Parameter Block

Gustavo Patow
Grup de Grafics de Girona,
IIiA, UdG, Spain
dagush@ima.udg.es

Fernando Lyardet
Darmstadt University of Technology
FB20 Telecooperation, Alexanderstr. 6,
D-64283 Darmstadt, Germany
fernando@tk.informatik.tu-darmstadt.de

ParamBlock	Object Behavioral

Also Known As

Data Block, Open Arguments, generic function-binding interface, Undetermined Generic Arguments, Property List as parameters.

Intent

Parameter Block is used to create a generic interface for parameter passing, decoupling the API declaration of the procedures and functions from the type and number of the parameters they receive.

Motivation

The development of complex software relies -among other things- on well defined interfaces or APIs between the different modules. Unfortunately, as development cycles tend to shorten, the need for greater parallelism of development activities makes it more difficult to have well established APIs, since everything is under construction. This uncertainty introduces a coordination problem for the development team due to continuous corrections and adaptations required to accommodate the updated interfaces and as such, it is costly and error prone.

A similar problem occurs when software introduces a plug-in architecture as a mechanism to allow future extensions. In this scenario, it is difficult to determine beforehand the information needs for every possible plug-in. Furthermore, as the number of possible relevant information for the plug-in can be arbitrarily high, it may be not practical to pass all possible parameters. Besides, sometimes it might be necessary that the arguments could be added or removed later, at runtime, in order to avoid cluttering the initial declaration with unused information or allowing expanding the passed parameters list.

We could use something like the C++ open argument list, like:

```
void drawInterface (color& background, ...);
```

This approach is error-prone since the caller is responsible for providing a well formatted parameter list with correct types for each object, and the called function needs to figure out what is passed and how.

Finally, another possibility could be a system of global variables but it will turn out to be cumbersome and will introduce more trouble than providing a different solution.

Context

There are a couple of situations where you would need to resort to this pattern to go through a developing problem:

- You are developing a flexible piece of software that could be extended in the future though a plug-in-like mechanism.
- You want a stable interface shared by different programming teams, but still heavily under development.
- You are also working with a programming language that does not naturally provide support for API extensions at run-time, like C++ or Delphi (contrary to Smalltalk that does allow this sort of run-time extensions).

Problem

How do you allow individual functions to augment or change their state (arguments) at runtime? What should be done when a procedural interface is needed with a generic purpose parameter-passing mechanism? This is especially true when creating plug-ins that need an *undetermined number* of parameters with *undetermined types*. A similar situation occurs when you provide a plug-in interface and the interface cannot determine what parameters the plug-in might require, and it is necessary to provide a fixed API for unknown implementations and clients.

A possible solution on strongly typed languages like C, is to provide a sufficient number of void pointers, and each API call would use as many as needed. Unfortunately, since it is impossible to know in advance how many of those void pointers should be provided, the approach is not practical. Furthermore, using a list of void pointers to the actual parameters is not a good solution, since proper casting should be done for each argument.

A runtime mechanism for accessing, altering, adding and removing arguments or parameters must be provided.

Of course, these problems become even worse **when you are also working with a programming language that does not naturally provide support for API extensions at run-time.**

Solution

Decouple the API declaration from the parameters it receives, to allow a generic procedural interface to be written with an arbitrary number of parameters being sent, each one with an arbitrary type.

A practical way to go through is to define a class **ParamBlock** which is a simple dictionary of pairs (a **MAP** [Sandu] or a **Property List** as parameter [Sommerlad] are good implementations for this), which is the only real parameter of the implemented procedures. Thus, from within the called function (functionToDefine), different parameters are accessed through a generic call with the identifier of the needed parameter as argument.

An **Anything** [Sommerlad] is probably the best solution for storing the parameters, since they are proven as extremely useful universal data parameters, allowing useful interfaces without overloading. If the language being used is strongly typed, like C++ or Delphi, it is possible to use the **RTTIVisitor** pattern [martin-dih] to obtain a type-safe reference to the corresponding subclass. Also, a **Visitor** pattern [GoF] could be used, but probably would add too much complexity to **Parameter Block**, blurring its advantages. Of course, should this be the case, all possible parameters should be descendants of the same Parameter abstract super class. Also, in case of extreme need, a **Casting Method** [Meyers] Pattern could be used.

With **Parameter Block**, the decision of which parameters are passed from the main application to the plug-in is left to the plug-in itself, and not hard-coded by the developers of the application.

On the other hand, **Parameter Block** becomes an indispensable tool when more than one team is using the same procedural API, which in turn could be changing continuously, becoming a moving target which becomes a potential development bottleneck and a serious headache. With **Parameter Block**, the problem is avoided by letting each user of the API to define its own parameters, customizing the procedural interfaces to suit their own needs.

Structure

When applying Parameter Block in your application, you will usually develop an auxiliary dictionary which will be a data structure that maps slot names to values. This dictionary is the only data structure that needs to be passed from the main application to the requested function.
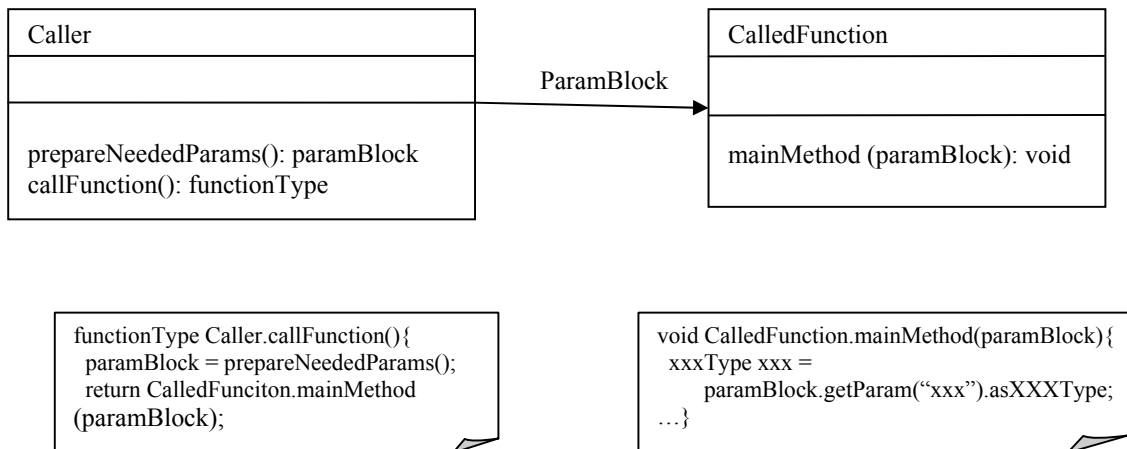
Caller		CalledFunction
	ParamBlock →	
prepareNeededParams(): paramBlock callFunction(): functionType		mainMethod (paramBlock): void

```
functionType Caller.callFunction(){
  paramBlock = prepareNeededParams();
  return CalledFunciton.mainMethod
(paramBlock);
```

```
void CalledFunction.mainMethod(paramBlock){
  xxxType xxx =
     paramBlock.getParam("xxx").asXXXType;
…}
```

Figure: the basic structure of the Parameter Block Pattern, where the interaction between the caller and the called function are shown.

This way, when implementing the method at the plug-in, something like this could be used:

```
void CalledFunction.mainMethod(paramBlock){

  …

  //we need the "XXX" parameter from the block:

  double xxx = paramBlock.getParamNamed("XXX").AsDouble();

  …

}
```

Note that here, implicitly, we are using an Anything Pattern to store the paramBlock elements. It must be mentioned that a parameter naming convention should be established from the very beginning, as it would be impossible to find the right names for each parameter without it.

Implementation

For the implementation of this pattern, a couple of steps should be taken:

1. Evaluate if there is a real need for using the Parameter Block pattern. The main forces that should be checked are
   - You need to provide a runtime mechanism for accessing, altering, adding and removing arguments or parameters. Unfortunately, Parameter Block also would result in a more cumbersome syntax (code bloat).
   - You are also working with a programming language that does not naturally provide support for API extensions at run-time. Unfortunately, this advantage is somewhat counter-forced by a loose of all semantic guidance from the

interfaces and checks from your programming language, and, if implemented, a slower parameter validation mechanism.

- There are several teams working on the same API, and this API is still under development. You must provide a mechanism for API extensions, at the same time providing a stable API to share by all teams working on a code.

It is very important to note that this solution has some drawbacks, as well, since the parameter passing mechanism ceases to be explicit and extra work must be done to have it working properly (parameter checking, completeness, …).

2. Decide the following participants:
   - Decide which the parameter names will be, and use them as are the key values by which the arguments will be looked for in the p*aramBlock*..
   - Called Function: The object with the function that needs the arguments to perform its tasks.
   - Decide which will the `Function Caller` be, since this is the class (or classes) responsible of the creation of the *paramBlock* that stores the passed arguments, and is also the subsystem responsible of registering the references and providing the corresponding arguments at runtime.

   It is also important no to forget to defina a *Dictionary*, which is a random access data structure where the arguments are stored and later retrieved by their *id* (probably just the parameter name).

3. In order for this mechanism to be completely generic, a registration of the parameters needed by the function should be performed first:

```
void CalledFunction.init(funcitonCaller){

  funcitonCaller.register("nameOfNeededParameter1");

  funcitonCaller.register("nameOfNeededParameter2");

  ...

}
```

   The advantage of using this pre-registering mechanism is that the caller knows in advance exactly which parameters will be needed by the *CalledFunction*, being able to implement mechanisms to avoid time/memory waste. This registering process should be done as soon as the plug-in or library is loaded, but could be delayed to any time **before** any call to the *CalledFunction*,

4. When calling the implementations for this), which is the only real parameter of the implemented procedures. In C++

```
funtionType CalledFunciton.functionToDefine
                          (ParamBlock& paramBlock){

  ...

}
```

5. The paramBlock is built by the Parameter Server by using the registered needed arguments for the calledFunction.
6. From within the called function (functionToDefine), different parameters are accessed through a generic call of the form

```
neededParam = paramBlock.getParam("name");
```

7. If the Anything Pattern is used for storing the elements in the paramBlock, the previous line would change to get advantage of the respective conversion functions:

```
neededType neededParam
            = ParamBlock.getParam("name").AsNeededType();
```

8. If more than one function will use this mechanism, the registration should be procedure-based. In this case, the paramBlock dictionaries would use triples of the form (FunctionID, ArgumentID, Value), which could be easily implemented as nested lists.

Variants

Property List: This pattern [Sommerlad] is used for attaching a flexible set of attributes to an object in run-time. Each attribute is given a name represented by a data value, and attributes can be added or removed on a per object basis.

Known Uses

- **Maya SDK:** Maya is a professional 3D modeling and animation system used in the professional film industry which uses this mechanism to add new functionality through user-defined plug-ins. Each plug-in register the parameters needed in a function **Initialize** and the method **Compute()** receives a structure called **dataBlock** which contains the needed run-time parameters. To downcast, Maya uses a CastingMethod pattern [meyers]. For example, to create a surface shader plug-in called *PhongNode*, the method *compute()* (the called function) receives a block of type *MDataBlock* that contains all the passed parameters. When the called function needs any parameter, it retrieves it from the *block* and converts it to the desired type (labeled with *(1)* in the code below). Maya's API also allows output parameters (marked with *(2)* in the code) to be returned this way:

```
MStatus PhongNode::compute(const MPlug& plug, MDataBlock& block ) {

     ...

    (2) MFloatVector& refrColor
            = block.inputValue(aRefractedColor ).asFloatVector();

     ...

    (1) MDataHandle outColorHandle = block.outputValue( aOutColor )

      MFloatVector& outColor = outColorHandle.asFloatVector();
```

```
            …

    }
```

- **Apache http server:** it provides a similar structure for its module extension API. thus allowing a modules separate functions that are called within the phases of request processing to communicate within this structure, without the surrounding code needing to know about the module's needs.

Consequences

Advantages:

- **Generic procedural API:** You can write a generic procedural interface that allows an arbitrary number of parameters being sent, each one with an arbitrary type. This is also good for Object Oriented Framework hook methods.
- **API changes at runtime:** You can add and remove arguments at runtime.
- **Less impact of API changes on development:** Parameter Block allows a degree of change without compromising communication between different modules, thus leveraging coordination effort of the development team.
- **Testability**: a generic testbed might be easily constructed by using configuration data to fill in the paramblocks.
- **Parameter Editor:** It is easy for you to build a parameter editor, since the information can easily be retrieved.
- **Arguments Evolution:** Arguments can evolve to first-class arguments as an application evolves. This is the case when an argument is recognized to have a recurrent appearance across different implementations.
- **Iterating on the parameters:** You would be able to traverse and, perhaps, process, iteratively the parameters in a function.

Disadvantages:

- **Code Bloat:** Syntax is more cumbersome in the absence of reflective support: normally, access to arguments is quite straightforward, but with this pattern it becomes different, more verbose.
- **Overdose Danger:** if you are only using one generic parameter, you loose all semantic guidance from the interfaces and checks from your programming language. Checking the types and correctness of the passed parameters becomes much more difficult.
- **Slower parameter validation:** If you implement mechanisms to detect wrong supply of arguments you might find that they are much slower. An example would be Smalltalk's "doesNotUnderstand:", which can be implemented to trap incorrectly formatted messages, can be orders of magnitude slower since the dictionary must be thoroughly checked for each call.
- **Runtime Overhead:** You will access individual parameters in a slower way than with conventional parameter passing.

See also

- Polymorphic parameter passing in general, since Parameter Block is a specific variation of the theme for passing arbitrary data around in a structured way.
- **Casting Method** [meyers]: use this in case of extreme need to downcast the parameter object.
- **Comand** [GoF]: This pattern encapsulates a request as a parameterized object, decoupling an object from the operations performed on it, while Undetermined Generic Arguments decouples the procedural interface of a command from the parameters it needs to work.
- **Arguments Object, Selector Object and Curried Object** [Noble]: these three patterns are intended to *simplify* an existing but complex protocol. Thus the need to know *in advance* the arguments a function must be provided.
- **Generic function-Binding Interface** [Bilas] is the use of two simultaneous patterns: Undetermined Generic Arguments and Command, but the Undetermined Generic Arguments is used with ordering in the dictionary as the key to access each parameter.
- **Accumulator** [Yelland]: is a variant of the Curried Object [Noble] which simplifies the protocol used to create objects.
- **MAP** [Sandu]: is one of the best way to represent the dictionary that holds the parameter collection
- **RTTIVisitor** [martin-dih] is what is used to avoid casting inside the defined function that uses this parameter passing mechanism. As mentioned above, **Visitor** [GoF] could also be used, but could easily render the solution unfeasible.

Acknowledgements

Bibliography

[Bilas]   Scott Bilas "A Generic Function-Binding Interface", in Game Programming Gems, Mark DeLoura Ed., Charles Rivers Media, pp. 56--67, ISBN 1584500492

[GoF]   Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, October 1994.

[martin-dih] "Dual-Inheritance Hierarchies", in "Design Patterns for Dealing with Dual Inheritance Hierarchies in C++", Robert C. Martin, C++ Report, April, 1997. Available at: http://www.objectmentor.com/

[meyers]   Effective C++, Scott Meyers, Addison-Wesley Publishing, 1992. ISBN 0-201-56364-9

[Noble]   "Arguments and Results" James Noble, Plop 1998

[Sandu]   "Collection Patterns" Dorin Sandu, Plop 2001

[Sommerlad]   "Do-it-yourself Reflection", Peter Sommerlad and Marcel Rüedi, EuroPlop 1998

[Yelland]   "Creating host compliance in a portable framework: a study in the reuse of design patterns". In Proceedings of 10th Conference on Object-Oriented Programming Languages, Systems and Applications, San Jose, CA, 1996