EventListener library

Frantz Maerten

Igeoss

Contents

1 Introduction

This package recreate an Event pattern similar to the one used in Java. It uses the patterns Listener and Observer/Observable to connect objects between them by mean of listeners, and create communication by propagating events.

`Event`s can be emitted from anywhere through a `Publisher` and be intercepted by a `Subscriber` which can listen to these `Event`s by using appropriate `Listener`s or having a method with the type of event as pointer argument.

In most common cases, client code will use a `Publisher` to emit custom `Event`s, and a `Subscriber` to receives events, the connection being done by the use of the `igs_connect` template function. Other classes of this package can be avoided (`EventFilter`, `Listener`, etc...).

2 Revisions

- 20060103
  Added priority Listener for a given Event type. The priority can be changed dynamically if necessary. Little drawback in speed.
- 20051213
  Simplified interface for `Publisher`.
- 20051212
  Added `MultiplexerListener` class to simplify `Listener` declarations.
  It is now possible to avoid creating a `Listener` class for a given `Event`. Three templated functions are introduced to simplify the mechanisum of connection/disconnection:
  - `igs_connect(Publisher*, Subscriber*, igs_functor/igs_function,int)`
  - `igs_disconnect(Publisher*, Subscriber*, igs_functor/igs_function,int)`
  - `igs_get_listener(Publisher*, Subscriber*, igs_functor/igs_function,int)`
- 20051126
  Added `ListenerForEvent<E>` class to simplify `Listener` declarations.
  Use now `std::list` instead of `std::vector`, for performance reasons.
- 20051014
  First revision of the package which is fully functional.

3 Performaces

All the tests were performed on a Pentium IV, 600 Mhz (Laptop Dell Latitude D800) using gcc-3.4.3 on Mandriva Linux 10.1

3.1 `Listener`

1. 0.85us per event, or 0.85 second for 1 millions of events
2. 15 times faster than `boost::signals`
3. 5 times faster than `Qt` Signal/Slot

It can be noted that the use of `ListenerForEvent<E>` is 5 times slower than using a classical `Listener`, since a special filter is used, and that a new pure virtual method is added.

3.2 `MultiplexerListener`

1. 1.02us per event, or 1.02 second for 1 millions of events
2. 14 times faster than `boost::signals`
3. 4.5 times faster than `Qt` Signal/Slot

4 Recommandations

1. We recommand to inherits your class from `Subscriber` in order to auto-removed the installed listeners (done in `Subscriber`).
2. In most cases you will only use `igs_connect` and `igs_disconnect` to connect/disconnect to a given event type. Using these functions allows to avoid using and creating a `Listener` for a given `Event`, as it is automaticaly created.

5 Features

1. Listeners support the notion of priority. When two listeners are connected to a same event type, they are called according to their priority. The priority can be changed at any time if necessary. When Listeners have the same priority, the are called in the order of registration. This priority is set to 0 by default, and can be specified by the `igs_connect` method (last parameter).

2. `Listener` and `Subscriber` supports `EventFilter` and can have more than one

3. `Subscribers` can:

   (a) Inherit from `Listener`, and override the virtual functions

   (b) Use `listener`s with the `add_listener()` method with a `igs::Function` connected to.

4. Source of the event (`Publisher`) available when:

   (a) Receiving an event. The `Event` class has the appropriate method.

   (b) A member function of a `Subscriber` is called due to an event. The `Publisher` is, in that case, available only within the function scope.

5. Pluggable `EventFilter`s for:

   (a) `Listener`

   (b) `Subscriber`

6. Block/unblock events from sending/receiving for:

   (a) `Listener` (sending)

   (b) `Publisher` (sending)

   (c) `Subscriber` (receiving)

7. Auto unregistration of:

   (a) `Subscriber`s and their associated `Listener`s when deleted

   (b) `Listener`s when a `Publisher` is deleted

8. Auto deletion of:

   (a) `Event`s after posting

   (b) `EventFilter`s when they are not used anymore

   (c) `Listener`s when a `Subscriber` is deleted

9. Auto listening of methods of type `myevent(EVENT*)`

6 Classes

This package is a very light one. It uses few classes to describe this event pattern:

6.1 Event

The argument which is propagated from the `Publisher` to the `Subscriber` by the use of `Listener`. There can be as many types of `Event`s as needed.
`Event` inherits from `GenObject`, and therefor is a smart-pointer. You don't have to worry about deallocations of events after emission.
Its interface is:

```
class Event {
public:
  Publisher* get_source() const ;
} ;
```

6.2 Listener

This class is responsible of doing the connection of a `Subscriber` to a `Publisher` for an `Event` of a given type. `Listener` have a unique ID, which is used to retrieve it from a `Subscriber`, in order to disconnect it or add `EventFilter`s if necessary.
Another derived class `ListenerForEvent<EVENT>` can also be used. This class specialize the `Listener` class for a given event template parameter `EVENT`. The pure virtual method `process_event(Event*)` is replaced by the specialized one `process_event(EVENT*)`, where `EVENT` is the template parameter.

6.3 Publisher

This class is responsible of posting `Event`s. Posting events can be done from the `Publisher` directly, or outside since the method `Publisher::post_event(Event*)` is public.
Its interface is:

```
class Publisher {
public:
  int post_event(Event* e) ;
  void block_all_events() ;
  void unblock_all_events() ;
} ;
```

and the related macro is:

```
igs_emit(EVENT) ;
```

6.4 Subscriber

Classe which connect to a `Publisher` in order to receives `Events` of a certain type, by the use of `Listener`. Any `Subscriber` interested in a given `Event` from a `Publisher`, can subscribe for receiving `Events` by using an appropriate `Listener`.
Its interface is:

```
class Subscriber {
public:
  Listener::ID add_listener(Listener* l) ;
  Listener* get_listener_with_id(const Listener::ID&) const ;
  bool remove_listener_with_id(const Listener::ID& id) ;
  void remove_all_listener() ;
  void add_event_filter(Listener::ID, EventFilter*) ;
  void add_event_filter(EventFilter*) ;
  void remove_all_event_filters() ;
  void block_events() ;
  void unblock_events() ;
} ;
```

and the related macro is:

```
igs_add_listener(PUBLISHER, LISTENER, FUNCTION)
```

6.5 EventFilter

When `Events` are posted, sometime it is interresting to filter them, in order to only receive only appropriate `Events`. This class is used for that purpose.
A derived class `EventFiltering` is also defined, which allows to automatically filter events of a given type. This class is used in the `ListenerForEvent<EVENT>`.
Its interface is:

```
class EventFilter {
public:
  virtual bool filter(Event* e) = 0 ;
} ;
```

6.6 MultiplexerListener

A derived class from `Listener`. Since receiving an event is characterized by a unique signature of a method (for example `void receives_event1(Event1*)`), it is possible to simplify the definition of the pair (Event,Listener). The use of the `MultiplexerListener` allows this feature, by only defining `Event` classes, and to listen to these events in a simple way.
The `Subscriber` methods to connect to any type of `Event` take only the event type as parameter. Using this class, it is unecessary, when creating a event of a given type, to create its corresonding listener.

6.6.1 `igs_connect` and `igs_disconnect`

Using the `igs_connect` and `igs_disconnect` global template functions facilitate the processus of connection and disconnection.
The interfaces are:

```
template <typename E>
Listener::ID igs_connect(Publisher* publisher,
                         Subscriber* susbscriber,
                         const basic::function1<void, E*>& f,
                         unsigned int priority=0) ;
template <typename E>
bool igs_disconnect(Publisher* puslisher,
                    Subscriber* subscriber,
                    const basic::function1<void, E*>& f) ;
```

and the related macros are:

```
igs_functor(SUBSCRIBER, METHOD) ;
igs_function(FUNCTION) ;
```

Example of use:

```
class Event1: public Event {} ;
class Event2: public Event {} ;
class Event3: public Event {} ;

class S: public Subscriber {
public:
  void event1(Event1*) {std::cerr << "1\n" ;}
  void event2(Event2*) {std::cerr << "2\n" ;}
  void event3(Event3*) {std::cerr << "3\n" ;}
} ;

int main() {
  Publisher p ;
  S         s ;
  igs_connect(&p, &s, igs_functor(&s, S::event1)) ;
  igs_connect(&p, &s, igs_functor(&s, S::event2)) ;
  igs_connect(&p, &s, igs_functor(&s, S::event3)) ;
  p.igs_emit(Event1) ;
  p.igs_emit(Event2) ;
  p.igs_emit(Event3) ;
  igs_disconnect(&p, &s, igs_functor(&s, S::event3)) ;
  p.igs_emit(Event3) ; // never received
}
```

Here, we see that no special `Listener` for each `Event` is created, and that the `MultiplexerListener` class is used through the `igs_connect` function to listen to the special events defined by

the S member functions. This class automatically recognize the Event type argument and do the necessary connections.
Note that the igs_connect returns the created Listener, which allows you block/unlock, add EventFilter and remove it from the Subscriber.

The reason why the igs_connect takes a Subscriber pointer and the repeated Subscriber pointer within the functor definition, is because you can use a functor from another class, or a C function as in the following code:

```
class S {
public:
  void event1(Event1*) {}
  void event2(Event2*) {}
} ;

void event3(Event3*) {}

int main() {
  Publisher* pub = new Publisher ;
  Subscriber sub ;
  S* s = new S ;

  igs_connect(pub, sub, igs_functor(s, S::event1)) ;
  igs_connect(pub, sub, igs_functor(s, S::event2)) ;
  igs_connect(pub, sub, igs_function(event3)) ;

  pub->igs_emit(Event3) ;
  ...
}
```

7 Example 1: Templated events

This example show a very simple way to use templated events:

```
template <typename T>
class TEvent: public Event {
public:
  TEvent(const T& t): t_(t){}
  const T& t() const {return t_ ;}
private:
  T t_ ;
} ;

class S: public Subscriber {
public:
  template <typename T>
  void receives_event(TEvent<T>* e) {
    std::cerr << "receives " << typeid(T).name() << ": " << e->t() << std::endl ;
  }
} ;

int main() {
  Publisher p ;
  S s ;

  igs_connect(&p, &s, igs_functor(&s, S::receives_event<double>)) ;
  igs_connect(&p, &s, igs_functor(&s, S::receives_event<std::string>)) ;
  p.igs_emit(TEvent<double>(1.23)) ;
  p.igs_emit(TEvent<int>(123)) ;
  p.igs_emit(TEvent<std::string>("Hello World")) ;

  std::cerr << std::endl ;

  igs_disconnect(&p, &s, igs_functor(&s, S::receives_event<double>)) ;
  igs_disconnect(&p, &s, igs_functor(&s, S::receives_event<std::string>)) ;
  igs_connect(&p, &s, igs_functor(&s, S::receives_event<int>)) ;
  p.igs_emit(TEvent<double>(1.23)) ;
  p.igs_emit(TEvent<int>(123)) ;
  p.igs_emit(TEvent<std::string>("Hello World")) ;
}
```

Will print:

```
S   receives d: 1.23
S   receives Ss: Hello World

S   receives i: 123
```

8 Example 2: Using `EventFilter`

```
class DoubleEvent: public Event {
public:
  DoubleEvent(double d): d_(d) {}
  double d() const {return d_ ;}
public:
  double d_ ;
} ;

class S: public Subscriber {
public:
  void event(DoubleEvent* e) {std::cerr << "d = " << e->d() << std::endl ;}
} ;

class SFilter: public EventFilter {
public:
  SFilter(double limit): limit_(limit) {}
  virtual bool filter(Event* e) {
    DoubleEvent* ee = dynamic_cast<DoubleEvent*>(e) ;
    if (ee && ee->d()>=limit_) return false ;
    return true ;
  }
private:
  double limit_ ;
} ;

int main() {
  Publisher* p = new Publisher ;
  S* s = new S ;

  igs_connect(p, s, igs_functor(s, S::event)) ;

  // Add the filter not to S but to the Listener
  Listener* l = igs_get_listener(p, s, igs_functor(s, S::event)) ;
  assert( l != NULL) ;
  l->add_event_filter(new SFilter(100)) ;

  p->igs_emit(DoubleEvent(50)) ;
  p->igs_emit(DoubleEvent(99)) ;
  p->igs_emit(DoubleEvent(100)) ; // never received
  p->igs_emit(DoubleEvent(200)) ; // never received
  p->igs_emit(DoubleEvent(1)) ;
}
```

Will print:

```
50
99
1
```

9 Example 3: Event recorder

```
class Event1: public Event {} ;
class Event2: public Event {} ;
class Event3: public Event {} ;
class Event4: public Event {} ;

class A: public Subscriber {
public:
  void event1(Event1*) {std::cerr << "1\n" ;}
  void event2(Event2*) {std::cerr << "2\n" ;}
  void event3(Event3*) {std::cerr << "3\n" ;}
  void event4(Event4*) {std::cerr << "4\n" ;}
} ;

int main() {
  EventRecorder recorder ;
  recorder.igs_record(Event1) ;
  recorder.igs_record(Event4) ;
  recorder.igs_record(Event2) ;
  recorder.igs_record(Event3) ;

  Publisher*  p = new Publisher ;
  A* a = new A ;
  igs_connect(p, a, igs_functor(a, A::event1)) ;
  igs_connect(p, a, igs_functor(a, A::event2)) ;
  igs_connect(p, a, igs_functor(a, A::event3)) ;
  igs_connect(p, a, igs_functor(a, A::event4)) ;

  recorder.play(p) ;
}
```

Will print:

```
1
4
2
3
```

10 Example 4: CD Player

```
class StartEvent: public Event {} ;
class StopEvent : public Event {} ;

class CDPlayer: public Subscriber {
public:
  void start(StartEvent*) {std::cerr << "Start the CD player\n" ;}
  void stop (StopEvent*)  {std::cerr << "Stop  the CD player\n" ;}
} ;

template <typename E>
class Button: public Publisher {
public:
  void click() {igs_emit(E) ;}
} ;

int main() {
  Button<StartEvent> start_button ;
  Button<StopEvent>  stop_button  ;
  CDPlayer           cd ;

  igs_connect(&start_button, &cd, igs_functor(&cd, CDPlayer::start)) ;
  igs_connect(&stop_button , &cd, igs_functor(&cd, CDPlayer::stop))  ;

  start_button.click() ;
  stop_button. click() ;
}
```

11 Example 5: RadioStation

```
using namespace listener ;

class Message: public Event {
public:
  Message(const std::string& m): m_(m) {}
  const std::string& message() const {return m_ ;}
private:
  std::string m_ ;
} ;

class RadioOff: public Event {
} ;

class RadioStation: public Publisher {
public:
  RadioStation(const std::string& name): name_(name) {}
  ~RadioStation()                 {igs_emit(RadioOff) ;}
  void send(const std::string& m) {igs_emit(Message(name_+": "+m)) ;}
  const std::string& name()       {return name_ ;}
private:
  std::string name_ ;
} ;

class Receiver: public Subscriber {
public:
  void message(Message* e)    {
    std::cerr << e->message() << std::endl ;
  }
  void radio_off(RadioOff* e) {
    RadioStation* s = dynamic_cast<RadioStation*>(e->get_source()) ;
    std::cerr << "Radio " << s->name() << " is now off" << std::endl ;
  }
} ;

int main() {
  RadioStation* radio1 = new RadioStation("105.5") ;
  RadioStation* radio2 = new RadioStation("98.3 ") ;
  Receiver*     r = new Receiver ;

  igs_connect(radio1, r, igs_functor(r, Receiver::message)) ;
  igs_connect(radio1, r, igs_functor(r, Receiver::radio_off)) ;
  igs_connect(radio2, r, igs_functor(r, Receiver::message)) ;
  igs_connect(radio2, r, igs_functor(r, Receiver::radio_off)) ;
```

```
  radio1->send("News") ;
  radio2->send("Music") ;

  delete radio1 ;
  radio2->send("Informations") ;

  igs_disconnect(radio2, r, igs_functor(r, Receiver::message)) ;
  radio2->send("Variety") ;

  delete radio2 ;
}
```

Will print:

```
  105.5: News
  98.3 : Music
  Radio 105.5 is now off
  98.3 : Informations
  Radio 98.3  is now off
```

12 Example 6: Priority

Show how to use the priority. In this example, 3 Subscribers are created ($s[i], 0 \leqslant i \leqslant 2$), with different priorities. The higuest priority is 0 (default value).

```
using namespace listener ;

class Event0: public Event {
} ;

class S: public Subscriber {
public:
  void receives(Event0*) {}
} ;

int main() {
  Publisher p ;
  S s[3] ;

  // Connections using priority (last argument)
  igs_connect(&p, &s[0], igs_functor(&s[0], S::receives), 110) ;
  igs_connect(&p, &s[1], igs_functor(&s[1], S::receives), 2761) ;
  igs_connect(&p, &s[2], igs_functor(&s[2], S::receives), 0) ;

  p.igs_emit(Event0) ;
}
```

Order of call:

```
s[2] // with priority 0
s[0] // with priority 110
s[1] // with priority 2761
```