Do-it-yourself Reflection

Authors

Peter Sommerlad and Marcel Rüedi
IFA Informatik, Zurich, +41-1-273 4646
sommerlad@ifa.ch, rueedi@ifa.ch

We wrote down this collection of design patterns, because they describe well-known practice. They all have been used in building flexible systems or frameworks. We call them "do-it-yourself reflection", because they follow some of the principles of reflective systems, accessing program elements by the program itself. They index program elements by data values (e.g. strings as names) not by program language identifiers.

The three patterns are Property List, Anything, and Registry. Property List and Anything deal with flexibility in type and number of attributes or parameters. Registry provides mechanisms for managing global resources or objects in a systematic and flexible way.

The Object System pattern [Noble98] submitted by James Noble to EuroPLoP 98 takes the core ideas of reflection to the extreme and shows you how to build your own reflective object system from scratch.

Shared Context

Consider you are developing a flexible piece of software, that has to be extended in the future. There are two approaches to it:

white-box extensibility: You define abstract interfaces to be competed by subclassing, or

black-box extensibility: You provide complete components that perform some tasks on behalf of clients components.

One of the key issues of white box extensibility is to define stable interfaces, that you can use for even unforseen extensions. Nevertheless, flexible interfaces should not impose too much overhead.

Both means of extensibility require some means of managing the used components and often managing some configuration data of these components in a concrete system.

Credits We thank our current and former colleages that worked with us and designed systems where we mined these patterns. Special credits to our EuroPLoP shepherd Robert Hirschfeld and all the organizers and workshop participants of EuroPLoP 98.

Property List

The Property List design pattern is used to attach a flexible set of attributes to an object at run-time. Each attribute is given a name represented by a data value (not an identifier in the programming language) and attributes can be added or removed on a per object basis.

Example We are building a framework for a graphical editor. A key abstraction in that framework is the class for graphical objects Gobject. We want subclasses of Gobject to represent geometrical forms. Let us consider just the attributes of Gobject and its descendents. We might end up with the following structure:

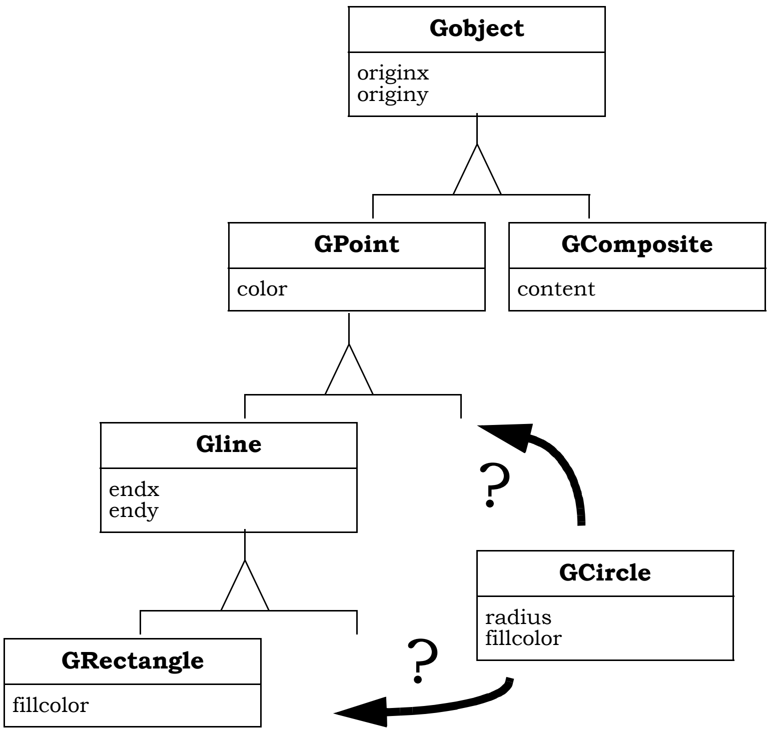
```
class Gobject {
int      xorigin, yorigin;
};
class GPoint extends Gobject {
int      color;
};
class GLine extends GPoint {
int      xend, yend;
};
class GRectangle extends GLine {
int      fillcolor;
}
class CompositeGobject extends Gobject {
Vector   content;
}
```

Now we are adding circles to our Gobject hierarchy.

```
class GCircle extends ??? {
int      radius;
int      fillcolor;
};
```

Should we inherit from GRectangle to obtain a fillcolor attribute, or should we refactor our hierarchy and use multiple inheritance to merge-in the attribute fillcolor? Both ways have their problems.

How can we incorporate additional attributes like line width. In a CompositeGobject storing the line width for each element might be



too much overhead. Nevertheless, users should be able to construct a composed graphical object using different line widths.

In addition we are developing a framework. Framework users might want to attach semantic information to our graphical objects, for example, associating a circle with a network node in a network topology view.

We are stuck with a situation where common attributes do not help us with defining the class hierarchy and where we do not know all the needed attributes up-front.

Context You are developing a flexible piece of software, that has to be extended in the future. You are using a static typed language without built-in support for very flexible data types.

Problem In the given context, several areas arise similar problems.

If you want to provide white-box extensibility, it might be hard to design the interfaces flexible enough to be used for later extension. For example, it might be hard to define the exact type and amount of data to be passed to an operation, which implementation is not yet written. Nevertheless you also do not want future extensions to break encapsulation. *How do you define parameters in a flexible way?*

If you prepare for black-box extensibility, client components might require your components to store information that you cannot guess in advance as the network node reference in the example. *How do you define the attributes of your components in a way they can be extended by client components?*

Both cases might force you to design class hierarchies, where objects sharing identical attributes cannot be related directly in the hierarchy. And, where factoring such shared attributes to a common base class would either break encapsulation or would bloat the root classes as it is the case with the fillcolor attribute in the example. *How do you implement these common attributes, showing that they are really the same to a programmer of a client component?*

A last situation to consider is when you have objects that follow some life-cycle, where specific attributes only make sense during a some period of time. *How do you implement attributes that should be attached or detached during runtime?*

In dynamic languages like Lisp or Smalltalk these issues do not arise, because the language environment itself provides enough flexibility. However, in type-safe compiled languages like C++ or also Java, defining flexible parameters or attributes to be attached or detached at run-time can be a design challenge.

In particular you want to address the following *forces*:

- Attributes should be attached/detached at run-time.
- Objects share attributes/parameters across the class hierarchy.
- You want to pass an open set of parameters to an operation.

Solution Provide the objects with a 'property list'. This is a data-structure that allows to associate names (e.g. string values) with arbitrary other values or objects. Each name identifies a so-called slot. Each defined slot will refer to or store an object or a value of a given type. The owner or the client code has to decide how to interpret the value returned, e.g. by “downcasting” it. A property list can be dyamically modified. It is stored in an object as a single attribute.

Class Client	***Collaborators***
Responsibility	Owner
• accesses Owners properties via Owners interface • defines what property names to use and what to do with the values	

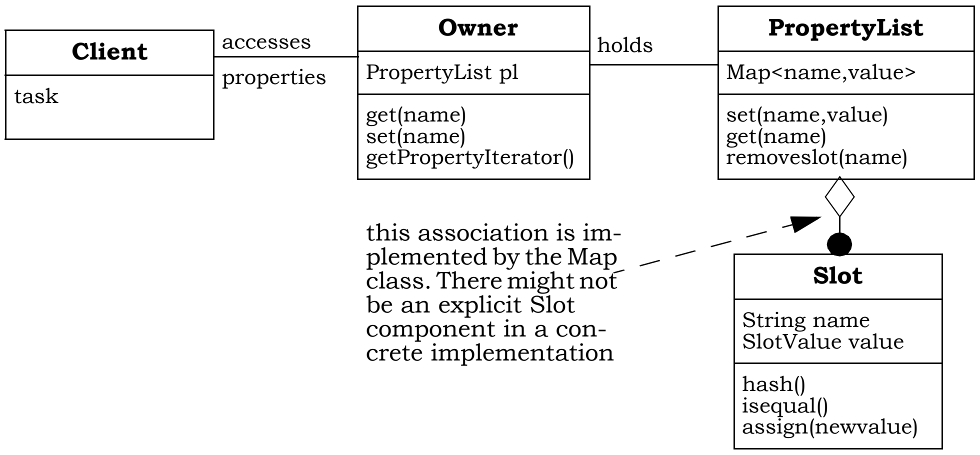
Class Owner	***Collaborators***
Responsibility	PropertyList
• holds a property list. • provides interface to get and set slot values via names • can map semantic operations to property list implementation	

Class Property List	***Collaborators***
Responsibility	
• provides dynamic table of slots with (name,value) pairs • provides access to slot values via names. • provides iteration over slots	

no CRC cards for slot names and slot values are shown

Using property lists throughout your class hierarchy allows you to use the same “attribute name” (i.e. slot name) for attributes with identical semantics (e.g. “line-width” might make sense for lines and hollow shapes in your graphical object hierarchy, but it will not be useful for points or filled shapes).

Structure If you use Property List in your program, you would usually define a utility data structure implementing these property lists. The data structure holds a map from slot names to values (aka dictionary). This results in the following components used: client, owner, property list and—depending on the implementation—a slot with a index type and value type.



Dynamics Since the property list is typically used as a flexible data storage, no very interesting dynamic scenarios can be shown.

Implementation To implement the Property List pattern perform the following steps

1 *Analyze carefully where you will need property lists.* Prototyping or explorative programming might be much easier if you use property lists as attributes. Look at the consequences section if the liabilities using property lists, like the code and run-time overhead, do not hinder you from its application.

2 *Decide what kind of slot index you need and what data types should be stored as slot values.* The most generic pair of types for the property list association would be (object,object) but this is only useful if your language and concrete problem allows you to create the index objects easily. A typical choice is to use strings for the slot indices. Depending on the implementation of the mapping having a slot index type with a good hash function might be preferable. Slot values are often object references, especially in single-rooted OO-languages, like in Java. If you only want to provide simple values (integers, doubles, characters) as slot values a C-union like structure might be sufficient. For example:

```
union slotvalues {
     char c;
     int  i;
     double d;
}; // the client will know what to use
```

3 *Implement your Property List component* with the data structure of your choice. Property lists are often implemented using hash-tables or some other style of implementing look-up. It is beyond the scope of this pattern description to discuss all options you might use for the name to value mapping and for storing the associations. A good advice might be to use the dictionary or map types your favorite class library provides. In Java there is already a property list standard class.

An important implementation decision is the behavior in the case a client tries to access a slot that is not in the property list. Either a new slot with a null value is created automatically, the property list returns just a null value to show the missing slot or it raises an exception.

4 *Implement your Owner class(es)*. These classes take a property list as an attribute. You have to decide if you want to implement all attributes as just mapping them to the property list. However, you might want to provide specific operations for clients, that hide the property list implementation of these attributes. Nevertheless, for easy extensibility by clients, provide generic set and get operations on the property list of your owner class(es).

You get a nice side-effect if you implement all attributes of the owner classes as slots in a property list. Then object serialization can be reduced to store the property list. This mechanism can be provided by the Owner classes´ base class in a generic way. Especially in the situation of prototyping a program this might be handy and allows a programmer to concentrate on functionality issues instead of building infrastructure.

5 *Implement Client components*. Very often these client components are the ones that define some or even most of the properties semantics. They will use the slot names to set, read and modify property values.

Variants **Property Lists as parameters.** Passing a property list as a parameter allows a client component to extend the amount of data passed to an operation at will. If you want to keep stable interfaces, shared by different teams, but are still heavily under development, property-lists as parameters or attributes can be a real life-safer (see also Anything).

Anything. The Anything pattern is similar to implementing nested property lists. However, because there are more issues to discuss, it is described as a separate pattern below.

Known Uses Most current component construction kits, like Delphi, Visual Basic or Java Beans have property lists attached to each component that can be used to define and fine-tune the look and behavior of the components.

Factory Method can be implemented nicely using a property list of Prototype objects [GHJV95], View Handler [POSA96] might use a property list to store managed windows.

Consequences The Property List pattern implies the following **benefits**:

- *Black-box extensibility.* A property list allows you to attach new values to an objects by client code without changing the object´s class. This mechanism assists in extending a system even if its basics are only available as “black-boxes” without the source code.

- *Extension with keeping object identity.* Property List allows client program code to extend existing objects with keeping their object identity, thus eliminating one of the drawbacks of wrapping the object to extend its functionality.

- *Iteration over object attributes.* It is possible to iterate over a property list´s entries. This allows you to implement generic mechanisms manipulating objects easily, even when your programming language does not provide you with the required meta-information on a class´ inner structure. Examples of such mechanisms are an object inspector and object serialization.

- *Per-object Attributes and possible memory optimization.* In systems where instances of a class play different roles, their required attribute space might vary heavily. Defining all attributes that might be needed through an objects lifetime as regular attributes in your programming language might impose a huge memory overhead. This benefit might make property lists useful even in languages like smalltalk.

- *Attribut sharing across class hierarchy.* Using the same slot name a program can simulate using identical attributes with objects not related within the class hierarchy. Consider our graphical object example: There classes Circle and Rectangle can share attribut ´fillcolor´even if they are not directly related in the hierarchy.

- *Flexible Parameters.* Using a Property List as a parameter in an interface definition allows for covariant extension of a system. Client program code can supply parameter slots, that are used by co-developed implementors of the interface.

However, the Property List pattern also has its **liabilities**:

- *Possible confusion.* If you use normal attributes and property lists there are different ways to access regular and dynamic attributes defined as properties. A programmer extending your components might be confused by the different syntactical ways for accessing attributes.

- *Type safety is left to the programmer.* The code using a property list must know how to interpret a given result. This might result in additional code for type checking and error handling. If your programming language provides meta-information it is at least possible to detect the type of a value retrieved from the property list and perform only allowed operations on it.

- *Naming of slots is not checked by a compiler.* This situation might result in hard-to-detect errors that wouldn´t occur with normal attributes. Just consider what happens with 'color' vs. 'colour' in our example.

- *Semantics of attributes not given by owner class code but by client code.* If several clients use the same slot names they might influence each other without notice, or they might have different perception on how to interpret a given attribut. For example, is ´color´ defining the line color or the fill color?

- *Run-time overhead can be substantial because of name-lookup.* Especially in contrast to a compiled indexed memory access a name lookup and eventually type-check can be orders of magnitudes more expensive. If you expect slow performance, carefully decide where you have other implementation options.

- *Misuse.* It is easy to misuse the Property List pattern to loose all type-safety of your language environment. This happens very often when the “hammer syndrome” occurs to a developer.[1] Some people might even consider the proposed persistence mechanism based on property lists a “misuse”.

- *Memory Management.* A design decision to be taken is if get() operation returns a copy of the slot value or a reference to it for performance reasons. In the latter case, if you are implementing Property List in a programming language where memory management is left to the programmer, you need additional mechanisms to keep track of memory used by slot values.

See Also Registry can be implemented using Property List. Anything is an alternative to Property List if deeper structure is required. Kent Becks Variable State [Beck96] is similar to this pattern in Smalltalk.

Credits The example using property lists for attributes of drawing objects is based on early versions of Erich Gamma´s JHotDraw drawing editor framework. Thanks to Dirk Riehle [Riehle96] who notified me (Peter) about the absence of a pattern version of Property List.

1. We refer to the situation typically described as “give someone a hammer (property list) and everything looks like a nail”.

Anything

The Anything design pattern provides a generic structured data container that is useful as universal (catch-all) operation parameter. In addition Anythings are well-suited as a flexible means of storing , retrieving and transmitting structured data values. This makes Anything an ideal implementation technique for configuration data.

Example We are developing a framework for creating HTML output. Each HTML construct is created by a component called renderer. We want to call each renderer in a generic way, but unfortunately they require completely different number and types of parameters.

```
class Reply; //collects output
struct Parameter {};
class Renderer { // abstract
public:
void RenderAll(Reply &r, const Parameter &p)=0;
};
class StringRenderer {
public:
void RenderAll(Reply &r, const Parameter &p);
};
struct StringRendererParameter : public Parameter {
String language; // allow for language specific strings
Map<String,String> values; // map language to value
};
class ImageRenderer {
public:
void RenderAll(Reply &r, const Parameter &p);
};
struct ImageRendererParameter : public Parameter {
String imagepath;
StringRendererParameter alternate;
String imagename;
};
```

For each Renderer subclass we need to describe the required parameters as a separate subclass. In addition we need some means to fill in the correct parameter subclass when activating a specific renderer object. In addition down-casts of the Parameter reference are required and these might fail at run-time.

One option out of this dilemma is to use C++´ open parameter list like

```
    void RenderAll(Reply &r, ...);
```

However, this is even more error-prone, because then the caller is responsible for providing some useful parameter list and the called operation needs to figure out what is passed how. In addition the (...) parameters are generally passed untyped.

It is beyond the scope of this example to show all other options and infrastructure possible. But for now, we see there are some design challenges.

Context You are developing a flexible or configurable piece of software in a typed language like C++.

Problem This pattern addresses several problems conjointly. It fits in, where Property List is not be sufficient. This is the case when you do not just want to keep a map to simple data values or objects, but structured data that also includes sequences of data, or just simple values as attributes or parameters.

How do you provide method parameters or object data attributes that fit the need for future subclasses?

Consider you are passing strcutured data values around. If you use classes to describle these data structures, each extension of the framework requires its own extension of the parameter class. This rises a whole set of other problems, like how and where to instantiate the paramer objects, where and how to specify the concrete data values, when to destroy the parameter objects. Using C++ open parameters like void foo(...) is also no optimal solution.

How do you make different subsystems compile-time independent and allow for “slippage”?

If you are developing a larger software system where subsystems should be modified and extended independently stability of the interfaces is a more important requirement. Such a cross-subsystem interface then requires flexible parameters. The use of a hierarchy of parameter classes is no option, because each extension will impose a re-compilation and eventually changes of both subsystems.

How do you provide a generic configuration or communication data strcuture that is easily extensible?

Look at our example above. For creatig HTML we might not want to specify all renderer parameters within your program code. This would require a re-compile for each data value changed. It would be nice to have a configuration mechanism that allows us to specify renderer objects used and their corresponding parameter values by a simple mechanism like a text file. In addition it would be nice to define a format that allows us to implement the configuration reader in a way that it does not needed to be changed for each framework extension.

If we´ve got such a generic file format with a generic reader, it is easy to extend it with a writer component. Than we can use the mechanisms to pass structured values across process/network boundaries. The receiver might not even be the same kind of program. It will know about what was sent and might only care about a part of the data received.

In particular you want to address the following *forces*:

- You want to pass an open set of structured data to an (abstract) operation.
- You want an internal and external representation of these data-structures that can be used universally.
- Client program code should not need to care about the memory management of these structured data values, even if their memory consumption will require it.
- The built-in type system of the language is not flexible enough to support these issues directly.

Solution Create an abstraction for structured values that is self-describing: Use an Anything. An Anything can represent simple values, like booleans, integers, floating point number, or strings. In addition it can keep an indexed sequence of Anythings or a sequence which allows indexing by names, like a property list.

The implementation of Anythings consists of a handle class Anything, and of a collection of implementation classes all derived from a root class AnyImpl defining the common abstract interface.

Class	***Collaborators***
Anything	client components AnyImpl
Responsibility • Handle class to be passed by value throughout the program. • Maps its interface to AnyImpl. • Provides serialized I/O on character streams.	

Class	***Collaborators***
AnyImpl	Anything AnyIntegerImpl AnyDoubleImpl AnyStringImpl AnyArrayImpl
Responsibility • abstract class defining interface for all Anything implementation types • implements reference counter and thus memory management following Counted Pointer	

To distinguish different stored values the AnyImpl subclasses provide a simple meta-information interface returning the specific kind of value stored. Nevertheless, class Anything allows client program code to retrieve the kind of value from it, as desired. To allow for normal program continuation, each value access method of class Anything provides a default value, which can be specified by the client, that is returned if a data type is retrieved that cannot be usefully converted.

The AnyArrayImpl class is used to keep dynamic growing sequences of Anythings. In addition to indexing via integers, it allows string indices using a hash-table lookup.

Class	***Collaborators***
AnyDoubleImpl	Anything AnyImpl
Responsibility • Anything implemetnation class holding floating point values. • access as string, integer, and boolean (non zero) is defined.	

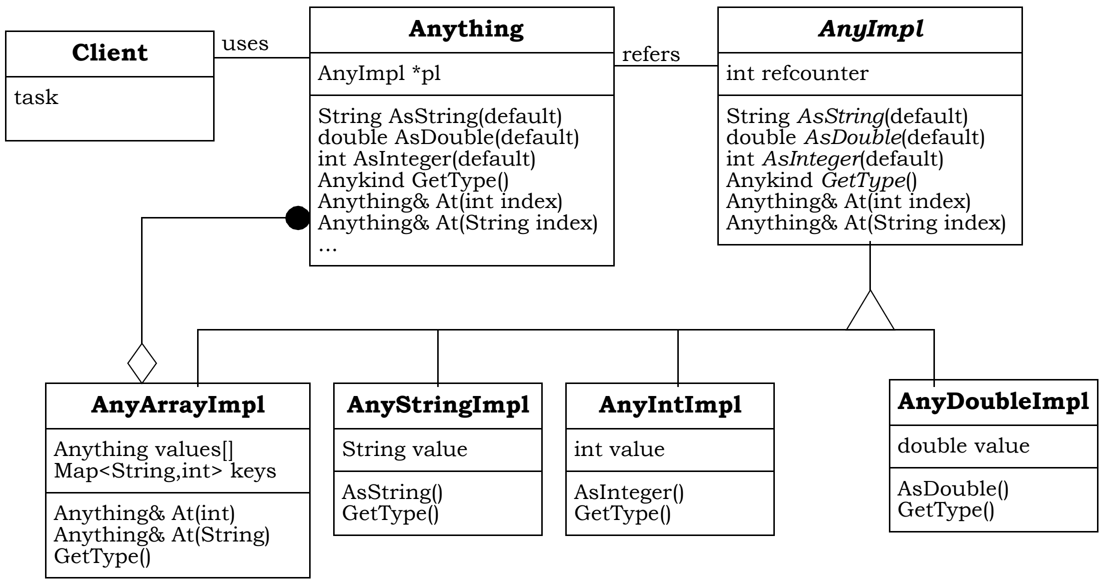
Class	***Collaborators***
AnyArrayImpl	Anything AnyImpl
Responsibility • implements sequences that can be indexed by integers and strings if a string index is defined.	

Additional infrastructure is needed to serialize Anything structures and to parse the file representation of them. A simple grammar is sufficient. An Anything representing some author information might look like:

```
{   # this is a comment, the brace starts a sequence
    # a slash followed by the name denotes a string index
    /name "Peter Sommerlad" # string index, string value
    /age 33 # couldn´t think of another integer
    /nofbooks 0.2 # one fifth of POSA :-)
    /patterns { # just a sequence
    "Property List" "Anything" "CommandProcessor"
    }
}
```

Reading such a text would result in an Anything holding an AnyArrayImpl sequence with 4 elements with index names “name”, “age”, “nofbooks”, and “patterns” respectively. The elements are internally represented by an AnyStringImpl, an AnyIntegerImpl, an AnyDoubleImpl and an AnyArrayImpl respectivly. The last one holds an anonymous sequens of three Anythings, each represented as an AnyStringImpl.

Structure The resulting Anything infrastructure classes follows a handle-body schema, where class Anything represents the monomorphic handle, that is passed by value (in C++). The AnyImpl hierarchy represent the polymorphic body objects, that are refered by the handle. These body instances are normally shared and reference counted. Thus memory management of the Anything content is automatic for the user (in C++).



Dynamics There are no specific interesting dynamics of using Anything.

Implementation To implement the Anything pattern perform the following steps:

1 *Define the Interface and the needed elementary data types required by your Anythings*. A typical Anything implementation supports strings, integers and floating point values as possible data elements within a simple anything. Other data values might be interesting and useful for your specific application as well. The interface of the Anything handle class allows access to any type that could be stored within the Anything. However, to provide a tolerant behavior, we add a default parameter to each access function, that is returned when the concrete value stored cannot be usefully converted to the requested type. For each of the possible elementary types we define a constructor as well. To distinguish the content type of an Anything, we introduce a home-made meta information within the Anything, that can be implemented by an enumeration type that is returned by a GetType() operation. The resulting interface might look like the following excerpt:

```
class Anything {
public:
    Anything();            // constructors
    Anything(int);
    Anything(double);
    Anything(const String&);
    Anything(const Anything &a);
    ~Anything();
    // coding of meta information
    enum EType {
        eNull,
        eInteger,
        eDouble,
        eString,
        eArray
    };
    EType GetType() const;
    Anything &operator= (int);
    Anything &operator= (double);
    Anything &operator= (const String&);
    Anything &operator= (const Anything &a);
    // conversion
    String AsString(const char *dflt= 0) const;
    int AsInteger(int dflt= 0) const;
    double AsDouble(double dlft= 0.0) const;
```

```
        //... some things omitted here
    private:
        AnyImpl *fAnyImpl;
    };
```

2 *Implement the Anything handle class*. The constructors, assignment operators and destructor have to take care about the reference counting (in C++). Otherwise the implementation is straightforward delegation to the AnyImpl object refered to by the Anything handle object.

3 *Implement the abstract implementation base class*. This class (AnyImpl) mimics the Anything interface with corresponding virtual functions (in C++). The AnyImpl abstract class might provide convenient default implementations of the access methods by just returning the default value passed as parameter. In addition it implements the reference counter and methods for its adjustment. Whenever the counter reaches zero an AnyImpl object deletes itself (in C**).

You have to decide how you share the AnyImpl subclasses between different Anything handle objects:

- share it, then any modification by the program is visible everywhere it is used,
- pass it around mostly as read-only data. This can be accomplished by a specific ReadOnlyAnything handle, that doesn´t allow modification to the refered data.
- implement a copy-on-write schema. This is the most advanced implementation and can impose a large run-time overhead if large data structures stored within an Anything.

4 *Decide how to handle empty Anythings*. There are several options to choose from:

- Use a null pointer value and check fAnyImpl pointer for validity before use.
- Implement a specific AnyNullImpl class. This is an application of the Null-Object pattern. You can consider applying also Singleton and share the AnyNullImpl instance (you have to adjust the self-deletion property of AnyImpl´s constructor).

- Use an instance of the AnyImpl class for implementing the case of an empty Anything.

5 *Implement the required concrete implementation classes*. It is usually straightforward. Nevertheless, you need to decide what useful conversions should be allowed. For example, retrieving a string from an internally stored integer is a convenient operation.

6 *Implement the Anything sequence/associative array*. These are conceptually two different things, but can be implemented together. Our experience in using Anythings showed, that the associative array is the more common use for filling in data (e.g. to mimic record structures), but extracting data is done both ways, via associative lookup and via iteration through the defined elements. Simple sequences are treated as anonymous entries in the associative array, just accessible via iteration or numerical indices. The resulting interface for such array access has to be given by classes Anything and AnyImpl:

```
long Append(const Anything &a);
// removal (sequence and associative array)
void Remove(long slot);
void Remove(const char *k);
// associative access
// query number of elements
long GetSize();
// returns slot index
long FindIndex(const char *k) const;
// just check if slot is defined
bool IsDefined(const char *k) const;
// returns name of slot (if any) - meta-info
const char *SlotName(long slot) const;
// accesses element slot in array
Anything &At(long slot);
Anything &At(const char *slotname);
```

For convenient syntax, you might overload the array-access operator in C++ to benefit from convenient syntax.

7 *Define the external format and implement it*. Implement write methods, dumping an AnyImpl on a stream and a corresponding parser reconstructing an Anything from a character sequence on the stream. If you intend to use the external format as configuration data, look for

a human understandable syntax (and possibly a pretty-printing output). The grammar might look like the following (ommitting the details of masking characters within strings and of treating white-space):

```
any  :   [0-9]+
     |   [0-9]+\.[0-9]*(E[+-]?[0-9]+)?
     |   string
     |   *
     |   { anyseq }
anysq:   any anysq
     |   ´/´ string any anysq
string:  ´"´ .* ´"´
     |   [A-Za-z][A-Za-z0-9_]*
```

If you want to ´deep-copy´ an Anything using a in-memory stream can be used to implement this operation.

8 *Decide where in your system you should use Anythings.* Anythings are great in some places as given by the problem statement above. As a flexible parameter passing mechanism in a framework they can be a real life saver. However, you should make your key abstractions in your framework explicit parameters, where used, to reveal more of the intention of your operations. Anythings can be handy to solve covariance problems, where different elements or extensions of your framework co-operate via standardized interfaces. The external format proved to be very useful to specify configuration data in a very flexible way. Such configuration data can be accessed via a Registry or used on a per-object basis like described in Named Object [RuSo98].

Known Uses ET++ (André Weinand)

Beyond SNiFF and SNiFF (Walter Bischofsberger)

Consequences The Anything pattern implies the following **benefits**:

- *Useful as universal data parameter.* Anythings have been proven to work, whenever an open set of data parameters need to be passed around.

- *Allows really useful interfaces without overloading.* A problem with extending a class-hierarchy is, that additional sub-classes might require additional parameter data. However, just adding a method with the same name and a different number of parameters does not provide to be useful, because such overloading is confusing and usually breaks the application of polymorphism.

- *Configuration data can be nicely represented.* Anythings are content extensible. Therefore, the readable external representation fits nicely the needs of typical configuration data.

- *Less complexity due to fewer classes.* Getting rid of parameter classes can reduce the number of elements a programmer needs to understand when using a framework. In addition, Anything as parameter can reduce confusion on where to put functionality, it does not belong in the parameter class.

- *Program to program communication.* Like other data structures that have a defined syntax or are self-describing, the external Anything format is ideal for passing values between programs. In contrast, for example, to Corba-IDL structs giving a fixed data-structure defined as an inter-process interface, an Anything can fulfil current and future needs of data-exchange without requiring a change to the interface syntax.

- *Automatically memory managed (in C++).* The application of the Counted Pointer idiom fits nicely in the structure. Sharing the implementation objects, i.e. the place where the data resides is efficient and restricts copying the data structure to a minimum.

  However, the Anything pattern also has its **liabilities**:

- *Less type safety.* Because an Anything can represent arbitrary data structures, the compiler can no longer check, if parameter passed really fulfil the requirements by the place used. However, because you define your own meta-information interface with GetType(), GetSize(), SlotName(), and IsDefined(), your program can control validity of data-type stored in an Anything at run-time.

- *Intent of parameter elements not always obvious.* Anythings define their own name-space managed by the program you get the same liabilities as Property List regarding the naming of slots. This situation requires carefully created and maintained documentation of the names used.

- *Code overhead for value lookup and member access.* Anythings require overhead for accessing elements by name compared to fixed data structures defined in your programming languages. In C++ operator overloading can help with this situation. But error-handling and determining if names required are defined can bloat the code. Using the default mechanism of the access methods can help a little.
- *Dangling elements in configuration data.* When your system uses large configuration data files and is changing over time, the question if that "/foobaz" thing in your configuration is still used arises quite often. You can either ignore the situation and make life harder for the maintainer of your system or you keep an eye on accurate documentation of the slot names used in your configuration data and expected by your program.
- *No real object, just data.* Anythings are implemented by classes. Nevertheless they represent just structured data. All semantics related to that data is de-coupled from it externally in your system.
- *Run-time overhead can be substantial because of name-lookup.* Especially in contrast to a compiled indexed memory access a name lookup and eventually type-check can be orders of magnitudes more expensive. If you expect slow performance, carefully decide where you have other implementation options.
- *Overuse*. Anythings are not "Everything". Make sure, that you are not ending up using Anything objects for your key abstractions in your system.

See Also A Registry with hierarchical name space can be implemented using Anything. Named Object [RuSo98] can be implemented nicely using Anythings for configuration data.

Credits Thanks go to all Anything inventors, some of them—as far as I was told—developed the idea independently [MäBi96]. I still wonder if combining sequences with hash-table-like access is a good idea or not. I suppose André Weinand and Erich Gamma will know. Special thanks to Andi Birrer who has kept up changing our Anything implementation throughout all our enhancements.

Registry

The Registry design pattern is useful for implementing globally accessible (configuration) data in an extensible way.

Example Consider our example with the Renderer classes from above. We are using a simple interpreter to call the Renderers given by their name. However, the class Renderer is intended to be subclassed, but we do not want to change our interpreter whenever a new Renderer subclass is invented.

How can we guarantee that every renderer called also exists? In contrast how can we ensure that our interpreter knows each new renderer?[2]

A second example in our HTML generator framework relating to this pattern is the global configuration data. Each renderer that needs to generate some URL pointing back to the system must know where the program is running, where images should be located, etc. How can we provide a means for all this shared information to be easily accessible without copying it around to each individual piece?

Context You are developing a flexible piece of software, to be extended in the future. This can be an operating system, a user interface platform or an application framework, for example.

Problem You want to store and access global (configuration) data or objects in a flexible and extensible way. Using global variables in your program is not be appropriate because global variables are considered ‘bad practice’ and initialization of those variables might not be possible after program compilation.

In addition some configuration data might be shared by different programs. Storing this configuration data in simple files can solve the global persistent storage problem, but it requires to implement reading and writing the file format whereever it is used. In addition it might be hard to implement access control to the file. Thus it might

2. In Java it is possbile to use the ClassLoader for this purpose.

get corrupted by concurrent access by several programs or by a misbehaving program.

One option to store such data is a database management system. But using a full-fledged database requires too much development and run-time overhead or too high license costs.

Future extensions of your system might require additional data or objects not known in advance. You do not want to hard-code the number and names of all of your configuration information to allow easy extensibility.

How do you implement a globally accessible and extensible data or object storage?

In particular you want to address the following *forces*:

- You want to access a set of global configuration data or objects.
- You want to extend the system in the future with yet unknown configuration data or objects.
- You want a single mechanism to store and access this configuration data.
- You want to modify the configuration data indepent of the program code using it.
- If you are building a single program, the built-in type system of the language is not flexible enough to deal with these issues directly by keeping track of global variables or objects for you.

Solution Implement a Registry, that maps objects or configuration data names to instances or values. In the case of configuration data, this registry holds all name-value pairs for each configuration element. If you want to keep track of a set of initialized objects the registry stores these objects references accessible by name.

Every configuration element or object is registered with its name and value or reference in the Registry by an installer component. Later on, client components access theses objects or data just using the global registry and the well-known names.

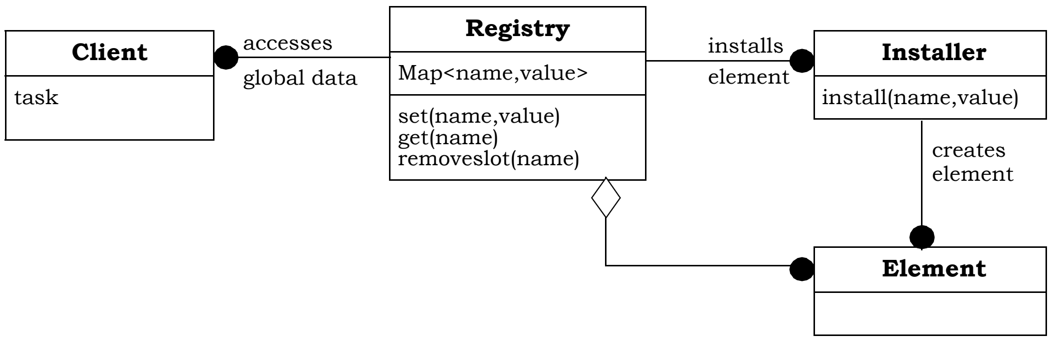
Class Client	***Collaborators***
Responsibility • needs access to global configuration data elements • retrieves data from Registry •	Registry Element

Class Registry	***Collaborators***
Responsibility • provides (hierarchical) storage for configuration data elements • implements retrieval and storage of data elements by name	Installer Client Element

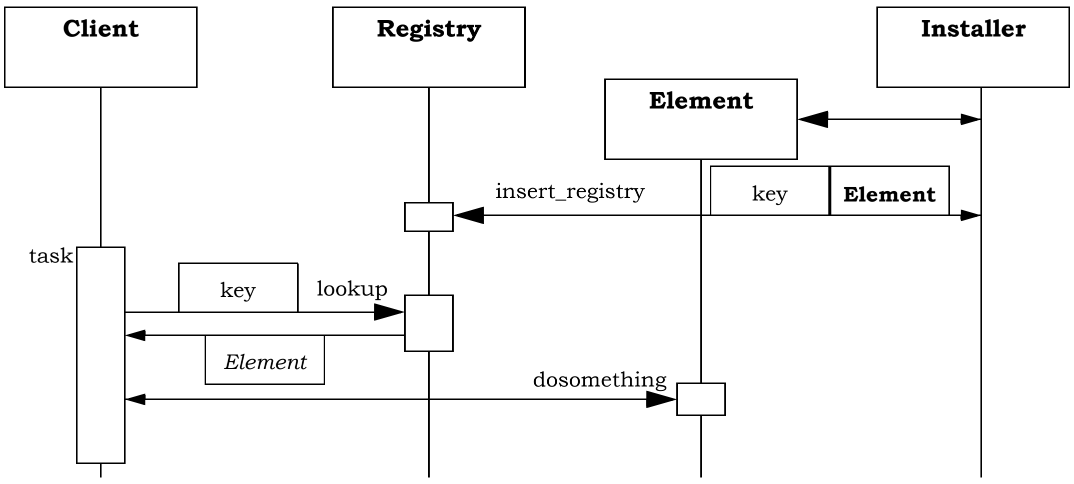
Class Installer	***Collaborators***
Responsibility • register configuration data elements with their names • fills the registry	Registry Element

Class Element	***Collaborators***
Responsibility • abstract component defining data values • can represent strings, numbers or anything	

Structure The client components are usually unaware of the installers, that put the data elements or objects in the global registry. Therefore you get a structure like the following.



Dynamics A typical scenario of using a registry consists of the phases installation and client access. During installation an installer component inserts or updates one or more registry entries. After that clients are able to access the registry and retrieve the names components.



Implementation To implement the Registry pattern follow the following steps:

1 *Define the scope of the registry.* If you are building an operating system or a comparably complex infrastructure, you have to create a registry that can be used by several programs. Thus the access to the configuration data in the registry should be given by your system API. On the other hand, if your target is a single program, consisting of several partly independent components your registry can well be refered to by a global variable or be implemented as a Singleton[GHJV95].

2 *Determine the name space.* The more components share the registry, the more likely it is that they use identical names for different elements. If you predict such name clashes, provide your registry with a hierarchical name space. One solution might be to have two levels, one for the program name the next one for the program specific entries. You get the highest level of flexibility if you allow for arbitrary nesting depth. However, a flat namespace can be easier to implement and maps easily to a simple file for persistency.

3 *Design the default behavior*. When several components are sharing the registry and it has a hierarchical name space separating their configuriation, it can be tedious and inefficient to install the same entries for each component separately. In this case, define a specific default category and modify the lookup procedure such that it consults the default category whenever a name search in a specific category failed.

4 *Implement the persistency mechanism of the registry*. It depends on your registry´s scope if the content needs to be persistent to program endings, system crashes or whatever. In the case you require a simple mechanism to store the registry you need to chose a file format. If the namespace is flat and values kept are simple a file format with a single line per entry can be sufficient. For hierarchical configuration information external Anything format has proven to be an ideal candidate. More complex or elaborated persistency mechanisms up to a database system should be considered if the requirements of the registry users promote their use.

5 *Implement the installers*. Usually installation of registry entries requires almost identical code. Therfore you can provide generic setup programs (in the case of a system wide registry) or installer components (in the case of an in-program registry). In our C++ example we define a preprocessor macro that installs new Renderer components in the registry of Renderer objects. The installer class serves the purpose to define a ‘static’ object that automatically installs an instance of the Renderer class whenever the corresponding renderer object file is linked to a program.

```
class RendererInstaller {
public:
    RendererInstaller(const char *name, Renderer *r);
};
#define RegisterRenderer(name) \
    static RendererInstaller _NAME2_(name,Registerer)\
        (_QUOTE_(name), new name())
#endif
```

Using this infrastructure the implementation file of a class MyRenderer needs to call the macro on top-level scope:

```
RegisterRenderer(MyRenderer);
```

6 *Implement the clients*. Clients using a registry can be implemented straight forward. One major design concern is chosing names. The second larger issue is implementing the default behavior of lookup in the registry fails. Just crashing the program or even the system should´t be your choice if the registry content is not sane. Depending on the available type system, you might need to 'downcast' the reference retrieved from the registry. This also might incur errors.

7 *Implement debugging support.* Finding errors in a system due to wrong registry contents is a tedious task. Especially when the size and structure of the registry hinder its understanding. Therefore you should provide means to analyse the registry, like a viewer, a converter to text format, and a search mechanism. In addition each programmer of client code should be obliged to document the entries that her codes uses.

Variants **Named Object** [RuSo98] is a special application of the Registry pattern together with object initialization by configuration data.

Known Uses Microsoft Windows uses a hierarchically structured Registry to store system-wide configuration data used by all installed programs. Usually setup programs install required information of the installed programs in the registry.

The UNIX operating system provides each process with its so-called environment. This environment stores globally name-value pairs, where each name and value is a string. It is used to pass globally set parameters from parent to child processes.

The X window system provides X client programs with a global registry of settings stored by the X server, that provide default values for user interface parameters in a hierarchical way. The xrdb program can be used to modify the settings for a running X server.

Consequences The Registry pattern implies the following **benefits**:

- *Centralized place for configuration data.* The Registry infrastructure defines a common and reusable way to store and retrieve global and possibly persistent configuration data. Therefore, extensions of your system do not have to re-invent and reimplement their own configuration data storage and retrieval systems.

- *Yet-unknown extensions or objects can be added later on.* The structure of the Registry content can be easily extended to accommodate new entry names and also new kinds of entries.

- *Sharing of configuration data.* Storing configuration data in a global registry makes it easy to share it among different clients. If a user wants to change the default system font it can be done in a single place, instead of individually changing properties of all used programs.

- *Different system behavior by multiple versions of the registry.* The separation of configuration data makes it easier to run a single installed component in different configurations by just exchaning the configuration data in the registry. If the registry allows easy exchange of a complete set of configuration information this is a useful operation. Another option is to have a system installed one time but provide different registries with configuration data to individual users.

  However, the Registry pattern also has its **liabilities**:

- *A global Registry requires mutual exclusion or copying.* Because the Registry is a shared global resource, it either requires locks if modifications are done to it, or each process or thread requires its individual copy. Both ways impose additional run-time overhead.

- *Naming of slots not checked by a compiler ('color' vs. 'colour').* Separate components might use identical names with desatrous results.

- *Semantics of registry entries not given by class code only by clients.* The entries kept in the registry do not have the ability to define where and how they are used.

- *Retirement of no longer useful registry entries almost impossible.* The decoupling of registry entries from the components using it might lead to a lot of dangling entries that are no longer used by anyone. However, implementing garbage collection in a registry requires inspection of all components using it. This is one reason why people suggest re-installing Microsoft Windows from time to time.

- *Name clashes possible.* A Registry cannot restrict the use of useful names. Different subsystems might want to use identical names. You can either cope that situation with strict rules on how to choose names and by hierarchically grouping name spaces. Another way observable is that subsystems generate unique but cryptic names automatically

- *Weak access control.* The Registry is intentionally a global resource. Therefore, any subsystem might access and even change its content. This can result in fatal situations (have you ever tried regedit under windows?) or trojan horses looking for weakly encrypted password information in the registry.

- *Database management system might be the more effective choice.* Some of the above mentioned benefits and liabilities are better balanced by a database management system. If you have to keep track of a huge amount of registry entries, a simple implementation of a Registry might not be able to cope with the situation. Then a DBMS might scale better

See Also Factory Method can be implemented using a Registry of Prototype objects [GHJV95]. A Manager [PLOPD3] also benefits from a Registry holding all managed objects. Flyweight [GHJV95] proposes the use of a registry of flyweight objects.

[Beck96] Kent Beck: *Smalltalk Best Practice Patterns*, Addison-Wesley, 1995

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[MäBi96] Kai-Uwe Mätzel, Walter Bischofberger: *The Any Framework A Pragmatic Approach to Fleibility*, USENIX COOTS Toronto, 1996

[Noble98] James Noble: *The Object System Pattern,* submitted to EuroPLoP 98, 1998

[POSA96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented Software Architecture–A System of Patterns*, J. Wiley & Sons, 1996

[PLOPD3] *Pattern Languages of Program Design 3*, Addison-Wesley, 1997

[Riehle96] Dirk Riehle: *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose,* UbiLab Tecnical Report 97.1.1, 1996

[RuSo98] Marcel Rüedi, Peter Sommerlad: *Named Object,* Proto-Pattern, submitted to Writing Workshop at EuroPLoP 98, IFA Informatik 1998