Computing and Informatics, Vol. 35, 2016, 653–686

IMPLEMENTING THE FACTORY PATTERN WITH THE HELP OF REFLECTION

Matúš Chochlík

Department of Informatics
Faculty of Management and Informatics
Univerzitná 8215/1
010 26 Žilina, Slovakia
e-mail: Matus.Chochlik@fri.uniza.sk

Abstract. Reflection, reflection-based programming and metaprogramming are valuable tools for many programming tasks, like the implementation of persistence and serialization-like operations, object-relational mapping, remote method invocation, automatic generation of user-interfaces, etc., and also for the implementation of several design patterns. C++ as one of the most prevalent programming languages still lacks support for standardised, compiler-assisted reflection. In this paper we introduce in short the Mirror reflection library which is a part of an ongoing effort to add reflection to C++ and we will show how reflection can be used to greatly simplify the implementation of object factories – classes constructing instances of other classes from various external data representations.

Keywords: Reflection, metaprogramming, generic programming, design patterns, factory pattern

1 INTRODUCTION

Design patterns, as described in [1, 2] and elsewhere, present the established solutions to commonly occuring problems in the process of designing and implementing a software. However, one of the issues with such patterns is that their own implementation can become problematic, tedious and error-prone, especially when they are implemented manually by a software development team.

1.1 The Factory Pattern

In this paper we describe a way to partially automate the implementation of the *Factory* design pattern in the C++ language using the *Mirror C++ reflection utilities*. Since various definitions of the *factory* pattern differ slightly in details, for the purpose of this paper by *factory* we mean a class which can create instances of a *product* type, but does not require that the caller chooses the manner of construction, nor he supplies required parameters directly in the native data representation of C++.

Such factories separate the caller, who wants an instance of the *product* type, from details of the construction process. The caller just selects an appropriate factory class that can work with the data in an external representation from which the instance (or instances) should be constructed and the factory handles the details: It picks up the most suitable constructor based on the available input data (or lets the application user to pick up the constructor in case of factories using some form of user interface), gathers the required parameters, converts them into the native C++ data types, calls the selected constructor with the converted arguments and returns the constructed instance to the caller.

The mechanism used to automate the implementation of such factory classes is the compile-time or run-time reflection.

1.2 Reflection

The term *reflection* refers to the ability of a computer program to examine and possibly modify its own structure and/or behavior. This includes altering the existing or building new data structures, doing changes to algorithms or changing the way the program code is interpreted [3]. Reflective programming is a particular kind of *metaprogramming* [2, 4].

Some of the more common use-cases for reflection are listed below:

- serialization or storing the persistent data in an external binary format or in XML, JSON, XDR, etc.
- deserialization or (re-)construction of class instances from external data representations, from the data stored in a RDBS, from the data entered by an application user through a user interface
- automatic generation of a relational schema from the application object model and object-relational mapping (ORM)
- support for scripting
- remote procedure calls (RPC)/remote method invocation (RMI)
- object inspection and manipulation in an user interface
- online object access through web services (WS)
- visualization of program data structures, data and the relations in the data

- automatic or semi-automatic implementation of certain software design patterns
- support for logging and debugging
- documentation or conceptual representation of a software system.

These and other use cases are examined for example in [5] and realized by projects like [6, 7, 8, 9, 10] and others.

A more detailed introduction to reflection and reflective programming and their usage can be found in other papers by the author [11, 12, 13, 14, 15] and other publications, for example [3, 16].

Support for reflective programming is more common in the high-level languages, usually executed inside of a virtual machine, an interpreter or another such run-time environment, like JAVA, C#, CLOS or Smalltalk [16, 17, 18] and less common or limited in the lower-level statically typed languages like Objective C, Lisp or Scheme [3].

2 RELATED WORK

2.1 Reflection in C++

Presently, C++ as one of the most popular (according to [19, 20]) multi-paradigm programming languages, lacks standard reflection. Some time ago, one of the attempts for standardized reflection for C++ was made in [21], but the work seems to be on hold recently [22]. A more recent attempt to add "rich pointers" and some support for static reflection to C++ [23, 24] looks more promising, but does not address several important aspects like strongly typed enumerations, template parameters, traversal of namespace members, etc. and is focused mainly on run-time reflection. This paper's author is also working on a proposal to add static reflection to standard C++ [15] which was mailed to the standardizing committee recently. Other recent interesting compile-time reflection proposals include [25, 26, 27].

There are several non-standard reflection systems, with varying degree of introspective and reflective capabilities, using various approaches.

The OpenC++ [28] is a programming toolkit that allows writing meta-level source-to-source transformation programs according to a meta-object-protocol [17, 29] at the program preprocessing/parse-time. The meta-level program is compiled by the GCC C++ compiler linked with an OpenC++ add-on. Unfortunately, only older compiler versions are supported which makes the approach non-portable and outdated. Other publications [30, 31] describe a similar source-to-source translator called *FOG*, adding support for aspect-oriented programming and basic compile-time reflection to C++. More common are run-time reflection systems like *Seal* C++ reflection system developed at CERN [22], or those described in [18, 32, 33, 34, 35, 36, 37]. There are also several static reflection facilities with some run-time features, like the one described in [38].

2.2 Reflection in Implementation of Desing Patterns

Other publications also suggest that reflection could be used for automated implementation of the *factory* design pattern, one example is [39], but the method described therein is somewhat limited because it requires that the constructed *product* types have a default constructor which is not applicable to all types.

This technique is also used by several other reflection facilities, with similar limitations. Other examples of design pattern implementation with the help of reflection can be found in [29, 40, 41].

In [37] the implementation of the *Inversion of Control* pattern with the help of reflection is discussed.

2.3 The *Mirror* Reflection Utilities

Mirror consists of a set of C++ libraries and utilities which implement portable and non-intrusive compile-time and run-time introspection and reflection for the C++ language.

These provide reusable metadata reflecting C++ program constructs like namespaces, types, classes, base class inheritance, member variables, class templates and their template parameters, free and member functions, constructors, etc., and implement an extensive set of metaprogramming tools for traversal and usage of these metadata in compile-time functional and run-time object-oriented algorithms.

It also contains several high-level utilities, for example the factory generator utility, described in more detail in this paper. It tries to adhere to the three principles attributed to well-designed reflective facilities [16]: *encapsulation*, *stratification* and *ontological correspondence*. Unlike many other reflection utilities, the Mirror is not tailored for a specific problem and is very versatile.

The library and the related tools are implemented as proof-of-concept for a proposal for standardised reflection for the C++ language [15]. The source code is available for download under the Boost Software License [42] from SourceForge.net at the following URL: `http://sourceforge.net/projects/mirror-lib/files/`.

The latest version [10] which is in active development, is using several new features from the recent ISO standard for the C++ language (C++2011). This version removes several design problems discovered during the development and usage of the previous versions and, in addition, it contains a run-time layer, built on top of the compile-time metaobjects, providing a dynamic, polymorphic object-oriented interface for their usage in run-time reflective algorithms. Recently, both an object-oriented compile-time layer and a type-erasure utility have also been added. This quaternary interface makes the library applicable in a wide range of scenarios.

Mirror has a hierarchic layered architecture where each layer provides services to the layers above:

Registering and basic metaobjects: Because the standard C++ provides only a limited set of metadata to build upon, the basic program constructs like name-

spaces, types, typedefs, classes, variables, functions, constructors, enumerated values, operators, etc. need to be registered before they can be reflected. Mirror tries to make the process of registering simple and convenient by providing a set of user-friendly registering macros and already has the intrinsic types and many of other common types, classes, templates and namespaces pre-registered. This is the lowest layer and is used by the applications only to register their components. This registering should be replaced in future by static meta-data provided by the compiler similar to the now standardized `type_traits` as described for example in [14].

Functional compile-time layer – Mirror: Mirror is a compile-time functional-style reflective programming library, which is based directly on the basic meta-data and suitable for generic programming, similar to the standard `type_traits` library. It allows to write compile-time metaprograms which enable the compiler to generate an efficient program code. Historically it is the first component of the Mirror reflection utilities.

Object-oriented compile-time layer – Puddle: Puddle is the OOP-style compile-time interface with some run-time features built on top of Mirror. It copies the metaobject concept hierarchy of Mirror, but provides a more object-oriented interface. It is still inherently static, allowing for extensive optimization by the compiler. This may seem to be negligable in cases when the reflection code is used together with high latency I/O or GUI operations but proves to be crucial improvement in many cases where too many virtual calls would cause significant overheads.

Object-oriented type-erasure utility – Rubber: Based on the object-oriented compile-time layer; its purpose is to remove the exact types of the instances of the metaobject concepts provided by the lower layers and merge them into a single polymorphic type for each compile-time metaobject concept. This enables the type-erased metaobjects to be stored in standard containers (like vectors, maps, sets, etc.) and used in non-template functions (for example C++ lambda functions). Unlike the Lagoon layer, it does not use virtual functions to achieve polymorphism. This utility is suited for situations where a combination of compile-time and run-time reflection is required.

Object-oriented run-time layer – Lagoon: Based on top of the Mirror and Puddle layers, it provides a run-time polymorphic interface, more suitable for run-time reflective programming, allowing the use of the provided metadata in a dynamic manner dependent on other data available only at run-time. One of its disadvantages is the performance penalty induced by virtual function calls and the inability of the compiler to inline such calls in many cases together with long compilation times. In the future this layer will allow the metainformation to be compiled into shared dynamic libraries separate from the applications and load them on-demand.

Several projects by the author and others (for example [43]) are using Mirror as a foundation for the implementation of their own functionality.

Since the source code of Mirror consists of more than 60 000 lines of code the scope of this paper does not allow detailed explanation and discussion of Mirror's implementation, for this purpose the reference should be consulted [10].

3 IMPLEMENTING THE FACTORY PATTERN

The purpose of the *factory* pattern is to separate its user, who requires a new instance of a *product* type, from the details of this instance's construction. The user just supplies the input data to the factory and waits for the new instance. There are several aspects that need to be considered when designing and implementing the factory.

The input data for the construction of an instance of the *product* may be stored in an external representation (XML fragment, RDBS database dataset, JSON document, etc.) or even entered by the user from a GUI and so on, and would need to be converted to the native C++ representation. The new instance also might be constructed as a copy of another already existing instance of the same type.

The product may be polymorphic and the exact type may not even be known to the user. It may have one or several constructors, each of which may require a different set of parameters. It may or may not have constructors with a specific signature, for example a default constructor.

Default constructor does not make sense for many types and forcing it just because the type will be used with a factory is unsystematic. For example what a "default" instance of `person` or `address` would look like – it would not have any meaning at all. Thus, well designed factories should not depend on the presence of a constructor with a specific signature.

Furthermore, it might be desirable that the constructor to be used for the construction of a particular instance is picked based on the available input data which is known only at run-time, but not when the factory is designed and implemented.

3.1 Design of a Factory

Let us consider a factory for a rather simple `vector` class representing a vector in 3-dimensional space defined as follows:

```
struct vector
{
  double _x, _y, _z;
  vector(double x, double y, double z):
    _x(x), _y(y), _z(z) { }
  vector(double w): _x(w), _y(w), _z(w) { }
  vector(void): _x(0.0), _y(0.0), _z(0.0) { }
  vector(const vector&) = default;
```

```
  // ... other declarations
};
```

A naive hand-coded implementation of a factory constructing `vector`s from some `Data` type (for example XML) might look like this:

```
class vector_factory
{
private:
  unsigned pick_constructor(Data data)
  {
    // somehow examine the data and pick
    // the most suitable constructor of the vector class
  }

  double extract(Data data, string param)
  {
    // somehow extract and convert the value
    // of a named parameter from the data
  }
public:
  vector create(Data data)
  {
    switch(pick_constructor(data))
    {
      case 0: return vector();
      case 1: return vector(extract(data, "w"));
      case 2: return vector(
            extract(data, "x"),
            extract(data, "y"),
            extract(data, "z")
          );
      default: throw exception(...);
    }
  }
};
```

Now, let us suppose that there is some pool of existing `vector` objects and let's extend the factory to use this pool and return copies if applicable:

```
class vector_factory
{
private:
  unsigned pick_constructor(Data data)
  {
    // same as before but also allow the copy
    // constructor to be picked if the data says so
  }
```

```
  double extract(Data data, string param);
public:
  vector create(Data data)
  {
    switch(pick_constructor(data))
    {
      // same as before, but add a new case
      // returning copies from the pool
      case 3: return vector_pool.get(data);
      default: throw exception(...);
    }
  }
};
```

Furthermore, let us add a `triangle` class, defined by three `vector`s and its factory that is using a `vector_factory` to recursivelly construct parameters for the construction of the `vector` parameters required by the constructor:

```
struct triangle
{
  vector _a, _b, _c;
  triangle(vector a, vector b, vector c):
    _a(a), _b(b), _c(c) { }
  triangle(const triangle&) = default;
  double area(void) const;
  // ... other declarations
};

class triangle_factory
{
private:
  unsigned pick_constructor(Data data);
  Data subdata(Data data, string param)
  {
    // somehow get a fragment of data related
    // to the specified parameter
  }

  vector_factory _fact_a, _fact_b, _fact_c;
public:
  triangle create(Data data)
  {
    switch(pick_constructor(data))
    {
      case 0: return triangle(
        _fact_a.create(subdata(data, "a")),
```

```
        _fact_b.create(subdata(data, "b")),
        _fact_c.create(subdata(data, "c"))
      );
      // ... etc.
    }
  }
};
```

And finally, let us add a `tetrahedron` class using a `vector` and a `triangle` and its factory:

```
struct tetrahedron
{
  triangle _base;
  vector _apex;

  tetrahedron(const triangle& base, const vector& apex)
   : _base(base), _apex(apex)
  { }

  tetrahedron(vector a, vector b, vector c, vector d)
   : _base(a, b, c), _apex(d)
  { }

  double volume(void) const;
  // ... other declarations
};

class tetrahedron_factory
{
private:
  unsigned pick_constructor(Data data);
  Data subdata(Data data, string param);

  // factories for the first constructor
  tetrahedron_factory _fact_base;
  vector_factory _fact_apex;

  // factories for the second constructor
  vector_factory _fact_a, _fact_b;
  vector_factory _fact_c, _fact_d;
public:
  tetrahedron create(Data data)
  {
    switch(pick_constructor(data))
    {
      case 0: return tetrahedron(
```

```
        _fact_base.create(subdata(data, "base")),
        _fact_apex.create(subdata(data, "apex"))
      );
      case 1: return tetrahedron(
        _fact_a.create(subdata(data, "a")),
        _fact_b.create(subdata(data, "b")),
        _fact_c.create(subdata(data, "c")),
        _fact_d.create(subdata(data, "d"))
      );
      // ... etc.
    }
  }
};
```

In order to support various input data representations, the functionality related to the manipulation with the data could be factored out of the factories and then used as parameters for them:

```
class xml_data_handler
{
    typedef XMLnode Data;

    template <typename T>
    T extract(Data data, string param);

    Data subdata(Data data, string param);

    template <typename Metadata>
    unsigned pick_constructor(
        Metadata metadata,
        Data data
    );
};

class json_data_handler { ... };
class rdbs_data_handler { ... };
// etc.

template <class DataHandler>
class tetrahedron_factory
{
private:
  DataHander _hander;

  // factories for the first constructor
  tetrahedron_factory _fact_base;
  vector_factory _fact_apex;
```

```
  // factories for the second constructor
  vector_factory _fact_a, _fact_b;
  vector_factory _fact_c, _fact_d;
public:
  tetrahedron create(DataHandler::Data data)
  {
    Metadata metadata = /* get metadata */;
    switch(_handler.pick_constructor(
      metadata,
      data
    ))
    {
      case 0: return tetrahedron(
        _fact_base.create(_handler.subdata(data, "base")),
        _fact_apex.create(_handler.subdata(data, "apex"))
      );
      case 1: return tetrahedron(
        _fact_a.create(_handler.subdata(data, "a")),
        _fact_b.create(_handler.subdata(data, "b")),
        _fact_c.create(_handler.subdata(data, "c")),
        _fact_d.create(_handler.subdata(data, "d"))
      );
      // ... etc.
    }
  }
};
```

When looking at the hand-coded factories above, it is obvious that implementing and maintaining (as the constructed types evolve and change) factories for several dozens of classes in a larger application is a highly repetitive, tedious and possibly error-prone process and at least partial automation is desirable.

3.2 Factory Dissection

Factory classes must generally handle several tasks which fit into two distinct and nearly orthogonal categories:

Product type-related

- **Constructor description** – providing the metadata describing the individual constructors, their parameters, etc.
- **Constructor dispatching** – calling the selected constructor with the supplied arguments which results in a new instance of the product type.

Input data representation-related

Input data validation – checking if the input data match the available constructors.

Constructor selection – examining the input data, comparing it to the metadata describing product's constructors and determining which constructor should be called.

Getting the argument values – determining where the argument values should come from and getting them:

Conversion from the external representation – this usually applies to intrinsic C++ types, but complex types could be converted directly, too.

Recursive construction by using another factory – this usually requires some form of cooperation between the parent and its child factories and it means that all the tasks discussed here must be repeated also for the recursivelly constructed parameter(s).

Copying an existing instance – for example from an object pool.

Parts from each category can be combined with parts from the other to create new factories what promotes code reusability. Factories constructing instances of a single product from various data representations share the product-related components and factories constructing instances of various product types from a single input data representation share the input-data-related parts. This approach has several advantages like better maintainability or the ability to develop the components separatelly and combine them later.

3.3 Initialization and Cleanup

An important aspect of factory implementation is that factory instances need to be properly initialized before they are used and cleaned-up when they are no longer needed. For factories using GUI to get the input data this involves programmatically building the GUI from appropriate components used to select the constructors, data input and validation.

Moreover, repeating this procedure for every constructed instance of product would be a significant waste of resources – the GUI should be set-up once, reused during the lifetime of the factory and properly cleaned-up afterwards. Factories doing the construction from other input data representation may require similar initialization and cleanup procedures.

3.4 Automating Factory Class Implementation

As noted above, several parts of every factory are related to its product type and the implementation of these parts follows distinct patterns that can be algorithmized. The input data for the algorithm is the metadata describing the product

type (specifically its constructors) and the result of the algorithm is the program code of the said parts. This is where reflection comes as the source of the metadata and metaprogramming and generic programming as the tools for implementing the metaalgorithm that is executed by a C++ compiler.

4 MIRROR'S FACTORY GENERATORS

Mirror implements two distinct frameworks for generating such factories; one working at compile-time and the other dynamically at run-time. The former's advantage is that the compiler has access to the whole code and can do extensive optimizations like function inlining or removing unused code. The disadvantage is that the types for which the factories are generated must be known at compile-time. The latter allows to construct factories for types described by run-time reflection which can be loaded dynamically. The cost is some performance overhead and possibly an increased code size.

4.1 Compile-Time Factory Generator

4.1.1 Input Data-Related Parts

The compile-time factory generator defines four concepts for the input data representation-related parts of the factory:

Manufacturer – defines the interface for a component of the factory responsible for the conversion of input data from an external representation to the native C++ data types or recursive construction by the means of another factory. More precisely models of this concept are templates that are parametrized by the type they are returning. The user-defined manufacturers usually provide several specializations for types that are considered atomic (Booleans, numbers, strings and possibly others) and the default implementation of the template makes the recursive construction. Manufacturers basically implement the same functionality as the `extract` and `subdata` functions in the factories in the examples above.

Enumerator – defines the interface for a component of the factory that handles the input of values of C++ enumerated types. Enumerator is similar in function to *Manufacturer*, but enumerations are specific and therefore handled differently.

Manager – interface for components responsible for choosing the constructor to be used in the construction of a particular instance. Manager does the job of the `pick_constructor` function from the examples above. A specialization for the `void` type of a concrete *Manufacturer* is used as a manager. Using this specialization instead of a separate class is advantageous, because we can use one less template parameter in the factory generator template, and the `void` type cannot be instantiated anyway.

Suppliers – optionally, if there is a pool of existing objects that can be used to create copies of product instances, the application can use models of the *Suppliers* concept to select and return an existing instance, like the `vector_pool` object in the examples above. There could even be several sources of existing instances – hence the plural name.

An application wishing to use a generated factory must implement types modelling these concepts (called factory generator plug-ins), but Mirror already provides several reusable implementations:

- A `wxWidgets`-based generator plug-in which allows the user to select the constructor and input the necessary data via automatically generated GUI.
- A `libpq`-based generator plug-in allowing to construct instances from a dataset obtained through a SQL query from a PostgreSQL database.
- A `SOCI`-based generator plug-in similar to the previous one but allowing to use other RDB systems.
- A generator plug-in using JSON (JavaScript Object Notation) parser allowing to construct instance from JSON documents (obtained for example from a RESTful web service).
- A `wxXML` and `rapidxml`-based plug-ins allowing to generate factories constructing instances from XML documents.
- A `Boost.Filesystem`-based generator plug-in for construction of objects from data stored in files in a directory hierarchy.
- A `Boost.Spirit`-based plug-in using a parser to get the input data from a simple C++-like scripting language.
- A plug-in that serves as a framework for the implementation of run-time factory generators described in Subsection 4.2.

The factory generator is a template which takes the product type and concrete templates modeling the manufacturer, enumerator, manager and suppliers concepts and "returns" the desired factory type, by using metadata describing the product obtained by reflection. The concrete instantiations of the manufacturer, enumerator and suppliers templates are generated as private member variables of the returned factory.

4.1.2 Design and Implementation of the Generator

The following pseudo-code shows roughly how the generator works, however, note that the actual implementation is more complex as explained further below.

The `pick_source` is a helper class that decides whether *Manufacturer*, *Enumerator* or *Suppliers* should be used as the source of a parameter value:

```
template <
  template <class> class Manufacturer,
```

```
  template <class> class Enumerator,
  template <class> class Suppliers,
  typename Product,
  // additional metadata used to decide
  // which of the above will be used
  // as the source of a parameter when
  // calling a constructor
> struct pick_source
{
  // defined based on the additional metadata
  typedef
    // either as ...
    Manufacturer<Product>
    // ... or as ...
    Enumerator<Product>
    // ... or as
    Suppliers<Product>
  type;
};
```

The `constructor` is a helper class encapsulating a single constructor of product and the parameter sources:

```
template <
  template <class> class Manufacturer,
  template <class> class Enumerator,
  template <class> class Suppliers,
  typename Product,
  // Additional metadata describing a single
  // constructor and its parameters
> struct constructor
{
private:
  std::tuple<
    pick_source<
      Manufacturer,
      Enumerator,
      Suppliers,
      Product,
      // the additional metadata
    >::type ...
  > sources;
public:
  Product operator()(void)
  {
    // use the product's constructor and use
    // the individual sources from the tuple
```

```
    // to get the required parameter values
    return Product(std::get<...>(sources)() ...);
  }
};
```

Now the factory uses a *Manager* to pick the constructor and a set of `constructors` one of which is (based on the picked index) used to construct a new instance:

```
template <
  template <class> class Manufacturer,
  template <class> class Enumerator,
  template <class> class Suppliers,
  typename Product,
  // Additional metadata describing the product
> struct factory
{
private:
  Manufacturer<void> manager;

  std::tuple<
    constructor<
      Manufacturer,
      Enumerator,
      Suppliers,
      Product,
      // the additional metadata
    >::type ...
  > constructors;

  Product dispatch_constructor(unsigned index)
  {
    // something equivalent to:
    switch(index)
    {
      case 0: return std::get<0>(constructors)();
      case 1: return std::get<1>(constructors)();
      case 2: return std::get<2>(constructors)();
      // etc.
    }
    // something failed
    throw exception(...);
  }
public:

  factory(InputData input_data)
   : manager(input_data)
```

```
   , constructors(input_data...)
  {
    // register all constructors
    // with the manager, do this
    // for all constructors
    // in the tuple
    manager.add_constructor(std::get<...>(constructors)...);
  }

  Product operator()(void)
  {
    // use the manager to pick the index
    // and then call the selected constructor
    return dispatch_constructor(manager.pick_index());
  }
};
```

4.1.3 Factory Generator Plug-Ins

As stated before, the input data-related parts of factories are implemented by the factory generator plug-ins. The *Manufacturer* template has the following interface:

```
template <typename Product>
class some_manufacturer
{
public:
        template <class InputData, class ConstructionInfo>
        some_manufacturer(
                InputData& input_data,
                ConstructionInfo ctr_info
        );

        template <class InputData>
        void finish(InputData& input_data);

        Product operator()(void);
};
```

The constructor takes two parameters: A representation of the input data which depends on the external representation. For RDBS plug-ins, the input data may be an open dataset; for GUI-generating plug-ins this may be the reference to the parent GUI component; for scripting language plug-ins this may be the reference to the script parser, etc.

In RDBS plug-ins the constructor is responsible for finding the right dataset column from which the input values will be read, but the constructor can also make changes to the input data; in the case of GUI plug-in the manufacturer can create

a new GUI component for the input of data necessary for the construction of a value of `Product` type, in case of a scripting plug-in the manufacturer can add new parsing rules to the parser.

The `ConstructionInfo` parameter provides contextual information about the construction. For example in a factory constructing instances of `tetrahedron` as described above, the manufacturers constructing the values of the individual vector coordinate may need to know that the `double` value is the coordinate `x`, of the vertex `a`, of the triangle `apex` of the tetrahedron. Such information is necessary for example in XML or JSON-based plug-ins which need to search a document tree hierarchy using the names of the input parameters.

The `finish` function is called after all other manufacturers in the factory are constructed and allows to do late-initialization, for example if access to sibling or child manufacturers is required.

The function call operator is responsible for the actual construction of instances of `Product` from the input data provided in the manufacturer's constructor. For RDBS plug-ins this involves calling the appropriate funtion of the database access library and converting the result into the `Product` type (in the `libpq` plug-in the `::PQgetvalue` function is used). A manufacturer in GUI plug-in reads, validates and converts the value entered by a user into a visual component, etc.

The *Enumerator* has the same interface and a very similar function as *Manufacturer*, but it converts (usually a textual) representation of the enumeration value name into the actual C++ value. GUI enumerators usually create visual components (like Pickboxes) that allow the user to choose from a list of options. In other plug-ins the enumerators usually store a list of names of valid enumeration values and match them to the actual input data.

The *Suppliers* component has also the same interface as the *Manufacturer*, but instead of creating new instances from external representations it can access a pool of existing instances and return them in the `Product operator ()(void)` operator.

The *Manager* is responsible for managing and choosing the right constructor and has the following interface:

```
template <>
class some_manufacturer<void>
{
public:
        template <class InputData, class Context>
        inline some_manufacturer(
                InputData& input_data,
                Context context
        );

        template <class InputData>
        void finish(InputData& input_data);

        template <class InputData, class ConstructorInfo>
```

```
InputData& add_constructor(
        const InputData& input_data,
        ConstructorInfo ctr_info
);

int index(void);
```

The constructor and the `finish` function have responsibilities very similar to those in *Manufacturer*. They take some representation of the input data and contextual information.

The `add_constructor` function is called by the factory generator once per each constructor in the `Product` type. The task of this function is to register a new constructor. It can do so by scanning both the input data and constructor metadata, provided by the `ConstructorInfo` parameter. The metadata includes information about the parameters of the constructor and some additional contextual information. In GUI plug-ins this function can for example add a new container component in which the child components generated by the nested *Manufacturer*s will be contained. In RDBS plug-ins this function may compile a list of dataset columns that need to be available and contain valid values in order the particular constructor is usable.

The `index` member function is called by the factory generator, when a new instance of `Product` is constructed and a constructor needs to be selected. In GUI plug-ins the index function can for example determine which component has focus (for example which page on a page-control component was selected by the user) and return the index of the matching constructor. In RDBS plug-ins the manager scans the current row of the open dataset and matches the names and types of the available columns to the metadata describing the constructors and return the constructor with the most parameters for which all values are available. In XML or JSON plug-in a document fragment hierarchy elements can be scanned, and again, the constructor with the most parameters that can be satisfied is selected.

4.1.4 Further Details

In order to be truly versatile, the `factory` generator must supply additional parameters to the concrete *Manufacturers*, *Enumerators*, *Suppliers* and *Managers*.

In contrast to the pseudo-code above, in Mirror these concepts take an additional template parameter called `Traits` which is specific to a concrete implementation of the plug-ins and may change their behaviour.

Also, in some situations, for example when generating GUI as a part of the `factory` some contextual data may be required, such as a `vector` is constructed not as the factory result but as a parameter for the construction of a `triangle` which is used as a parameter in the construction of a `tetrahedron`. This information can help the factory to adjust the GUI accordingly.

Mirror's implementation takes care of these details, for the full source code please refer to the Mirror's source repository and/or documentation [10]. Detailed

description of the implementation of one of the Mirror's factory generator plug-ins will be the topic of another paper.

4.1.5 Usage

The first example shows what the library user needs to do to re-construct a class from a relational database using the `libpq`-based factory. To work properly, it expects that a table with the following definition is present in a database called "test" and contains some data.

```
CREATE TABLE person (
  first_name TEXT NOT NULL,
  middle_name TEXT.
  family_name TEXT NOT NULL,
  birth_date DATE,
  sex VARCHAR(6),
  weight FLOAT,
  height FLOAT
);
```

We want to re-construct instances of a user-defined `person` class from the data stored in a PostgreSQL database. The `person` class and the `gender` enumeration used by it have the following definition:

```
#include <string>
#include <ctime>

namespace test {

enum class gender
{
        female,
        male
};

struct person
{
        std::string first_name;
        std::string middle_name;
        std::string family_name;

        std::tm birth_date;
        gender sex;

        float weight;
        float height;
```

```
        person(
                const std::string& _first_name,
                const std::string& _family_name,
                const std::tm _birth_date,
                gender _sex,
                float _weight,
                float _height
        ): first_name(_first_name)
         , family_name(_family_name)
         , birth_date(_birth_date)
         , sex(_sex)
         , weight(_weight)
         , height(_height)
        { }

        person(
                const std::string& _first_name,
                const std::string& _middle_name,
                const std::string& _family_name,
                const std::tm _birth_date,
                gender _sex,
                float _weight,
                float _height
        ): first_name(_first_name)
         , middle_name(_middle_name)
         , family_name(_family_name)
         , birth_date(_birth_date)
         , sex(_sex)
         , weight(_weight)
         , height(_height)
        { }
};

} // namespace test
```

The enclosing `test` namespace, the enumeration and the class with its member variables and constructors need to be registered with Mirror in the following way (the native C++ types are already pre-registered):

```
#include <mirror/mirror.hpp>
#include <mirror/meta_class.hpp>
#include <mirror/meta_enum.hpp>
#include <mirror/pre_registered/type/native.hpp>

MIRROR_REG_BEGIN

MIRROR_QREG_GLOBAL_SCOPE_NAMESPACE(test)
```

```
MIRROR_REG_ENUM_BEGIN(test, gender)
  MIRROR_REG_ENUM_VALUE(female)
  MIRROR_REG_ENUM_VALUE(male)
MIRROR_REG_ENUM_END

MIRROR_REG_CLASS_BEGIN(struct, test, person)
 MIRROR_REG_CLASS_MEM_VARS_BEGIN
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, first_name)
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, middle_name)
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, family_name)
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, birth_date)
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, sex)
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, weight)
  MIRROR_REG_CLASS_MEM_VAR(_, _, _, height)
 MIRROR_REG_CLASS_MEM_VARS_END
 MIRROR_REG_CONSTRUCTORS_BEGIN
  MIRROR_REG_COPY_CONSTRUCTOR(public)

  MIRROR_REG_CONSTRUCTOR_BEGIN(public, 1)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, first_name)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, family_name)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, birth_date)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, sex)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, weight)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, height)
  MIRROR_REG_CONSTRUCTOR_END(1)

  MIRROR_REG_CONSTRUCTOR_BEGIN(public, 2)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, first_name)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, middle_name)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, family_name)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, birth_date)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, sex)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, weight)
   MIRROR_REG_CONSTRUCTOR_PARAM(_, height)
  MIRROR_REG_CONSTRUCTOR_END(2)
 MIRROR_REG_CONSTRUCTORS_END
MIRROR_REG_CLASS_END

MIRROR_REG_END
```

The registering macros are verbose, but Mirror can auto-detect many propreties of the registered member variables, constructors, functions, etc. (like the type, constness, linkage or access specifiers) or reasonable defaults can be used. The _ arguments in the code above indicate that Mirror should detect the values.

Once the user-defined types are registered, the Mirror's `libpq`-based factory generator can be used out of the box, by including the appropriate header file:

```
#include <mirror/utils/libpq_factory.hpp>
```

The usage is quite straightforward. The object-relational mapping is created automatically from the meta-data describing the `person` (or any other reflected class), when a new instantiation of the factory generator for a particular `Product` type is created, by means of the `libpq_factory_maker` class.

```
#include <mirror/stream.hpp>
#include <iostream>
#include <stdexcept>

int main(void)
{
  try
  {
  using namespace mirror;
  // connect to the database
  libpq_fact_db test_db("dbname=test");
  // query data about persons
  libpq_fact_result persons =
    test_db.query("SELECT * FROM person");
  // make a person object factory
  libpq_factory_maker::factory<test::person>::type
    person_factory(persons.data());
  // go through the rows of the data-set
  while(!persons.empty())
  {
    std::cout <<
    // make a person from the data
    // and write it to std output
    // using a JSON formatter
    stream::to_json::from(
      person_factory(),
      [&persons](std::ostream& out)
      {out << "person_" << persons.position();}
    ) << std::endl;
    persons.next();
  }
  }
  catch(const std::exception& error)
  {
    std::cerr << error.what() << std::endl;
  }
  return 0;
}
```

For the full source code of the example above and for other examples refer to [10]. Figure 1 shows a screenshot of an input dialog window generated by the `wxWidgets` factory generator plug-in, for a `tetrahedron` class which was described above. The dialog contains a hierarchy of visual components for the selection of constructors to be used (each constructor is contained on a separate `wxChoicebook` panel) and for the input of parameter values. The input fields are validated using `wxValidator`s which prevent the user from entering invalid values.

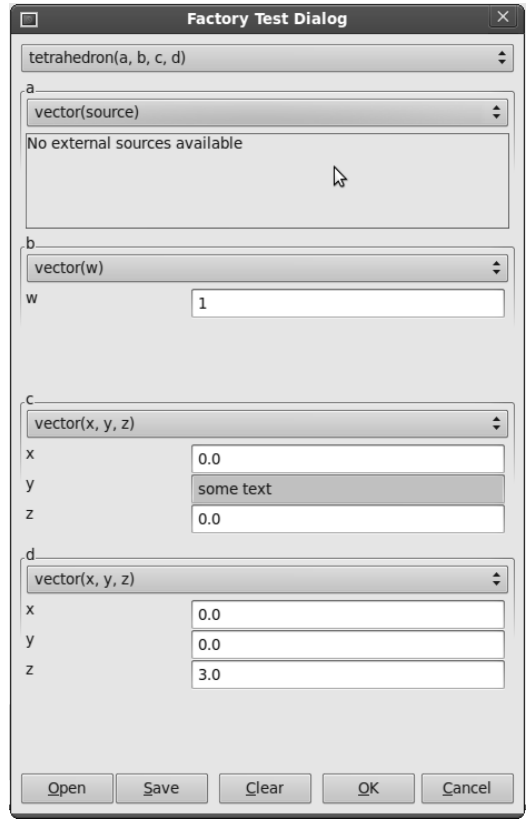

Figure 1. Example of a graphic user interface dialog constructed by an automatically generated factory

4.2 Run-Time Factory Generator

As already mentioned above, the drawback of the compile-time factory generator is that it requires the metadata reflecting the product type at compile-time. There are situations when it is desirable to generate a factory for a type specified only at

run-time (for example based on user preference). The polymorphic factory generator implemented by the *Lagoon* layer of Mirror aims to provide a solution.

4.2.1 Interfaces

Instead of compile-time concepts, Lagoon defines a set of abstract interfaces (structs with virtual functions), several of which are used to represent parts of a polymorphic factory:

First there is the `polymorph_factory_context` interface that stores information about the context in which a factory component is used. For the exact definition see [10].

```
struct polymorph_factory_context { ... };
```

Then, there is the `polymorph_factory_manager` interface for objects having the same role as the *Managers* in the compile-time factories; it is responsible for selecting a constructor. Note, that the manager uses a `raw_ptr` type to pass on data. `raw_ptr` is a type erasure for pointers – basically a more safe `void*`.

```
struct polymorph_factory_manager
{
  virtual raw_ptr data(void) { return raw_ptr(); }

  virtual raw_ptr add_constructor(
    raw_ptr data,
    const shared<meta_constructor>& constructor,
    const polymorph_factory_context& context,
    bool backward_iteration
  ) { return raw_ptr(); }

  virtual void finish(raw_ptr data) { }

  virtual int index(void) = 0;
};
```

The `polymorph_factory_composite` is an interface for manufacturers of complex types – i.e. types that recursivelly use other factories to supply parameters for their constructors.

```
struct polymorph_factory_composite
{
  virtual raw_ptr data(void) { return raw_ptr(); }
  virtual void finish(raw_ptr data) { }
  virtual void on_create(void) { }
};
```

Next, there is the `polymorph_factory_arrayer` interface for components creating a sequence of elements used in container type construction.

```
struct polymorph_factory_arrayer
{
  virtual raw_ptr data(void) { return raw_ptr(); }
  virtual void finish(raw_ptr data) { }

  struct element_producer
  {
    virtual ~element_producer(void) { }
    virtual void reset(void) = 0;
    virtual void make_next(void) = 0;
  };

  virtual void assign_producer(element_producer& producer)
    { }
  virtual void create(element_producer& producer) = 0;
};
```

Then follow the interfaces for manufacturers, enumerators and suppliers.

```
template <typename Product>
struct polymorph_factory_manufacturer
{
    virtual void finish(raw_ptr parent_data) { }
    virtual Product create(void) = 0;
};

struct polymorph_factory_enumerator
{
    virtual void finish(raw_ptr parent_data) { }
    virtual int create(void) = 0;
};


struct polymorph_factory_suppliers
{
    virtual void finish(raw_ptr parent_data) { }
    virtual raw_ptr get(void) = 0;
};
```

Finally the `polymorph_factory_builder` is an interface for a builder class that on-demand creates the individual factory components as required for a specific product type. The process is described in greater detail below:

```
struct polymorph_factory_builder
{
  // Defined for each native Product type
  virtual std::shared_ptr<
    polymorph_factory_manufacturer<Product>
```

```
> make_manufacturer(
  raw_ptr parent_data,
  const shared<meta_parameter>& ctr_param,
  const polymorph_factory_context& context
) = 0;

virtual std::shared_ptr<
  polymorph_factory_manager
> make_manager(
  raw_ptr parent_data,
  const polymorph_factory_context& context
) = 0;

virtual std::shared_ptr<
  polymorph_factory_composite
> make_composite(
  raw_ptr parent_data,
  const shared<meta_parameter>& func_param,
  const polymorph_factory_context& context
) = 0;

virtual std::shared_ptr<
  polymorph_factory_composite
> make_composite(
  raw_ptr parent_data,
  const shared<meta_type>& product,
  const polymorph_factory_context& context
) = 0;

virtual std::shared_ptr<
  polymorph_factory_arrayer
> make_arrayer(
  raw_ptr parent_data,
  const shared<meta_type>& element,
  const polymorph_factory_context& context
) = 0;

virtual std::shared_ptr<
  polymorph_factory_suppliers
> make_suppliers(
  raw_ptr parent_data,
  const shared<meta_parameter>& ctr_param,
  const polymorph_factory_context& context
) = 0;

virtual std::shared_ptr<
```

```
    polymorph_factory_enumerator
  > make_enumerator(
    raw_ptr parent_data,
    const shared<meta_parameter>& ctr_param,
    const polymorph_factory_context& context
  ) = 0;
};
```

These serve to build polymorphic factories which are represented by the `polymorph_factory` interface:

```
struct polymorph_factory
{
  virtual raw_ptr new_(void) = 0;
};
```

The `meta_type` interface among (many) other things declares the `make_factory` function, that uses a builder to dynamically create a factory specific for the type reflected by that `meta_type`:

```
struct meta_type
{
  // ...

  virtual std::unique_ptr<
    polymorph_factory
  > make_factory(
    polymorph_factory_builder& builder,
    raw_ptr build_data
  ) = 0;

  // ...
};
```

The implementations of this function (in concrete `meta_type`s reflecting concrete types) basically use the compile-time generator to create a factory that has special manufacturers, enumerators, suppliers and managers that use the polymorphic components created by the factory builder that was passed as argument to `make_factory`. In this way the concrete metatypes and concrete implementations of factory builders can be combined at run-time.

4.2.2 Usage

The following example shows the usage of a polymorphic dynamically generated factory to create a `std::vector` of values of user-defined type, from a text in a simple scripting language:

```
#include <mirror/mirror_base.hpp>
#include <mirror/pre_registered/basic.hpp>
```

```
#include <mirror/pre_registered/class/std/vector.hpp>
#include <mirror/pre_registered/class/std/map.hpp>
#include <mirror/utils/quick_reg.hpp>
#include <lagoon/lagoon.hpp>
#include <lagoon/utils/script_factory.hpp>
#include <iostream>

namespace morse {

enum class signal : char { dash = '-', dot = '.' };
typedef std::vector<signal> sequence;
typedef std::map<char, sequence> code;

} // namespace morse

// register the user defined types
MIRROR_REG_BEGIN

MIRROR_QREG_GLOBAL_SCOPE_NAMESPACE(morse)
MIRROR_QREG_ENUM(morse, signal, (dash)(dot))

MIRROR_REG_END

int main(void)
{
  try
  {
  using namespace lagoon;

  // use a builder implemented by Lagoon
  c_str_script_factory_builder builder;
  // and its input data
  c_str_script_factory_input in;
  auto data = in.data();

  // reflect the morse::code type
  auto meta_morse_code =
    reflected_class<morse::code>();

  // use the metatype and the builder
  // to make a new factory for instances
  // of morse::code
  auto morse_code_factory =
    meta_morse_code->make_factory(
    builder,
    raw_ptr(&data)
```

```
  );

// the input text
const char input[] = "{ \
  {'A', {dot, dash}}, \
  {'B', {dash, dot, dot, dot}}, \
  {'C', {dash, dot, dash, dot}}, \
  {'D', {dash, dot, dot}}, \
  {'E', {dot}}, \
  // ... rest ommited for sake of brevity
  {'Z', {dash, dash, dot, dot}}, \
  {'1', {dot, dash, dash, dash, dash}}, \
  {'2', {dot, dot, dash, dash, dash}}, \
  // ... rest ommited for sake of brevity
  {'9', {dash, dash, dash, dash, dot}}, \
  {'0', {dash, dash, dash, dash, dash}} \
}";
// set the input data
in.set(input, input+sizeof(input));

// use the factory to create a new instance
raw_ptr pmc = morse_code_factory->new_();

// cast the raw pointer to the right type
morse::code& mc = *raw_cast<morse::code*>(pmc);

// go through the values in the container
for(auto i=mc.begin(), e=mc.end(); i!=e; ++i)
{
  // print the character and its morse code
  std::cout
  << "Morse code for '"
  << i->first
  << "': ";
  auto j = i->second.begin(),
     f = i->second.end();
  while(j != f)
  {
  std::cout << char(*j);
  ++j;
  }
  std::cout << std::endl;
}

// use the metatype to delete the instance
meta_morse_code->delete_(pmc);
```

```
  }
  catch(std::exception const& error)
  {
  std::cerr << error.what() << std::endl;
  }
  return 0;
}
```

This application prints the following to the standard output (abbreviated here):

```
Morse code for '0': -----
Morse code for '1': .----
Morse code for '2': ..---
...
Morse code for '9': ----.
Morse code for 'A': .-
Morse code for 'B': -...
Morse code for 'C': -.-.
...
Morse code for 'Z': --..
```

5 CONCLUSION AND FURTHER WORK

In this paper we described a method for semi-automated implementation of the factory design pattern in C++ by using the Mirror reflection utilities.

The Mirror utilities are still in development and serve as a proof-of-concept implementation for a proposal to add support for reflection to the C++ standard.

The factories generated by the utility provided by Mirror can be used by applications directly or can be composed into more complex components. The same approach used to invoke constructors by the factories can be used to call arbitrary functions. The (still experimental) invoker-generator in Mirror can be used to generate such invoker classes.

Another planned utility – the manipulator generator will use a similar approach to generate classes which can manipulate existing instances of arbitrary reflectible types. Such manipulators could be used to serialize existing instances to XML, JSON, etc. or to show them in GUI and allow the user to manipulate their values.

REFERENCES

[1] Gamma, E.—Helm, R.—Johnson, R.—Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Proffesional, 1995. ISBN 0-201-63361-2.

[2] Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley Professional, 2001.

[3] Kirby, G. N. C.: Reflection and Hyper-Programming in Persistent Programming Systems. Ph.D. Thesis, University of St Andrews, 1992.

[4] Abrahams, D.—Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley Professional, 2004.

[5] Spertus, M.: Use Cases for Compile-Time Reflection. Proposal for ISO/IEC JTC1 SC22 WG21, N3403=120093, 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3403.pdf.

[6] A C++ Reflection-Based Data Dictionary. http://sourceforge.net/projects/crd/.

[7] Property Set Library (PSL). http://sourceforge.net/projects/psl/.

[8] QxOrm Library. http://sourceforge.net/projects/qxorm/.

[9] Template Reflection Library. http://sourceforge.net/projects/trl/.

[10] Chochlík, M.: Mirror C++ Reflection Library Documentation. http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/.

[11] Chochlík, M.: Support for Object-Oriented Parallel Programs for Grids and Clusters. Proceedings of 2nd International Workshop on Grid Computing for Complex Problems, Bratislava, Slovakia, 2006, pp. 88–96.

[12] Chochlík, M.: Generating Object Factory Classes with the Mirror Reflection Library. Journal of Information, Control and Management System, Vol. 8, 2010, No. 2. ISSN 1336-1716.

[13] Chochlík, M.: Portable Reflection for C++ with Mirror. Journal of Information and Organizational Sciences, Vol. 35, No. 1. ISSN 1846-3312.

[14] Chochlík, M.: Static Reflection. Draft of Proposal for ISO/IEC JTC1 SC22 WG21, C++ Programming Language, Library Working Group, 2012. http://kifri.fri.uniza.sk/~chochlik/jtc1_sc22_wg21/std_cpp_refl.pdf.

[15] Chochlík, M.: Static Reflection. Proposal N3996, Programming Language C++, SG7, Reflection, 2014. https://isocpp.org/files/papers/n3996.pdf.

[16] Bracha, G.—Ungar, D.: Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 2003, pp. 85–104.

[17] Chiba, S.: A Metaobject Protocol for C++. Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 1995.

[18] Chuang, T.-R.—Kuo, Y. S.—Wang, C.-M.: Non-Intrusive Object Introspection in C++. Software – Practice and Experience, Vol. 32, 2002, pp. 191–207.

[19] Ohloh.net: Compare Languages Tool. http://www.ohloh.net/languages/compare.

[20] Programming Language Popularity. http://langpop.com/.

[21] Stroustrup, B.: XTI An Extended Type Information Library. http://lcgapp.cern.ch/project/architecture/XTI_accu.pdf.

[22] Roiser, S.—Mato, P.: The Seal C++ Reflection System. Computing in High Energy and Nuclear Physics (CHEP), September 2004.

[23] Berris, M. D.—Austern, M.—Crowl, L.: Rich Pointers. Proposal for ISO/IEC JTC1 SC22 WG21, N3340=120030, 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3340.pdf.

[24] Berris, M. D.—Austern, M.—Crowl, L.—Singh, L.: Rich Pointers with Dynamic and Static Introspection. Proposal for ISO/IEC JTC1 SC22 WG21, N3410=120100, 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3410.pdf.

[25] Santoia Silva, C.—Auresco, D.: Adding Attribute Reflection to C++. Proposal N3984, Programming Language C++, SG7, Reflection, 2014. https://isocpp.org/files/papers/n3984.pdf.

[26] Santoia Silva, C.—Auresco, D.: Yet Another Set of C++ Type Traits. Proposal N3987, Programming Language C++, SG7, Reflection, 2014. https://isocpp.org/files/papers/n3987.pdf.

[27] Tomazos, A.—Kaeser, C.: Type Member Property Queries (Rev 2). Proposal N4027, Programming Language C++, SG7, Reflection, 2014. https://isocpp.org/files/papers/n4027.pdf.

[28] The OpenC++ Project. http://opencxx.sourceforge.net.

[29] Tanter, E.—Noy, J.—Caromel, É. D.—Cointe, P.: Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '03), Anaheim, California, USA, 2003.

[30] Willink, E. D.—Muchnick, V. B.: Weaving a Way Past the C++ One Definition Rule. Proceedings of European Conference on Object Oriented Programming, Lisbon, June 14, 1999.

[31] Willink, E. D.: Preprocessing C++: Meta-Class Aspects. Proceedings of the Eastern European Conference on the Technology of Object Oriented Languages and Systems (TOOLS EE 99), Blagoevgrad, Bulgaria, June 1999.

[32] Devadithya, T.—Chiu, K.—Lu, W.: C++ Reflection for High Performance Problem Solving Environments. Proceedings of the 2007 Spring Simulation Multiconference, Norfolk, Virginia, USA, 2007, Vol. 2, pp. 435–440.

[33] Madany, P. W.—Islam, N.—Kougiouris, P.—Campbell, R. H.: Reification and Reflection in C++: An Operating Systems Perspective. 1992.

[34] Vollman, D.: Metaclasses and Reflection in C++. http://www.vollmann.ch/pubs/meta/meta.html, 2000.

[35] Madina, D.—Standish, R. K.: A System for Reflection in C++. Proceedings AUUG 2004: Always on and Everywhere, 2004.

[36] Knizhnik, K.: Reflection for C++. http://www.garret.ru/cppreflection/docs/reflect.html.

[37] de Bayser, M.—Cerqueira, R.: A System for Runtime Type Introspection in C++. 2012.

[38] Watte, J.: Static Reflection in C++ Using Minimal Repetition. http://www.enchantedage.com/cpp-reflection.

[39] Kovacs, R.: Creating Dynamic Factories in .NET Using Reflection. http://msdn.microsoft.com/en-us/magazine/cc164170.aspx.

[40] Tatsubori, M.—Chiba, S.: Programming Support of Design Patterns with Compile-Time Reflection. Proceedings OOPSLA '98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, 1998.

[41] The Visitor Pattern. `http://www.oodesign.com/visitor-pattern.html`.

[42] Boost Software License. Version 1.0, `http://www.boost.org/LICENSE_1_0.txt`.

[43] The Firestarter Project. `https://github.com/teotwaki/firestarter`.

Matúš Chochlík earned his Ph.D. degree at the Faculty of Management Science and Informatics at the University of Žilina, where he currently works as Assistant Professor in the Department of Informatics. His research is centered on implementation of compile-time reflection for the C++ programming language, meta-programming and generic programming techniques. He is also interested in hardware accelerated computer graphics, GPU programming and graphics algorithms.