Callback Implementations in C++

patterns and complete source code available at http://www.primenet.com/~jakubik/callback.html

Abstract: This paper decomposes the callback solution proposed in [Hickey] to show all of the patterns involved in transforming a simple solution for having two separate objects talk to each other to a sophisticated callback solution. After exposing all of the patterns in Hickey's callbacks, the paper shows how the decomposition exposes a decision point where a more common pattern could be applied. By applying this pattern, the solution evolves into a different callback library that, while slightly less efficient, is simpler and more flexible than the callbacks presented in [Hickey].

INTRODUCTION

Callbacks are most easily described in terms of the telephone system. A function call is analogous to calling someone on a telephone, asking her a question, getting an answer, and hanging up; adding a callback changes the analogy so that after asking her a question, you also give her your name and number so she can call you back with the answer. In this paper, the original caller is the *client*, and the callee who calls the client back is the *server*.

Callbacks have a simple goal, but a wide variety of implementations. All callback solutions allow the server to call back to a wider variety of clients without the server needing hard coded knowledge about the client.

This paper does not describe callbacks as a general pattern, instead it focuses on showing a variety of callback solutions and the patterns used to create them. A more general discussion of callbacks can be found in [Berczuk]. Some examples of where callbacks are useful can be found in [Lea], and [Schmidt95a].

The range of callback solutions is organized as an evolution. First the paper shows how a simple client and server interaction can be evolved, by applying patterns, into the callback solution presented in [Hickey]. The paper then shows how alternative patterns can be applied to form a new branch in the evolution that leads to a simpler and more flexible solution. Finally the paper compares Hickey's solution with the simpler solution to determine which solution is better, and thus which patterns lead to better solutions.

Many of the callback solutions presented in this paper are ideal solutions for different contexts. Most of these solutions are specific to C++. Features of C++ such as static typing, automatic type coercion, manual memory allocation and deallocation for heap based objects, pointers, references, and value semantics lead to challenges and solutions that won't be found in most languages.

The assumed goal is to create a callback library that can be used to allow library code to call client code. A further goal is for the callbacks themselves to be reusable so that many different libraries could use them. All of the callback solutions are for use within a single process; inter-process communication and persistence are beyond the scope of these patterns.

FORCES[1]

A common set of forces is identified for all of the callback implementations so that all implementations can be evaluated against the same criteria.

- *Automatic:* Users should be able to use the library without writing extra code, even when adding new clients and servers to their system. Flexibility should be in the library, not in adapters written by users.
- *Flexible:* The library should support callbacks to functions, function objects, and member functions of objects.
- *Loose Coupling:* Libraries should be independent of client code.
- *Not Type Intrusive:* Users should not have to alter their class hierarchies to use the library. This is important for users trying to use two different libraries they don't have source code for. Type intrusion only occurs in statically typed languages such as C++.

---

[1]Many of the forces identified here were originally identified in [Hickey] as requirements for a generic callback mechanism.

- *Simple Interface:* The library's interface should be simple and easy to use.
- *Space Time Efficiency*: The solution should be compact and fast.

A NOTE ON GLUE OBJECT SCENARIOS

This paper uses GLUE object scenarios to show the design of the callback solutions. Figure 2 shows examples, more can be found in [Huni]. Some changes were made to the GLUE object scenarios as seen in [Huni]: the notation was extended so that pointers to functions and member functions could be shown explicitly, the notation was altered to make it easy to draw with CorelFLOW. GLUE object scenarios were chosen to show implementation details without including all of the source code in the body of the paper.

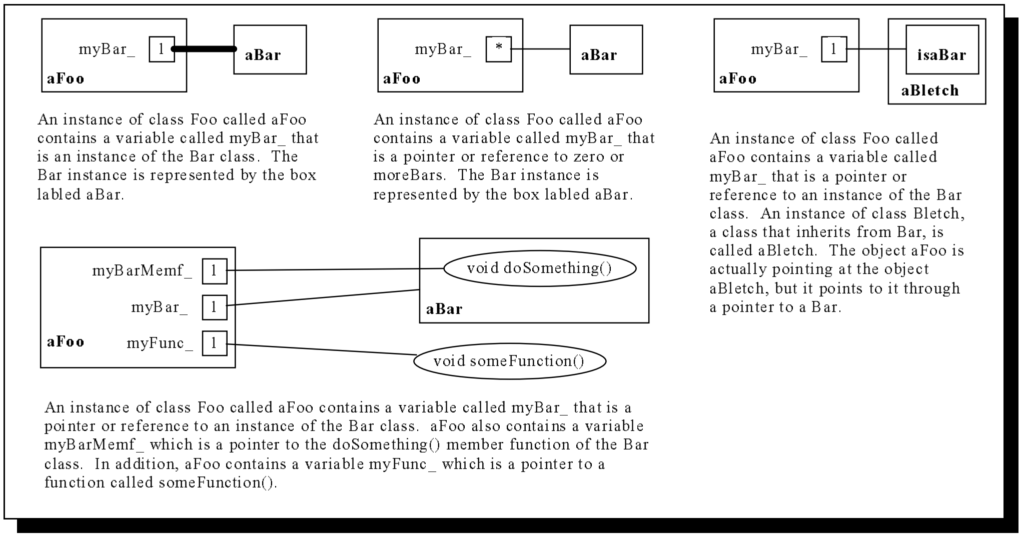

Figure 1: Examples of GLUE object scenarios and descriptions

OVERVIEW

Figure 2 shows the patterns in the order that they are applied in this paper.

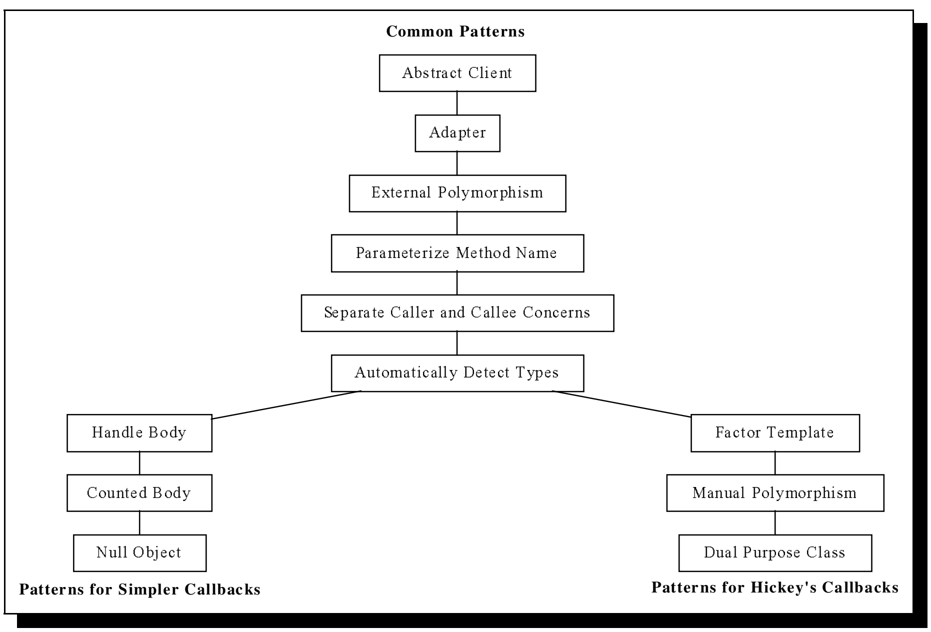

Figure 2: Order of pattern application showing alternate solutions as branches.

SIMPLE SOLUTION

The examples show implementations of a Button object and a CdPlayer object.[2] The Button is a generic library object while the CdPlayer is part of the client code. The CdPlayer has a playButton and a stopButton.

When the Button is pushed, it tells its client that it has been pushed. How a Button gets pushed is not important. The important thing is that Button will need some way to tell the CdPlayer that it was pushed.

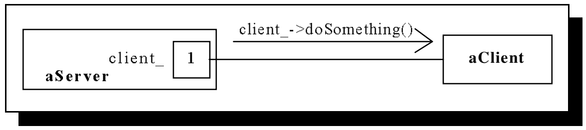

Figure 3: Abstract view of simple solution

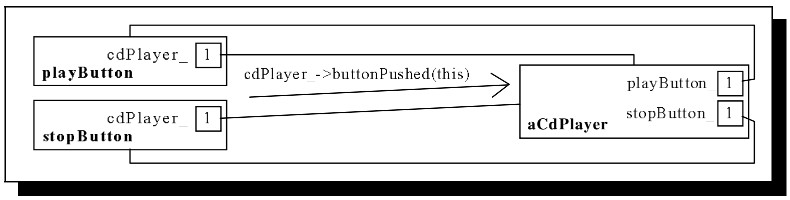

Figure 4: View of actual simple application

Figure 3 shows an abstract view of this solution. One object knows about the existence of another object and can tell it to do something.

Actually implementing an application in Figure 4 is more complex. There are two Buttons now. When pushed, each Button will tell the CdPlayer the same thing: `buttonPushed(this)`. The CdPlayer needs a way to tell which button was pushed, so Buttons always pass a pointer to themselves as part of the message. The CdPlayer stores pointers to its Buttons so it can tell which Button was pushed.

The simple implementation is limited; the client and server are tightly coupled so no part of the solution can be reused as is. What the CdPlayer has to do to figure out which Button was pushed is also a problem.

source

```
#include <iostream.h>
class Button;
class CdPlayer
{
public:
    void buttonPushed(Button* pButton_)
    {
      if (pButton_ == _pPlayButton) this->playButtonPushed(pButton_);
      else if (pButton_ == _pStopButton) this->stopButtonPushed(pButton_);
    }
    void setPlayButton(Button* pButton_) {_pPlayButton = pButton_;}
    void setStopButton(Button* pButton_) {_pStopButton = pButton_;}
    void playButtonPushed(Button*) {cout << "PLAY" << endl;}
    void stopButtonPushed(Button*) {cout << "STOP" << endl;}
private:
    Button* _pPlayButton;
    Button* _pStopButton;
};
//----------------------------------------
class Button
{
public:
    Button(CdPlayer* pCdPlayer_): _pCdPlayer(pCdPlayer_) {}
    void push() {_pCdPlayer->buttonPushed(this);}
private:
    CdPlayer* _pCdPlayer;
};
```

[2]Using Buttons and CdPlayers to demonstrate callbacks is an idea taken from [Hickey].

```
//--------------------------------------
main()
{
    CdPlayer aCdPlayer;
    Button playButton(&aCdPlayer);
    Button stopButton(&aCdPlayer);
    aCdPlayer.setPlayButton(&playButton);
    aCdPlayer.setStopButton(&stopButton);
    playButton.push();
    stopButton.push();
    return 0;
}
```

force resolution

- *Automatic:* No additional code has to be written outside of the client and server.
- *Flexible:* Not flexible at all.
- *Loose Coupling:* The client and server are tightly coupled, they can't be used separately.
- *Not Type Intrusive:* Resolved.
- *Simple Interface:* Resolved except the CdPlayer must figure out which Button was pushed.
- *Space Time Efficiency:* Resolved except for minor problems: extra space is used to store button pointers in the CdPlayer, and extra time is used to decide which Button was pushed.

ABSTRACT CLIENT

In this section, the Abstract Client pattern [Martin] is applied to remove the biggest problem: the coupling of the Button and CDPlayer. This technique is widely used and many patterns are built using it. Many patterns found in [Gamma] including Bridge, Chain of Responsibility, Observer, and Strategy use Abstract Client. The Reactor Pattern [Schmidt95a], the Completion Callback pattern [Lea], and the Conduits+ framework [Huni] are heavy users of Abstract Client as well. This is not an exhaustive list. This pattern is central to object oriented programming, and it is the easiest callback pattern to apply.

In C++ there are two options for specifying the abstract client interface: the specification can be made explicit by creating an abstract base class that all clients must inherit from, or the specification can be made implicit by templating the server by the type of the client so that how the server uses the template type specifies the client interface. [Martin] only mentions the explicit implementation. The implicit version of this pattern is the foundation of STL [Musser] and is mentioned as an alternative to explicit implementation in the discussion of the Strategy pattern [Gamma].

The Abstract Client pattern is often built into languages. Dynamically typed object oriented languages support run time checked implicit Abstract Clients. In dynamically typed languages, this is so much a fundamental way of how the language works that users would not consider this a pattern as much as a way of life. Java provides an interface facility especially for specifying Abstract Client interfaces. Even non-object-oriented languages use techniques similar to Abstract Client; the equivalent in C is passing function pointers.

It is important to notice how common Abstract Client is because everywhere it is used, any callback implementation could be used. The broadness of the callback interface is the biggest reason for choosing different implementations. If the abstract client defines an interface with five or more methods, you should use one of the first three patterns in the evolution: Abstract Client, Adapter [Gamma], or External Polymorphism [Schmidt95b]. If the Abstract Client only defines one or two methods, the final solution in this paper or Hickey's solution would be best. Of course, if you just want the simplest and most straightforward callback, Abstract Client is the pattern for you.

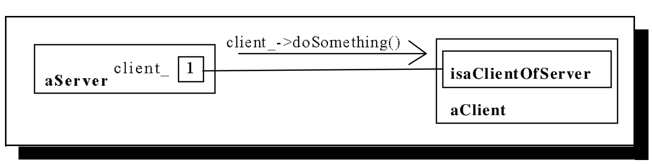

Figure 5: Abstract view of Abstract Client transformed solution

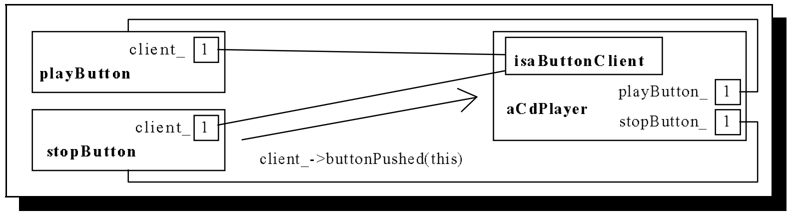

Figure 6: View of actual transformed application

Figure 5 shows the explicit implementation of Abstract Client. ClientOfServer is the abstract client interface from which all Clients must be derived. Servers only see the abstract interface instead of the actual client type. The implicit Abstract Client implementation will be used as part of the External Polymorphism pattern in a later section.

Figure 6 shows how well this pattern works. Buttons now only know about ButtonClients. The Button and the ButtonClient interface can now be part of a reusable library that can be used without the CdPlayer.

source

The most major change is the addition of the ButtonClient class.

```
class Button;
class ButtonClient
{
public:
    virtual void buttonPushed(Button*) = 0;
};
```

The Button class is written in terms of the Button client instead of in terms of the CdPlayer.

```
class Button
{
public:
    Button(ButtonClient* pClient_):_pClient(pClient_){}
    void push() {_pClient->buttonPushed(this);}
private:
    ButtonClient* _pClient;
};
```

The CdPlayer now inherits from ButtonClient; the ButtonPushed method is now virtual by virtue of inheritance.

```
class CdPlayer:
    public ButtonClient
{
public:
    /*virtual*/ void buttonPushed(Button* pButton_);
   // ...
};
```

force resolution

- *Automatic:* If clients can inherit from the abstract client interface and the abstract interface suits the needs of the client, there is no need to write additional code. The next section shows how an adapter can be added if this is not the case.
- *Flexible:* This solution is not flexible on its own.
- *Loose Coupling:* Mostly resolved. The server no longer needs to know the type of the client.
- *Not Type Intrusive:* The client must alter its type by inheriting from an abstract base class.
- *Simple Interface:* No change.
- *Space Time Efficiency:* A small amount of time overhead is added by trading a direct method call for a polymorphic method call. None of the callback solutions will be more efficient.

ADAPTER

The Adapter pattern [Gamma] can be used to make the Abstract Client more *flexible* and eliminate the *type intrusiveness* of the solution. In addition, applying the Adapter pattern will allow the interface of the CdPlayer to be simplified and thus provide better resolution of the *simple interface* force.

There are two kinds of Adapters: the class adapter and the object adapter. The class adapter inherits from the abstract client and the actual client; the object adapter inherits from the abstract client and contains a reference to the actual client. The adapter receives messages from the server through the abstract client interface and then forwards these messages to the actual client. This section uses the class adapter form of the pattern. The next section on the External Polymorphism pattern [Schmidt95b] will use the object adapter structure.

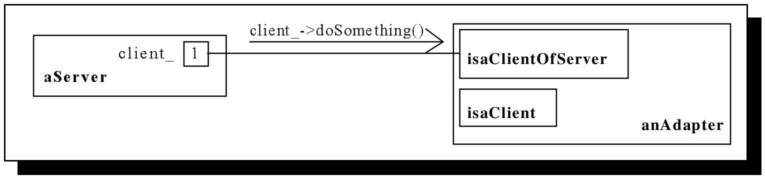

Figure 7: Abstract view of class adapter

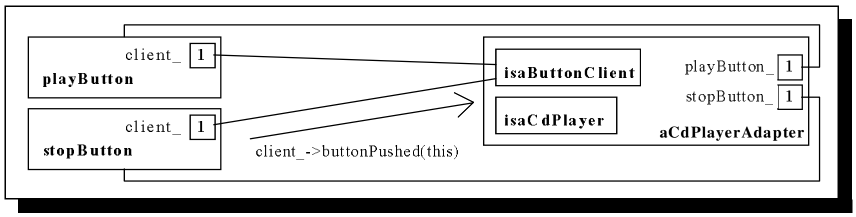

Figure 8: Application after applying the Adapter pattern

Figure 7 shows the structure of a class adapter. The adapter implements the interface specified in ClientOfServer by invoking methods on itself that it inherited from Client.

Figure 8 shows Adapter applied to the application. The CdPlayerAdapter must know about both the general Button library and the CdPlayer, thus it is not suitable to be a part of the Button library. Since the CdPlayerAdapter is specific to CdPlayers, the logic for determining which button was pushed can be moved into it to completely decouple the Buttons and CdPlayer. Buttons still know nothing about CdPlayers, and now CdPlayers know nothing about buttons.

source

What is interesting about the CdPlayer is what is not there.

```
class CdPlayer
{
public:
    void playButtonPushed(Button*) {cout << "PLAY" << endl;}
    void stopButtonPushed(Button*) {cout << "STOP" << endl;}
};
```

Both the inheritance of the ButtonClient and the logic for determining which button was pushed move into the CdPlayerAdapter. Most of the CdPlayerAdapter’s code was moved from the CdPlayer.

```
class CdPlayerAdapter: public ButtonClient, private CdPlayer
{
public:
    /*virtual*/ void buttonPushed(Button* pButton_)
    {
      if (pButton_ == _pPlayButton) this->playButtonPushed(pButton_);
      else if (pButton_ == _pStopButton) this->stopButtonPushed(pButton_);
    }
    void setPlayButton(Button* pButton_) {_pPlayButton = pButton_;}
    void setStopButton(Button* pButton_) {_pStopButton = pButton_;}
private:
    Button* _pPlayButton;
    Button* _pStopButton;
};
```

The `main` function changes slightly and is not shown here. Main creates a CdPlayerAdapter where before it created a CdPlayer.

force resolution

- *Automatic:* Adapters are not automatic. Each new client type needs its own adapter. The next step in the evolution will investigate making adapters automatic.

- *Flexible:* Resolved. Different forms of adapters could adapt the abstract client interface to call functions or function objects.
- *Loose Coupling:* Resolved. The CdPlayer no longer needs to know anything about Buttons.
- *Not Type Intrusive:* Resolved. The CdPlayer no longer inherits from the abstract client interface. This adaptation would have been required if the CdPlayer were part of a third party library.
- *Simple Interface:* No change.
- *Space Time Efficiency:* One more level of indirection is added. The overhead is still minor.

EXTERNAL POLYMORPHISM

The Adapter pattern had the drawback that the user has to manually create new adapters. This section uses the External Polymorphism pattern [Schmidt95b] to *automate* adapter creation, at the expense of some *flexibility* and the reintroduction of some *coupling*.

External Polymorphism starts with an object adapter, and generalizes it by using the implicit form of the Abstract Client pattern. The adapter class is templated by the type of the actual client it will work with to allow template instantiation to be used to create adapters for different clients. All clients for which the adapter template is instantiated must implement the methods that the adapter template will call, otherwise the compiler generates errors at instantiation time.

External polymorphism allows the clients to use the more permissive implicit Abstract Class implementation while the server still sees the explicit Abstract Class interface. This allows the server to keep a list of differently typed clients to call back to without requiring all clients to inherit from the same base class. This fact is not shown in the example.

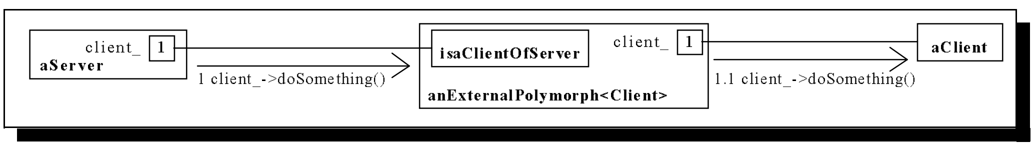

Figure 9: Abstract view of External Polymorphism

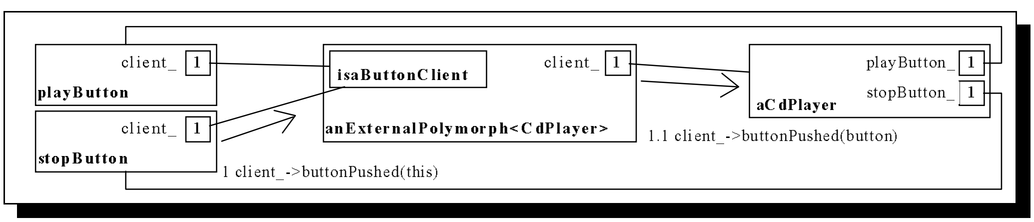

Figure 10: Application after applying External Polymorphism

Figure 9 shows that External Polymorphism takes the explicit Abstract Client implementation and turns it into the implicit Abstract Client implementation.

Figure 10 shows the new structure of the application. When the ExternalPolymorph is instantiated with CdPlayer as its client type, the compiler will generate an error if CdPlayer does not have a method `buttonPushed` that can be passed a `Button*`.

source

The big change at this step is the ExternalPolymorph. This is a more general adapter which does not know anything about CdPlayers. To discover the interface all clients must have, we must look at all methods invoked on the client in the ExternalPolymorph's code. The only method called is `myButtonPushed`, so that is the only method CdPlayer must implement. This method has to do what the old `CdPlayer::buttonPushed` did.

```
template <class T>
class ExternalPolymorph: public ButtonClient
{
public:
    ExternalPolymorph(T* pT_):_pT(pT_){}
    /*virtual*/ void buttonPushed(Button* pButton_) {_pT->myButtonPushed(pButton_);}
private:
    T* _pT;
};
```

force resolution

- *Automatic:* Somewhat resolved. The creation of adapters is automated, but further adaptation may be necessary. There may be clients that don't want to use the method names assumed by the ExternalPolymorph.

- *Flexible:* Not flexible. No single ExternalPolymorph template is able to call back to functions, function objects, and member functions.
- *Loose Coupling:* Mostly resolved. This force is no longer fully resolved because the CdPlayer must know about Buttons.
- *Not Type Intrusive:* No change.
- *Simple Interface:* No change.
- *Space Time Efficiency:* ExternalPolymorph could cause code bloat from template instantiation, but the template is small so the problem should not be bad.

PARAMETERIZE METHOD NAME

This step of the evolution applies the Parameterize Method Name pattern [Jakubik] to address some of the *flexibility* that was lost when the callback solution was made more *automatic* in the previous step. Applying this pattern will also resolve the *loose coupling* force. The client interface will be simplified, but the server will be complicated with some object lifetime issues.

The Parameterize Method Name pattern allows the name of the called back method to be specified as a parameter when creating a callback object. This pattern assumes that a single callback object will call back to a single member function of some object. Though this feature was not used, the previous patterns that were applied, Abstract Client, Adapter, and External Polymorphism, allowed multiple methods to be called back through a single entity. After applying this pattern, callbacks will start to resemble function pointers.

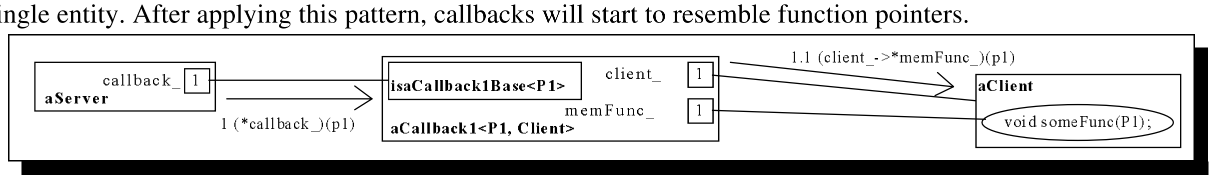

Figure 11: Abstract view after Parameterize Method Name applied to External Polymorphism

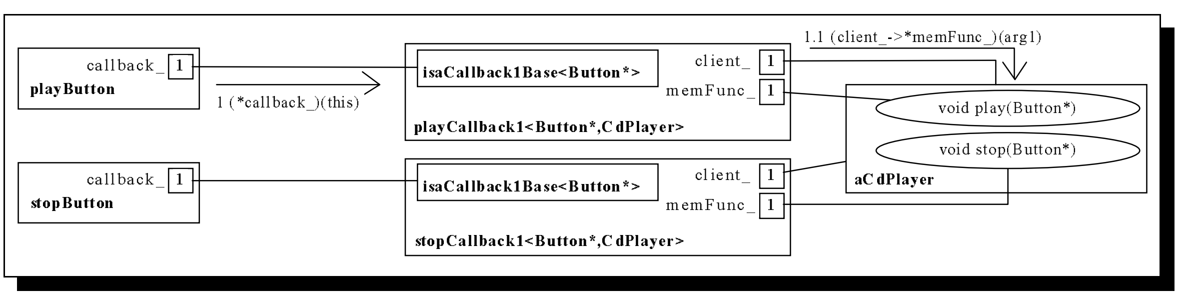

Figure 12: Application after applying Parameterize Method Name

Figure 11 shows that little changed structurally between the External Polymorphism solution and this solution. The callbacks are now a completely separate entity and could be in a library of their own.

Figure 12 shows that there is no longer a ClientOfButton class. The server now simply declares methods through which a particular kind of Callback1Base reference can be passed. The template argument of Callback1Base specifies what kind of argument the server will pass to its clients.

An actual callback is another template that automatically derives itself from the appropriate base class. Clients only have to instantiate the template with the type of client to be called back, the appropriate argument and return types, and the actual client and member function to call back to.

It is more difficult to apply Parameterize Method Name in C++ than it is in dynamically typed languages. More dynamic languages can check for type safety at run time and avoid the tricky templates inheriting from templates structure necessary in C++ to gain static type safety without sacrificing true parameterization of method names.

source

One major change is the addition of the Callback1Base and Callback1. Callback1Base and Callback1 are more generic replacements for ButtonClient and ExternalPolymorph. The templatizing of Callback1Base, the additional template argument in Callback1, and the pointer to a member function argument in Callback1's constructor all work together to allow the method name to be parameterized.

```
template <class P1>
class Callback1Base
{
public:
  virtual void operator()(P1) const = 0;
  virtual Callback1Base<P1>* clone() const = 0;
};
//----------------------------------------
template <class P1, class Client>
class Callback1: public Callback1Base<P1>
{
public:
  typedef void (Client::*PMEMFUNC)(P1);
  Callback1(Client& client_,PMEMFUNC pMemfunc_): _client(client_), _pMemfunc(pMemfunc_){}
  /*virtual*/ void operator()(P1 parm_) const {(_client.*_pMemfunc)(parm_);}
  /*virtual*/ Callback1<P1,Client>* clone() const {return new Callback1<P1,Client>(*this);}
private:
  Client& _client;
  PMEMFUNC _pMemfunc;
};
```

The Button class is changed to use callbacks. In the past, the Button was passed a reference to its client and assumed that it was not responsible for destroying the client. Now that a callback is being passed, the Button must worry about object lifetime issues. For now, the Button assumes that it is passed a Prototype [Gamma] of the callback it will use and calls the Prototype's clone method to get a pointer to its own copy of the callback.

```
class Button
{
public:
  Button(Callback1Base<Button*>& callback_): _pCallback(callback_.clone()) {}
  ~Button() {delete _pCallback;}
  void push() {(*_pCallback)(this);}
private:
  Callback1Base<Button*>* _pCallback;
};
```

The CdPlayer is simplified. The callbacks simply call back directly to the correct method; no adaptation is necessary.

```
#include <iostream.h>
class CdPlayer
{
public:
  void playButtonPushed(Button*) {cout << "PLAY" << endl;}
  void stopButtonPushed(Button*) {cout << "STOP" << endl;}
};
```

Finally, main has to create callbacks. Notice how the method names to be called back are specified in the callback constructors.

```
main()
{
  CdPlayer aCdPlayer;
  Callback1<Button*,CdPlayer> playCallback(aCdPlayer,&CdPlayer::playButtonPushed);
  Callback1<Button*,CdPlayer> stopCallback(aCdPlayer,&CdPlayer::stopButtonPushed);
  Button playButton(playCallback);
  Button stopButton(stopCallback);
  playButton.push();
  stopButton.push();
  return 0;
};
```

force resolution

- *Automatic:* Improved. Parameterizing the method name allows a wider variety of clients to be called back without requiring adaptation. As was seen in the example, this is a big help to clients that are a client to some constant number of servers that all have the same type. In C++, there is still room for more flexibility. C++ allows automatic type coercion to be performed on argument types, but the current callback implementation requires an exact signature match between the callback type and the method signature.
- *Flexible:* No change.
- *Loose Coupling:* Resolved. The CdPlayer does not need to know any more about Buttons. The code could be changed so that the called back method is not passed a `Button*`, but leaving an argument in makes the Callback code a little more interesting.
- *Not Type Intrusive:* No change.
- *Simple Interface:* No change. The CdPlayer is simplified, but the Buttons must worry about cloning and deleting callbacks. There are chances for memory leaks or dangling pointers.
- *Space Time Efficiency:* The space and time overhead of using member function pointers is small. More use of templates means more opportunities for code bloat. The only big problem is the cloning of callbacks. Cloning an object means creating an object on the heap. Heap allocation can be expensive if it gets out of hand.

SEPARATE CLIENT AND SERVER INTERFACE SPECIFICATIONS

Normally it is possible to call a function and ignore its return type. It is also normally possible to pass arguments to a function that don’t exactly match what the function expected and allow the compiler to automatically cast the arguments to the appropriate type. The previous section’s callback implementation did not allow this behavior, but by applying the Separate Client and Server Interface Specifications pattern [Jakubik] this behavior can be achieved and the *Automatic* force can be resolved. This pattern specifies a specific application of the implicit Abstract Client.

Applying this pattern requires separate template arguments for the types of arguments the server will invoke the callback with, and the type of the member function pointer the callback will invoke. This separation of the callback interface specifications gives the compiler the opportunity to perform type coercion. If the member function pointer passed in by the user turns out to have a type that the compiler can not coerce the arguments to work with, the compiler will generate a compile time error at the time the template is instantiated.

Applying this pattern goes further than just allowing automatic casting of arguments. Objects other than pointers to member functions can be specified as long as the same operators can be applied to that type. Pointers to member data can be passed instead of pointers to member functions when the member data is either a pointer to a member function, or a function object. While these alternatives are legal C++, some compilers have problems compiling these more advanced forms.

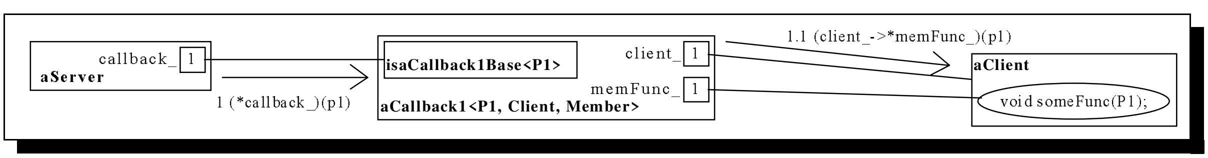

Figure 13: Abstract view after applying Separate Client and Server Interface Specifications

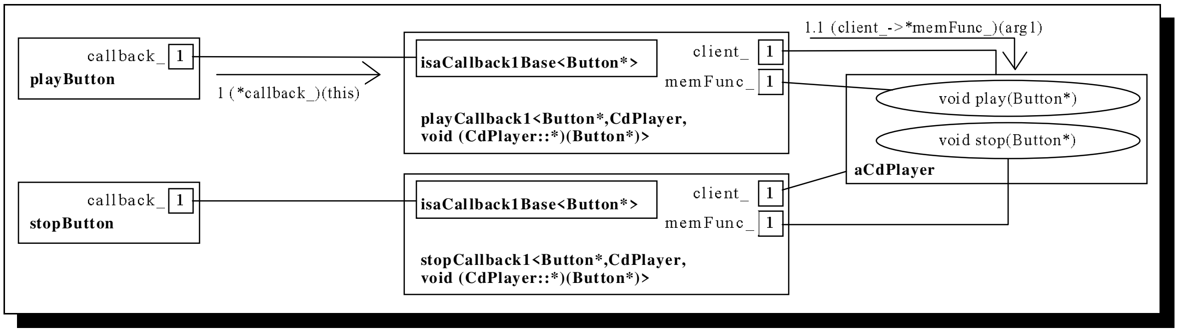

Figure 14: Application after applying Separate Client and Server Interface Specifications

Figures 13 and 14 show the only change: a template argument was added to Callback1.

source

Callback1 has an added template argument, Member, that is now used wherever a pointer to a member function type was used. To increase the flexibility of what types can be used as the Client and Member, the implementation of operator() now uses '->*' instead of '.*'. The operator '->*' can be overloaded to be passed arbitrary arguments. The only requirement for this template is that the operator '->*' must result in something that the function call operator can be applied to.

```
template <class P1, class Client, class Member>
class Callback1: public Callback1Base<P1>
{
public:
  Callback1(Client& client_, Member member_): _client(client_), _member(member_) {}
  /*virtual*/ void operator()(P1 parm_) const {((&_client)->*_member)(parm_);}
  /*virtual*/ Callback1<P1,Client,Member>* clone() const
  {return new Callback1<P1,Client,Member>(*this);}
private:
  Client& _client;
  Member _member;
};
```

No other major changes occur in the code until the function `main`. When declaring the callbacks, the full type of the member function must be specified.

```
main()
{
  CdPlayer aCdPlayer;
  Callback1<Button*,CdPlayer, void (CdPlayer::*)(Button*)>
    playCallback(aCdPlayer,&CdPlayer::playButtonPushed);
  Callback1<Button*,CdPlayer, void (CdPlayer::*)(Button*)>
    stopCallback(aCdPlayer,&CdPlayer::stopButtonPushed);
  Button playButton(playCallback);
  Button stopButton(stopCallback);
  playButton.push();
  stopButton.push();
  return 0;
};
```

force resolution

- *Automatic:* Resolved.
- *Flexible:* Somewhat resolved. This still is not a direct callback to a function pointer or function object, but they can be called through data member pointers.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* The callback template is becoming unwieldy for clients to instantiate.
- *Space Time Efficiency:* The new template parameter adds more opportunities for code bloat.

AUTOMATICALLY DETECT TYPES

The pattern Automatically Detect Types [Jakubik] can be used to simplify the interface. The `make_pair` templated function in STL [Musser] uses this pattern to automatically detect the types of each element of the pair.

A user only needs to call `make_pair` passing in the objects the user wants to make a pair out of. The template function `make_pair` is able to detect the types of the objects passed to it and declare a pair accordingly. Function templates find their template arguments implicitly by looking at the types of the arguments they are passed [Ellis]. As long as the template class's template arguments are all types as opposed to a constant value of some type, a template function can be used to automatically detect the types of the arguments it is passed and use these types when instantiating the template.

Applying the pattern provides a template function that will automatically detect the types to be used when instantiating the class template and return the created object. This pattern is helpful in cases where the user does not need a local copy of the template object created by the function, or the user can store the local copy as a reference to a simpler base class that does not have as many template arguments. This pattern depends on the fact that the arguments passed to the constructor of a templated class usually are sufficient to specify the type arguments for instantiating that template class.

No diagrams are shown because there are no changes to them.

source

To implement the pattern, a `make_callback` templated function is added to the source code.

```
template <class P1, class Client, class Member>
Callback1<P1,Client,Member>*
make_callback(Callback1Base<P1>*,Client& client_, Member member_)
{return new Callback1<P1,Client,Member>(client_,member_);}
```

The other major change to the code is in `main`, where `make_callback` is called. The callback returned by `make_callback` can be referred to using an easier to specify Callback1Base pointer. An artificial parameter is passed to `make_callback` that specifies what kind of callback the created callback must be derived from, and what kinds of arguments that callback expects.

```
main()
{
  CdPlayer aCdPlayer;
  Callback1Base<Button*>* pPlayCallback =
    make_callback((Callback1Base<Button*>*)0, aCdPlayer, &CdPlayer::playButtonPushed);
  Callback1Base<Button*>* pStopCallback =
    make_callback((Callback1Base<Button*>*)0, aCdPlayer, &CdPlayer::stopButtonPushed);
```

Once the callbacks are used, they must be deleted.

```
  Button playButton(*pPlayCallback);
  delete pPlayCallback;
  Button stopButton(*pStopCallback);
  delete pStopCallback;
  playButton.push();
  stopButton.push();
  return 0;
};
```

force resolution

- *Automatic:* No change.
- *Flexible:* No change.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* Mostly resolved. Instantiating callbacks is easier. The pointer semantics are still a problem.
- *Space Time Efficiency:* `make_callback` adds extra heap allocation and a new opportunity for code bloat.

Evolution of Hickey's Callback

FACTOR TEMPLATE

The callback library would be easier to use if the callbacks supporter value semantics. Hickey's solution is the most efficient possible: applying Manual Polymorphism [Jakubik]. Before that pattern can be applied, all of the data used by the derived class, Callback1, must be placed in the base class, Callback1Base. This can be done by using the pattern Type Destroyer is Restorer [Jakubik] to allow the pattern Factor Template [Jakubik] to be applied aggressively.

Normally the results of applying Factor Template must be tested to make sure code bloat is reduced, but in this case the test is not necessary. Regardless of the results, Factor Template must be applied aggressively before Manual Polymorphism can be applied.

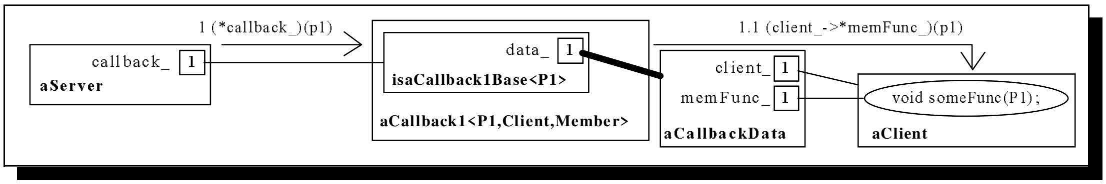

Figure 15: Abstract view of solution after applying Factor Template pattern

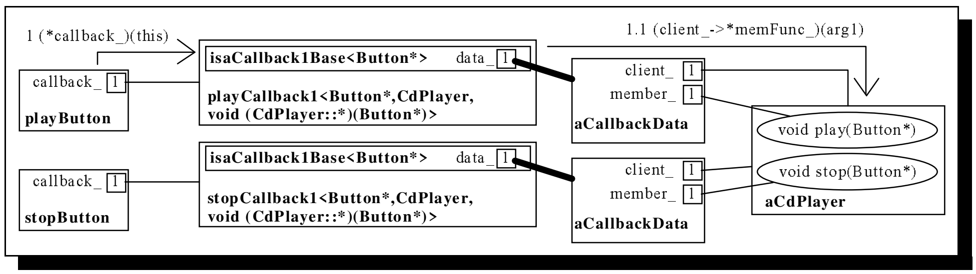

Figure 16: View of application after applying Factor Template pattern

Figures 15 and 16 show how the data that was stored in Callback1 is now factored into CallbackData and stored by value inside of Callback1Base. Instead of templating CallbackData or Callback1Base by the client and method types, CallbackData contains generic representations of the client reference and the pointer to member function. Object pointers can be stored as void pointers; there is no equivalent for member function pointers so a fixed size character array is used instead. Storing member function pointers in character arrays limits the flexibility that this solution will be able to achieve.

source

The CallbackData object is added for storing data.

```
class CallbackData
{
public:
  typedef void (CallbackData::*PMEMFUNC)();
  CallbackData(const void* pClient_, const void* ppMemfunc_): _pClient(pClient_)
  {memcpy(_ppMemfunc,ppMemfunc_,sizeof(PMEMFUNC));}
  const void* pClient() const {return _pClient;}
  const void* ppMemfunc() const {return _ppMemfunc;}
private:
  char _ppMemfunc[sizeof(PMEMFUNC)];
  const void* _pClient;
};
```

Callback1Base is altered to contain CallbackData and to allow derived classes to initialize that data. A pointer to a pointer to a member function is passed to the Callback1Base constructor instead of a pointer to a member function. This allows the pointer to be passed as a void pointer.

```
template <class P1>
class Callback1Base
{
public:
  virtual void operator()(P1) const = 0;
  virtual Callback1Base<P1>* clone() const = 0;
protected:
  Callback1Base(const void* pClient_, const void* ppMemfunc_): _data(pClient_,ppMemfunc_) {}
  const CallbackData& data() const {return _data;}
private:
  CallbackData _data;
};
```

The Callback1 constructor now passes a pointer to the client and a pointer to the pointer to a member function to its base class constructor. This is where type destruction occurs. While Callback1 knows the actual types of the data, Callback1Base sees only void pointers. Callback1's function call operator extracts data from the data object stored in the base class and casts it to the appropriate types to rebuild type information. The same object that destroyed the type information rebuilds and uses it. This ensures type safety.

```
template <class P1, class Client, class PMemfunc>
class Callback1: public Callback1Base<P1>
{
public:
  Callback1(Client& client_, PMemfunc pMemfunc_): Callback1Base<P1>(&client_,&pMemfunc_) {}
  /*virtual*/ void operator()(P1 parm_) const
  {
    Client* pClient = (Client*) data().pClient();
    PMemfunc& pMemfunc = (*(PMemfunc*)(data().ppMemfunc()));
    (pClient->*pMemfunc)(parm_);
  }
// ...
};
```

The `make_callback` function changes significantly. Instead of just being passed something for type Member, `make_callback` is much more explicit in its declaration so that only member function pointers can be passed to it. By being this explicit, there now has to be two `make_callback` functions, one for pointers to member functions, the other for pointers to const member functions. This change ensures that only pointer to member function sized objects will be stored in the character array in CallbackData, and it limits the flexibility of this solution.

```
template <class P1, class Client, class TRT, class CallType, class TP1>
inline Callback1<P1,Client,TRT(CallType::*)(TP1)>*
make_callback(Callback1Base<P1>*, Client& client_, TRT (CallType::* const& memFunc_)(TP1))
{
  typedef TRT (CallType::*PMEMFUNC)(TP1);
  return new Callback1<P1,Client,PMEMFUNC>(client_, memFunc_);
}
//----------------------------------------
template <class P1, class Client, class TRT, class CallType, class TP1>
inline Callback1<P1,Client,TRT(CallType::*)(TP1)const>*
make_callback(Callback1Base<P1>*,
              Client& client_,
              TRT (CallType::* const& memFunc_)(TP1) const)
{
  typedef TRT (CallType::*PMEMFUNC)(TP1) const;
  return new Callback1<P1,Callee,PMEMFUNC>(client_, memFunc_);
}
```

force resolution

- *Automatic:* No change.
- *Flexible:* Less resolved. The solution is less flexible because of the pointer to member function must be stored in a fixed size array. While some flexibility could be preserved, this is the solution in [Hickey].
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* No change.
- *Space Time Efficiency:* No change. The data factoring may have helped or harmed code bloat, but is not the point of these changes anyway.

MANUAL POLYMORPHISM

It is now possible to apply Manual Polymorphism and allow callbacks to be passed by value. Normally you cannot pass an object by value in C++ and still get polymorphic behavior from it. Manual Polymorphism manually implements polymorphic calls by using the C implementation of the Abstract Client pattern between the base class and derived class.

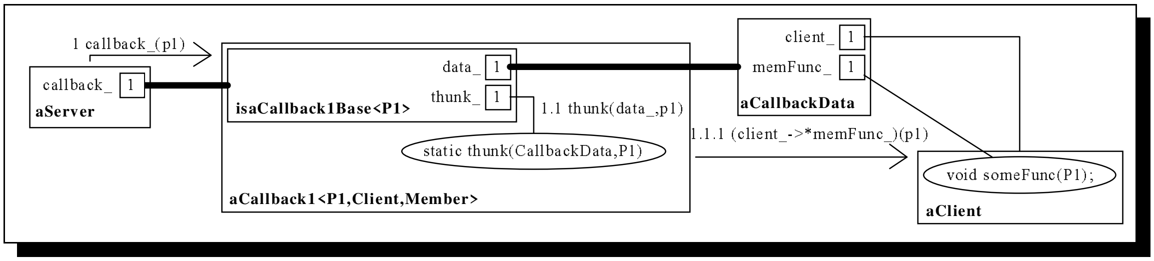

Figure 17: Abstract view of callback solution after applying Manual Polymorphism

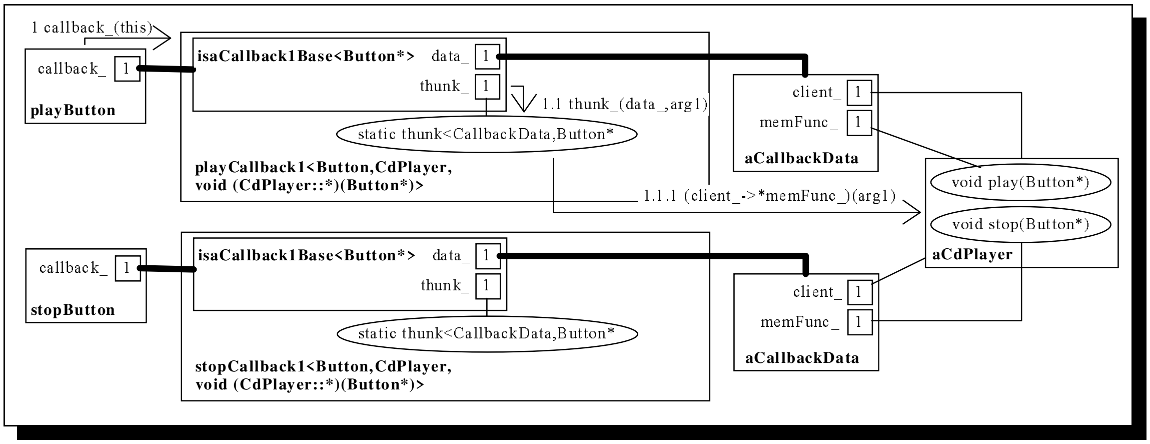

Figure 18: View of application after applying Manual Polymorhpism pattern

As can be seen from Figure 17, the server can now store a Callback1Base by value. Callback1Base gains a pointer to a function that points to a static member function that is a part of Calback1. The server invokes Callback1Base which in turn invokes its function pointer, passing a copy of the CallbackData. Finally, the static member function calls back the client.

Despite the complexity shown in Figure 18, the solution eliminates all use of the heap and is very efficient. When looking at the source code and the force changes at the end of this section, remember that all of the difficult changes and the problems associated with them occurred in the previous step.

source

A function pointer called `_thunk` is added to Callback1Base. The function call operator is no longer virtual and is now implemented as invoking the function pointed to by `_thunk`. The clone virtual method is no longer needed because Callback1Base can now be passed by value.

```
template <class P1>
class Callback1Base
{
public:
  void operator()(P1 p1_) const {_thunk(_data, p1_);}
protected:
  typedef void (*THUNK)(const CallbackData&, P1);
  Callback1Base(THUNK thunk_, const void* pClient_, const void* ppMemfunc_):
    _thunk(thunk_), _data(pClient_,ppMemfunc_) {}
private:
  CallbackData _data;
  THUNK _thunk;
};
```

Callback1 initializes Callback1Base's `_thunk` data member to point at Callback1's new static member function. This static member function is almost identical to the function call operator in the previous Callback1.

```
template <class P1, class Client, class PMemfunc>
class Callback1: public Callback1Base<P1>
{
public:
  Callback1(Client& client_, PMemfunc pMemfunc_):
    Callback1Base<P1>(thunk,&client_,&pMemfunc_) {}
private:
  static void thunk(const CallbackData& data_, P1 parm_)
  {
    Client* pClient = (Client*) data_.pClient();
    PMemfunc& pMemfunc = (*(PMemfunc*)(data_.ppMemfunc()));
    (pClient->*pMemfunc)(parm_);
  }
};
```

The `make_callback` functions are simply changed to return a callback by value and no longer create a callback on the heap.

The Button is changed to use value semantics with the Callback1Base.

```
class Button
{
public:
  Button(const Callback1Base<Button*>& callback_): _callback(callback_) {}
  void push() {_callback(this);}
private:
  const Callback1Base<Button*> _callback;
};
```

The value semantics simplify `main` as well. It is no longer necessary to keep pointers to the results of `make_callback` so that the created callbacks could be deleted later. Instead the results of `make_callback` can be passed directly to Button without any fear of memory leaks.

```
main()
{
  CdPlayer aCdPlayer;
  Button playButton
  (
    make_callback((Callback1Base<Button*>*)0, aCdPlayer, &CdPlayer::playButtonPushed)
  );
  Button stopButton
  (
    make_callback((Callback1Base<Button*>*)0, aCdPlayer, &CdPlayer::stopButtonPushed)
  );
  playButton.push();
  stopButton.push();
  return 0;
}
```

force resolution

- *Automatic:* No change.
- *Flexible:* No change.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* Resolved. Value semantics makes using callbacks a lot simpler and safer.
- *Space Time Efficiency:* Mostly resolved. The callbacks are now very efficient. Absolutely no heap allocation is used. The only potential problem is code bloat.

DUAL PURPOSE CLASS

All that is left is to add support for calling back to functions and function objects. Unfortunately, it is impossible to add support for calling back to function objects. Function objects are just objects with the function call operator overloaded and so can come in any size. This callback library only works with fixed size entities less than or equal to the size of a member function pointer. Calling back to functions is possible, and Hickey chose to implement this without adding any additional overhead to this class by using the Dual Purpose class pattern [Jakubik].

The Dual Purpose Class pattern crams two classes with mutually exclusive roles into a single class. Anonymous unions are used to avoid most of the space overhead this might cause. This pattern runs against good object oriented practice, and is only used to preserve the factoring of data necessary to support Manual Polymorphism.

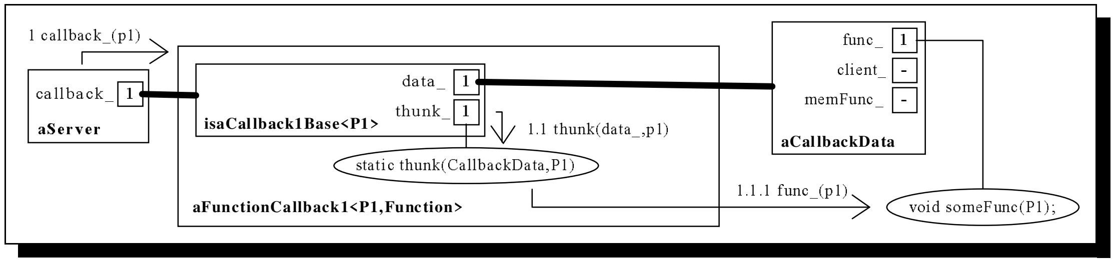

Figure 19: Abstract view of calling back to a function

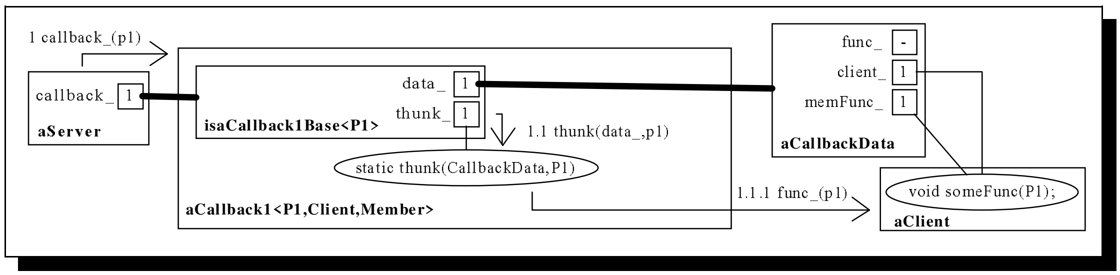

Figure 20: Abstract view of calling back to a member function

Figure 19 and Figure 20 show the two different kinds of callbacks that this library will support. As far as the classes that are part of the callback library are concerned, the only changes are whether an instance of the Callback1 or the FunctionCallback1 class is used. When calling back to a function in Figure 19, the FunctionCallback1 is used. FunctionCallback1 provides a thunk that interprets the passed in CallbackData as containing a function pointer and invokes the function. In Figure 20 Callback1 works just like it always has, it reads the client and member function pointers from the data object and invokes the member function.

By separating Callback1 and FunctionCallback1 into separate classes, it is easy to see how the underlying Callback1Base and CallbackData are used differently by the two classes. FunctionCallback1 always stores a function pointer in the underlying data structure and thus knows that it is safe to extract a function pointer. Callback1 always stores a member function pointer and a pointer to a client, and knows that it is safe to extract a member function pointer and pointer to a client. The Dual Pupose class pattern was made safe by again applying the Type Destroyer is Restorer pattern.

source

The CallbackData object must have a role added to it. The union keeps this additional responsibility from adding any space overhead.

```
class CallbackData
{
public:
  typedef void (CallbackData::*PMEMFUNC)();
  CallbackData(const void* pClient_, const void* ppMemfunc_): _pClient(pClient_)
  {memcpy(_ppMemfunc,ppMemfunc_,sizeof(PMEMFUNC));}
  CallbackData(const void* pFunc_): _pFunc(pFunc_), _pClient(0) {}
  const void* pClient() const {return _pClient;}
  const void* ppMemfunc() const {return _ppMemfunc;}
  const void* pFunc() const {return _pFunc;}
private:
  union
  {
    char _ppMemfunc[sizeof(PMEMFUNC)];
    const void* _pFunc;
  };
  const void* _pClient;
};
```

The only change to Callback1Base is that an additional constructor is added for when Callback1Base is initialized with information for calling back to a function instead of a member function.

```
template <class P1>
class Callback1Base
{
// ...
protected:
  Callback1Base(THUNK thunk_, const void* pFunc_): _data(pFunc_), _thunk(thunk_) {}
// ...
};
```

Callback1 does not have to be changed at all. Instead, Callback1Func is created to handle the details of calling back to functions.

```
template <class P1, class Func>
class Callback1Func: public Callback1Base<P1>
{
public:
  Callback1Func(Func f_): Callback1Base<P1>(thunk, f_) {}
private:
  static void thunk(const CallbackData& data_, P1 p1_) {(Func(data_.pFunc()))(p1_);}
};
```

A `make_callback` function is added for creating callbacks to functions. Because of the manual polymorphism, this function must make sure it is passed function pointers and not function objects. Function objects might be too big to store in CallbackData.

```
template <class P1, class TRT, class TP1>
inline Callback1Func<P1, TRT(*)(TP1)> make_callback(Callback1Base<P1>*, TRT(*func_)(TP1))
{return Callback1Func<P1,TRT(*)(TP1)>(func_);}
```

A function is added for testing the new calling back to a function facility.

```
void buttonTest(Button*) {cout << "BUTTON TEST" << endl;}
```

A little bit of new code is added to the end of main to test calling back to a function.

```
main()
{
  // ...
  Button testButton(make_callback((Callback1Base<Button*>*)0, buttonTest));
  testButton.push();
  return 0;
}
```

force resolution

- *Automatic:* No change.
- *Flexible:* Mostly resolved. It is now possible to call back to functions, but it is not possible to callback to function objects with this callback.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* No change.
- *Space Time Efficiency:* No change.

Evolution of a Simpler Callback

HANDLE BODY

This step in the evolution occurs immediately after the section entitled Automatically Detect Types. The evolution of Hickey's callback is a separate branch of the evolution.

Hickey's callbacks applied Factor Template, and Manual Polymorphism to allow the callbacks to be passed by value while preserving polymorphic behavior. Applying Manual Polymorphism restricted the flexibility of the callback solution by requiring all of the data used by the derived classes to be stored in a fixed sized base class.

The Handle Body pattern [Coplien92] accomplishes the same thing without adding the fixed size requirement. Instead, the Handle Body pattern creates a separate class that is the handle and passes the handles around by value. The handle contains a pointer to an abstract body that provides that actual implementation and polymorphic behavior. The body is just an explicit Abstract Client implementation. Handles forward messages to a derived body. The Handle Body pattern is most useful in languages that do not have automatic garbage collection.

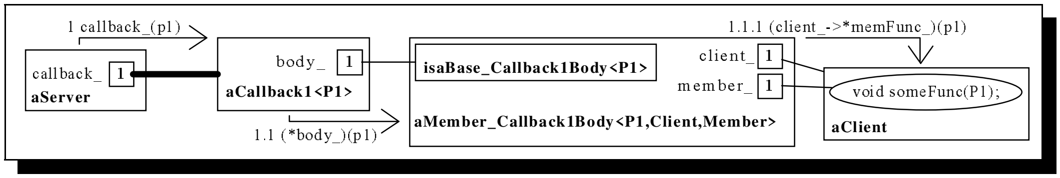

Figure 21: Abstract view of Handle Body based callback

Figure 21 shows that the server contains the handle, aCallback1<P1>, by value. The handle has a reference to the body, Member_Callback1Body<P1,Client,Member>. The handle class is new, the base body class used to be the callback base class, and the member body class used to be the callback class. These classes only changed their names.

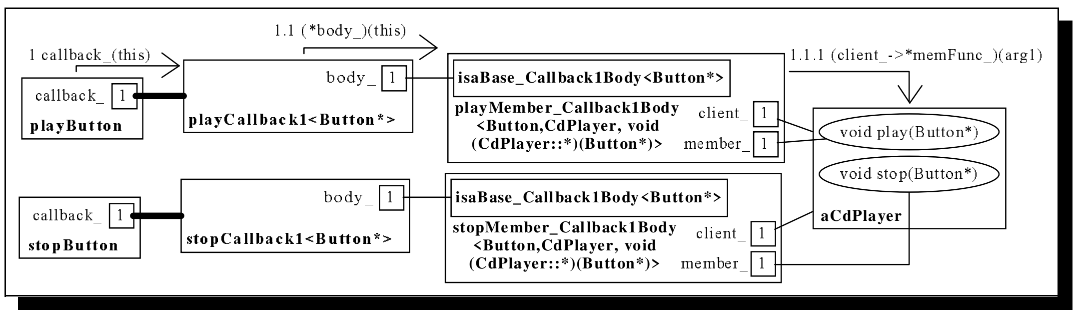

Figure 22: Application after applying Handle Body pattern

It is easy to see from Figure 22 that the Handle Body pattern adds an extra level of indirection while giving the callbacks value semantics.

source

The callback body classes are just the old callback classes from the Automatically Detect Types section that have been renamed. The handle class, Callback1, is new and appears below. Whenever a handle is copied, it makes a new copy of the body it was pointing to. Whenever a handle is destroyed, it destroys the body it refers to. The handle assignment operator deletes its old body and copies the body of the right hand side handle. All of this copying and destroying is safe, but expensive.

```
template <class P1>
class Callback1
{
public:
  Callback1(Base_Callback1Body<P1>* body): body_(body) {}
  Callback1(const Callback1<P1>& callback): body_(callback.body_->clone()) {}
  ~Callback1() {delete body_; body_ = 0;}
  Callback1<P1>& operator=(const Callback1<P1>& callback)
  {
    if (this != &callback)
    {
      delete body_;
      body_ = callback.body_->clone();
    }
    return *this;
  }
  void operator()(P1 p1) {(*body_)(p1);}
private:
  Base_Callback1Body<P1>* body_;
};
```

`make_callback` is changed to return a callback by value. The handle's body copying policies cause `make_callback` to create and delete extra copies of the body.

```
template <class P1, class Client, class Member>
Callback1<P1>
make_callback(Callback1<P1>*, Client& client_, Member member_)
{return Callback1<P1>(new Member_Callback1Body<P1,Client,Member>(client_,member_));}
```

Button's `push` method and destructor are simplified by callback's value semantics.

```
class Button
{
public:
  Button(const Callback1<Button*>& callback_):_callback(callback_){}
  ~Button() {}
  void push() {_callback(this);}
private:
  Callback1<Button*> _callback;
};
```

`main` is similarly simplified by removing all heap maintenance from it.

```
main()
{
  CdPlayer aCdPlayer;
  Button playButton
  (make_callback((Callback1<Button*>*)0, aCdPlayer,&CdPlayer::playButtonPushed));
  Button stopButton
  (make_callback((Callback1<Button*>*)0, aCdPlayer, &CdPlayer::stopButtonPushed));
  playButton.push();
  stopButton.push();
  return 0;
}
```

force resolution

- *Automatic:* No change.
- *Flexible:* No change.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* The interface is simplified by hiding all heap allocation behind the handle object and `make_callback` function.
- *Space Time Efficiency:* While heap allocation is hidden, there are excessive amounts of it.

COUNTED BODY

Excess heap activity makes the *space time efficiency* of the last solution awful. Luckily, this problem is addressed by Handle Body's companion pattern, Counted Body [Coplien94] [Coplien92].

The idea behind Counted Body is that each body object could contain a count of how many handles are currently referring to it. When a handle is copied, the body is shared instead of copied and the body's reference count is incremented. When a handle is deleted, it decrements the body's reference count. If the reference count is zero after the handle decrements it, the handle deletes the body. Counted Body eliminates excess heap allocation and the user gets better performance from the same simple interface.

The diagrams are omitted since the changes are easier to see in the code.

Source

A counted body class encapsulates the changes to the body classes.

```
class CountedBody
{
public:
  CountedBody(): count_(0) {}
  virtual ~CountedBody() {}
  void incCount() {count_++;}
  void decCount() {count_--;}
  int count() {return count_;}
private:
  int count_;
};
```

The body base class inherits from counted body. The clone interface is removed.

```
template <class P1>
class Base_Callback1Body:
  public CountedBody
{
public:
  virtual void operator()(P1) const = 0;
};
```

The handle replaces body cloning with body incrementing. Deleting bodies is replaced by decrementing bodies. After decrementing the body, the handle deletes it if the reference count is zero. Decrementing bodies is interesting enough to be encapsulated in a private helper method. Incrementing bodies is similarly wrapped for symmetry.

```
template <class P1>
class Callback1
{
public:
  Callback1(Base_Callback1Body<P1>* body): body_(body) {this->incBodyCount();}
  Callback1(const Callback1<P1>& callback): body_(callback.body_) {this->incBodyCount();}
  ~Callback1() {this->decBodyCount(); body_ = 0;}
  Callback1<P1>& operator=(const Callback1<P1>& callback)
  {
    if (body_ != callback.body_)
    {
      this->decBodyCount();
      body_ = callback.body_;
      this->incBodyCount();
    }
    return *this;
  }
  void operator()(P1 p1) {(*body_)(p1);}
private:
  Base_Callback1Body<P1>* body_;
  void incBodyCount() {body_->incCount();}
  void decBodyCount() {body_->decCount(); if(body_->count() == 0) delete body_;}
};
```

Nothing else needs to be changed to support the Counted Body pattern.

force resolution

- *Automatic:* No change.
- *Flexible:* No change.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* No change.
- *Space Time Efficiency:* Mostly resolved. The excess heap allocation is eliminated. Potential code bloat from template instantiation is the only issue that will keep this force from being completely resolved.

NULL OBJECT

The last issue that will be addressed may have been forgotten. The original goal was to create a callback that was *flexible* enough to call back to functions, function objects, and member functions. As unlikely as it sounds, this problem can be addressed with the Null Object pattern [Anderson].

The Null Object pattern suggests deriving a specific class from an explicit Abstract Client in order to represent the absence of that client. Servers often have to deal with the fact that they might not have a client. Making the server always check to see if it really has a client leads to lots of conditional statements in the code. Instead of writing the server to work with no clients, the server can use a specially derived null client. The null client is given the correct behavior for if there is no client. Conditional statements are traded for polymorphic calls.

This seems to have nothing to do with providing callbacks to functions and function objects, but a connection does exist. In the discussion of Null Objects, a more general but less recognizable form of the pattern was identified: the Special Purpose Class. Special Purpose Class points out that wherever the same condition is tested in several places in a program, it may be possible to create a special purpose class to replace the conditional tests with polymorphism. This general pattern can be applied to the callback library to support callbacks to functions and function objects. It is also useful to apply the Null Object pattern to allow the callback to have a default constructor. Default constructors are required for classes that are stored in arrays. While the application of the two patterns will look similar enough to suggest that they really are the same pattern, both solutions might really be instances of the even more general Don't Be Stupid, Just Use Object Oriented Programming pattern. It is difficult to balance generalizing patterns to avoid overwhelming people with too many patterns against making the pattern specific enough that people can figure out how to use it.

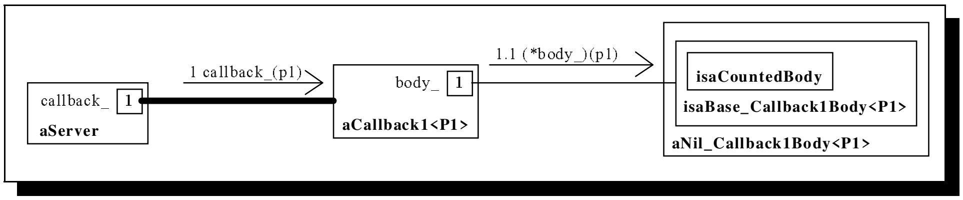

Figure 23: Abstract view of a callback transformed by Null Object pattern

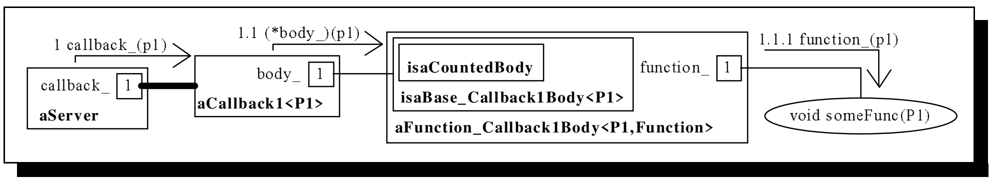

Figure 24: Abstract view of Special Purpose Object pattern application

Figures 23 and 24 show that adding in Nil callbacks, and callbacks to functions and function objects involves adding two new kinds of body classes. Function_Callback1Body uses the implicit Abstract Client implementation to allow it to call back both functions and function objects. The syntax for invoking a function or function object in C++ is the same. This use of the Abstract Client pattern was inspired by STL generic algorithms such as `sort`.

source

A Nil_Callback1Body's function call operator just throws an exception.

```
template <class P1>
class Nil_Callback1Body: public Base_Callback1Body<P1>
{
public:
  /*virtual*/ void operator()(P1) const{throw "nil callback invoked";}
};
```

The handle class's new default constructor creates a Nil_Callback1Body to be used as the handle's body.

```
template <class P1>
class Callback1
{
public:
  Callback1(): body_(new Nil_Callback1Body<P1>) {}
// ...
};
```

The Function_Callback1Body template is similar to the Member_Callback1Body. Objects that implement an appropriate function call operator, including function pointers, can be used as the `Function` type for this template.

```
template <class P1, class Function>
class Function_Callback1Body: public Base_Callback1Body<P1>
{
public:
  Function_Callback1Body(Function& function_): _function(function_) {}
  /*virtual*/ void operator()(P1 parm_) const {_function(parm_);}
private:
  Function _function;
};
```

An additional `make_callback` function makes it easy to create callbacks to functions and function objects.

```
template <class P1, class Function>
Callback1<P1>
make_callback(Callback1<P1>*, Function function_)
{return Callback1<P1>(new Function_Callback1Body<P1,Function>(function_));}
```

A dummy function and some test is added to test nil callbacks and calling back to functions.

```
void buttonTest(Button*) {cout << "BUTTON TEST" << endl;}
main()
{
  // ...
  Button testFunc(make_callback((Callback1<Button*>*)0,buttonTest));
  testFunc.push();
  Callback1<Button*> nullCb;
  Button testNull(nullCb);
  try
  {
    testNull.push();
  }
  catch(const char * p)
  {
    cout << "caught exception: " << p << endl;
  }
  return 0;
};
```

force resolution

- *Automatic:* No change.
- *Flexible:* Callbacks to functions, function objects, and member functions are supported.
- *Loose Coupling:* No change.
- *Not Type Intrusive:* No change.
- *Simple Interface:* No change.
- *Space Time Efficiency:* No change.

Conclusions

The author would have never discovered how to implement the simpler callbacks without first identifying all of the patterns behind Hickey's callbacks. Patterns document important design decisions and show where alternative decisions could be made.

Once two solutions to the same problem have had all of their patterns identified, by comparing the quality of the solutions, the patterns that lead to better solutions can be identified.

Hickey's callbacks are extremely efficient. No heap allocation is necessary when using them. Despite this efficiency, the diagrams and source code attest to the fact that the underlying implementation is complex. At least one widely used compiler can't compile the `make_callback` functions used by this solution. Another popular compiler can't compile the const method version of `make_callback`. The fixed storage size requirements of this library not only restrict its flexibility to fall short of the original goal, but also causes problems when porting the code to systems where pointers to member functions are not all the same size. Only someone who has to count every processor cycle would think that this is the best callback solution.

The code for the simpler callbacks is easier to understand, maintain and port. The `make_callback` functions are simpler, less restrictive, and can be compiled by more compilers. The flexibility of the final solution exceeds the original goals. The only apparent problem is the required heap allocation. Research has shown that heap allocation can be cheap; it is possible to write general purpose heap allocators that perform well [Wilson][Neely]. Even though most compilers come with relatively poor heap allocators, those heaps are still efficient enough that heap allocation is commonly used in C++ programs. The simpler solution is better for most uses of callbacks.

Handle Body, Counted Body, and Null Object led to the better solution and should be highly recommended. Factor Template (aggressively applied), Manual Polymorphism, and Dual Purpose Class should specify that they should only be used when both every processor cycle counts and the patterns can be used together to eliminate heap allocation. The rest of the patterns can be recommended, but without being able to compare these patterns against alternatives, it is impossible to say if these patterns tend to lead to better solutions than alternatives.

More solution comparisons need to be performed to improve pattern assessments.

The callback libraries presented in this document are simplified versions of what would be used in practice. A complete version of the simpler callback library is available at ftp://ftp.primenet.com/users/j/jakubik/callback/. In that directory you will find a readme file with more information on how to download and use the library.

References

[Anderson] Anderson, Bruce, archive of the null object thread started by Bruce Anderson, "Pattern Proposal 'Null Object'", HTTP://ST-WWW.CS.UIUC.EDU/FTP/PUB/PATTERNS/MAIL-ARCHIVE/NULL-OBJECTS, January 5, 1994.

[Berczuk] Berczuk, Stephen P., archive of the callback thread started by Stephen Berczuk, "Handler/Callback Pattern", HTTP://ST-WWW.CS.UIUC.EDU/FTP/PUB/PATTERNS/MAIL-ARCHIVE/CALLBACK, January 31, 1994.

[Coplien92] Coplien, James O., ADVANCED C++ PROGRAMMING STYLES AND IDIOMS, Addison-Wesley, Reading, MA, 1992.

[Coplien94] Coplien, James O. "Setting the Stage", C++ REPORT, 6(8) 1994.

[Ellis] Ellis, M. A. and B. Stroustrup, THE ANNOTATED C++ REFERENCE MANUAL, Addison-Wesley, 1990.

[Gamma] Gamma, Erich., Richard Helm, Ralph Johnson, and John Vlissides, DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading MA, 1995.

[Hickey] Hickey, R., "Callbacks in C++ Using Template Functors", C++ REPORT, 7(2), 1995.

[Huni] Huni, Hermann, Ralph Johnson, and Robert Engel, "A Framework for Network Protocol Software", FTP://ST.CS.UIUC.EDU/PUB/PATTERNS/PAPERS/CONDUITS+.PS, 1995.

[Jakubik] Jakubik, Paul A., "Callbacks in C++", HTTP://WWW.PRIMENET.COM/~JAKUBIK/CALLBACK.HTML, 1996.

[Lea] Lea, Doug. "Completion Callbacks", HTTP://G.OSWEGO.EDU/DL/PATS/CB.HTML.

[Martin] Martin, Robert C., "Discovering Patterns in Existing Applications", PATTERN LANGUAGES OF PROGRAM DESIGN, edited by James O. Coplien and Douglas C. Schmidt, Addison-Wesley, Reading MA, 1995.

[Musser] Musser, David R., and Atul Saini, STL TUTORIAL AND REFERENCE GUIDE: C++ PROGRAMMING WITH THE STANDARD TEMPLATE LIBRARY, Addison-Wesley, 1996.

[Neely] Neely, Michael S., "An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation", FTP://FTP.CS.UTEXAS.EDU/PUB/GARBAGE/NEELY-THESIS.PS.GZ, 1996.

[Schmidt95a] Schmidt, Douglas C. "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", PATTERN LANGUAGES OF PROGRAM DESIGN, edited by James O. Coplien and Douglas C. Schmidt, Addison-Wesley, Reading MA, 1995.

[Schmidt95b] Schmidt, Douglas C. "External Polymorphism in C++", HTTP://WWW.CS.WUSTL.EDU/~SCHMIDT/EUROPLOP-96.PS.Z, 1996.

[Wilson] Wilson, Paul R., Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic Storage Allocation: A Survey and Critical Review, 1995 INTETNATIONAL WORKSHOP ON MEMORY MANAGEMENT, Kinross, Scotland, UK, 1995, Springer Verlag LNCS.