Reflection

The *Reflection* architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

**Also Known As**  Open Implementation, Meta-Level Architecture

**Example**  Consider a C++ application that needs to write objects to disk and read them in again. Since persistence is not a built-in feature of C++, we must specify how to store and read every type in the application. Many solutions to this problem, such as implementing type-specific store and read methods, are expensive and error-prone. For example, whenever we change the class structure of the application, we must modify these methods as well.

Other solutions to the lack of persistence raise other problems. For example, we could provide a special base class for persistent objects from which application classes are derived, with inherited store and read methods overridden. Changes to the class structure require us to modify these methods within existing application classes. Persistence and application functionality are strongly interwoven.
Instead we want to develop a persistence component that is independent of specific type structures. However, to store and read arbitrary C++ objects, we need dynamic access to their internal structure.

**Context**  
Building systems that support their own modification a priori.

**Problem**  
Software systems evolve over time. They must be open to modifications in response to changing technology and requirements. Designing a system that meets a wide range of different requirements a priori can be an overwhelming task. A better solution is to specify an architecture that is open to modification and extension. The resulting system can then be adapted to changing requirements on demand. In other words, we want to design for change and evolution. Several forces are associated with this problem:

- Changing software is tedious, error prone, and often expensive. Wide-ranging modifications usually spread over many components and even local changes within one component can affect other parts of the system. Every change must be implemented and tested carefully. Software which actively supports and controls its own modification can be changed more effectively and more safely.
- Adaptable software systems usually have a complex inner structure. Aspects that are subject to change are encapsulated within separate components. The implementation of application services is spread over many small components with different interrelationships [GHJV95]. To keep such systems maintainable, we prefer to hide this complexity from maintainers of the system.
- The more techniques that are necessary for keeping a system changeable, such as parameterization, subclassing, mix-ins, or even copy and paste, the more awkward and complex its modification becomes. A uniform mechanism that applies to all kinds of changes is easier to use and understand.
- Changes can be of any scale, from providing shortcuts for commonly-used commands to adapting an application framework for a specific customer.
- Even fundamental aspects of software systems can change, for example the communication mechanisms between components.
Reflection

Solution

Make the software self-aware, and make selected aspects of its structure and behavior accessible for adaptation and change. This leads to an architecture that is split into two major parts: a meta level and a base level.

The meta level provides a self-representation of the software to give it knowledge of its own structure and behavior, and consists of so-called metaobjects. Metaobjects encapsulate and represent information about the software. Examples include type structures, algorithms, or even function call mechanisms.

The base level defines the application logic. Its implementation uses the metaobjects to remain independent of those aspects that are likely to change. For example, base-level components may only communicate with each other via a metaobject that implements a specific user-defined function call mechanism. Changing this metaobject changes the way in which base-level components communicate, but without modifying the base-level code.

An interface is specified for manipulating the metaobjects. It is called the metaobject protocol (MOP), and allows clients to specify particular changes, such as modification of the function call mechanism metaobject mentioned above. The metaobject protocol itself is responsible for checking the correctness of the change specification, and for performing the change. Every manipulation of metaobjects through the metaobject protocol affects subsequent base-level behavior, as in the function call mechanism example.

For the persistence component, located at the base level of our example application, we specify metaobjects that provide run-time type information. For example, to store an object, we must know its internal structure and also the layout of all its data members. With this information available we can recursively iterate over any given object structure to break it down into a sequence of built-in types. The persistence component 'knows' how to store these. If we change the run-time type information we also modify the behavior of the store method. For example, objects of classes that are no longer persistent are no longer stored. Following similar strategies for every method, we can construct a persistence component that is able to read and store arbitrary data structures.
The meta level consists of a set of metaobjects. Each metaobject encapsulates selected information about a single aspect of the structure, behavior, or state of the base level. There are three sources for such information:

- It can be provided by the run-time environment of the system, such as C++ type identification objects [DWP95].
- It can be user-defined, such as the function call mechanism in the previous section.
- It can be retrieved from the base level at run-time, for example information about the current state of computation.

All metaobjects together provide a self-representation of an application. Metaobjects make information, which is otherwise only implicitly available, explicitly accessible and modifiable. Almost every system internal can be described in this way. For example, in a distributed system there may be metaobjects that provide information about the physical location of base-level components. Other base-level components can use these metaobjects to determine whether their communication partners are local or remote. They can select the most efficient function call mechanism to communicate with them. The function call mechanisms themselves may be provided by other metaobjects. Further examples include type structures, real-time constraints, inter-process communication mechanisms and transaction protocols.

However, what you represent with metaobjects depends on what should be adaptable. Only system details that are likely to change or which vary from customer to customer should be encapsulated by metaobjects. System aspects that are expected to stay stable over the lifetime of an application should not be.

The interface of a metaobject allows the base level to access the information it maintains or the service it offers. For example, a metaobject that provides location information about a distributed component will provide functions to access the name and identifier of the component, information about the process in which it is located, and information about the host on which the process runs. A metaobject that implements a function call mechanism will offer a method of activating a specific function of a specific addressee, including input and output parameter passing. A metaobject does not allow the base level to
modify its internal state. Manipulation is possible only through the metaobject protocol or by its own computation.

The base level models and implements the application logic of the software. Its components represent the various services the system offers as well as their underlying data model. The base level also specifies the fundamental collaboration and structural relationships between the components it includes. If the software includes a user interface, this is also part of the base level.

The base level uses the information and services provided by the metaobjects, such as location information about components and function call mechanisms. This allows the base level to remain flexible—its code is independent of aspects that may be subject to change and adaptation. Using the metaobject’s services, base-level components do not need to hard-code information about the concrete locations of communication partners—they consult appropriate metaobjects for this information.

Base-level components are either directly connected to the metaobjects on which they depend, or submit requests to them through special retrieval functions. These functions are also part of the meta level. The first type of connection is preferred if the relationship between the base level and the metaobject is relatively static. The base-level component always consults the same metaobject, for example if an object needs type information about itself. The second type of connection is used if the metaobjects used by the base level vary dynamically, as in the case of the store procedure of our persistence component.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Level</td>
<td>• Meta Level</td>
</tr>
</tbody>
</table>

**Responsibility**
- Implements the application logic.
- Uses information provided by the meta level.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta Level</td>
<td>• Base Level</td>
</tr>
</tbody>
</table>

**Responsibility**
- Encapsulates system internals that may change.
- Provides an interface to facilitate modifications to the meta-level.
The *metaobject protocol* (MOP) serves as an external interface to the meta level, and makes the implementation of a reflective system accessible in a defined way. Clients of the metaobject protocol, which may be base-level components, other applications, or privileged human users, can specify modifications to metaobjects or their relationships using the base level. The metaobject protocol itself is responsible for performing these changes. This provides a reflective application with explicit control over its own modification.

To continue our example above, a user may specify a new function call mechanism to be used for communication between base-level components. As a first step, the user provides the metaobject protocol with the code of this new function call mechanism. The metaobject protocol then performs the change. It may do this, for example, by generating an appropriate metaobject that includes the user-defined code for the new mechanism, compiling the generated metaobject, dynamically linking it with the application, and updating all references of the ‘old’ metaobject to the ‘new’ one.

The metaobject protocol is usually designed as a separate component. This supports the implementation of functions that operate on several metaobjects. For example, modifying metaobjects that encapsulate location information about distributed components eventually requires an update of the corresponding function call mechanism metaobjects. If we delegate the responsibility for such changes to the metaobjects themselves, consistency between them is hard to maintain. The metaobject protocol has a better control over every modification that is performed, because it is implemented separately from the metaobjects.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metaobject Protocol</td>
<td>• Meta Level</td>
</tr>
<tr>
<td>Responsibility</td>
<td>• Base Level</td>
</tr>
</tbody>
</table>

- Offers an interface for specifying changes to the meta level.
- Performs specified changes
To perform changes, the metaobject protocol needs access to the internals of metaobjects. If it is further entitled to change connections between base-level objects and metaobjects, it also needs access to base-level components. One way of providing this access is to allow the metaobject protocol to directly operate on their internal states. Another safer but more inefficient, way of providing it is for metaobjects and base-level components to provide a special interface for their manipulation, only accessible by the metaobject protocol.

Since the base-level implementation explicitly builds upon information and services provided by metaobjects, changing them has an immediate effect on the subsequent behavior of the base level. In our example, we changed the way base-level components communicate. However, in contrast to a conventional modification, the system was changed without modifying base-level code.

The general structure of a reflective architecture is very much like a Layered system (31). The meta level and base level are two layers, each of which provides its own interface. The base-level layer specifies the user interface for exploiting application functionality. The meta-level layer defines the metaobject protocol to modify the metaobjects.

However, in contrast to a layered architecture, there are mutual dependencies between both layers. The base level builds on the meta level, and vice-versa. An example of the latter occurs when metaobjects implement behavior that is executed in case of an exception. The kind of exception procedure that must be executed often depends on the current state of computation. The meta level retrieves this information from the base level, often from different components to those providing the interrupted service. In a pure layered architecture, these bidirectional dependencies between layers are not allowed. Every layer only builds upon the layers below.
For our persistence component example we specify metaobjects that provide introspective access to the type structure of our application—that is, they can access information about the application’s structure or behavior, but cannot modify it. We can obtain information about the name, size, data members and superclasses of a given type or object. An additional metaobject specifies a function that allows a client to instantiate objects of arbitrary types. We use this function, for example, when restoring an object structure from a data file. The metaobject protocol includes functions for adding new, and modifying existing, run-time type information.

The body of the persistence component is independent of the concrete type structure of our application. For example, the store procedure only implements the general algorithm for recursively breaking down a given object structure into a sequence of built-in types. If it needs information about the inner structure of user-defined types, it consults the meta level. Data members with built-in types are directly stored. All other data members are further decomposed.

Dynamics

It is almost impossible to describe the dynamic behavior of reflective systems in general. We therefore present two scenarios based on the persistence component example. See the Implementation section for details of the metaobject protocol and metaobjects involved.

Scenario I illustrates the collaboration between base level and meta level when reading objects stored in a disk file. All data is stored in an appropriate order, and a type identifier proceeds every object. The scenario further abstracts from special cases, such as reading strings, static members, and restoring cycles in the object structure. The scenario is divided into six phases:

- The user wants to read stored objects. The request is forwarded to the \texttt{read()} procedure of the persistence component, together with the name of the data file in which the objects are stored.
- Procedure \texttt{read()} opens the data file and calls an internal \texttt{readObject()} procedure which reads the first type identifier.
- Procedure \texttt{readObject()} calls the metaobject that is responsible for the creation of objects. The ‘object creator’ metaobject instantiates an ‘empty’ object of the previously-determined type. It returns a handle to this object and a handle to the corresponding run-time type information (RTTI) metaobject.
• Procedure `readObject()` requests an iterator over the data members of the object to be read from its corresponding metaobject. The procedure iterates over the data members of the object.

• Procedure `readObject()` reads the type identifier for the next data member. If the type identifier denotes a built-in type—a case we do not illustrate—the `readObject()` procedure directly assigns the next data item from the file to the data member, based on the data member's size and offset within the object. Otherwise `readObject()` is called recursively. This recursion starts with the creation of an 'empty' object if the data member is a pointer. If not, the recursively called `readObject()` operates on the existing layout of the object that contains the data member.

• After reading the data, the `read()` procedure closes the data file and returns the new objects to the client that requested them.
Scenario II illustrates the use of the metaobject protocol when adding type information to the meta level. Consider a class library used by the application that changes to a new version with new types. To store and read these types, we must extend the meta level with new metaobjects. Adding this information can be performed by the user, or automatically, using a tool. For reasons of simplicity we unify the classes type_info and extTypeInfo as specified in the Implementation section. The scenario is divided into six phases which are performed for every new type:

- A client invokes the metaobject protocol to specify run-time type information for a new type in the application. The name of the type is passed as an argument.

- The metaobject protocol creates a metaobject of class type_info for this type. This metaobject also serves as a type identifier.

- The client calls the metaobject protocol to add extended type information. This includes setting the size of the type, whether or not it is a pointer, and its inheritance relationships to other types. To handle the inheritance relationship, the metaobject protocol creates metaobjects of class baseInfo. These maintain a handle to the type_info object for a particular base class and its offset within the new type.

- In the next step, the client specifies the inner structure for the new type. The metaobject protocol is provided with the name and type of every data member. For every data member the metaobject protocol creates an object of class dataInfo. It maintains a handle to the type_info object for the type of the member, its name, and whether or not it is a static data member. The dataInfo object also maintains the absolute address of the data member if it is static, otherwise its offset within the new type.

- The client invokes the metaobject protocol to modify existing types that include the new type as a data member. Appropriate data member information is added for every type. Since this step is very similar to the previous one, we do not illustrate it in the object message sequence chart that follows.

- Finally, the client calls the metaobject protocol to adapt the ‘object creator’ metaobject. The persistence component must be able to instantiate an object of the new type when reading persistent data.
The metaobject protocol automatically generates code for creating objects of the new type, based on the previously-added type information. It further integrates the new code with the existing implementation of the ‘object creator’ metaobject, compiles the modified implementation, and links it with the application.
The following guidelines help with implementing a Reflection architecture. Iterate through any subsequence if necessary.

1. **Define a model of the application.** Analyze the problem domain and decompose it into an appropriate software structure. Answer the following questions:
   - Which services should the software provide?
   - Which components can fulfill these services?
   - What are the relationships between the components?
   - How do the components cooperate or collaborate?
   - What data do the components operate on?
   - How will the user interact with the software?

   Follow an appropriate analysis method when specifying the model.

   ➪ The persistence component in our C++ disk-storage example is part of a warehouse management application [Coad95]. We identify components that represent physical storage, such as warehouses, aisles and bins. We also identify components for orders and items. It is a requirement that we can resume computation with a valid state after system crashes. Both the physical structure of the warehouse and its current population of items must therefore be made persistent. We need two components to achieve this. A persistence component provides the functionality for storing and reading objects. A file handler is responsible for locking, opening, closing, unlocking and deleting files, as well as for writing and reading data.

2. **Identify varying behavior.** Analyze the model developed in the previous step and determine which of the application services may vary and which remain stable. There are no general rules for specifying what can alter in a system. Whether a certain aspect varies depends on many factors such as the application domain, the environment of the application and its customers and users. An aspect that is likely to vary in one system may stay stable in others. The following are examples of system aspects that often vary:
   - Real-time constraints [HT92], such as deadlines, time-fence protocols and algorithms for detecting deadline misses.
   - Transaction protocols [SW95], for example optimistic and pessimistic transaction control in accounting systems.
• Inter-process communication mechanisms [CM93], such as remote procedure calls and shared memory.

• Behavior in case of exceptions [EKM+94], [HT92], for example the handling of deadline misses in real-time systems.

• Algorithms for application services [EKM+94], such as country-specific VAT calculation.

The Open Implementation Analysis and Design Method [KLLM95] helps with this step.

➢ To keep the persistence component example simple, we do not consider an adaptation of application behavior.

3 Identify structural aspects of the system, which, when changed, should not affect the implementation of the base level. Examples include the type structure of an application [BKSP92], its underlying object model [McA95], or the distribution of components [McA95] in a heterogenous network.

➢ Our implementation of the persistence component must be independent of application-specific types. This requires access to run-time type information, such as the name, size, inheritance relationships and internal layout of each type, as well as the types, order and names of their data members.

4 Identify system services that support both the variation of application services identified in step 2 and the independence of structural details identified in step 3. For example, implementing resumable exceptions in C++ requires explicit access to the exception handling mechanism of the language. Other examples of basic system services are:

• Resource allocation

• Garbage collection

• Page swapping

• Object creation

➢ The persistence component must instantiate arbitrary classes when reading persistent objects.
5 Define the metaobjects. For every aspect identified in the three previous steps, define appropriate metaobjects. Encapsulating behavior is supported by several domain-independent design patterns, such as Objectifier [Zim94], Strategy, Bridge, Visitor, and Abstract Factory [GHJV95]. For example, metaobjects for function call mechanisms can be implemented as strategy objects, and multiple implementations of components can be implemented with the Bridge pattern. Visitor allows you to integrate new functionality without modifying existing structures. Sometimes you may find appropriate domain-specific patterns that support this step, for example the Acceptor and Connector patterns for developing distributed systems [Sch95]. Another example is the Detachable Inspector pattern [SC95a], which supports the addition of run-time facilities such as debuggers and inspectors. Detachable Inspector builds on the Visitor pattern. Encapsulating structural and state information is supported by design patterns like Objectifier [Zim94] and State [GHJV95].

The metaobjects that provide the run-time type information for our persistence component are organized as follows:

The C++ standard library class `type_info` is used for identifying types [DWP95]. Its interface offers functions for accessing the name of a type, for comparing two types, and for determining their system internal order. Every type in the application is represented by an instance of class `type_info`.

```cpp
class type_info {
    //...
    private:
        type_info(const type_info& rhs);
        type_info& operator=(const type_info& rhs);
    public:
        virtual ~type_info();
        int operator==(const type_info& rhs) const;
        int operator!=(const type_info& rhs) const;
        int before(const type_info& rhs) const;
        const char* name() const;
    }
```

None of the other classes of the run-time type information system are part of the C++ standard.

A class `extTypeInfo` provides access to information about the size, superclasses, and data members of a class. Clients can also determine whether the type is built-in or a pointer.
The method `bases()` returns an object of class `baseIter`, which is an iterator over either all base classes of a given type or just its direct base classes. If the type is built-in, the method returns a NULL iterator. Analogously, the method `data()` returns an object of class `dataIter`. It iterates either over all data members of a given type, including inherited ones, or just the data members declared specifically for this type. If the type is built-in, the method returns a NULL iterator.

A class `BaseInfo` offers functions for accessing type information about a base class of a class, as well as to determine its offset in the class layout.

```cpp
class BaseInfo {
    // ...
    public:
        const type_info* type() const;
        const long offset() const;
    }
```

A class `DataInfo` includes functions that return the name of a data member, its offset and its associated `type_info` object.

```cpp
class DataInfo {
    // ...
    public:
        const char* name() const;
        const type_info* type() const;
        const bool isStatic() const;
        const long offset() const;
        const long address() const;
    }
```
Define the metaobject protocol. Support a defined and controlled modification and extension of the meta level, and also a modification of relationships between base-level components and metaobjects.

There are two options for implementing the metaobject protocol:

- Integrate it with the metaobjects. Every metaobject provides those functions of the metaobject protocol that operate on it.
- Implement the metaobject protocol as a separate component.

An advantage of the latter approach is that the control of every modification of the reflective application is localized at a central point. Functions that operate on several metaobjects are easier to implement. In addition, a separate component can shield metaobjects from unauthorized access and modification, if its implementation follows patterns such as Facade [GHJV95] or Whole-Part (225). The Singleton idiom [GHJV95] helps ensure that the metaobject protocol can only be instantiated once.

If implemented as a separate component, the metaobject protocol usually does not serve as a base class for classes that define metaobjects—it just operates on them. It only makes sense to specify the metaobject protocol as a base class from which concrete metaobject classes are derived if it applies to every metaobject.

We provide a class MOP which defines the metaobject protocol for the meta level of our persistence component example. It is implemented as a singleton and operates directly on the internal structure of all classes declared in the previous step.

Type information is accessible by two functions.

```cpp
const type_info* getInfo(char* typeName) const;
const extTypeInfo* getExtInfo(char* typeName) const;
```

The first function allows clients to access the standard type information about an object. The second function accesses the extended type information that we defined specifically for our runtime type information system. We need this function because objects of the standard class type_info do not provide access to user-defined information. All other type information—such as that about base classes—is accessible through the extTypeInfo object.
New type information metaobjects can be initialized with two functions, one for instantiating type_info objects and one for creating extTypeInfo objects.

```c
void newTypeId(char* typeName);
void newTypeInfo(char* typeName,
                 bool builtIn, bool pointer);
```

The newTypeInfo() function also calculates and sets the size of a type. The function deleteInfo() deletes all available information about a type, but only if no other class of the system contains a reference to an object of that type.

```c
void deleteInfo(char* typeName);
```

We define four functions for adding new or modifying existing type information. The functions addBase() and deleteBase() respectively add and remove base class information, while the functions addData() and deleteData() respectively add and delete data member information.

```c
void addBase(char* typeName, char* baseName);
void addData(char* typeName,
            char* memberType, char* memberName);
void deleteBase(char* typeName, char* baseName);
void deleteData(char* typeName, char* memberName);
```

Before executing changes, all functions perform consistency checks. For example, to set base class information, corresponding type_info and extTypeInfo objects must be available.

Two functions support modification of the ‘object creator’ metaobject.

```c
void addCreationCode(char* typeName);
void deleteCreationCode(char* typeName);
```

Internally, the metaobject protocol needs functions for calculating type sizes and offsets of base classes and data members. These functions are compiler-dependent and must therefore be changed when using a different compiler. One way to support changing these functions is provided by the Strategy pattern [GHJV95]. To maintain type_info and extTypeInfo objects, the metaobject protocol maintains two maps, tMap and eMap. These maps offer functions to add, remove and find elements.
Most functions of the metaobject protocol can be implemented straightforwardly. Calculating offset and sizes and manipulating the 'object creator' metaobject requires higher implementation effort. The following code defines the `addBase()` function.

```c++
void MOP::addBase(char* typeName, char* baseName) {
    BaseInfo* base;
    // Is extended type information for type typeName
    // and type information for type baseName available?
    if (!eMap.element(typeName) ||
        !tMap.element(baseName))
        // error handling ...
    // Instantiate the baseInfo object for type baseName
    base = new BaseInfo(tMap[baseName]);
    // Calculate the offset of the base class.
    base->baseOffset = calcOffset(typeName, baseName);
    // Add the new baseInfo object to the list of
    // bases within the extTypeInfo object for
    // type typeName
    eMap[typeName]->baseList.add(base);
}
```

Robustness is a major concern when implementing the metaobject protocol. Errors in change specifications should be detected wherever possible. Changes should also be reliable. The metaobject protocol described above, for example, checks the availability of appropriate type information metaobjects when adding new base class and data member information. Before deleting its type information, it also checks whether a type is used as a base class or data member.

Robustness also means maintaining consistency. For example, if we add a data member to a specific type, we must recalculate the size of all types that include the changed type as a base class or a data member. In addition, any modification should only affect those parts of the system that are subject to change. Finally, clients of the metaobject protocol should not take responsibility for integrating changes into the meta level. Ideally, a client only specifies a change, and the metaobject protocol is responsible for its integration. This avoids direct manipulation of source code.
7 Define the base level. Implement the functional core and user interface of the system according to the analysis model developed in step 1.

Use metaobjects to keep the base level extensible and adaptable. Connect every base-level component with metaobjects that provide system information on which they depend, such as type information, or which offer services they need, such as object creation in our persistence component. To handle system services, use design patterns such as Strategy, Visitor, Abstract Factory and Bridge [GHJV95], or idioms like Envelope-Letter [Cope92]. For example, the context class component of the Strategy pattern represents the base-level component, and the strategy class hierarchy the metaobjects. When applying the Visitor pattern, the metaobjects are the visitors, and the object structure represents the base-level components.

Provide base-level components with functions for maintaining the relationships with their associated metaobjects. The metaobject protocol must be able modify every relationship between the base level and the meta level. For example, when replacing a metaobject with a new one, the metaobject protocol must update all references to the replaced metaobject. The metaobject protocol operates either directly on internal data structures of base-level components, or uses a special interface the base-level components provide.

If the metaobjects to be used are not known a priori, provide the meta level or the metaobject protocol with appropriate retrieval functions, such as the \texttt{getInfo()} and \texttt{getExtInfo()} functions in the persistence component example.

Metaobjects often need information about the current state of computation. For example, the ‘object creator’ in our persistence component example must know what type it should instantiate. This information can either be passed as a parameter to the metaobjects, the metaobjects can retrieve it from other metaobjects, or the metaobjects can retrieve it from appropriate base-level components.

Changes to metaobjects affect the subsequent behavior of base-level components to which they are connected. Changing a relationship between the base level and the meta level affects only a specific base-level component, the one that maintains the modified relationship.
The implementation of the read() method of our persistence component follows the first scenario depicted in the Dynamics section. The method implements a general recursive algorithm for reading objects from a data file. The method consults the meta level to get information about how to read user-defined types. Reading built-in types or strings is hard-coded within its implementation. To obtain information about types, read() consults the getInfo() and getExtInfo() functions of the metaobject protocol. For creating objects of arbitrary types, read() is directly connected with the 'object creator' metaobject.

The structure of the store() method is similar to that of the read() method. It first opens the data file to be read, then calls an internal storeObject() method that stores the object structure. Finally, store() closes the data file.

The most challenging part of implementing store() is the detection of cycles in the object structure to be stored—it is essential to avoid storing duplicates and running into infinite recursion. To achieve this, the method marks the structure with a unique identifier which is also stored, before storing the object. If we return to an object that is so marked, we then just store its identifier.

The following simplified code illustrates the structure of the storeObject() method. It abstracts from several details, such as the storage of static data members.

```c++
void Persistence::storeObject(void* object, char* typeName) {
    type_info* objectId;
    extTypeInfo* objectInfo;
    baseIter* iterator;

    // Get type information about the object to be stored
    objectId   = mop->getInfo(typeName);
    objectInfo = mop->getExtInfo(typeName);
    iterator   = objectInfo->data();

    // Mark the object to avoid storing duplicates
    markObject(object);

    // Object is of built-in type?
    if (objectInfo->isBuiltIn())
        storeBuiltIn(object, objectId);
```
In the previous sections we explained the Reflection architecture of our persistence component example. How we provide run-time type information is still an open issue.

Unlike languages like CLOS or Smalltalk, C++ does not support reflection very well—only the standard class `type_info` provides reflective capabilities: we can identify and compare types. One solution for providing extended type information is to include a special step in the compilation process. In this, we collect type information from the source files of the application, generate code for instantiating the metaobjects, and link this code with the application. Similarly, the ‘object creator’ metaobject is generated. Users specify code for instantiating an ‘empty’ object of every type, and the toolkit generates the code for the metaobject. Some parts of the system are compiler-dependent, such as offset and size calculation.
As illustrated in the code examples, we use pointer and address arithmetic, offsets, and sizes of types and data members to read and store objects. Since these features are considered harmful, for example by incurring the danger of overwriting object code, the persistence component must be implemented and tested very carefully.

**Variants**  
*Reflection with several meta levels.* Sometimes metaobjects depend on each other. For example, consider the persistence component. Changes to the run-time type information of a particular type requires that you update the ‘object creator’ metaobject. To coordinate such changes you may introduce separate metaobjects, and—conceptually—a meta level for the meta level, or in other words, a meta meta level. In theory this leads to an infinite tower of reflection. Such a software system has an infinite number of meta levels in which each meta level is controlled by a higher one, and where each meta level has its own metaobject protocol. In practice, most existing reflective software comprises only one or two meta levels.

An example of a programming language with several meta levels is RbCl [MY92]. RbCl is an interpreted language. RbCl base-level objects are represented by several meta-level objects. These are interpreted by an interpreter that resides at the meta metal level of RbCl. The metaobject protocol of RbCl allows users to modify the metaobjects that represent RbCl base-level objects, the metaobject protocol of the meta meta level the behavior of the RbCl metaobject interpreter.

**Known Uses**  
*CLOS.* This is the classic example of a reflective programming language [Kee89]. In CLOS, operations defined for objects are called *generic functions,* and their processing is referred to as *generic function invocation.* Generic function invocation is divided into three phases:

- The system first determines the methods that are applicable to a given invocation.
- It then sorts the applicable methods in decreasing order of precedence.
- The system finally sequences the execution of the list of applicable methods. Note that in CLOS more than one method can be executed in response to a given invocation.
The process of generic function invocation is defined in the metaobject protocol of CLOS [KR891]. Basically, it executes a certain sequence of meta-level generic functions. Through the CLOS metaobject protocol users can vary the behavior of an application by modifying these generic functions or the generic functions of the metaobjects they call.

**MIP** [BKSP92] is a run-time type information system for C++. It is mainly used for introspective access to the type system of an application. Every type of a C++ software system is represented by a set of metaobjects that provide general information about that type, its relationships to other types, and its inner structure. All information is accessible at run-time. The functionality of MIP is separated into four layers:

- The first layer includes information and functionality that allows software to identify and compare types. This layer corresponds to the standard run-time type identification facilities for C++ [SL92].

- The second layer provides more detailed information about the type system of an application. For example, clients can obtain information about inheritance relationships for classes, or about their data and function members. This information can be used to browse type structures.

- The third layer provides information about relative addresses of data members, and offers functions for creating 'empty' objects of user-defined types. In combination with the second layer, this layer supports object I/O.

- The fourth layer provides full type information, such as that about friends of a class, protection of data members, or argument and return types of function members. This layer supports the development of flexible inter-process communication mechanisms, or of tools such as inspectors, that need very detailed information about the type structure of an application.

The metaobject protocol of MIP allows you to specify and modify the metaobjects that provide run-time type information. It offers appropriate functions for every layer of the MIP functionality.

MIP is implemented as a set of library classes. It also includes a toolkit for collecting type information about an application, and to generate code for instantiating the corresponding metaobjects. This
code is linked to the application that uses MIP and is executed at the beginning of the main program. The toolkit can be integrated with the ‘standard’ compilation process for C++ applications. A special interface allows users to scale the available type information for every individual class or type.

**PGen** [THP94] is a persistence component for C++ that is based on MIP. It allows an application to store and read arbitrary C++ object structures.

The example used to explain the Reflection pattern is based mainly on MIP and PGen. Although simplified, the description of the persistence component, the class declarations for the metaobjects and the metaobject protocol widely reflect the original structure of MIP and PGen.

**NEDIS.** The car-dealer system NEDIS [Ste95] uses reflection to support its adaptation to customer- and country-specific requirements. NEDIS includes a meta level called *run-time data dictionary*. It provides the following services and system information:

- Properties for certain attributes of classes, such as their allowed value ranges.
- Functions for checking attribute values against their required properties. NEDIS uses these functions to evaluate user input, for example to validate a date.
- Default values for attributes of classes, used to initialize new objects.
- Functions specifying the behavior of the system in the event of errors, such as invalid input or unexpected ‘null’ values of attributes.
- Country-specific functionality, for example for tax calculation.
- Information about the ‘look and feel’ of the software, such as the layout of input masks or the language to be used in the user interface.

The run-time data dictionary is implemented as a persistent database. A special interface allows users to modify any information or service it provides. Whenever the run-time data dictionary
Reflection

changes, special tools check and eventually restore its consistency. The run-time data dictionary is loaded when starting the software. For reasons of safety it cannot be modified while NEDIS is running.

**OLE 2.0** [Bro94] provides functionality for exposing and accessing type information about OLE objects and their interfaces. The information can be used to dynamically access structural information about OLE objects, and to create invocations of OLE interfaces. For example, the run-time environment of Visual Basic [Mic95] checks the correctness of method calls to an object before dynamically invoking it. A similar concept is specified for Corba [OMG92].

Further examples of languages and systems that use a Reflection architecture include Open C++ [CM93], RbCl [IMY92], AL-1/D [OIT92], R2 [HT92], Apertos [Yok92] and CodA [McA95]. Even more examples can be found in [IMSA92], but note that although all examples provide reflective facilities, not all of them really implement a Reflection architecture as described by this pattern.

**Consequences**

A Reflection architecture provides the following **benefits**:

*No explicit modification of source code.* You do not need to touch existing code when modifying a reflective system. Instead, you specify a change by calling a function of the metaobject protocol. When extending the software, you pass the new code to the meta level as a parameter of the metaobject protocol. The metaobject protocol itself is responsible for integrating your change requests: it performs modifications and extensions to meta-level code, and if necessary re-compiles the changed parts and links them to the application while it is executing.

*Changing a software system is easy.* The metaobject protocol provides a safe and uniform mechanism for changing software. It hides all specific techniques such as the use of visitors, factories and strategies from the user. It also hides the inner complexity of a changeable application. The user is not confronted with the many metaobjects that encapsulate particular system aspects. The metaobject protocol also takes control over every modification. A well-designed and robust metaobject protocol helps prevent undesired changes of the fundamental semantics of an application [Kie92].

*Support for many kinds of change.* Metaobjects can encapsulate every aspect of system behavior, state and structure. An architecture based
on the Reflection pattern thus potentially supports changes of almost any kind or scale. Even fundamental system aspects can be changed, such as function call mechanisms or type structures. With the help of reflective techniques it is also possible to adapt software to meet specific needs of the environment or to integrate customer-specific requirements.

However, a Reflection architecture has some significant liabilities:

**Modifications at the meta level may cause damage.** Even the safest metaobject protocol does not prevent users from specifying incorrect modifications. Such modifications may cause serious damage to the software or its environment. Examples of dangerous modifications include changing a database schema without suspending the execution of the objects in the application that use it, or passing code to the metaobject protocol that includes semantic errors. Similarly, bugs in pointer arithmetic can cause object code to be overwritten.

The robustness of a metaobject protocol is therefore of great importance [Kic92]. Potential errors within change specifications should be detected before the change is performed. Each change should only have a limited effect on other parts of the software.

**Increased number of components.** It may happen that a reflective software system includes more metaobjects than base-level components. The greater the number of aspects that are encapsulated at the meta level, the more metaobjects there are.

**Lower efficiency.** Reflective software systems are usually slower than non-reflective systems. This is caused by the complex relationship between the base level and the meta level. Whenever the base level is unable to decide how to continue with computation, it consults the meta level for assistance. This reflective capability requires extra processing: information retrieval, changing metaobjects, consistency checking, and the communication between the two levels decrease the overall performance of the system. You can partly reduce this performance penalty by optimization techniques, such as injecting meta-level code directly into the base level when compiling the system.
Not all potential changes to the software are supported. Although a Reflection architecture helps with the development of changeable software, only changes that can be performed through the metaobject protocol are supported. As a result, it is not possible to integrate easily all unforeseen changes to an application, for example changes or extensions to base-level code.

Not all languages support reflection. A Reflection architecture is hard to implement in some languages, such as C++, which offers little or no support for reflection. C++ only provides type identification. Reflective applications in C++ often build on language constructs such as pointer arithmetic to handle arbitrary objects, and need tool support for dynamically modifying meta-level code. This is, however, tedious and error-prone. In such languages it is also impossible to exploit the full power of reflection, such as adding new methods to a class dynamically. However, even in languages that do not provide reflective capabilities, it is possible to build reflective systems that are changeable and extensible, such as the C++ systems NEDIS [EKM+94], MIP [BKSP92] and Open C++ [CM93].

See Also

The Microkernel architectural pattern (171) supports adaptation and change by providing a mechanism for extending the software with additional or customer-specific functionality. The central component of this architecture—the microkernel—serves as a socket for plugging in such extensions and for coordinating their collaboration. Modifications can be made by exchanging these 'pluggable' parts.

An earlier version of this pattern appeared in [PLoP95].

Credits

One of the first works on reflection is the Ph.D. thesis by Brian Cantwell Smith [Smi82]. This describes reflection in the context of procedural languages. An overview of reflective concepts can be found in [Mae87].

We thank the members of PLoP’95 Working Group 1 for their valuable criticism and suggestions for improvement of an earlier version of this pattern, especially Douglas C. Schmidt and Aamod Sane. Special thanks also go to Linda Rising and David E. DeLano from AG Communication Systems, and Brian Foote and Ralph Johnson from the University of Illinois at Urbana Champaign. Their detailed review of an earlier version of this pattern helped to shape this description.