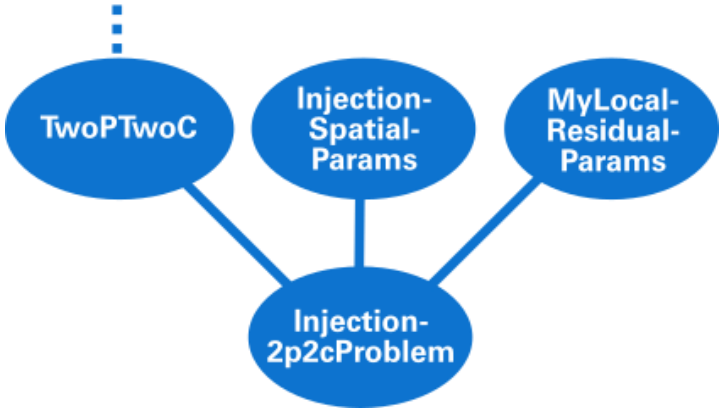



# The DuMu$^x$ Property System

Flexible Compile-Time Parameters

DuMu$^x$

# Parameters and Properties

- `Parameters` are read and set at **run-time**
  - Can have a default value which is used if the user doesn’t provide a value at run-time

- `Properties` are known and set at **compile-time**
  - Can be used e.g. as template parameters (are types or have constexpr qualifier)
  - No run-time penalty, enable the compiler to optimize

# Template parameters

- Many classes depend on variable types that have to be known at compile-time

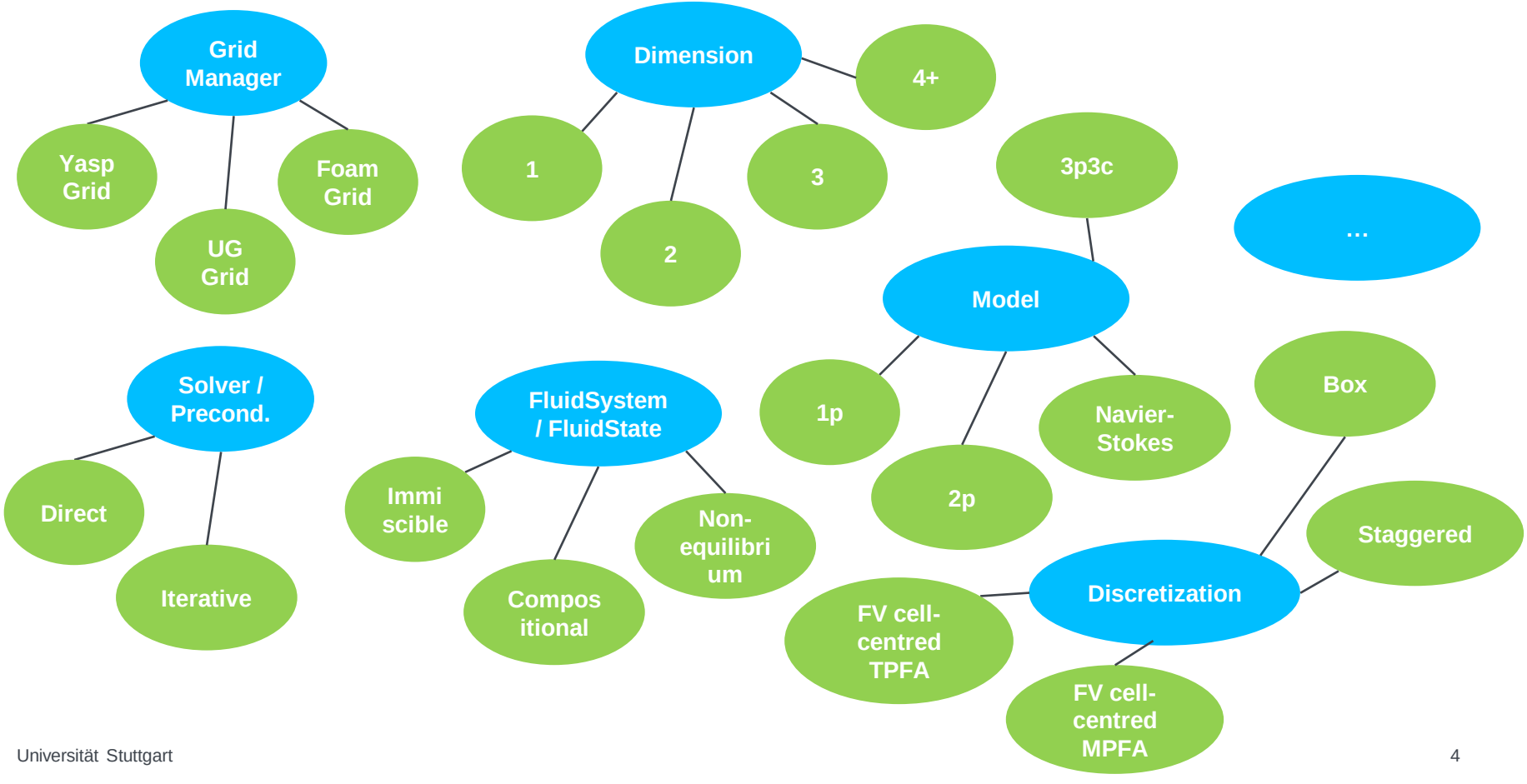
*Example from the C++ standard library*

```
template <typename T>
class vector; // class declaration of std::vector
...

std::vector<int> v; // a vector of ints
```

- This allows flexibility → generic programming
- Less code duplication
- Efficient → decisions are made at compile-time

# Examples of compile time parameters



# Too many template parameters

- For some classes, providing all of these parameters can be very cumbersome and error-prone.

*Example from dune-pdelab:* ***9*** *template parameters*

```
using DGGO2 = Dune::PDELab::GridOperator<GFS,GFS,LOP,MBE,
NumberType,NumberType,NumberType,DGCC2,DGCC2>;

DGGO2 dggo2(fs.getGFS(),dgcc2,fs.getGFS(),dgcc2,lop,MBE(5));
```

# Traits classes

- Traits classes are the usual way to group template parameters.

```
struct MyGridOperatorTraits {
  using FromGFS = …;
  …
};

using DGGO2 = Dune::PDELab::GridOperator<MyGridOperatorTraits>;
```

- Inheritance can lead to unexpected results:

```
struct MyBaseTraits {
  using Scalar = int;
  using Vector = std::vector<Scalar>; };
struct MyDoubleTraits : public MyBaseTraits {
  using Scalar = double; };
…
MyDoubleTraits::Vector v{1.41421, 1.73205};
```

- v is a `std::vector<int>`!

# Design of the Property System

- The DuMu$^x$ property system is based on **C++ template specialization** hidden behind **preprocessor magic**.
- A hierarchy of nodes -- called **type tags** -- is defined. All parameters are labeled and attached to the appropriate nodes in this acyclic graph.
- The labels are called **property tags**, whereas the parameters actually attached are called **properties**.
- The definition of properties may **depend on** arbitrary other properties, which may be **overwritten** at any higher node of the acyclic graph.
- The only requirement for properties is that they may not exhibit **cyclic dependencies**.

# Better than traits classes?

- Implement the example above using the property system:

```
NEW_TYPE_TAG(BaseTag);
SET_TYPE_PROP(BaseTag, Scalar, int);
SET_TYPE_PROP(BaseTag, Vector,
              std::vector<GET_PROP_TYPE(BaseTag, Scalar)>);

NEW_TYPE_TAG(DoubleTag, INHERITS_FROM(BaseTag));
SET_TYPE_PROP(DoubleTag, Scalar, double);

using Vector = GET_PROP_TYPE(DoubleTag, Vector);
Vector v{1.41421, 1.73205};
```

- `v` is a `std::vector<double>`!

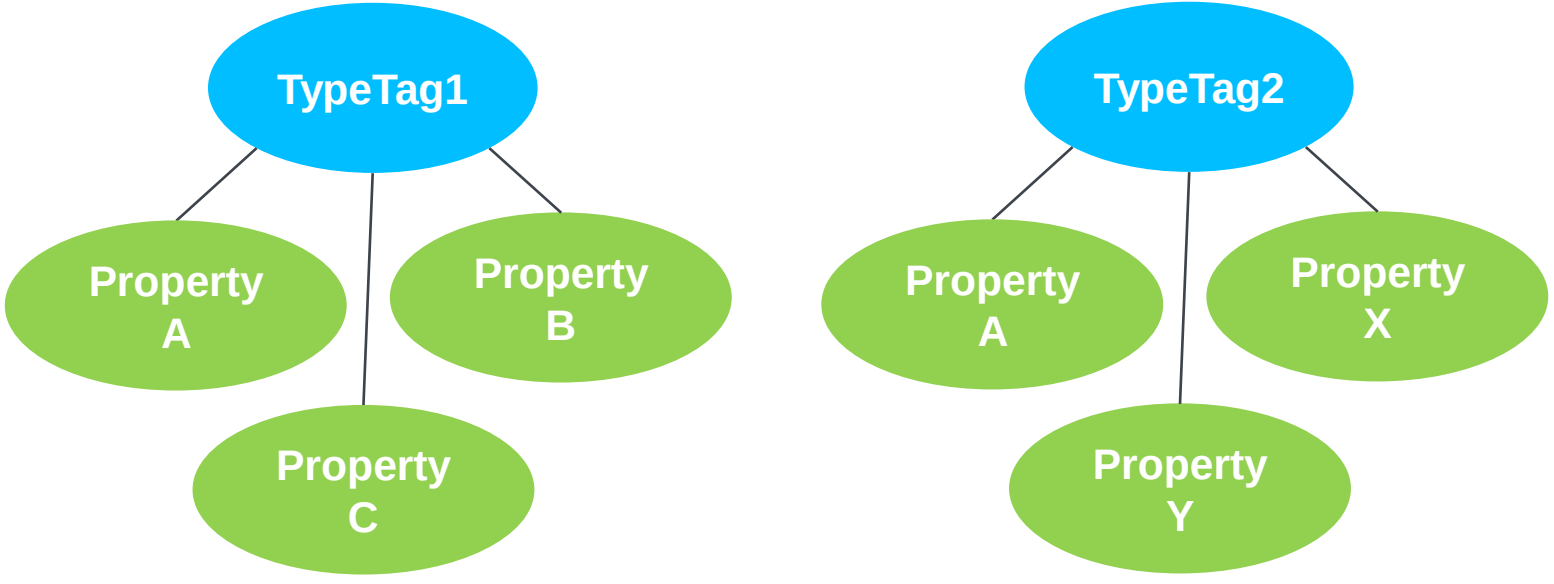
# Configuring and setting up models at compile time

- The DuMu$^X$ way → `Properties`
  - A so-called `TypeTag` bundles all necessary information → only ONE template parameter
  - `Properties` (data types and values) can be retrieved using pre-processor macros

```
template<class TypeTag>
class InjectionProblemTwoP
{
  using VolumeVariables = typename GET_PROP_TYPE(TypeTag, VolumeVariables);
  constexpr auto useIFS = GET_PROP_VALUE(TypeTag, EnableBoxInterfaceSolver);
  ...
};
```

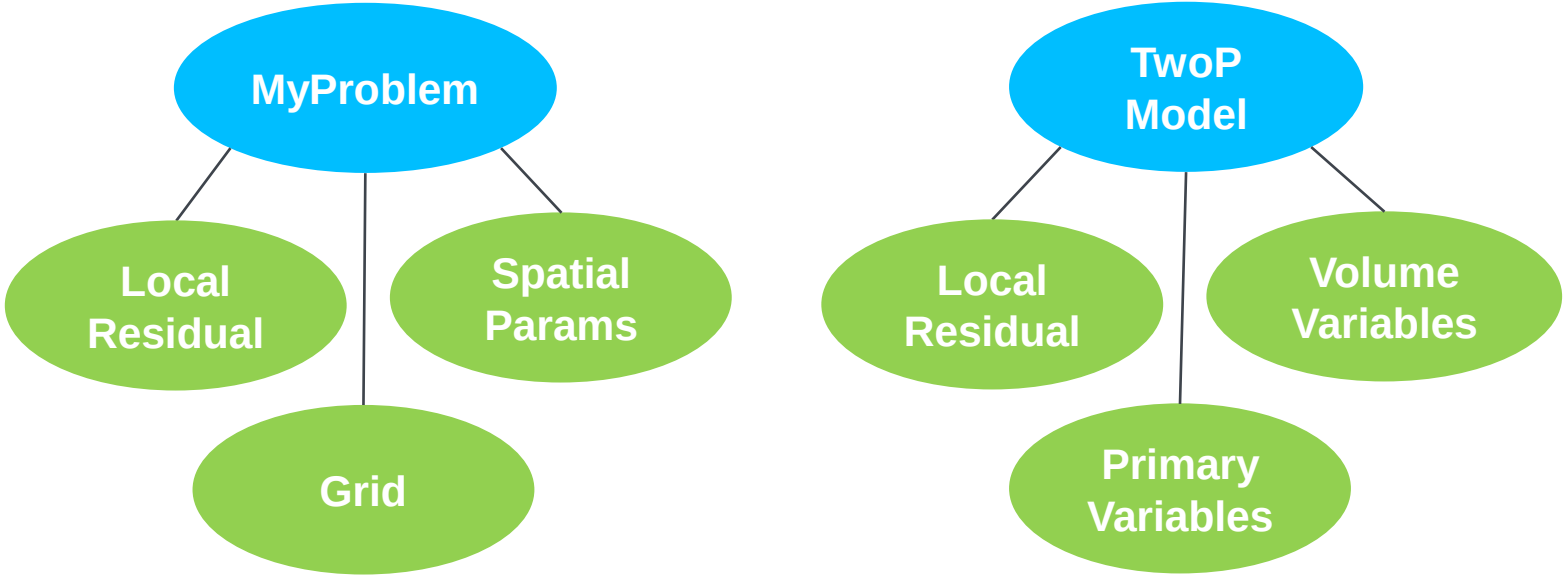
# The DuMu$^x$ property system

- Extension → tree of so called `TypeTag` nodes
- Each `TypeTag` is associated with `Properties`



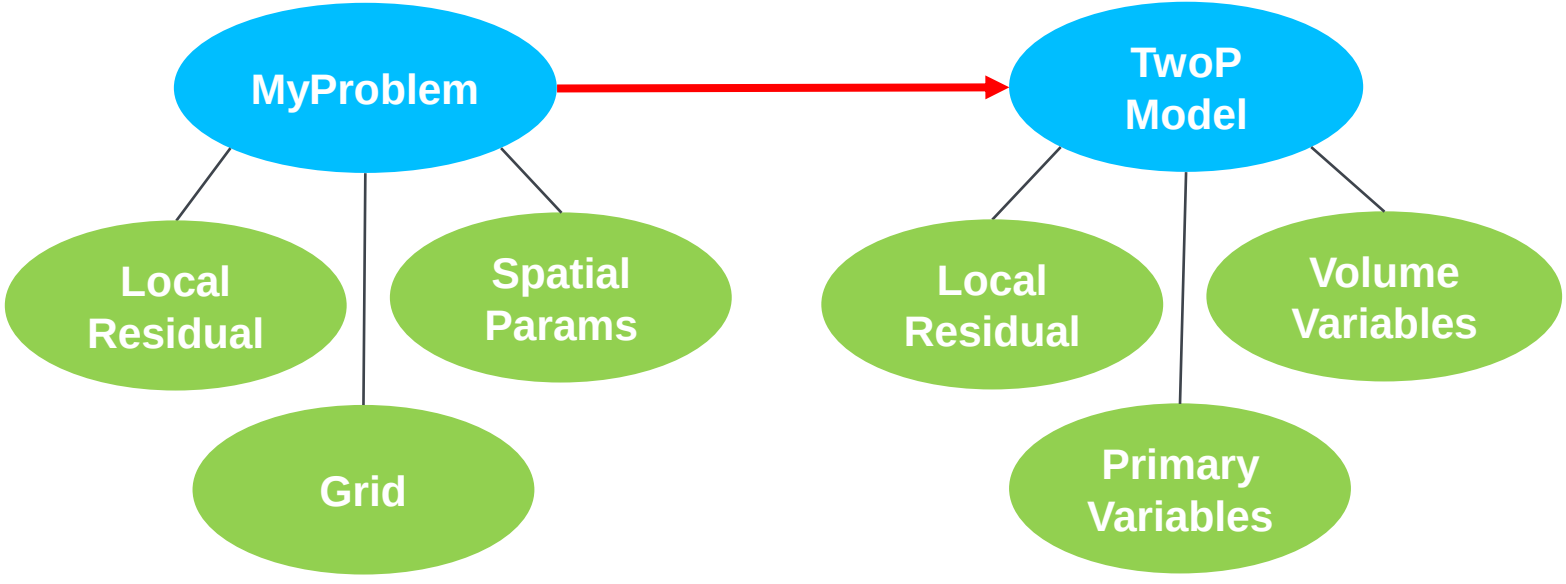
# The DuMu$^x$ property system

- Tree of so called `TypeTag` nodes
- Each `TypeTag` is associated with `Properties`



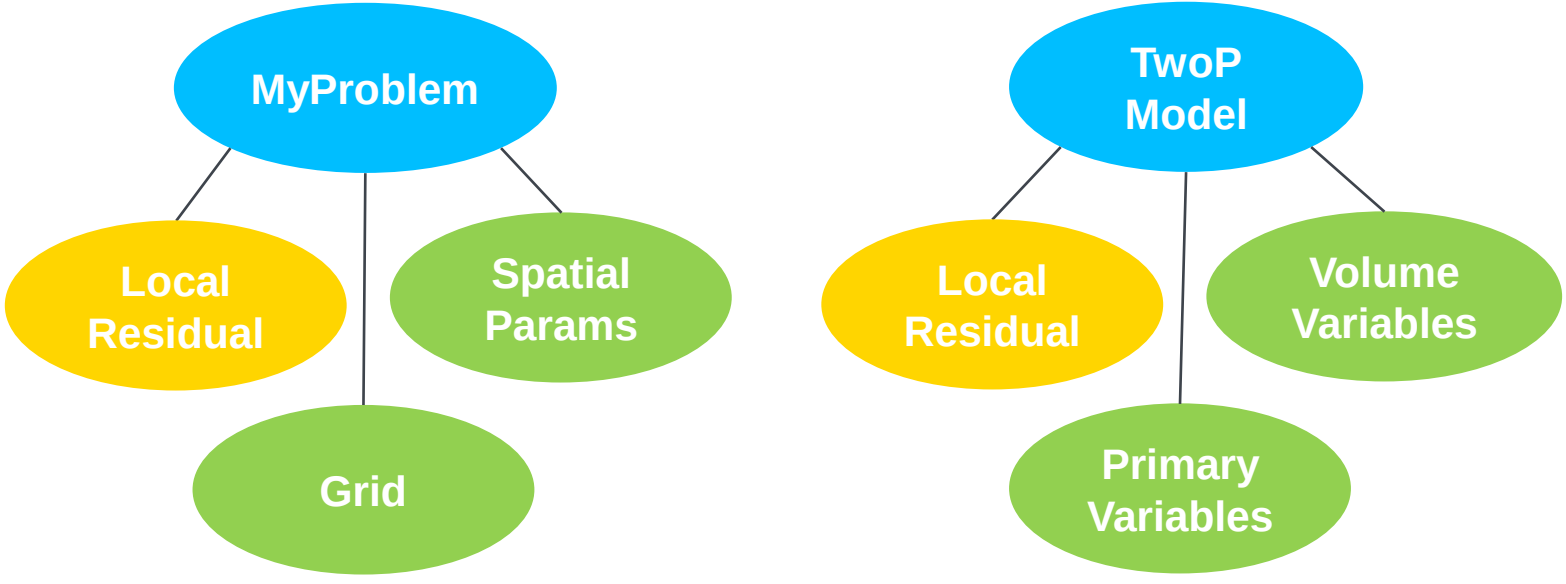
# The DuMu$^x$ property system

- Tree of so called `TypeTag` nodes
- Each `TypeTag` is associated with `Properties`



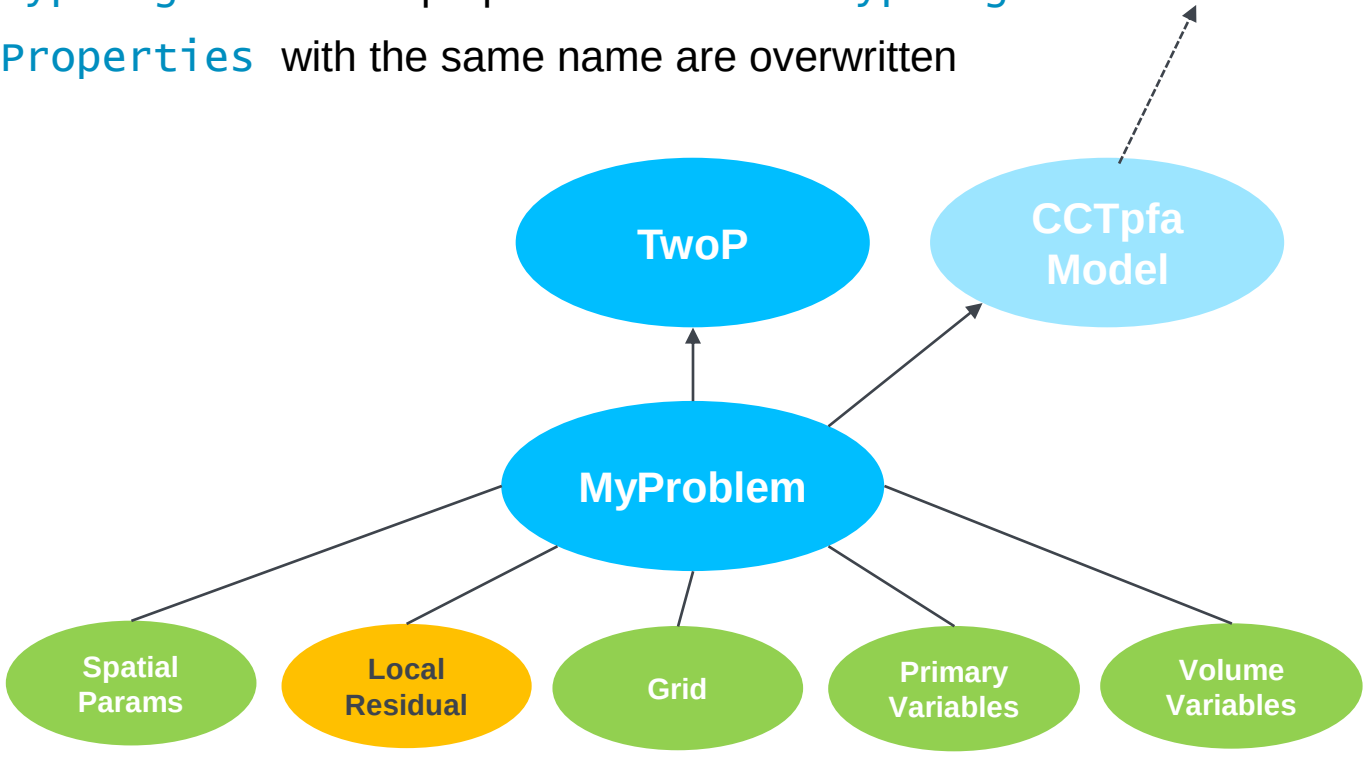
# The DuMu$^x$ property system

- Tree of so called `TypeTag` nodes
- Each `TypeTag` is associated with `Properties`

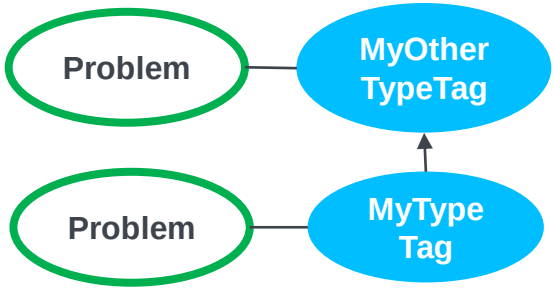


# The DuMu$^x$ property system

- Hierarchy / Inheritance
  - `TypeTag`s can inherit properties from other `TypeTag`s
  - `Properties` with the same name are overwritten



# How to use



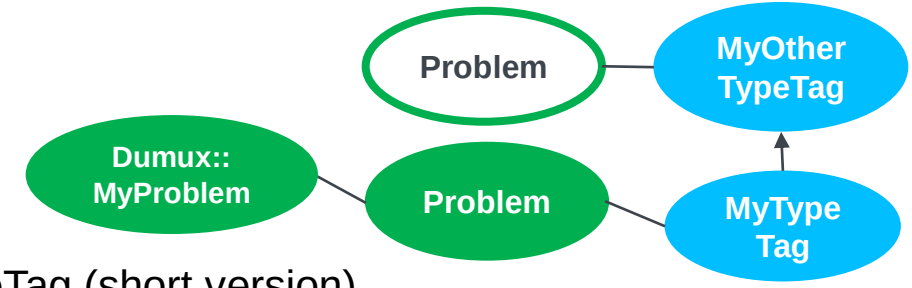

- Creating new `TypeTag` nodes

```
namespace Properties {
NEW_TYPE_TAG(MyTypeTag);
NEW_TYPE_TAG(MyOtherTypeTag, INHERITS_FROM(MyTypeTag));
} // end namespace Properties
```

- Creating new `PropTag`s (empty, unset properties) → Property names are unique!

```
namespace Properties {
NEW_PROP_TAG(Problem);
} // end namespace Properties
```

# How to use




- Setting **type** properties for a specific TypeTag (short version)

```
namespace Properties {
SET_TYPE_PROP(MyTypeTag, Problem, Dumux::MyProblem<TypeTag>);
} // end namespace Properties
```

- Setting **type** properties for a specific TypeTag (long version)

```
namespace Properties {
SET_PROP(MyTypeTag, Problem) {
    using type = Dumux::MyProblem<TypeTag>;
};
} // end namespace Properties
```

# How to use

- Setting **boolean** properties for a specific TypeTag (short version)

```
namespace Properties {
SET_BOOL_PROP(MyTypeTag, EnableBoxInterfaceSolver, true);
} // end namespace Properties
```

- Setting **boolean** properties for a specific TypeTag (long version)

```
namespace Properties {
SET_PROP(MyTypeTag, EnableBoxInterfaceSolver) {
    static constexpr bool value = true;
};
} // end namespace Properties
```

# How to use

- Getting the property type set for a specific TypeTag

```
namespace Dumux {
template <class TypeTag>
class Problem
{
     using Scalar = typename GET_PROP_TYPE(TypeTag, Scalar);
} // end namespace Dumux
```

# How to use

- „Top-level“ classes in DuMu$^{X}$ use these aliases and the property system
- Changing a property e.g. in the `Problem` will affect all classes using the same TypeTag – without further changing anything in those classes!
- Example: see *Exercise*

# How to use

- As seen earlier:
- Each model defines a set of properties grouped in a TypeTag
  - e.g. `TwoP, TwoPTwoC, TwoPNI`
- By deriving your problem TypeTag from those TypeTags your problem inherits all type information needed to set up the model at compile time!

# Hierarchy of TypeTags used for the two-phase incompressible problem