Representation of configuration options with subclasses of `Parameters`

Wojciech Śmigaj

October 19, 2021

Outline

Met Office

Outline

Motivation

Met Office

Many classes in JEDI receive an `eckit::Configuration` object in their constructor and call its methods to retrieve values of YAML configuration options.

```
DifferenceCheck::DifferenceCheck(const eckit::Configuration &conf, ...)
  : FilterBase(conf, ...),  // sets this->config_ to conf
    ref_(conf.getString("reference")),
    val_(conf.getString("value"))
{
  ...
}

void DifferenceCheck::applyFilter(...) const {
  const float missing = util::missingValue(missing);
  float vmin = config_.getFloat("minvalue", missing);
  float vmax = config_.getFloat("maxvalue", missing);
  ...
}
```

Disadvantages:

1. Users need to read the implementation to find the list of supported options.
2. There's no obvious place for the documentation of these options.
3. Unrecognised (e.g. mistyped) options are silently ignored.
4. Error messages do not provide sufficient information about the error location.

These disadvantages can be overcome by using the `Parameters` framework.

Main components of the framework

Met Office

- `Parameters` is an abstract base class; each of its concrete subclasses represents a set of key-value pairs on a single level of the YAML hierarchy.
- A `Parameter<T>`, `RequiredParameter<T>` or `OptionalParameter<T>` represents a single key-value pair, with the value expected to be of type T. They differ in their handling of the absence of their key from the YAML file:
  - A `RequiredParameter<T>` throws an exception.
  - A `Parameter<T>` is set to a default value specified its construction.
  - An `OptionalParameter<T>` is set to a special value meaning “missing”.

Example

YAML file

Met Office

Consider a thinning filter inspecting observations one-by-one and retaining only those located far enough from all previously retained ones. It might accept the following YAML options:

```
# Minimum distance between two retained observations
min distance (km): 1000
# Inspect observations in random order? Optional; by default, true.
shuffle: true
# Optional; if not set, use a seed based on the current (calendar) time.
random seed: 12345
```

Example

Parameters subclass

Met Office

```
#include "oops/util/parameters/OptionalParameter.h"
#include "oops/util/parameters/Parameter.h"
#include "oops/util/parameters/Parameters.h"
#include "oops/util/parameters/RequiredParameter.h"

class ThinningParameters : public oops::Parameters {
  OOPS_CONCRETE_PARAMETERS(ThinningParameters, Parameters)
 public:
  oops::RequiredParameter<float> minDistance{
    "min distance (km)",
    "Minimum distance between two retained observations",
    this};
  oops::Parameter<bool> shuffle{
    "shuffle",
    "Inspect observations in random order",
    true, this};
  oops::OptionalParameter<int> randomSeed{
    "random seed",
    "Seed for the random number generator shuffling observations. "
    "If omitted, taken from the current time",
    this};
};
```

Example

Usage of the Parameters subclass

Met Office

```
const eckit::Configuration &config = ...;

// Load configuration options into a Parameters object
ThinningParameters params;
params.validateAndDeserialize(config);

// Access the value of a Parameter or RequiredParameter
if (params.shuffle.value()) {
  // Access the value of an OptionalParameter
  const boost::optional<int> &randomSeed =
    params.randomSeed.value();
  if (randomSeed != boost::none)
    srand(*randomSeed);
  else
    srand(time(nullptr));
}
```

Example

A multi-level YAML hierarchy

Let us allow the user to optionally specify a range of latitudes for which the filter should produce extra debugging output:

```
min distance (km): 1000
verbose latitudes:
  min: 10
  max: 20
```

Example

Extensions to the Parameters subclass

Met Office

```
/// A closed interval.
class RangeParameters : public oops::Parameters {
  OOPS_CONCRETE_PARAMETERS(RangeParameters, Parameters)
 public:
  oops::RequiredParameter<float> min{"min", "Lower bound", this};
  oops::RequiredParameter<float> max{"max", "Upper bound", this};
};

/// Options controlling the operation of a thinning filter.
class ThinningParameters : public oops::Parameters {
  OOPS_CONCRETE_PARAMETERS(ThinningParameters, Parameters)
 public:
  ...
  oops::OptionalParameter<RangeParameters> verboseLatitudes{
    "verbose latitudes",
    "Zonal band containing observations for which verbose output "
    "should be produced",
    this};
};
```

Example

Usage of the extended `Parameters` subclass

```
const boost::optional<RangeParameters> &verboseLatitudes =
  params.verboseLatitudes.value();

// Within the loop inspecting observations
const float obsLatitude = ...;
if (verboseLatitudes != boost::none &&
    obsLatitude >= verboseLatitudes->min.value() &&
    obsLatitude <= verboseLatitudes->max.value()) {
  ...  // Print observation to the log
}
```

More advanced features

Met Office

Detailed information about the following more advanced features can be found on the ReadTheDocs webpage:

- `Parameter` objects can hold not only numbers, Booleans and `Parameters` subclasses, but also
  - strings, lists (vectors) and maps
  - instances of JEDI-specific classes: `util::DateTime`, `util::Duration`, `oops::Variables` and `ufo::Variable`
  - `eckit::LocalConfiguration` objects.
- Numeric `Parameter` objects can be constrained to accept only values from a specific range.
- `PolymorphicParameter` objects can represent YAML sections in which the list of allowed keys depends on the value of a "discriminator" key.
- `Parameters` objects can be converted back to `LocalConfiguration` objects (particularly useful when interfacing with Fortran code).

Remarks

Met Office

- Passing a description to a `Parameter`'s constructor is optional. Alternatively, documentation can be placed in a Doxygen comment, but then it will not be added to the JSON schema. Compare:

```
oops::Parameter<bool> shuffle{
  "shuffle",
  "Inspect observations in random order",
  true, this};

/// Inspect observations in random order
oops::Parameter<bool> shuffle{
  "shuffle", true, this};
```

- Most calls to the `value()` method can be omitted: `(Required/Optional)Parameter<T>` objects overload the conversion operator to `const T&`. Therefore the following lines are equivalent:

```
if (params.shuffle.value()) { ... }
if (params.shuffle) { ... }
```

Outline

Refactoring an interface implementation

Met Office

1. Identify all YAML options accessed by the class.

```
DifferenceCheck::DifferenceCheck(
    const eckit::Configuration &conf, ...)
  : FilterBase(conf, ...),  // sets this->config_ to conf
    ref_(conf.getString("reference")),
    val_(conf.getString("value")) {
  ...
}

void DifferenceCheck::applyFilter(...) const {
  const float missing = util::missingValue(missing);
  float vmin = config_.getFloat("minvalue", missing);
  float vmax = config_.getFloat("maxvalue", missing);
  ...
}
```

Refactoring an interface implementation

Met Office

2. Define a subclass of `Parameters` (or a more specialised class such as `ufo::FilterParametersBase` or `oops::ModelParametersBase`) that will hold options of the refactored class. Add a member variable representing each option.

```
/// Options controlling the operation of the DifferenceCheck filter.
class DifferenceCheckParameters : public FilterParametersBase {
  OOPS_CONCRETE_PARAMETERS(DifferenceCheckParameters, FilterParametersBase)
 public:
  oops::RequiredParameter<std::string> ref{
    "reference", "Reference variable", this};
  oops::RequiredParameter<std::string> val{
    "value", "Test variable", this};
  oops::OptionalParameter<float> minvalue{
    "minvalue",
    "Minimum allowed value of the difference "
    "'test variable - reference variable'",
    this};
  oops::OptionalParameter<float> maxvalue{
    "maxvalue",
    "Maximum allowed value of the difference "
    "'test variable - reference variable'",
    this};
};
```

Refactoring an interface implementation

Met Office

3. Add a public typedef `Parameters_` to the class being refactored. Make it an alias for the newly defined `Parameters` subclass.

```
class DifferenceCheck : public FilterBase {
 public:
  typedef DifferenceCheckParameters Parameters_;
  ...
};
```

4. Change the signature of the constructor: replace the `const eckit::Configuration &` parameter by `const Parameters_ &`.

```
  class DifferenceCheck : public FilterBase {
   public:
-   DifferenceCheck(const eckit::Configuration &conf, ...);
+   DifferenceCheck(const Parameters_ &params, ...);
    ...
  }
```

Note

In implementations of `State` it is also necessary to change the signatures of `read()` and `write()`, and in implementations of `Increment` additionally `dirac()`. Refer to the Doxygen documentation.

Refactoring an interface implementation

Met Office

5. Replace any calls to methods of the `Configuration` object by references to member variables of `Parameters_`.

```
  DifferenceCheck::DifferenceCheck(
-     const eckit::Configuration &conf, ...)
+     const Parameters_ &params, ...)
-   : FilterBase(conf, ...),  // sets this->config_ to conf
+   : FilterBase(params, ...),
+     params_(params),        // save a copy of params in a member variable
-     ref_(conf.getString("reference")),
+     ref_(params.reference.value()),
-     val_(conf.getString("value")) {
+     val_(params.value.value()) {
    ...
  }

  void DifferenceCheck::applyFilter(...) const {
    const float missing = util::missingValue(missing);
-   float vmin = config_.getFloat("minvalue", missing);
+   float vmin = params_.minvalue.value().value_or(missing);
-   float vmax = config_.getFloat("minvalue", missing);
+   float vmax = params_.maxvalue.value().value_or(missing);
    ...
  }
```

Refactoring an interface implementation

Met Office

6. If the `Configuration` object used to be passed to Fortran code, call the `toConfiguration()` method of the `Parameters_` object to convert it to a `Configuration` object.

```
- GeometryQG::GeometryQG(const eckit::Configuration & conf,
+ GeometryQG::GeometryQG(const GeometryQgParameters & params,
                         const eckit::mpi::Comm & comm)
    : comm_(comm) {
-   qg_geom_setup_f90(keyGeom_, conf);
+   qg_geom_setup_f90(keyGeom_, params.toConfiguration());
    ...
  }
```

   Note: If the number of options passed to Fortran is small, consider passing each of them in a separate argument (of type `real`, `integer`, `oops_variables` etc.) instead.

7. For each parameter with a default value, remove the call to `f_conf % has()` (it will always return `.true.`).

```
- if (f_conf % has("qtotal")) then
-   call f_conf % get_or_die("qtotal", self % qtotal)
- else
-   self % qtotal = 1
- endif
+ call f_conf % get_or_die("qtotal", self % qtotal)
```

Refactoring an interface implementation

Met Office

Note

Your code does not need to call `validateAndDeserialize()` anywhere. It will be done by `oops` before it constructs the class you are refactoring.

Refactoring an interface implementation

Met Office

Examples

- ErrorCovariance: `lorenz95::ErrorCovarianceL95`
- Geometry: `lorenz95::Resolution`, `qg::GeometryQG`
- Increment: `lorenz95::IncrementL95`
- LinearModel: `lorenz95::TLML95`
- Model: `lorenz95::ModelL95`, `qg::ModelQG` (part 1), `qg::ModelQG` (part 2)
- ObsFilter: `ufo::Thinning` (C++ implementation), `ufo::GNSSROOneDVarCheck` (Fortran implementation)
- ObsOperator: `ufo::ObsGnssroBendMetOffice` (option values passed to Fortran individually rather than in a Configuration object), `ufo::ObsRadianceRTTOV` (`Parameters` subclass defined, but `ObsOperator` constructor still takes a `Configuration` object)
- State: `lorenz95::StateL95`

Outline

Refactoring an oops application

Met Office

1. Identify all YAML options accessed by the application.

```
template <typename MODEL, typename OBS>
class HofX3D : public Application {
 public:
  int execute(const eckit::Configuration & fullConfig) const {
    // Setup observation window
    const util::Duration winlen(fullConfig.getString("window length"));
    const util::DateTime winbgn(fullConfig.getString("window begin"));
    ...

    // Setup geometry
    const eckit::LocalConfiguration geometryConfig(fullConfig, "geometry");
    const Geometry_ geometry(geometryConfig, this->getComm());
    ...
  }
}
```

Refactoring an oops application

Met Office

2. Define a subclass of `Parameters` that will hold these options. Add a member variable representing each option.

```
/// \brief Top-level options taken by the HofX3D application.
template <typename MODEL, typename OBS>
class HofX3DParameters : public Parameters {
  OOPS_CONCRETE_PARAMETERS(HofX3DParameters, Parameters)
 public:
  typedef typename Geometry<MODEL>::Parameters_ GeometryParameters_;

  RequiredParameter<util::DateTime> windowBegin{
    "window begin",
    "Only observations taken after 'window begin' will be "
    "included in obs spaces"
    this};
  RequiredParameter<util::Duration> windowLength{
    "window length",
    "Observations taken after ('window begin' + 'window length') "
    "will not be included in obs spaces"
    this};
  RequiredParameter<GeometryParameters_> geometry{
    "geometry", "Model geometry", this};
  ...
};
```

Refactoring an oops application

Met Office

Notes

- The contents of many YAML sections will be passed to constructors of various oops interfaces, such as `Geometry` or `State`. These can typically be stored in instances of `InterfaceName<MODEL>::Parameters_`. There are exceptions; use the `HofX4D` and `GenEnsPertB` applications as guides.
- Some oops interfaces have not been refactored to use `Parameters` yet and their constructors still take a `Configuration` object. Use `(Optional/Required)Parameter<LocalConfiguration>` objects to represent the options passed to these constructors.

Refactoring an oops application

Met Office

3. At the start of the `execute()` function, create an instance of the new `Parameters` subclass and pass the top-level `Configuration` object to its `validateAndDeserialize()` method.

```
template <typename MODEL, typename OBS>
class HofX3D : public Application {
  typedef HofX3DParameters<MODEL, OBS> HofX3DParameters_;

 public:
  int execute(const eckit::Configuration & fullConfig) const {
    // Deserialize parameters
    HofX3DParameters_ params;
    params.validateAndDeserialize(fullConfig);
    ...
  }
}
```

Refactoring an oops application

Met Office

4. Replace any calls to methods of the `Configuration` object by references to member variables of the `Parameters` object.

```
  template <typename MODEL, typename OBS>
  class HofX3D : public Application {
    typedef HofX3DParameters<MODEL, OBS> HofX3DParameters_;
   public:
    int execute(const eckit::Configuration & fullConfig) const {
      ...

      // Setup observation window
-     const util::Duration winlen(fullConfig.getString("window length"));
+     const util::Duration winlen = params.windowLength.value();
-     const util::DateTime winbgn(fullConfig.getString("window begin"));
+     const util::DateTime winbgn = params.windowBegin.value();
      ...

      // Setup geometry
-     const eckit::LocalConfiguration geometryConfig(fullConfig, "geometry");
-     const Geometry_ geometry(geometryConfig, this->getComm());
+     const Geometry_ geometry(params.geometry.value(),
+                              this->getComm(), oops::mpi::myself());
      ...
    }
  }
```

Outline

Advantages of `Parameters`

Met Office

1. The type of the parameter passed to the class constructor indicates where to find declarations of all options supported by the class:

   ```
   DifferenceCheck(const DifferenceCheckParameters &params, ...);
   ```

2. Options can be described in strings passed to constructors of `Parameter` objects:

   ```
   oops::OptionalParameter<float> minvalue{
     "minvalue",
     "Minimum allowed value of the difference "
     "'test variable - reference variable'",
     this};
   ```

   or documented in Doxygen comments:

   ```
   /// Minimum allowed value of the difference
   /// 'test variable - reference variable'",
   oops::OptionalParameter<float> minvalue{"minvalue", this};
   ```

Advantages of `Parameters`

Met Office

3. `Parameters` objects can generate a JSON schema specifying the expected structure of the YAML tree. When the contents of a `Configuration` object are loaded ("deserialized") into a `Parameters` object, they can be validated against that schema. Unrecognised keys are treated as errors.
4. Error messages pinpoint the location of the offending key in the YAML tree:

```
Test "ufo/ObsFilters/testFilters" failed with unhandled
 exception: Error: YAML validation failed.
  Location:      /observations/2/obs filters/0
  Invalid value: {"filter":"Difference Check",
                  "filter variables":[{"name":"variable1"}],
                  "maxvalues":-3,
                  "reference":"var4@MetaData",
                  "value":"var3@MetaData"}
  Cause:         additional properties are not allowed
                  ('maxvalues' was unexpected)
```

Advantages of Parameters

Met Office

5. In future, schemas specifying the structure of YAML files accepted by applications and tests will be automatically exported to files, making it possible to
   - validate YAML files even before job submission
   - edit YAML files in Visual Studio Code with auto-completion and on-the-fly error detection.