functions
{
    SendPUandPhi
    {
        type fieldsToSmartRedis; // gets loaded because of the RTS
        libs ("libsmartredisFunctionObjects.so");
        fields (p U phi); // <- dont care about their types! just send these fields out plz
        patches (internal);
    }
}
```

How do you think the code will look like?

Design patterns for common mechanisms - objectRegistry ii

Right, should be simple enough

Trivial implementation; works for like 2 secs

```cpp
// Nice start, maybe a pure function (hopefully no side effects)
void Foam::smartRedisClient::sendGeometricFields
(
    const volScalarField& p,
    const volVectorField& U,
    const surfaceScalarField& phi
) const; // blah blah
```

But, what happens when someone wants to send temperature?

Design patterns for common mechanisms - objectRegistry iii

Exploit: Fields are registered to the mesh they were created on:

API improvements

```cpp
void Foam::smartRedisClient::sendGeometricFields
(
    const fvMesh& mesh // less dependencies -> stable API
    // You selfish ppl, this means less compilation time for YOU!
) const
{
    const auto& p = mesh.lookupObject<volScalarField>("p");
    const auto& U = mesh.lookupObject<volVectorField>("U");
    const auto& phi = mesh.lookupObject<surfaceScalarField>("phi");
    // the same blah blah from before
}
```

Much better, ensured a stable interface, but still needs changes to account for new fields

Design patterns for common mechanisms - objectRegistry iv

Generic programming to the rescue!

Externally configurable so more easily testable

```
void Foam::smartRedisClient::sendGeometricFields
(
    const dictionary dict, // now configurable
    const fvMesh& mesh
) const
{
    wordList fields = dict.lookup("fields"); // get fields list from config
    checkFieldsExist<SupportedTypes>(fields, mesh);
    sendFields<SupportedTypes>(fields, mesh);
}
```

Delegation to templated methods is good, gives control over supported types through type lists. C++ is, in the end, a typed language.

Design patterns for common mechanisms - objectRegistry v

```cpp
template<class... Types> bool Foam::smartRedisClient::checkFieldsExist (
    const wordList& fieldNames, const objectRegistry& obr
) const {
    // static_assert at least one template argument
    forAll(fieldNames, fi) {
        // Fold expressions to check if a matching name of any of the types is found
        if (!(obr.foundObject<Types>(fieldNames[fi]) || ...)) {
            // Be transparent with the poor user seeing this for the first time
            word supportedTypes = word("(") + nl;
            ((supportedTypes += tab + Types::typeName + nl), ...);
            supportedTypes += ")";
            FatalErrorInFunction
                << "Field " << fieldNames[fi] << " not found in objectRegistry"
                << " as any of the supported types:" << nl
                << supportedTypes
                << exit(FatalError); }
    }
    return true;
}
```

Closing remarks

- C++ can be tedious at times
  - But it's improving, I hope
- Watch your state
  - Functional programming makes a big deal out of a loop because of ”no state”
  - ”for element in collection” looks easier for science computing
  - Sometimes OOD is not the best way though
- Program for interfaces, really
  - Providing a stable and intuitive API is key
  - If everything takes raw pointers and returns raw pointers to stuff, you will have fat bugs
  - OOD is widely popular in CFD software but we would like to see other paradigms adopted

Licensing

Questions?

Sources and further reading i

[1] Mohammed Elwardi Fadeli. *Unit testing framework for OpenFOAM.* 2024. URL: https://github.com/FoamScience/foamUT.

[2] Gerson Kurz. *Introduction to Parallel Programming with MPI and OpenMP.* Source of the great 'pit stops' analogy. Dec. 2016. URL: http://p-nand-q.com/programming/cplusplus/registering_global_objects.html.

[3] Tomislav Maric et al. *Combining Machine Learning with Computational Fluid Dynamics using OpenFOAM and SmartSim.* 2024. DOI: 10.48550/ARXIV.2402.16196. URL: https://arxiv.org/abs/2402.16196.

[4] OpenCFD. *OpenFOAM source code.* Apr. 2024. URL: https://develop.openfoam.com/Development/openfoam.

[5] Refactoring.Guru. *Design Patterns in C++.* 2024. URL: https://refactoring.guru/design-patterns/cpp.