External control

An object-structural pattern for mapping hierarchical structures to external control mechanisms

Guntram Berti
berti@math.tu-cottbus.de
Numerische Mathematik & wissenschaftliches Rechnen
BTU Cottbus

Technical Report NMWR-98-1
October 8, 1998

Abstract

Controlling parameters is a necessity in every non-trivial application. We present a pattern that aids in decoupling external control mechanics from the internal component structure, while supporting the automatic mirroring of internal structure into external representation. Furthermore, we show how to use this pattern in the implementation of a **C++** component for stream-based control that has been used successfully in Scientific Computing applications.

1 Introduction

A common problem in the development of software applications is the necessity to control parameters at program run time. These parameters typically exhibit an internal structure determined by the compositional hierarchy of the involved components, for example the classes in an object-oriented programming style. A classical way of parameter control under UNIX is by command-line parsing. However, a traditional parse-loop approach has several drawbacks:

- Knowledge about parameters has to be centralized in the command-line parsing loop, such that the internal component structure is mirrored explicitly in the program code, contradicting efforts of detail encapsulation.
- In the loop code, there has to be explicit write access to internal data.
- It is difficult to extend this approach to another way of parameter control.

A more modern approach to parameter control is the construction of a graphical user interface (GUI). The following remarks apply in this case:

- We must pay attention not to hard-wire the details of a particular GUI-framework into the application components, as GUI components typically tend to depend strongly on the environment, thus creating obstacles to portability.
- We often want not only a GUI, but also the possibility to run applications in batch mode, supplying the parameters by non-interactive methods.

A general mechanism for controlling program parameters should therefore satisfy the following requirements:

- Knowledge of controllable parameters is localized to the class owning them.
- There is no need to specify explicit mutation operations for these parameters in the class interface, which would make them accessible to components other than the parameter controlling component.
- The natural hierarchy of class or component aggregation is preserved by the control component, changes in the internal aggregation structure of components stay limited to these components. No user interface code has to be adapted to these changes.
- The concrete nature of the external parameter supply mechanism remains opaque, giving the possibility to use several mechanisms at once.

We present in the following a pattern supporting this type of parameter control. The format of the papers remainder essentially follows that one introduced in [3].

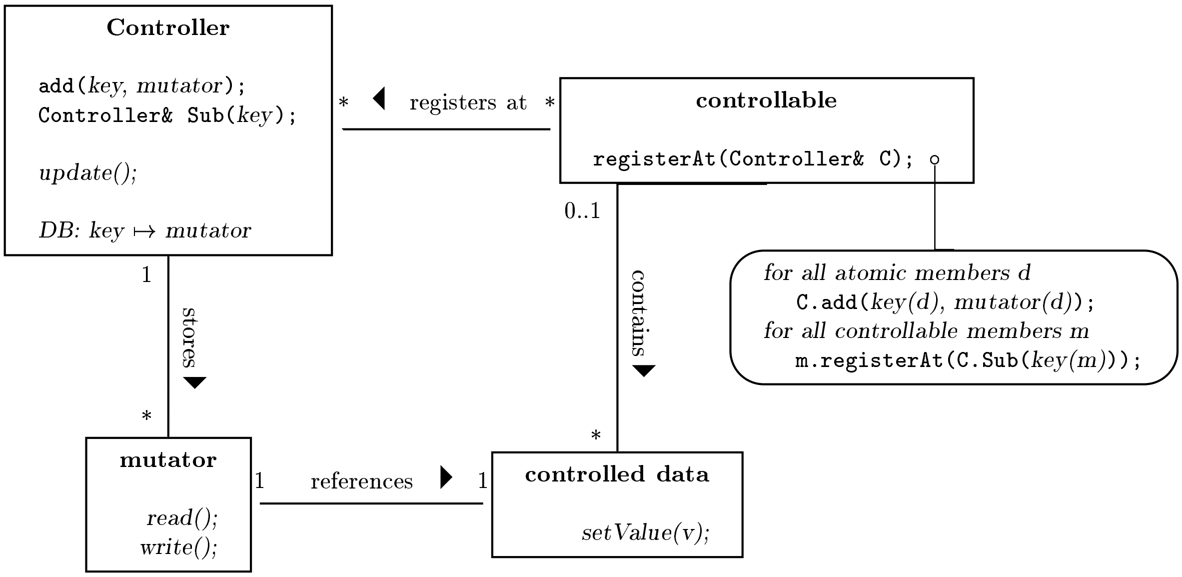

Figure 1: Structure of participants. Methods typeset in *slanted face* are not part of the *external control* pattern.

2 The *external control* pattern

2.1 Intent

Give a unified way of controlling parameters by external mechanisms, preserving the natural hierarchical structure of data, and without committing to a particular technique of parameter control.

2.2 Motivation

As pointed out in the introduction, ad-hoc parameter control for complex components can lead to inflexibilities and error-prone, tedious programming work. Therefore, it is desirable to have a unified way of providing run-time control to program parameters that automatically mirrors the internal component structure to an abstract parameter control device. *Automatically* means that the external control structure is built at run time without the need to expose the internal structure of the components involved.

2.3 Applicability

This pattern is most useful if large sets of hierarchically structured parameters have to be controlled in a uniform manner by an external mechanism that is unrelated to the components owning the parameters. It works best if there are few if any interdependencies between parameters.

2.4 Structure and Participants

The main participants are:

- **controller** : A component that encapsulates a database of program parameters to be controlled externally. This class hides the actual mechanism of parameter setting, and may be seen as a parameter control *server*. Internally, it maintains a sort of mapping between *keys* (e.g. strings) and generalized references to parameters, so-called *mutators* (see below). The main functionality includes:
  - `add(key,mutator)` for enlarging the mapping
  - `Sub(key)` for adding a new layer of hierarchy
- **controllable aggregate** (or *controllable* for short): A class containing data to be controlled externally. A *controllable* may be regarded as a *client* of *controller*. This aggregate may consist partly of *controllables*, giving raise to another hierarchical layer, and partly of data considered atomic in this context and termed *controlled data* for out purposes. Functionality includes:
  - `registerAt(controller&)` for adding local parameters to the *controller*
- **controlled data** is a *role* played by components or simple data types considered

atomic by the pattern. Typical examples for *controlled data* are build-in data types. The required functionality depends on the *mutator* type chosen and is not part of the pattern.

- **mutators** are abstract references to instances of *controlled data*. A *mutator* is responsible for performing the actual action of assigning values to *controlled data*, and acts as a *proxy* (see [3]) between `controller` and *controlled data*. Thus it is possible to introduce additional actions such as call-back triggering in the case of value assignment to *controlled data*.

2.5 Collaboration

A typical usage scenario includes:

- A *controller* tied to a physical device (stream, GUI)
- A *component owner* triggering the registering of top-level components to the *controller*
- A *user* triggering the parameter update in the *controller*, e.g. a "real" user clicking "OK" in a window or the command that reads in a parameter file

It has been left open how a *controller* triggers an update of its mutators or how a mutator actually changes the value of the data it references. An example for a stream-oriented implementation which can be adapted to a GUI is given below.

2.6 Consequences

The *external control* pattern has the following consequences:

- Once existing classes with parameters that are to be controlled are adapted to the *controllable*-interface, they typically do not have to be changed when new types of *controller* are created, except to profit from new features (e.g. using a slide-bar for a real-valued parameter)
- The name of an entity inside a hierarchy is determined by the context, i.e. the possessing class, which is usually better than letting the class itself decide that.
- Name conflicts may occur, especially when the context decides not to introduce another level of hierarchy, i.e. passes the *controller* itself (instead of an appropriately named *sub-controller*) to its members.
- Difficulties arise if mutual constraints of class parameters are complex.
- If changing a parameter has to trigger additional action, some notification mechanism using call-backs must be created.

2.7 Implementation

The following issues have to be considered when implementing the pattern:

- *controller* must itself be able to play the role of *controlled data* to enable hierarchical composition.
- When implementing the *mutator* functionality, additional requirements will arise for *controlled data*, for example the ability to be read from a stream.

2.8 Sample Code

We have a developed a **C++** version that implements the pattern for data streams as external control mechanism, and that should be extensible to an interactive device. The corresponding refinement of the *controller* concept uses STL associative containers (see [4]) for storing name-mutator pairs, in *controller*, and *mutators* employ stream input operators to get values:

```
class Mutator {
public:
  virtual void read (istream& in)        = 0;
  virtual void print(ostream& out) const = 0;
  virtual ~Mutator() {}
};
```

A very simple implementation is provided by the following template:

```
template<class T>
class TypedMutator : public Mutator {
protected:
  T& v;
public:
  TypedMutator(T& vv) : v(vv) {}
  virtual void read(istream& in)
    { in  >> v;}
  virtual void print(ostream& out) const
    { out << v;}
};

template<class T>
inline TypedMutator<T>* GetMutator(T& t)
{ return new TypedMutator<T>(t);}
```

Hence, for a *controlled data* object of type `T`, the following operators will be required:

```
istream& operator>>(istream&, T      &);
ostream& operator<<(ostream&, T const&);
```

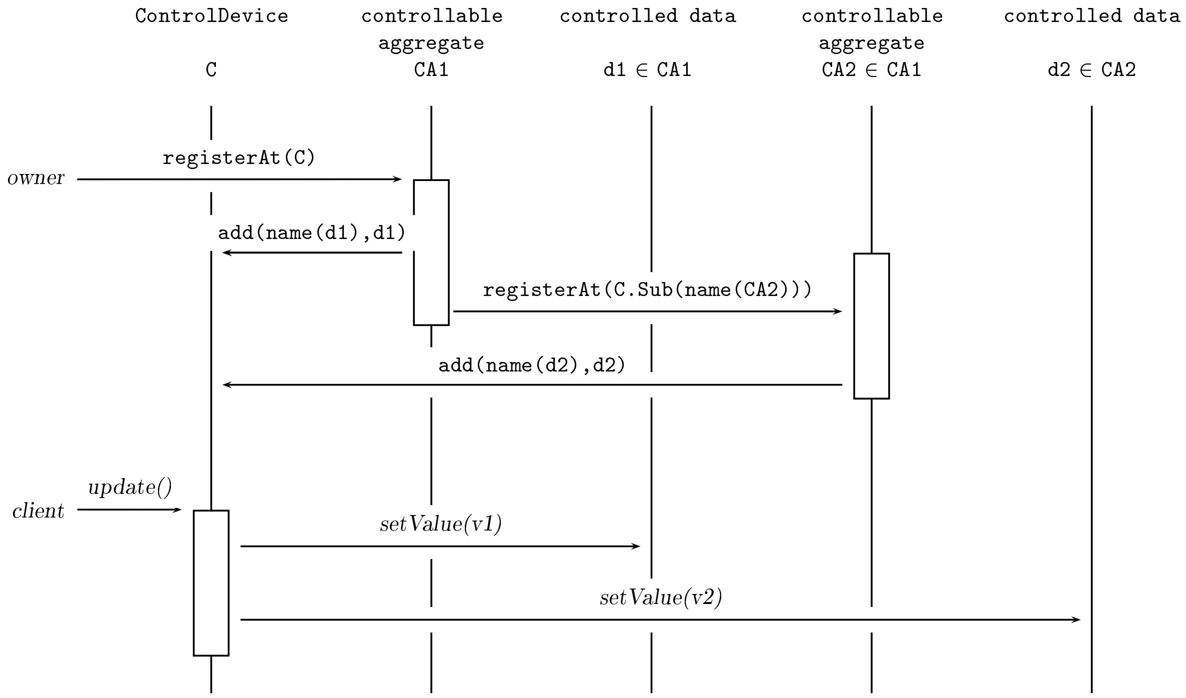

Figure 2: collaboration diagram for *external control*

An abstract base for *controllable* is given below. In order to register at a `ControlDevice`, it is not required for a class to derive from this base; however this is preferable if the class is to be used in a polymorphic context.

```
class controllable {
public:
  virtual void registerAt(ControlDevice&) = 0;
  virtual ~controllable()                 = 0;
};
```

Finally, `ControlDevice` implements the *controller* concept using a *letter-envelope* pattern (see e.g. [2]):

```
class ControlDevice  {
  ctrl_dev_impl* impl;
public:
  ControlDevice(ctrl_dev_impl* imp = 0)
    : impl(imp) {}

  void          add(string const& nm,
                    Mutator*      value);
  ControlDevice SubDevice(string const&);

  void update();

  void registerAt(ControlDevice&);
};
```

Note that the class `ControlDevice` itself has a `register_at` method, thus allowing recursive construction of hierarchies. The `update()` method triggers the reading of the underlying stream(s).

A typical piece of code using this stream-oriented `ControlDevice` looks like the following:

```
int main() {
  ControlDevice Ctrl = FileCtrl("ctrl.in");

  int nx,ny;
  Ctrl.add("no-of-x-values",GetMutator(nx));
  Ctrl.add("no-of-y-values",GetMutator(ny));

  class PlotFunction;
  PlotFunction.register_at(Ctrl);
  Ctrl.update();

  /* program code starts here */
  PlotFunction.plot(nx,ny);

  // ....
}
```

A typical implementation of a `controllable` class looks like this:

```
class A : public controllable {
private:
  int    a; // controlled data
  double x;
```

```
  B       b; // controllable aggregate

  // ...
};

void A::register_at(ControlDevice& Ctrl) {
  // flat registering
   Ctrl.add("x",GetMutator(x));
   Ctrl.add("a",GetMutator(a));
  // mirror hierarchy in Ctrl
   ControlDevice B_Ctrl=Ctrl.subDevice("B");
   b.register_at(B_Ctrl);
}
```

2.8.1 Extensions and Refinements

As **C++** streams already represent a kind of abstraction, this implementation directly covers parameter control from files and command lines simultaneously – we simply create a stream from the command line arguments and attach this stream to the `ControlDevice`. Also, an implementation of `ControlDevice` that supports user-edited text fields in a GUI framework should not be to difficult.

The most promising place for extensions, however, is the interplay between *mutators* on the one hand, and the referenced controlled data and their enclosing *controllable aggregate* on the other hand. For example, we can replace the direct assignment to controlled data with an indirect assignment, giving the *controllable aggregate* the possibility to plug in appropriate action:

```
template<class T, class AOp>
class VirtualAssignmentMutator
 : public mutator {
 T    t_tmp;
 AOp  assign;
public:
  virtual void read(istream& in)
    { in >> t_tmp; assign(t_tmp);}
  // ...
};
```

If in a *controllable aggregate* there is an item x that has to stay within some bounds, say $x_{\min} \leq x \leq x_{\max}$, we can achieve this as follows:

```
Ctrl.add("x",
         GetAssignMutator(
           BoundCheckingAssign(x,xmin,xmax)));
```

This, of course, gives just an impression of what can be done in this context. Especially, in the presence of dependent parameters or mutual constraints, a system of appropriate call-backs can be inserted at this place.

2.9 Known Uses

The component described above was successfully used in some applications for the numerical solution of partial differential equations, see [1] and [5].

2.10 Related Patterns

External control can be seen as refinement of the *Control* part of the well-known *model-view-control* (MVC) pattern. Like the *composite* pattern, it enables the hiding of complex internal structure. It also bears some resemblance to the *visitor* pattern, albeit its purpose is somewhat inverse. The dependance of the interface of *controler* – the *visitors* counterpart – on concrete types of *controled data* is not very elaborate, at least in our example, thanks to doing "just" I/O and the use of templates. For more information on these patterns, see again [3].

References

[1] G. Bader and G. Berti. Design principles of reusable software components for the numerical solution of PDE problems. In W. Hackbusch and G. Wittum, editors, *Concepts of Numerical Software (to appear)*, Braunschweig, 1998. GAMM, Vieweg Verlag.

[2] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[5] K. Schenk, G. Bader, and G. Berti. Analysis and approximation of multicomponent gas mixtures. In M. Feistauer, K. Kozel, and R. Rannacher, editors, *Proceedings of the 3rd Summer Conference Numerical Modelling in Continuum Mechanics*, Prague, 1997.