Software Design Patterns in Research Software with examples from OpenFOAM
This webinar is about Design Patterns in Research Software, and I’ll be using examples from my own work with OpenFOAM, a GPL open-source, but trademarked software:

This content is not approved or endorsed by OpenCFD Limited, producer and distributor of the OpenFOAM software via www.openfoam.com, and owner of the OPENFOAM® and OpenCFD® trademarks.
Fluid phases that do not mix are separated by sharp interfaces (3D surfaces).

Fluid phases exchange mass, momentum, and energy at fluid interfaces.

Fluid interfaces deform, break up, and merge.

Direct Numerical Simulations aim to resolve all scales, while ensuring convergence, volume conservation and (parallel) computational efficiency.

Multiphase flows are everywhere

- Fuel-cells, Lab-On-a-Chip, ship/offshore hydrodynamics, coating processes, 3D printing, ...
Level Set / Front Tracking \([1, 2, 3, 4]\) on unstructured meshes \([5, 6, 7, 8]\) combines

- Phase-indication (marker field): which fluid phase occupies point \(x\) at time \(t\)?
- Signed-distance calculation (redistancing): curvature approximation.
- Front (3D surface mesh) reconstruction: topology changes.
- Point-search operations: vertex-cell (front-mesh) mapping.
- Velocity interpolation.
Research software development for LENT is done by Tobias Tolle, Jun Liu, and myself.
The quality of the method is determined by validation & verification studies.

- There was another IDEAS/ECP webinar (2021-04-07) that covers a workflow for increasing research software quality in this context.

The sub-algorithms build a hierarchy, whose elements should be interchangeable at runtime without changing existing code.
Cannot talk about the hierarchy without understanding its elements first.

- Complex things (e.g. Front-Mesh communication) are **abstracted** in C++ as User-Defined Types (UDT, **classes**).

- A class **encapsulates** its data (attributes, **data members**).

- A class implements **behavior**: member **functions** that change the data members.

- Access specifiers
  - +: accessible from outside (public)
  - -: inaccessible from outside (private)

- Private data (-) = narrow focus.
Cannot talk about the hierarchy without understanding the interactions between its elements (UML)

- Classes **inherit (derive)** from other classes: A inherits from B.
- Classes **contain (composit) objects** of other classes: A contains C.

**Dynamic polymorphism**: addressing an object of the derived class via a pointer to the base class can be used to set the type of the object at runtime.

```cpp
configData input{"path/to/file"};
smart_pointer<A> Aptr = A::New(input);
Aptr->behave(); // B chosen in input!
```
Software Design Patterns in Research Software

What are software design patterns useful for?

Support programming on a higher-level of abstraction

- A high-level of abstraction is crucial - thinking in terms of complex objects; not getting lost in low-level details.

- Design patterns modularize abstractions’ functionality and their interaction:
  - What do parcels require from the mesh in order to evolve?
  - Which objects are written with runTime.write()? 

```cpp
// Perform mesh changes
mesh.update();

// Update moving reference frame
MRF.update();

// Make the fluxes relative to the mesh-motion
fvc::makeRelative(phi, rho, U);

// Evolve the particle cloud
parcels.evolve();

// Evolve the surface film
surfaceFilm.evolve();

// Write data
runTime.write();
```
Software Design Patterns [9]: code structures that **combine inheritance and composition** and have emerged repeatedly as **best-practice solutions for specific design problems**.

Software Design Patterns (examples from OpenFOAM)

- **Template Method**: boundary conditions, viscosity models, discretization schemes, ...
- **Strategy**: transport models, solvers and pre-conditioners, ...
- **Observer**: dynamic mesh handling, IO synchronization, ...
- **OpenFOAM’s Creational Pattern**: Runtime-Type Selection (RTS), **used everywhere**.

Not covered in this webinar

- **Facade**: Level Set / Front Tracking (Additional Slides)
- **Curiously Recurring Template Pattern (CRTP)**: Discrete Parcel Method (Additional Slides)
Virtual member function: implements different behavior in a derived class.
OpenFOAM’s boundary conditions

```plaintext
fvPatchField
+updateCoeffs() : void
+evaluate() : void

zeroGradientFvPatchField
+updateCoeffs() : void
+evaluate() : void

fixedValueFvPatchField
+updateCoeffs() : void
+evaluate() : void
```
Viscosity model hierarchy

and the nu Template Method

```cpp
// Return the laminar viscosity.
virtual tmp<volScalarField> nu() const = 0;
```
The Template Method is the virtual member function (method) to be overridden, it has nothing to do with C++ templates.

**Best practice**: utilize virtual member functions (dynamic polymorphism) to extend existing libraries without modifying them.
A single class contains different sub-algorithms.

Sub-algorithms can be selected at runtime.

Combining sub-algorithms does not require programming.

Basically the composition of the Template Method for sub-algorithm hierarchies.

**Best practice:** when unsure about sub-algorithm combinations, implement the Strategy Pattern.
Foam::lduMatrix

```cpp
    solverPerf = lduMatrix::solver::New
    ( psi.name() + pTraits<Type>::componentNames[cmpt],
      *this, bouCoeffsCmpt, intCoeffsCmpt, interfaces,
      solverControls
    )->solve(psiCmpt, sourceCmpt, cmpt);
```

selects a linear solver as a (solution) Strategy.
autoPtr<lduMatrix::preconditioner> preconPtr = lduMatrix::preconditioner::New(*this, controlDict_);

Each lduMatrix::solver selects its pre-conditioner as a (preconditioning) Strategy.
From GoF Design Patterns Book [9]: “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

**Subject**

- Has a state that is updated when the subject is modified.
- Forwards the `update` call to a list of its observers.
  
```cpp
void subject::update()
{
    for (auto& observer : observers_)
        observer.update();
}
```

**Observers**

- Implement the `update` interface.
- Register themselves to the subject via their constructor.
Example: Particles tracked along Lagrangian trajectories in an Eulerian (background) mesh

- Lagrangian-cloud particles know which cell they are in.
- The Eulerian mesh is the subject that changes state.
- Lagrangian particle cloud is an observer.
- Vice-versa is also relevant, resulting in 6-way coupling (mass, momentum, energy exchange $\times 2$).
Example: Particles tracked along Lagrangian trajectories in an Eulerian (background) mesh

1. The Eulerian mesh (subject) changes state: it is refined.
2. The Eulerian mesh (subject) updates its observers
   
   ```cpp
   for (auto& observer : observers)
       observer.update(cellMap);
   ```
3. The Lagrangian cloud is an observer
   
   ```cpp
   for (auto& particle : cloud)
   {
       auto found = cellMap.find(particle.cellLabel());
       if (found)
       {
           auto newCellLabel = cloud.find(particle, cellMap);
           particle.setCellLabel(newCellLabel);
       }
   }
   ```
**Observer IV**

**subject**

- observers_: observerList
- state_: state

+ addObserver(const observer&) : void
+ removeObserver(const observer&) : void
+ update() : void

**observer**

subject_: const subject&

+ observer(const subject&)  
+ update() : void

**concreteObserver**

+ update() : void
Example: write all data that should be written using the same output frequency

A single

```java
runTime.write();
```
call in the solver application, and

```java
for (regIOobject& : regIOobjects)
    regIOobject.writeObject();
```

in the `Time` class is better than manually typing

```java
if (runTime.writeTime())
{
    alpha.write();
    surfaceMesh.write();
    cloud.write();
    ...
}
```
in a solver application. It is necessary for reactive flows.
Foam::Time controls simulation (write) time and it is an objectRegistry. Foam::Time::write() loops over all registered fields and writes them to the drive.
while (runTime.loop()) // runTime state
{
    #include "CourantNo.H"

    // Pressure-velocity PISO corrector
    {
        #include "UEqn.H"
        // --- PISO loop
        while (piso.correct())
        {
            #include "pEqn.H"
        }
    }

    laminarTransport.correct();
    turbulence->correct();
    runTime.write(); // runTime state
}

- A CFD solver is a procedural application.
- Fields (velocity, pressure, density, temperature, ...)
  are global variables, modified by FVM differential
  operators / solution algorithms.
- Observer Pattern simplifies custom post-processing
  using OpenFOAM Function Objects (not C++ function
  objects).
Software Design Patterns in Research Software
Observer VIII

- **runTime** is the **subject** that changes state:
  - time-step increment
  - reached output time
  - reached end time

- **Function Objects** are the **observers**.
  - They access other (mesh or time) observers and "work" on them: compute the maximal and minimal temperature, sample the velocity over a line segment, ...

- **OpenFOAM Function Objects** change solver behavior without modifying solver application's source code.
Time changes state in ‘read’ and ‘run’ and forwards appropriate calls to ‘functionObject’.

**OpenFOAM Function Objects**

<table>
<thead>
<tr>
<th>Time</th>
<th>functionObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>-functionObjects_ : functionObjectList</td>
<td></td>
</tr>
<tr>
<td>+read() : bool</td>
<td>+read() : void</td>
</tr>
<tr>
<td>+run() : bool</td>
<td>+execute() : void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>concreteFunctionObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>+read() : void</td>
</tr>
<tr>
<td>+execute() : void</td>
</tr>
<tr>
<td>+write() : void</td>
</tr>
<tr>
<td>+end() : void</td>
</tr>
</tbody>
</table>
OpenFOAM Function Objects

- Observer is also used within Function Objects themselves: `fvMesh` is an `objectRegistry`, FOs fetch objects registered to the mesh and perform live (post-)processing tasks as the simulation runs.

- This saves research time and HPC resources (green computing): live post-processing can be used to stop large-scale simulations as soon as the results are too erroneous.
Geometric Fields:

- Values grouped into **internal** values and **boundary patch** values.
- Internal values associated with **cell centers** (alternatively: face centers or cell corner-points), boundary with face centers (alternatively, face corner-points).
The mesh connectivity changes with mesh refinement / unrefinement.
- GeometricFields do not map to the mesh.
- Mesh motion stretches/compresses finite volume faces.
  - Volumetric fluxes change magnitudes.
- Each time the mesh is updated, the fields are updated.
- `fvMesh` is a Subject, `GeometricFields` are the Observers.
Best practice

- Use when the same member function (write, map, execute, read, update) must be called for many objects.
Using a Creational Pattern to construct objects (select types) at runtime makes the solver application highly configurable.

No modification to the solver application is required to select boundary conditions, dynamic mesh handling, discretization and interpolation schemes, models, ...
Runtime Type Selection (RTS) is OpenFOAM’s Creational Pattern.

**RTS constructs OpenFOAM objects based on user input.**
- Ease-of-use: RTS tables provide information about available types and their parameters.
- Simplifies research: “constructing” the PDE discretization and solution via configuration files.

OpenFOAM’s RTS in a nutshell:
- RTS stores a class-static hash-table that maps strings to a virtual member function pointer.
- This so-called RTS table is initialized for the base class in its implementation file.
- The RTS table is extended in implementation files of derived classes.
- The RTS code is generated using preprocessor macros
  - RTS declaration and definition
  - RTS table extension
**Best practice:** if a research software provides a creational pattern, learning how to use it simplifies testing and saves time in research, compared to hacking your own "if-then-else" code for different types.
OpenFOAM RTS macros expanded with **gcc -E**: no need to learn how this works to use it

```cpp
typedef autoPtr<implicitSurface> (*ITstreamConstructorPtr)( ITstream is );
typedef Foam::HashTable<ITstreamConstructorPtr, Foam::word, Foam::Hash<Foam::word> > ITstreamConstructorTableType;
typedef Foam::HashTable<std::pair<Foam::word, int>, Foam::word, Foam::Hash<Foam::word> > ITstreamConstructorCompatTableType;
static ITstreamConstructorTableType* ITstreamConstructorTablePtr_;
static std::unique_ptr<ITstreamConstructorCompatTableType> ITstreamConstructorCompatTablePtr_;
static ITstreamConstructorCompatTableType& ITstreamConstructorCompatTable();
static void ITstreamConstructorTablePtr_construct(bool load);

```

```cpp
template<class implicitSurfaceType>
struct addAliasITstreamConstructorToTable
{
    explicit addAliasITstreamConstructorToTable ( const Foam::word& k, const Foam::word& alias, const int ver ) {  
        ITstreamConstructorCompatTable() .set(alias, std::pair<Foam::word, int>(k,ver));
    }
};

```

```cpp
template<class implicitSurfaceType>
struct addITstreamConstructorToTable
{
    static autoPtr<implicitSurface> New ( ITstream is ) {
        return autoPtr<implicitSurface>(new implicitSurfaceType(is));
    } explicit addITstreamConstructorToTable ( const Foam::word& k = implicitSurfaceType::typeName ) {
        ITstreamConstructorTablePtr_ -> construct(true);
        if (!ITstreamConstructorTablePtr_ -> insert(k, New)) {
            std::cerr << "Duplicate entry " << k << " in runtime table " << "implicitSurface" << std::endl;
            Foam::error::safePrintStack(std::cerr);
        }
    } ~addITstreamConstructorToTable() {
        ITstreamConstructorTablePtr_ -> construct(false);
    } addITstreamConstructorToTable (const addITstreamConstructorToTable&) = delete;
    void operator= (const addITstreamConstructorToTable&) = delete;  
};
```
Software Design Patterns [9]: design structures that combine inheritance and composition and have emerged repeatedly as best-practice solutions for specific design problems.

Software Design Patterns (examples from OpenFOAM)

- **Template Method**: boundary conditions, viscosity models, discretization schemes, ...
- **Strategy**: transport models, solvers and preconditioners, ...
- **Observer**: dynamic mesh handling, IO synchronisation
- **OpenFOAM’s Creational Pattern**: Runtime-Type Selection (RTS), used everywhere.

Not covered in this webinar

- **Facade**: Level Set / Front Tracking (Additional Slides)
- **Curiously Recurring Template Pattern (CRTP)**: Discrete Parcel Method (Additional Slides)
We didn’t cover everything, but

- **Type Lifting** for geometric fields and differential operators “+” **generic traits** for tensor rank calculation “+” **Template Method** and **RTS** for discretization and interpolation schemes + **Strategy** and **RTS** for linear solvers “=”

OpenFOAM’s Domain-Specific Language for Partial Differential Equation discretization.

```cpp
fvScalarMatrix TEqn
(
    fvm::ddt(T)
    + fvm::div(phi, T)
    - fvm::laplacian(DT, T)
    ==
    fvOptions(T)
);
TEqn.solve();
```
Design Patterns speed up research, if there is a high degree of methodological uncertainty: we don’t know which algorithms will work, in which combination.

Avoiding dogmatism: not every design question has to be answered by a pattern.

When dealing with legacy research code, it helps a lot understand its design principles: cargo-cult programming is quicker, but can tank research projects in the long-run.
Funded by the German Research Foundation (DFG) - Project-ID 265191195 - CRC 1194

Interaction between Transport and Wetting Processes

Z-INF sub-project (Prof. Dr. rer. nat. Dieter Bothe, Prof. Dr. Christian Bischof)
Additional Slides and References
OpenFOAM’s UML legend

- **Truncated**
- **PublicBase**
- **ProtectedBase**
- **PrivateBase**
- **Undocumented**
- **Templ<T>** with `<int>`
- **Used**

- **m_usedClass**
OpenFOAM uses Generic Programming (GP) for type lifting and traits.

- **Type lifting**: same code is re-used without modification with completely unrelated types. In OpenFOAM, everything is type-lifted for all tensors (scalar, vector, tensor, symmetric tensor, spherical tensor).

- **Template specialization**: e.g. specializing a fixed value tensor boundary condition as a scalar total pressure boundary condition.

- **Traits**: determine the tensor rank of the return type of $\nabla v$ (used in differential operators).
C++ Generic Programming in OpenFOAM (crash course) II
C++ templates: if-then-else for types

```cpp
template<class Type>
Type sum(const UList<Type>& f)
{
    if (f.size())
    {
        Type Sum = pTraits<Type>::zero;
        TFOR_ALL_S_OP_F(Type, Sum, +=, Type, f)
        return Sum;
    }
    else
    {
        return pTraits<Type>::zero;
    }
}
```

- **template** (Merriam Webster dictionary)
  - a gauge, **pattern, or mold** (such as a thin plate or board) used as a **guide to the form of a piece being made**
  - a molecule (as of DNA) that serves as a **pattern for the generation of another macromolecule** (such as messenger RNA)
  - something that establishes or **serves as a pattern**
template<class Type>
class fixedValueFvPatchField
{
  public fvPatchField<Type>

  A boundary condition class template is
  type-lifted all tensors.
  The same is done for arithmetic field
  operators, discretization schemes, ...
```cpp
#define makePatchTypeFieldTypedef(fieldType, type) \
    typedef type##FvPatchField<fieldType> \
    CAT4(type, FvPatch, CAPITALIZE(fieldType), Field);

class totalPressureFvPatchScalarField :
    // fixedValueFvPatchField<scalar>
public fixedValueFvPatchScalarField

- Specialized boundary conditions for pressure, temperature, velocity,...
```
The return-type of the gradient function template is determined based on the argument.

- The gradient of a scalar field is a vector field.
template<class arg1, class arg2>
class outerProduct
{
    public:

    typedef typename typeOfRank
    <
        typename pTraits<arg1>::cmptType,
        direction(pTraits<arg1>::rank) + direction(pTraits<arg2>::rank)
    >::type type;

    Traits determine the component types of scalars, vectors, tensors.
    Component type and rank traits promote outer product type.
    **One only needs this if the research involves extending the set of differential operators.**
    Type-lifting is enough for 99% of research using OpenFOAM.
template<class Type>
class inletOutletFvPatchField
:
  public mixedFvPatchField<Type>
{

protected:

  // Protected data

  //- Name of flux field
  word phiName_;
Facade hides the complexity of sub-algorithms, for example, the order of execution:

```cpp
void lent::advect()
{
  frontReconstructionModel->reconstructFront(); // Updates Front-Mesh communication.
  distanceFieldCalculator->calcSignedDistances();
}
```
Best practice: implement sub-algorithms as Strategies, test them individually, then integrate them in a specific execution order using Facade.
template<typename Parameter>

class MyType
    : public Parameter

- Class template inheriting from its template parameter.
- Used in generic programming for policy-based design: extending the host class (*MyType*) interface by inheriting from the template parameter (*Parameter*).
Curiously Recurring Template Pattern (CRTP) is **curiously recurring and nested** for the Lagrangian / Eulerian Discrete Parcel Method.

```cpp
namespace Foam {

    template<class CloudType>
    class ReactingCloud :
    {
        public CloudType,
        public reactingCloud
    
    
        typedef ReactingCloud
        <
            ThermoCloud
            <
                KinematicCloud
                <
                    Cloud<basicReactingParcel>
                >
            >
        >
        basicReactingCloud;
    
}


