Callback Implementations in C++

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

Abstract: *This paper presents pattern histories for deriving two different callback libraries. The pattern histories document the important design decisions behind each library. The two resulting libraries are compared to determine which patterns led to a better solution. The patterns community is identifying many different patterns that solve the same problems. This paper shows that it is possible to use objective comparisons of the resulting solutions, as opposed to relying on personal reactions, the expertise of the author, and finding three previous uses of the pattern, to determine which patterns lead to better solutions.*

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. Each solution differs in how wide a variety of clients can be called back and how much the server is coupled to 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 [Schmidt].

The range of callback solutions is organized in two pattern histories. A pattern history is more than a list of patterns since it shows the details of how each pattern was applied and how each pattern affected the solution. A pattern history is not a pattern language since it is made up of a series of standalone patterns instead of a web of interconnected patterns. The patterns do not tell the reader which patterns should be applied next. A pattern history simply documents which patterns were applied and how they were applied. First a complete pattern history is described that evolves a simple client-server interaction into the efficient callback solution presented in [Hickey]. Afterwards an alternative branch to the history that leads to a simpler and more flexible solution is presented. Finally we compare the more efficient solution with the simpler solution to determine which 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++. Aspects 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 allow a server library to call client code without being coupled to it. A further goal is for the callback solutions themselves to be reusable so that many different libraries can 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

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. Any extra code that must be added to allow a callback to a new client should be generated automatically instead of by the library user.
- *Flexible:* The library should support callbacks to functions, function objects, and member functions of objects.
- *Loosely Coupled:* Libraries should be independent of client code.

- *Not Type Intrusive:* Users should not have to alter their class hierarchies to use the library. Some class hierarchies, such as those in third party libraries, can't be altered. Type intrusion only occurs in statically typed languages such as C++.
- *Simple Interface:* The library's interface should be simple and easy to use.
- *Space Time Efficient*: 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 1 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, and altered to make it easier to draw. GLUE object scenarios were chosen to show implementation details without including all of the source code in the body of the paper.

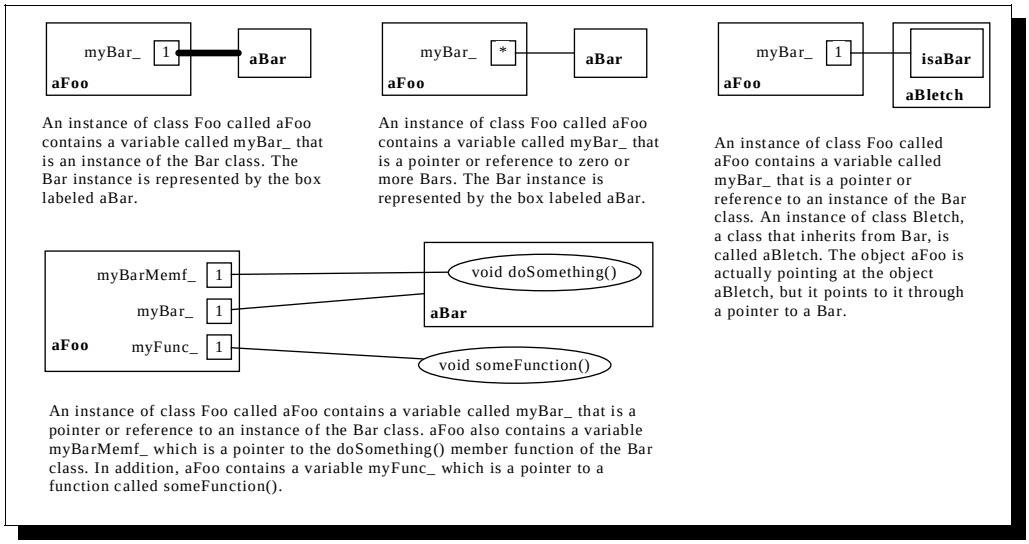

Figure 1: Examples of GLUE object scenarios and descriptions

OVERVIEW

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

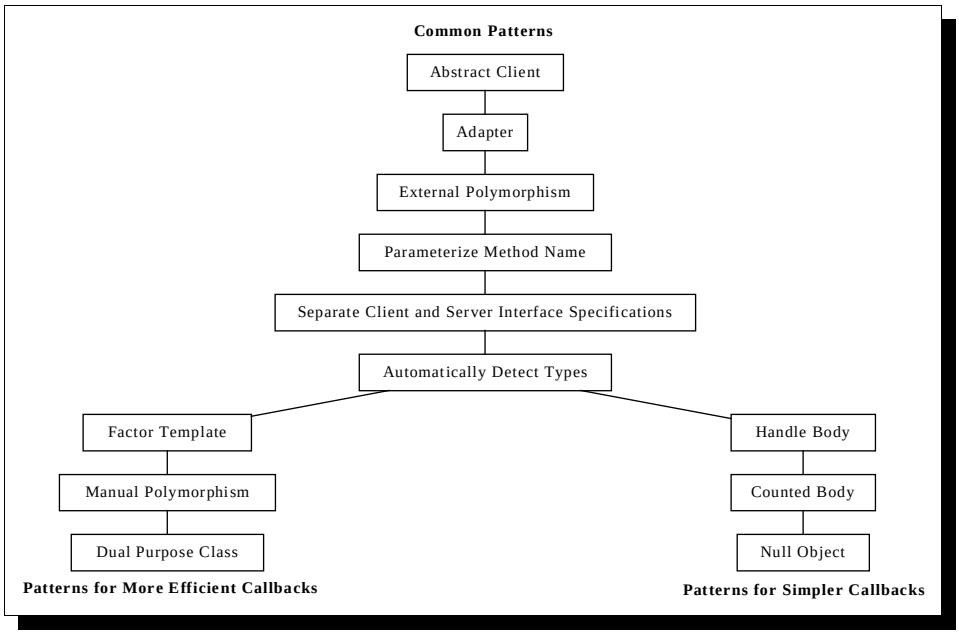

Figure 2: Order of pattern application showing alternate solutions as branches.

Common Callback History

SIMPLE SOLUTION

The examples show implementations of a `Button` object and a `CdPlayer` object. 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.

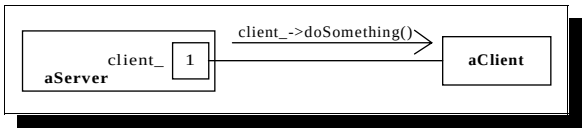

Figure 3: Abstract view of simple solution

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

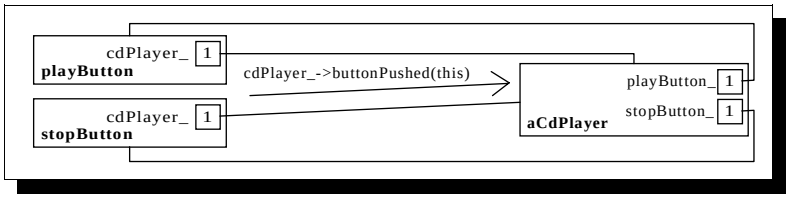

Figure 4: View of actual simple application

Actually implementing an application (Figure 4) is more complex. There are two `Button`s 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 `Button`s always pass a pointer to themselves as part of the message. The `CdPlayer` stores pointers to its `Button`s 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;
};
//---------------------------------------
main()
{
  CdPlayer aCdPlayer;
  Button playButton(&aCdPlayer);
  Button stopButton(&aCdPlayer);
  aCdPlayer.setPlayButton(&playButton);
  aCdPlayer.setStopButton(&stopButton);
  playButton.push();
  stopButton.push();
  return 0;
}
```

force resolution

- *Automatic:* Resolved. No additional code has to be written outside of the client and server.
- *Flexible:* Not resolved. Not flexible at all.
- *Loosely Coupled:* Not resolved. The client and server are tightly coupled, they can’t be used separately.
- *Not Type Intrusive:* Resolved.
- *Simple Interface:* Mostly resolved. The `CdPlayer` must figure out which `Button` was pushed.
- *Space Time Efficient:* Mostly resolved. 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. Patterns found in [Gamma] including Bridge, Chain of Responsibility, Observer, and Strategy use Abstract Client. The Reactor Pattern [Schmidt], the Completion Callback pattern [Lea], and the Conduits+ framework [Huni] also use Abstract Client. 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 the client interface is specified by the way the server uses the template type. [Martin] does not mention the implicit 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. Java provides an interface facility especially for explicitly specifying Abstract Client interfaces. Even non-object-oriented languages use techniques similar to Abstract Client; the equivalent in C is parameterizing a function name by passing function pointers.

Anywhere Abstract Client is used, another 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 history: Abstract Client, Adapter [Gamma], or External Polymorphism [Cleeland]. If the Abstract Client only defines one or two methods, the final solution in this paper or the more efficient solution would be best. Of course, if you just want the simplest and most straightforward callback, Abstract Client is the pattern for you.

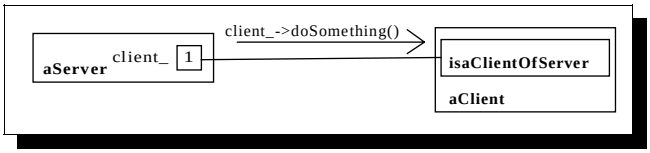

Figure 5: Abstract view of solution transformed by Abstract Client pattern

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.

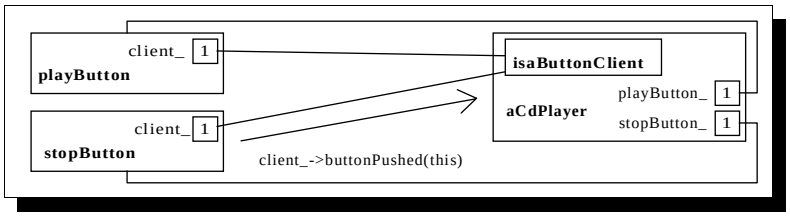

Figure 6: View of actual transformed application

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 `ButtonClient` 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:* Partially resolved. 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:* Not resolved. This can not call back to functions and function objects.
- *Loosely Coupled:* Mostly resolved. The server no longer needs to know the type of the client.
- *Not Type Intrusive:* Not resolved. The client must alter its type by inheriting from an abstract base class.
- *Simple Interface:* No change: mostly resolved. The `CdPlayer` still has to figure out which `Button` was pushed.
- *Space Time Efficient:* Mostly resolved. 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.

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 object adapter form of the pattern.

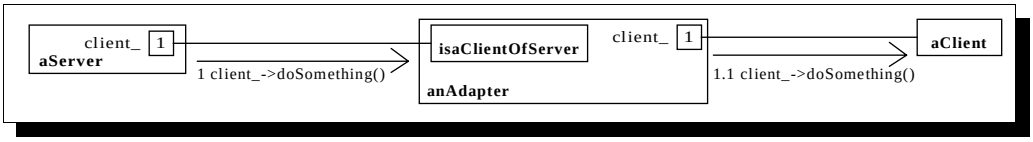

Figure 7: Abstract view of class adapter

Figure 7 shows the structure of an object adapter. The adapter implements the interface specified in `ClientOfServer` by invoking methods on the `Client` it refers to.

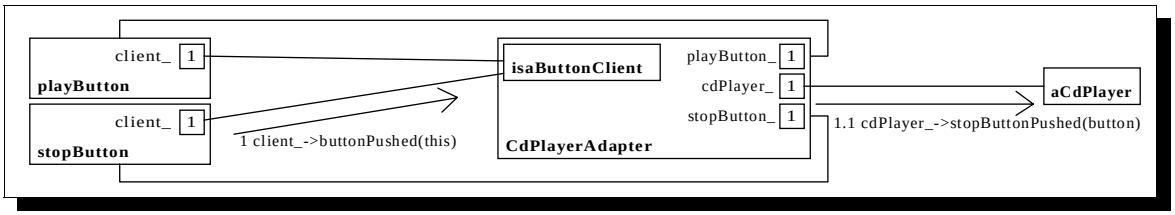

Figure 8: Application after applying the Adapter pattern

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 the adapter 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
{
public:
  CdPlayerAdapter(CdPlayer* pCd_): _pCd(pCd_) {}
  /*virtual*/ void buttonPushed(Button* pButton_)
  {
    if (pButton_ == _pPlayButton) _pCd->playButtonPushed(pButton_);
    else if (pButton_ == _pStopButton) _pCd->stopButtonPushed(pButton_);
  }
  void setPlayButton(Button* pButton_) {_pPlayButton = pButton_;}
  void setStopButton(Button* pButton_) {_pStopButton = pButton_;}
private:
  Button* _pPlayButton;
  Button* _pStopButton;
  CdPlayer* _pCd;
};
```

The `main` function now creates a `CdPlayerAdapter`. The `Button`s are now initialized with references to the `CdPlayerAdapter`, and the `CdPlayerAdapter` is told which `Button` is which.

```
main()
{
  CdPlayer aCdPlayer;
  CdPlayerAdapter aCdPlayerAdapter(&aCdPlayer);

  Button playButton(&aCdPlayerAdapter);
  Button stopButton(&aCdPlayerAdapter);

  aCdPlayerAdapter.setPlayButton(&playButton);
  aCdPlayerAdapter.setStopButton(&stopButton);

  playButton.push();
  stopButton.push();

  return 0;
}
```

force resolution

- *Automatic:* Not resolved. Each new client type needs its own adapter. The next step in the pattern history will automate adapter creation.
- *Flexible:* Resolved. Different forms of adapters could adapt the abstract client interface to call functions or function objects.
- *Loosely Coupled:* Resolved. The `CdPlayer` no longer needs to know anything about `Button`s.
- *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: mostly resolved. The `CdPlayer` is simplified, but all of the `Button` discrimination code must still be written by the library user.
- *Space Time Efficient:* Mostly resolved. 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 [Cleeland] 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 Client implementation while the server still sees the explicit Abstract Client 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.

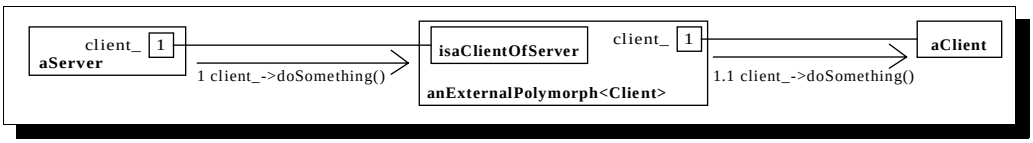

Figure 9: Abstract view of External Polymorphism

Figure 9 shows that External Polymorphism takes the explicit Abstract Client implementation and turns it into the implicit Abstract Client implementation. The object adapter is now a template called `ExternalPolymorph`.

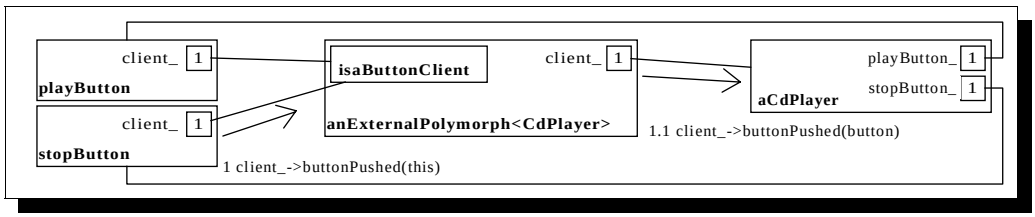

Figure 10: Application after applying External Polymorphism

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;
};
```

The Button discrimination code is moved back into the CdPlayer, but the CdPlayer does not have to inherit from ButtonClient.

```
class CdPlayer
{
public:
  void myButtonPushed(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;
};
```

In the rest of the code all uses of the `CdPlayerAdapter` are replaced by uses of the `ExternalPolymorph`.

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 resolved. No single `ExternalPolymorph` template is able to call back to functions, function objects, and member functions.
- *Loosely Coupled:* Mostly resolved. This force is no longer fully resolved because the `CdPlayer` must know about `Buttons`.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* No change: mostly resolved. The `Button` discrimination code has been moved back into the `CdPlayer`. It has not gone away.
- *Space Time Efficient:* Mostly resolved. `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 history 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 *loosely coupled* 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 method called back 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 instead.

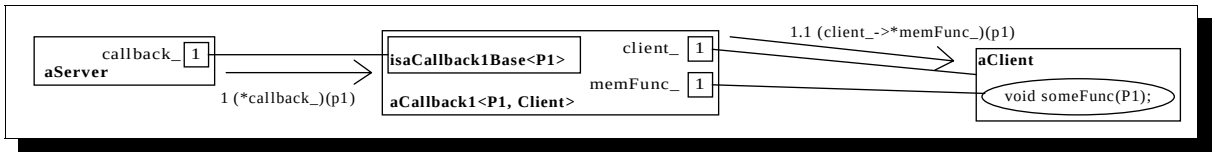

Figure 11: Abstract view after Parameterize Method Name applied to External Polymorphism

Figure 11 shows that little changed structurally between the External Polymorphism solution and this solution. While the structural changes are limited to the addition of the member function pointer, the callbacks are now a completely separate entity and could be in a library of their own.

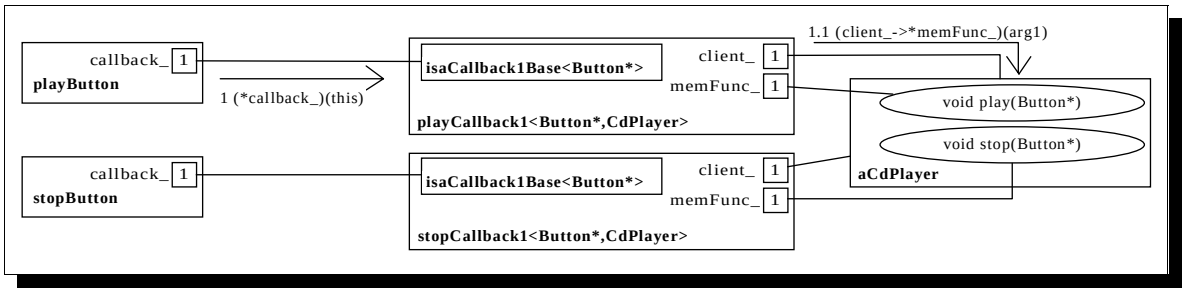

Figure 12: Application after applying Parameterize Method Name

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. Most 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` as more generic replacements for `ButtonClient` and `ExternalPolymorph`. The `Callback1Base` template, 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*> const & 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 use multiple servers of 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: not resolved. This solution can not call back to functions and function objects.
- *Loosely Coupled:* Resolved. The `CdPlayer` does not need to know about `Button`s any more. 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: resolved.
- *Simple Interface:* No change. The `CdPlayer` is simplified, but the `Button`s must worry about cloning and deleting callbacks. There are chances for memory leaks or dangling pointers.
- *Space Time Efficient:* Not resolved. 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.

```
class Base {};
class Derived: public Base { public: operator int(){return 0;} };
class Server{ void registerCb(Callback1Base<Derived&> const &); };
class Client
{
  void error1(Base&);          // arg does not match - requires trivial conversion
  int  error2(Derived&);       // return type does not match
  void error3(int);            // arg does not match - needs user defined conversion
  void error4(Derived&) const; // const not allowed by current callback
  void ok(Derived&);           // only this method can be called back
};
```

The code above illustrates the problem. The `Server` specifies that a `Callback1Base<Derived&>` can be registered to be called back. The `Client` wants to create callbacks for each of its methods, but `Callback1` is too restrictive. It specifies the exact type of pointer to member function that can be used. Even though all of `Client`'s member functions could legally be called passing a `Derived&` as the argument and ignoring the return value, the `Callback1` will only accept a method with the exact signature and return type of `Client::ok`. The problem is that `Callback1` over-specifies the requirements on the client.

The Separate Client and Server Interface Specifications pattern [Jakubik] addresses this problem, resolves the *Automatic* force, and improves the *Flexible* force. This pattern specifies a particular 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 function pointer, or a function object. While these alternatives are legal C++, some compilers have problems compiling these more advanced forms.

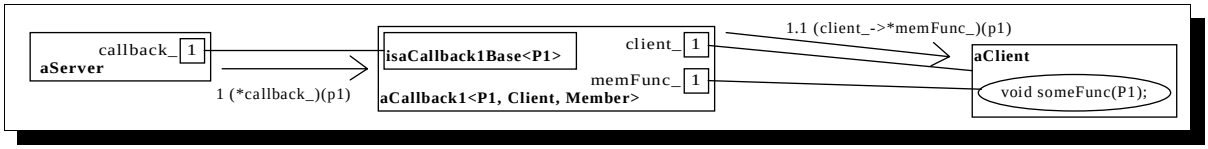

Figure 13: Abstract view after applying Separate Client and Server Interface Specifications

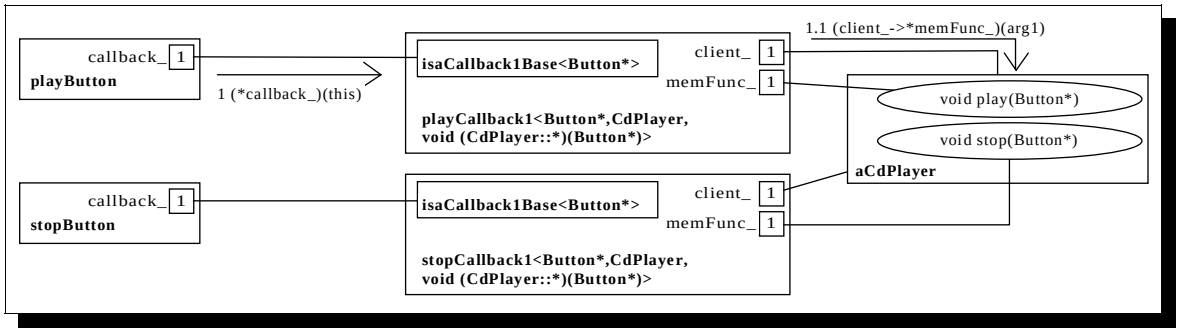

Figure 14: Application after applying Separate Client and Server Interface Specifications

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 before. 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. Additional callback classes would have to be added to the library to support callbacks with different numbers of arguments or with return types. Techniques for automatically generating these different callback types is beyond the scope of this paper. An example of this kind of automation can be found in the callback library available at [Jakubik].
- *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.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* Not resolved. The callback template is becoming unwieldy for clients to instantiate.
- *Space Time Efficient:* Not improved. 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

We want to automatically detect the types of the client and member function pointer and use them to create a callback derived from the server-specified kind of `Callback1Base`. Ideally we would just create a static member function template in `Callback1Base` as follows:

```
template <class P1>
class Callback1Base
{
public:
  // ...
  template <class Client, class Member>
  static Callback1Base<P1>* make(Client&, Member);
  // ...
};
```

While the proposed C++ standard allows member templates, most compilers do not. To make this code usable today, we simply use a function template and pass an artificial parameter to specify exactly what kind of `Callback1Base` the server is expecting.

```
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. Note how 0 is cast to be a pointer of the appropriate type to allow `make_callback` to detect the type of callback to create.

```
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: resolved.

- *Flexible:* No change: somewhat resolved. No direct callback to functions or function objects.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* Mostly resolved. Instantiating callbacks is easier. The pointer semantics are still a problem.
- *Space Time Efficient:* Not improved. `make_callback` adds extra heap allocation and a new opportunity for code bloat.

History of an Efficient Callback

FACTOR TEMPLATE

The callback library would be easier to use if the callbacks supported value semantics. The most efficient way to implement pass by value while preserving polymorphism is to apply Manual Polymorphism [Jakubik]. Manual Polymorphism will allow `Callback1Base` objects to be passed by value while preserving polymorphic behavior. To avoid slicing off data when passing the base class by value, all of the data currently being stored in derived classes must be moved to the base class.

The Factor Template method is frequently applied to templates to move all data and methods that don't depend on the template arguments up into a base class. This pattern is used to reduce the amount of code bloat caused by instantiating the template. We must move all data out of the derived class template, but the data in the derived class depends on its template arguments. We can remove this dependency by applying the Type Destroyer is Type Restorer pattern [Jakubik]. Type Destroyer is Type Restorer allows the derived class to remove the type information from the data and pass it to the base class to be stored in a generic form. When we want to use the data, the derived class will safely restore the type information as it extracts it from the base class.

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 since we are not trying to avoid code bloat. Regardless of the effects on code bloat, all data must be moved to the base class before Manual Polymorphism can be applied.

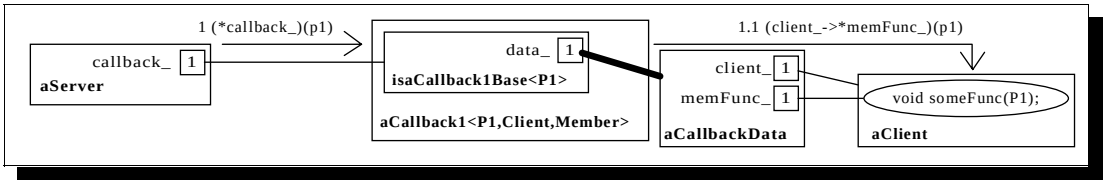

Figure 15: Abstract view of solution after applying Factor Template pattern

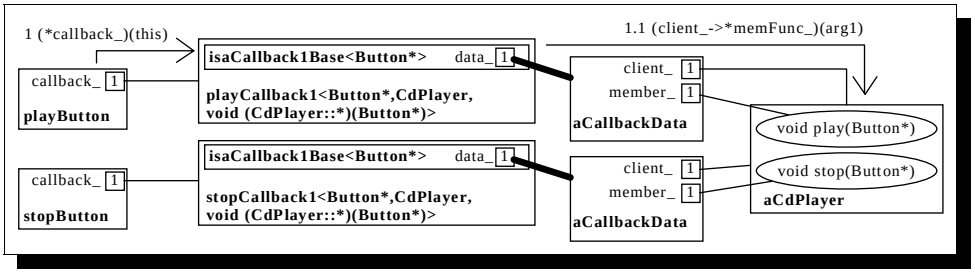

Figure 16: View of application after applying Factor Template pattern

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. Some compilers vary the size of member function pointers. For these compilers, care must be taken to make sure that the typedef `PMEMFUNC` is for the largest kind of member function pointer.

```
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 must be two `make_callback` functions, one for pointers to member functions, the other for pointers to const member functions. This change ensures that only member function pointers 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: resolved.
- *Flexible:* Less resolved. The solution is less flexible because the member function pointer must be stored in a fixed size array. The callback can no longer invoke functions and function objects through member data pointers.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* No change: mostly resolved. Callbacks must be passed by reference and cloned. We are now ready to apply Manual Polymorphism and address this problem.
- *Space Time Efficient:* No change. The data factoring may have helped or harmed code bloat, but the excess heap allocation is the bigger concern.

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.

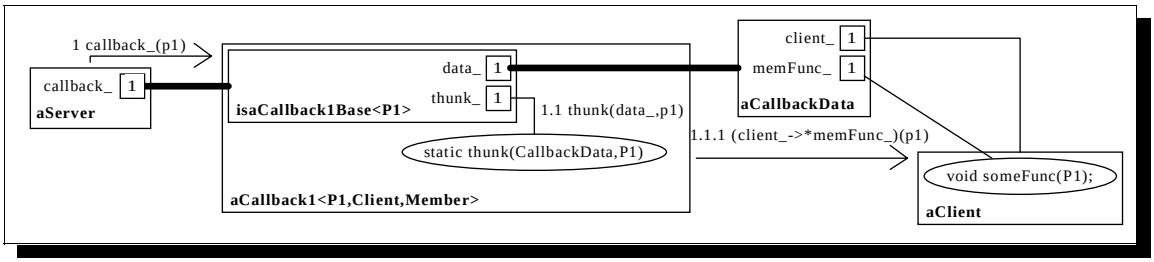

Figure 17: Abstract view of callback solution after applying Manual Polymorphism

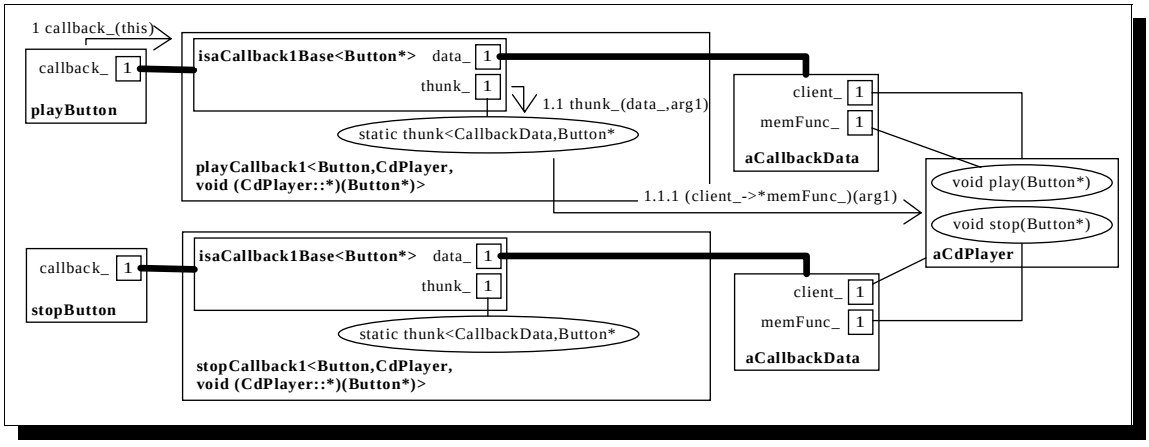

Figure 18: View of application after applying Manual Polymorhpism pattern

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 `Callback1`. 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::_thunk` 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: resolved.
- *Flexible:* No change: less resolved. Callbacks can only call back to member functions.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* Resolved. Value semantics makes using callbacks a lot simpler and safer.
- *Space Time Efficient:* 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; they can be 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 can be implemented 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.

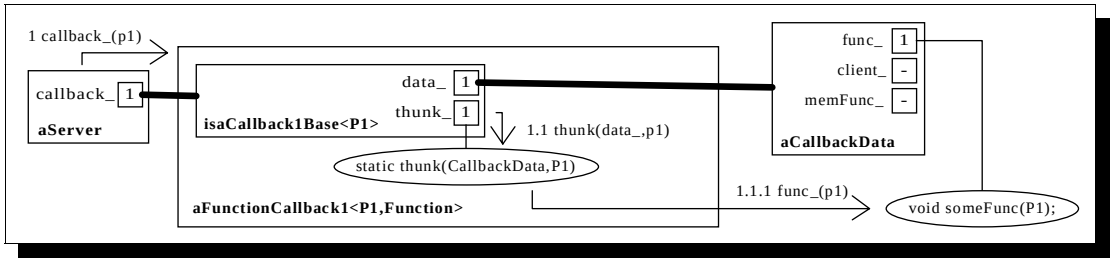

Figure 19: Abstract view of calling back to a function

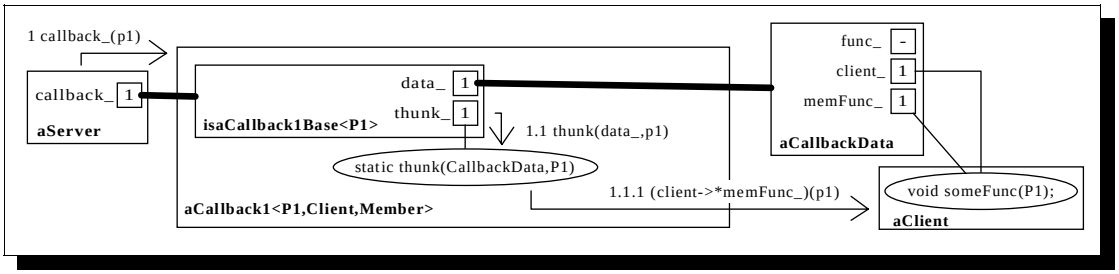

Figure 20: Abstract view of calling back to a member function

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 Purpose Class pattern was made safe by reapplying 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: resolved.
- *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.

- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* No change: resolved.
- *Space Time Efficient:* No change: mostly resolved. The only potential problem is code bloat from the templates.

History of a Simple Callback

HANDLE BODY

This step in the history occurs immediately after the section entitled Automatically Detect Types. The history of the more efficient callback is a separate branch of the history.

The more efficient 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 size 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 the 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.

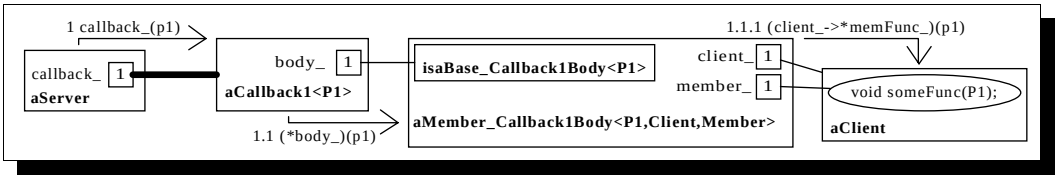

Figure 21: Abstract view of Handle Body based callback

Figure 21 shows that the server contains the handle, `aCallback1<P1>`, by value. The handle has a reference to the body, `aMember_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.

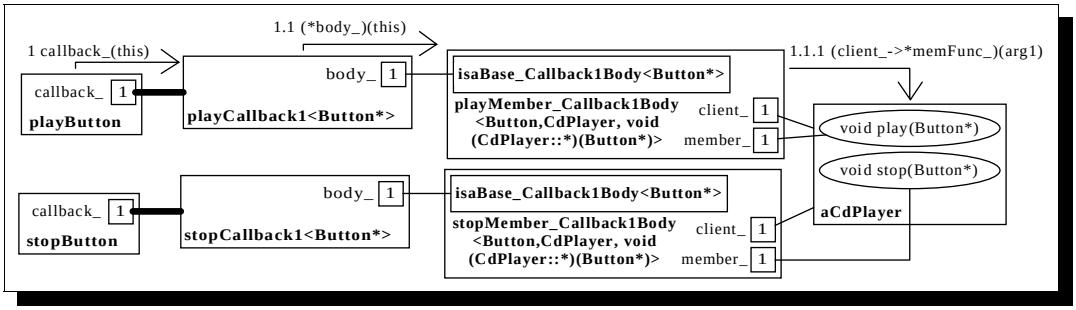

Figure 22: Application after applying Handle Body pattern

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; `Callback1Base` is now `Base_Callback1Body`, `Callback1` is now `Member_Callback1Body`. The new handle class, `Callback1`, 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::push` and `Button::~Button` are simplified by `Callback1`'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: resolved.
- *Flexible:* No change: somewhat resolved. No direct callback to functions and function objects.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* Resolved. The interface is simplified by hiding all heap allocation behind the handle object and `make_callback` function.
- *Space Time Efficient:* Not resolved. 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: resolved.
- *Flexible:* No change: somewhat resolved. Still no direct callback to functions and function objects.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* No change: resolved.
- *Space Time Efficient:* 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, there is more to a pattern than its structure. Both patterns are just basic uses of inheritance and polymorphism. The Null Object pattern is specific, which makes it very easy for an inexperienced programmer to see where to apply it. The Special Purpose Class pattern is more general, but more difficult for an inexperienced programmer to use. 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.

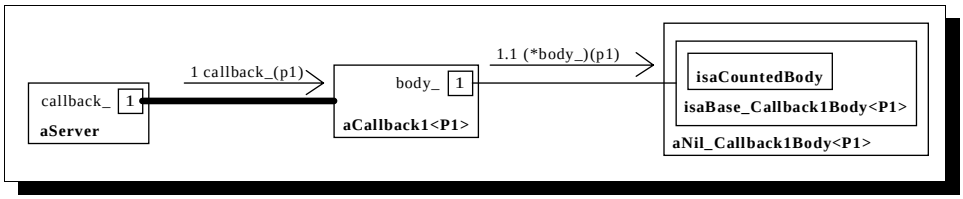

Figure 23: Abstract view of a callback transformed by Null Object pattern

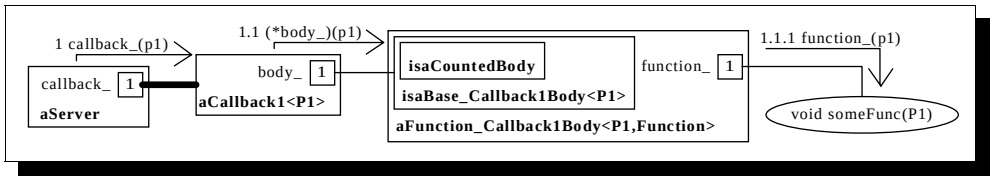

Figure 24: Abstract view of Special Purpose Object pattern application

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 is the same in C++. 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_));}
```

The `buttonTest` function and some additional code in `main` 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: resolved.
- *Flexible:* Resolved. Callbacks to functions, function objects, and member functions are supported.
- *Loosely Coupled:* No change: resolved.
- *Not Type Intrusive:* No change: resolved.
- *Simple Interface:* No change: resolved.
- *Space Time Efficient:* No change: mostly resolved. The templates could cause code bloat.

Comparing the Histories

Now that the pattern histories have been described in detail, they can be compared to determine which patterns led to better solutions and are expected to lead to better solutions when applied to other problems. The forces discussed throughout this paper give a coarse comparison of the two solutions. Both solutions completely resolved most of the forces, but did not completely resolve the *Space Time Efficient* force. In addition, the Efficient solution did not resolve the *Flexible* force. These shortcomings need to be examined in more detail.

SPACE TIME EFFICIENT

Neither solution is as space time efficient as the original object interaction presented in this paper. Each solution replaces simple method invocation with several method calls. Table 1 shows actual measurements of the time overhead for each solution. The table shows the number of high resolution clock ticks it took to perform a particular

operation 1000 times. The numbers include the loop overhead which was approximately 100 clock ticks. There were over 1.1 million clock ticks per second. The tests were performed under Windows NT on a 100 MHz Pentium processor using the Visual C++ compiler. While the timing mechanism was not as accurate as a processor clock count, it was accurate enough for rough comparisons. As expected, it takes longer to create and destroy a Simple Callback since it allocates and destroys a callback body on the heap while the Efficient callback does not perform any heap allocation. A separate test found that it took 29,000 clock ticks to heap allocate and destroy 1000 callback bodies. An unexpected result was that invoking a Simple callback is significantly faster than invoking an Efficient callback. Copy constructing and destroying a Simple callback is significantly faster than creating and destroying one since the heap allocation and deallocation is avoided. Copying and destroying an Efficient callback is even faster since the copy constructor and destructor are trivial (compiler supplied) and are actually simple stack operations.

Operation	Efficient Callback	Simple Callback
create and destroy	1800	33000
invoke	1400	1050
copy and destroy	150	1800
assign	150	400

Table 1: Performance comparison in high resolution clock ticks.

Many applications create several callbacks early on, and invoke them repeatedly. These applications can use the Simple callback with no performance penalty at all. Other applications continually create and destroy callbacks throughout the life of the application. While it appears that these applications would benefit more than others from using the Efficient callback, the benefits are not as large as they would appear to be. Typically applications that continuously create and destroy callbacks are creating them for other objects that are also being continuously created and destroyed on the heap. Instead of an order of magnitude improvement in performance, these applications are likely to see at most a factor of two improvement by using the Efficient callbacks. Applications that do not use the heap will benefit the most from using Efficient callbacks instead of Simple callbacks. Applications that only create a few callbacks or dynamically create objects at the same time new callbacks are created will benefit the least from using Efficient callbacks.

The largest time difference, for creating and destroying callbacks, is proportional to the amount of time required to allocate and deallocate memory on the heap. [Wilson] and [Neely] show that heap performance can be improved considerably over typical heap implementations, thus better compilers can reduce the heap overhead and reduce the advantage of the Efficient callback library.

Table 2 compares the size of the callbacks. The size of the Simple callbacks includes the size of the handle (four bytes). The table shows that there are many cases where the Simple callbacks are smaller than the Efficient callbacks.

Callback to…	Efficient Callback	Simple Callback
member function	24	20-36
function	24	16
nil	-	12
function object	-	12 + ?[1]

Table 2: Size in bytes of callbacks.

The table does not show the additional space efficiencies that the Simple callback can provide. Typically, the client keeps a copy of the callback it registers with a server so it can easily deregister it later. When a Simple callback is copied, only the handle (four bytes) is copied; the body is shared. Efficient callbacks never share storage.

The pattern histories also warned about the potential for code bloat. These two libraries use templates very similarly and should bloat at about the same rate.

While the Efficient callback is faster in most cases and is sometimes smaller than the Simple callback, it is impossible to say which is more *space time efficient.* When many copies of the same callback are kept, when there

[1]The size of a Simple callback to a function object depends on the size of the function object. Function objects can be any size and are stored by value inside of the callback body.

are many callbacks to functions, or when the are many nil callbacks, the Simple callback will use significantly less storage.

FLEXIBLE

After considering the *Space Time Efficient* force, neither solution comes out as definitely better. The flexible force shows more significant differences. As has already been pointed out in the pattern history, the Simple callback supports callbacks to function objects while the Efficient callback does not. The Efficient callback looks even less flexible if potential extensions to the library are considered.

Two extensions, inspired by [Laüfer], are Composition and Partial Application.

Consider the following functions:

```
Y f(X);
X g(Z);
```

Composition allows the two functions above to be combined into a single function equivalent to `Y f(g(Z))`.

Consider another function:

```
void f(X,Y);
```

Partial application allows a single argument to be applied to the above function to create a new function `void g(Y)` that when invoked will call `void f(X,Y)` passing the argument passed during Partial Application as well as passing the Y argument passed to `void g(Y)`.

The goal is to see if both callback libraries can be extended to support Composition and Partial Application for functions and function objects. Since both libraries implement callbacks as function objects, this would also mean supporting Composition and Partial Application of callbacks.

To support composition, the callback would have to store function pointers or function objects for each of the functions to be called. This storage would look similar to how the callbacks currently support calling back to a function, and how the Simple callbacks currently support callbacks to function objects.

By squeezing in storage for a second function pointer in `CallbackData`, the Efficient callbacks could support composition of functions, but not function objects. Callback composition would not be possible.

Adding composition to the Simple callback library is easier. A new kind of callback body template can be added that can store two different kinds of functions or function objects. Instead of `make_callback`, a `compose` function could be added to automatically detect the types of functions or function objects it is passed and automatically create an appropriate callback and callback body. Callback, function, and function object composition would be allowed and any combination of functions and function objects could be used.

Supporting Partial Application is more difficult. The callback libraries would have to be extended to support storage of arbitrary parameters. Since there is no bound on the size of this parameter, the Efficient callback would not be able to store it. To extend the Efficient callback library to allow Partial Application, the library would have to be changed to allocate storage on the heap, and the efficiency would be lost.

For the Simple callback library, since it always stores its data on the heap, creating another kind of callback body template that is capable of storing arbitrary parameters is easy. Instead of `make_callback`, a new function called `apply` would be used to automatically detect the type of function or function object to partially apply arguments to as well as automatically detect the types of the applied arguments to create an appropriate callback and callback body.

Clearly the Simple library is more flexible and is easier to extend with new functionality. Each extension is represented by a new kind of callback body and a new kind of creation template function. The different behaviors are cleanly separated.

COMPARING PATTERNS

The Simple callback library is the best choice for most applications. Even when extreme efficiency is necessary, deciding between the two libraries takes careful consideration since an application that invokes callbacks more often than it creates callbacks may run faster if it uses the Simple callbacks. It is also important to consider that the run time performance of the Simple callbacks is bound by the performance of the heap. Unless the application avoids all use of the heap or significantly limits its use, using the Efficient library won't significantly speed up the application.

The pattern history can now be read to easily determine which patterns led to what kind of solution. The start of both pattern histories is exactly the same. The patterns applied in the common part of the history are Abstract Client, Adapter, External Polymorphism, Parameterize Method Name, Prototype, Separate Client and Server Interface Specifications, and Automatically Detect Types. The fact that they are used in both histories suggests that they are commonly used. Beyond that, with no other patterns to compare them to, not much can be said about them.

Some conclusions can be drawn about how one pattern was applied in the common history. The Prototype pattern was applied to resolve ownership issues with callbacks. Both final solutions eliminated the callback cloning that the Prototype pattern added. If a detailed pattern history was not given in this paper, it may have appeared that the Prototype pattern should be avoided. Since the details of how the pattern was applied are known, a less sweeping generalization can be made. Using the Prototype pattern to resolve ownership issues is an abuse of the pattern. The prototype pattern should only be used in the object creation situations it was intended for. The ownership issues are better taken care of by the Handle Body and Counted Body patterns.

The unique part of each history is more interesting to look at. The first difference between the two histories appears when patterns are applied to get pass-by-value semantics while keeping polymorphic behavior. The Efficient callback library achieved this by applying the Manual Polymorphism pattern. Before Manual Polymorphism could be applied, all data had to be moved into the base class to avoid slicing. The data was moved by first applying the Type Destroyer is Type Restorer pattern to remove the dependencies between the data and the template arguments. Once the dependency was removed, the Factor Template pattern was used to factor the data into the base class.

Casual inspection of the Efficient callback branch of the history would suggest that all three patterns mentioned so far tend to lead to less flexible solutions. This is another place where presenting the pattern history in detail pays off. The detailed pattern history shows that this was not a typical application of Factor Template. The Factor Template pattern is normally used when data or methods in a template class do not depend on the template arguments. The use of Factor Template in the Efficient library history is not typical since the data required coercion to be factored, thus the results from this history should not be used to judge how good the Factor Template method is.

All of the work performed to make applying Manual Polymorphism possible highlights its shortcomings. The requirement that all data must be stored in the base class is limiting and difficult to achieve with interesting class hierarchies. The problems that Manual Polymorphism causes get worse when the history tries to extend the Efficient library to be able to call back to functions. Instead of separating all of the behavior into a separate derived class, Dual Purpose Class had to be used to wedge the function data into the same class where the object and member function data can be stored.

Manual Polymorphism limits the flexibility of the Efficient callbacks. By forcing all data into the base class, derived classes are forced to live with the amount of storage the base class provides. Even though the syntax and template semantics allow more flexibility, the implementation of the Automatically Detect Types has to be changed to restrict the types that can be passed to the `make_callback` function to those that are guaranteed to be the correct size. According to the paper the Efficient callback solution first appeared in, [Hickey], the alterations to the `make_callback` function are not standard C++ but seemed to work on several C++ compilers. There is at least one popular compiler on which this nonstandard use of templates does not work.

If all of the problems caused by Manual Polymorphism were not enough, additional issues come up when using a compiler that uses more than one size of member function pointer. At least one popular compiler uses several different member function pointer sizes. For these compilers, care must be taken to make sure that the size of the buffer set aside for member function pointers is large enough for all kinds of member function pointers.

Clearly Manual Polymorphism causes enough problems that it should only be used in applications where heap allocation must be avoided at all costs.

This leaves Type Destroyer is Type Restorer and Dual Purpose Class from the Efficient callback history. In this pattern history, these patterns are used to force a particular solution to be possible. Since the final solution that these patterns allowed has many shortcomings, these patterns should not be recommended. Without these patterns, the developer would have been forced to rethink the decision to use Manual Polymorphism. Until someone comes up with an example where these patterns are necessary and there are no good alternatives, they should be avoided and any use should require significant justification.

The Simple callbacks take a very different route to achieve the same goals. Instead of applying Manual Polymorphism, the Simple callback's history applied the Handle Body pattern. This pattern avoids the slicing issues, and all of the data factoring machinations needed to prevent slicing, by passing a simple handle around instead of the

actual polymorphic class. The handle can then implement a policy to ensure that the bodies are copied or deleted when necessary. The Counted Body pattern provided an efficient policy for controlling the lifetime of bodies and to allow the sharing of bodies among handles. Finally Null Object provided an example of how to implement special purpose behavior in the callback bodies. The special purpose behavior is added by using the basic properties of object-oriented programming and polymorphism. These patterns lead to a much more flexible solution that can be extended without cramming several mutually exclusive behaviors into a single class. The flexibility of the Simple callbacks, including the potential extensions discussed in this paper, and the efficiency of the resulting implementation suggest that for C++ these patterns balance the forces involved.

Conclusions

Pattern histories are an effective way to show the effects of applying different patterns. While it was not mentioned in this paper, the pattern history also proved to be an effective form of design documentation. The author would have never discovered how to implement the Simple callbacks without first identifying the pattern history for the Efficient callback library. That pattern history exposed why Manual Polymorphism was being used and made it easy to see that an alternative set of patterns could be applied.

The patterns community is beginning to identify many different patterns that all solve the same problem. Since patterns are supposed to lead to better solutions, the solutions that these patterns lead to need to be compared to better determine when each pattern should be used. When solutions are compared, they should be presented along with a pattern history so the reader can decide if each pattern was properly applied. Without a detailed discussion of the pattern history, Factor Template would have appeared to have led directly to a bad solution and Prototype would have appeared too inefficient to be useful anywhere.

The detailed pattern histories are also necessary to allow other people to rebut the results presented in a paper. Presenting a sketchy list of patterns applied to a solution and moving straight to conclusions about which patterns should be used would not be any better than claiming that a pattern should be used because an expert wrote it and it has been used in three or more software systems.

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.

Acknowledgments

This paper owes a lot to the article *Callbacks in C++ Using Template Functors* by Richard Hickey [Hickey]. The more efficient callback solution was originally presented in [Hickey]. The idea for using a CD player as the running example was also taken from that paper. The forces in this paper were inspired by the requirements for a good callback solution presented in [Hickey]. I would not have found the patterns *Separate Client and Server Interface Specifications*, *Automatically Detect Types*, *Factor Template*, *Type Destroyer is Type Restorer*, *Manual Polymorphism*, and *Dual Purpose Class* without seeing their uses in Hickey's solution. Finally, and most importantly, it was by identifying all of the patterns used in Hickey's callbacks that the alternative history leading to the simpler solution was discovered. The initial inspiration for this paper was reading [Hickey] and an early version of [Cleeland] and comparing these to my own callback work.

This paper was evaluated in a writer's workshop at PLoP '96 and was improved based on those comments. The comments received when submitting the paper to the TOOLS conference were also very helpful. I want to thank Doug Schmidt for encouraging me to write this paper, Paul Greco, Mike Slocum, and Grant Gibson for critiquing early versions of the paper, and my wife, Angela, for constantly proofreading the paper and putting up with me while I wrote it.

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.

[Cleeland] Cleeland, Chris, Douglas C. Schmidt, and Tim Harrison, "External Polymorphism", PATTERN LANGUAGES OF PROGRAM DESIGN, volume 3, edited by Robert Martin, Frank Buschmann, and Dirke Riehle, Addison-Wesley, Reading MA, 1997.

[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.

[Laüfer] Laüfer, Konstantin, "A Framework for Higher-Order Functions in C++", PROC. CONF. OBJECT-ORIENTED TECHNOLOGIES (COOTS), Monterey, CA, June 1995.

[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, volume 1, 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.

[Schmidt] 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.

[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.