Why another wrapper?

The usual idiom used when writing zmq::poll based code is based on a block of if_conditions, testing for the occurrence of one event or the other:

```
zmq::poll();
if event
actionHandler(state);
if another_event
another_actionHandler(state);
```

This might be error prone and hard to maintain in the long run. On the other hand more complex behavior is at least difficult to implement (for example varying actions based on state changes). And this is where the wrapper comes handy: it clearly separates poll functionality from actions and state (possible in 4 different ways) and provides the boiler plate of zmq::poll functionality.

How it works?

The user just writes the actions and assembles them together with the other required artifacts (events, sockets). There are 4 distinct ways of writing the actions and mixing them with state:

1: as free functions (they can share state at global level) - see server.cpp example

```
        typedef void (EVT_OP)(zmq::socket_t* s);
        void op1 (zmq::socket_t* s)
        { /*...*/}
        void op2 (zmq::socket_t* s)
        { /*...*/}
        reactor<EVT_OP> r;
r.add(s, ZMQ_POLLIN, &op1);
r.add(s1, ZMQ_POLLIN, &op2);
while (1) {
 r();
}
```

2. as free functions sharing common state (state can be at scope level) - see server2cpp example

```
        struct State
        {/*...*/};
        typedef void (EVT_OP)(zmq::socket_t* , State* );
        void op1 (zmq::socket_t* s, State* state)
        {/*...*/}
        void op2 (zmq::socket_t* s, State* state)
        {/*...*/}
        reactor<EVT_OP> r;
r.add(s, ZMQ_POLLIN, &op1);
r.add(s1, ZMQ_POLLIN, &op2);
State state;
while (1)
```

```
{
 r(&state);
}
```

3. polymorphic functors (each might keep its own state) - see server.cpp example

```
        struct IReactorEvent
        {
        virtual void operator()(zmq::socket_t* s)  = 0;
        virtual ~IReactorEvent(){};
        };
        struct ReactorEvent1 : IReactorEvent
        {
         virtual void operator() (zmq::socket_t* s)
         {/*...*/}
        };
        struct ReactorEvent2 : IReactorEvent
        {
         virtual void operator() (zmq::socket_t* s)
         {/*...*/}
        };
        reactor<IReactorEvent> r;
ReactorEvent1 e1(state);
r.add(s, ZMQ_POLLIN, &e1);
ReactorEvent2 e2(state);
r.add(s1, ZMQ_POLLIN, &e2);
while (1) {
 r();
}
```

4. “duck typing” interface that can hide arbitrary, non-related classes - see server1.cpp example

```
        template <class T>
        struct ReactorEvent1
        {/*...*/
        template <class T>
        struct ReactorEvent2
        {/*...*/
reactor<PollEventInterface> r;
ReactorEvent1<STATE> e1(State0);
PollEventInterface pe1(&e1);
r.add(s, ZMQ_POLLIN, &pe1);
ReactorEvent2<STATE> e2(State1);
PollEventInterface pe2(&e2);
r.add(s1, ZMQ_POLLIN, &pe2)[
while (1)
{
 r();
}
```

4. was introduced for scenarios when virtual functions from 3. are:
- small enough to be inlined (but cannot be due to virtualness)
-used in inner, tight loops (and this is mostly the case)
For possible penalties see Technical Report on C++ Performance: http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf

Notes:
A>There are other possible implementation for action handlers: std::function & boost::fusion, not taken into account for this implementation (std::function for performance (http://www.boost.org/doc/libs/1_44_0/doc/html/function/misc.html#id1285008) and boost::fusion for being different enough not to justify the complexity)

B>More complex functionality can be easily obtained by reconfiguring at runt-ime State and/or Actions, for example (see server1.cpp) :

```
reactor<PollEventInterface> r;
//assuming we have 2 PollEventInterface (pe1, pe2)
        //initially connected  to 2 EventHandlers (e1, e2)
        //and another EventHandler (e3)
//all using a common State, pe1 and pe2 initially triggered by s & s1
while (1)
{
 state.changedTo(SOME_STATE) ? pe2 = &e3 : pe2 = &e2 ;
 r();
}
```

C>The whole wrapper is provided by 2 headers, one being optional (implements the “duck typing” functionality)
D>The code should compile with no problems using:
-VS2010 on Windows
-g++ > 4.3 on Linux (using --std=gnu++0x)
for older compilers “auto” is most probably not supported and has to be replaced with the correct type
E>For more complicated requirements see Matt Weinstein’s Reactor Pattern (git://github.com/mjw9100/zmq_reactor.git)