Roman Ulan

Wzorce silników zdarzeń w C++

Część 2: Wzorzec Reactor i jego implementacja z wykorzystaniem współdziałających wzorców

W poprzednim artykule przedstawiłem wzorzec *Reactor* niemalże wyłącznie z imienia i nazwiska – w oparciu o koncepcyjną implementację. Natomiast w ninejszym pokażę nieco bardziej profesjonalną (i nieco bardziej złożoną) odsłonę tego wzorca, wykorzystując inne współdziałające wzorce oraz stosując w pełni zasady GRASP i SOLID.

O serii „Wzorce silników zdarzeń"

Intencją serii „Wzorce silników zdarzeń" jest dostarczenie usystematyzowanej wiedzy bazowej początkującym programistom, projektantom i architektom. Przez projektanta lub architekta rozumiem tutaj rolę pełnioną w projekcie.

W kolejnych artykułach będę przedstawiał najbardziej użyteczne wzorce rozwiązujące znaczną część problemów powiązanych z tą niebanalną tematyką. Każdy z wzorców postaram się dokładnie opisać od strony technicznej (obiektów i ich powiązań na podstawie schematów UML oraz implementacji w C++), a także wad i zalet w konfrontacji wydajności i prostoty w fazie utrzymania.

WZORZEC REACTOR I JEGO ZŁOŻONOŚĆ OBIEKTOWA

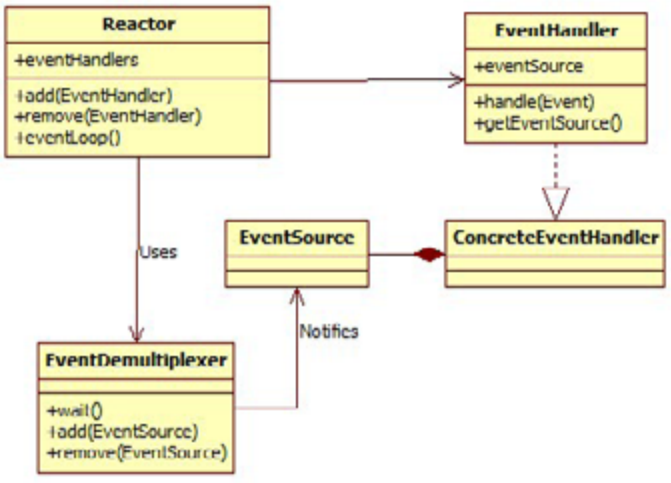
By poprawnie pisać w ogóle kod przy użyciu C++ (i ogólnie języków obiektowych), należy poprawnie stosować paradygmat *OOP*, wykorzystując przy tym zasady wspomagające: GRASP i SOLID.

Podstawową zasadą w *OOP* jest tworzenie obiektów zgodnie z ich naturą (odzwierciedlanie rzeczywistości).

Rodzi się więc pytanie: czym w rzeczywistości jest *Reactor*?

Na bazie poprzedniej części artykułu można stwierdzić: *Reactor* pozwala na rejestrowanie w nim obiektów obsługujących zdarzenia, na które *Reactor* czeka i pobiera za pośrednictwem demultipleksera zdarzeń.

Już z tej krótkiej definicji można wyodrębnić przynajmniej trzy obiekty: *Reactor*, *EventHandler* oraz *EventDemultiplexer*. Czy to już wszystko? Nie, by uzyskać pełną listę obiektów, musimy „rozebrać na czynniki pierwsze" wszystkie mniej lub bardziej złożone obiekty, a tym w w/w definicji Reactor'a jest demultiplekser zdarzeń (ED). Mianowicie ED zgłasza fakt wystąpienia zdarzenia dla konkretnego źródła zdarzeń. Co za tym idzie do naszej listy obiektów wzorca dochodzi, na pierwszy rzut oka niewidoczny, EventSource.



Rysunek 1. Diagram klas wzorca Reactor

POWRÓT DO PRZYKŁADU

Przypominam, że nasz przykład to klasyczny już program serwerowy, do którego podłączają się programy klienckie (np. telnet), wysyłają wiadomość i otrzymują zwrotne echo.

Demultipleksacja zdarzeń jest oparta o *epoll* (młodszy brat poll'a i select'a).

We wstępie nadmieniłem, że przedstawiona uprzednio implementacja wzorca jest jedynie koncepcyjna, i to prawda, a jej wady to między innymi słaba izolacja komponentów z różnych domen, która osłabia podatność tego kodu na testowanie. Może nie widać tego na pierwszy rzut oka, ponieważ zastosowałem tam wiele uproszczeń i ograniczeń obsługi błędów, ale przetestowanie tego kodu jednostkowo czy modułowo wymaga pewnej niepotrzebnej gimnastyki.

Co zatem zrobić, by nasz kod nieco ulepszyć?

Punktem wyjścia jest tradycyjny podział aplikacji na warstwy:

1. Frontend – GUI
2. Middleware
3. Domeny funkcjonalne

Warstwa GUI implementuje elementy odpowiadające za kontakt z użytkownikiem. W naszej aplikacji *frontend* właściwie nie istnieje, jest to typowa serwerowa, *backend'owa* aplikacja.

Middleware to warstwa sprzęgania obiektów domen funkcjonalnych w obiekty i funkcje realizujące przypadki użycia (*use cases* lub *user stories*).

Natomiast warstwa domen funkcjonalnych to niskopoziomowe obiekty będące instrumentami odgrywającymi pojedyncze akordy przypadków użycia.

Do takiej segregacji można podejść dwojako: „od ogółu do szczegółu" lub odwrotnie: „od szczegółu do ogółu". Obie metodologie są poprawne, a ich wybór to właściwie (chyba) tylko kwestia usposobienia osoby projektującej. Mnie bliższa jest ta druga (choć nie zawsze nią podążam), i na jej bazie sprecyzuję komponenty warstwy domen funkcjonalnych (przez komponent rozumiem zestaw obiektów z jednej domeny funkcjonalnej), a są to:

1. `IO` – komponent reużywalnych obiektów komunikacji takich jak: `Socket`, `Epoll`,...
2. `EventEngines` – komponent reużywalnych obiektów (wzorców) silników zdarzeń i obiektów/interfejsów z nimi ściśle powiązanych, są to: `Reactor`, `EventHandler`, `EventDemultiplexer`, `EventSource`.

TTS Company

Największy wybór profesjonalnego oprogramowania w Polsce !

... w ofercie programy ponad 300 producentów ...

Adobe · Microsoft · Kroll Ontrack. · progeSOFT · TENABLE Network Security

BRICSYS · Symantec. · KERIO · SMARTBEAR · ALTOVA

zend · DameWare part of the solarwinds family · JetBRAINS · McAfee · SPARX SYSTEMS

FLUKE networks · IDM · neuxpower · famatech · embarcadero

intel Software · TechSmith · ComponentOne · ELCOMSOFT PROACTIVE SOFTWARE · KASPERSKY

www.OprogramowanieKomputerowe.pl

Colasoft Maximize Network Value · CITRIX · Acronis Compute with confidence · PHD Virtual · Stellar DATA RECOVERY

veeam · FLEXERA SOFTWARE · Navicat · Nagios · mindjet

WINZIP · tripwire · RARLAB WinRAR · vmware · COREL

RIT Labs · Mind Technologies SUPPORTING HUMAN PERFORMANCE · TeamViewer · Quark · solarwinds

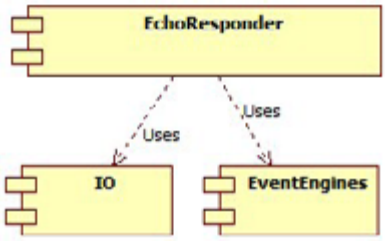
globalscape securely connected · eset · telerik deliver more than expected · ABBYY · INFRAGISTICS DESIGN / DEVELOP / EXPERIENCE

Więcej informacji: (22) 868 40 42 sales@tts.com.pl

Sprzedaż ▸▸ Dystrybucja ▸▸ Import na zamówienie

TTS Company Sp. z o.o. ul.Kryniczna 2 03-934 Warszawa e-mail: tts@tts.com.pl www.tts.com.pl

Na warstwę *middleware* składa się jedynie jeden komponent: `EchoResponder`, czyli implementacja naszego serwera echo na bazie komponentów: `IO` i `EventEngines`.



Rysunek 2. Podział na warstwy serwera echo

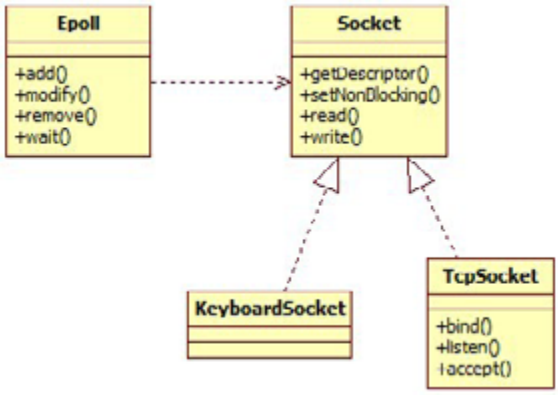
To, co prawdopodobnie chodzi teraz po głowie, to: „jak to teraz zrealizować?!". Spróbujemy nadal poruszać się obraną ścieżką od „szczegółu do ogółu".

PODSTAWOWY WZORZEC WSPOMAGAJĄCY – WRAPPER-FACADE (W-F)

Praktycznie nie ma programisty, który by nie użył (nadużył) tego wzorca – chyba jednym z częściej słyszanych zdań bywa: „(...) ok, to tu zrobimy sobie wrapperek (...)". Oczywiście nie chcę przez to powiedzieć, że robienie wrapperów jest całkowicie złe, ale z doświadczenia wiem, że często w kodzie kończy się to kilkoma lub kilkunastoma poziomami wrapperów, co zamiast upraszczać kod, komplikuje go (odpowiedzialności za zarządzanie stanem rdzennego obiektu rozpływają się na całą hierarchię, duplikuje się obsługa błędów...) i spowalnia jego wykonywanie (znaczny przyrost metod na stosie wywołań).

Co zrobić zatem, by nie przesadzić? Odpowiedź jest mocno związana z poprzednim zdaniem, a właściwie z jego częścią traktującą o stanie. Mianowicie nomenklatura *POSA* mówi, że wzorzec ten nominowany jest do enkapsulacji funkcji i struktur danych (często niskopoziomowych, *error-prone*) w obiekty, w rozumieniu paradygmatu *OOP*, ze spójnym stanem i przejrzystym interfejsem. Typowym przykładem mogą być tu obiekty oferowane przez np. ACE, które przykrywają natywne, specyficzne dla systemu operacyjnego API, takie jak gniazda czy wątki. Wracając już do meritum odpowiedzi, by nie przesadzić, przed postanowieniem dodania kolejnego wrappera należy się zastanowić, czy mamy kolejny stan/schemat zachowań, który wykracza poza to, co oferuje obecny. Jeśli tak (i co gorsza powtarza się to w innych miejscach w kodzie), to rozwiązaniem nie jest dokładanie kolejnego poziomu abstrakcji, lecz refaktoryzacja obecnego wrappera i kodu z niego korzystającego.

W naszym przykładzie jest domena obsługi gniazd. Na potrzeby tego projektu stworzyłem prosty zestaw obiektów:



Rysunek 3. Diagram klas komponentu IO

Dla sprostowania dodam, że w listingach przedstawiam jedynie elementy kluczowe z punktu widzenia koncepcji, a całość kodu jest dostępna na moim publicznym GitHub'ie (link w ramce „W sieci").

Epoll

Klasa `Epoll` to proste *RAII* (idiom wspomniany w poprzednim artykule), które wrappuje nam funkcjonalność demultipleksera zdarzeń opartego o `epoll`.

Listing 1. Deklaracja klasy `Epoll`

```
class Epoll
{
public:
  typedef boost::shared_ptr<Epoll> Ptr;

  struct EventType
  {
    static int Error;
    static int In;
    static int Out;
    static int Hup;
    static int RdHup;
  };
  typedef std::set<int> EventTypes;

  struct Event
  {
    Event(const EventTypes&, Socket::Descriptor);
    EventTypes eventTypes;
    Socket::Descriptor descriptor;
  };
  typedef std::vector<Event> Events;

public:
  explicit Epoll(int p_size = 100);
  virtual ~Epoll();

private:
  Epoll(const Epoll&);
  Epoll& operator=(const Epoll&);

public:
  void add(Socket::Ptr, const EventTypes&) const;
  void modify(Socket::Ptr, const EventTypes&) const;
  void remove(Socket::Descriptor) const;
  void wait(Events&) const;

private:
  const int m_size;
  const int m_epollFd;
};
```

W uprzedniej implementacji `Epoll` był ukryty w obiekcie `Reactor`, gdzie na sztywno był zapisany zestaw zdarzeń, których chcemy się spodziewać z gniazd (jedno z ograniczeń poprzedniej wersji).

W tym wypadku `Epoll` to osobny obiekt dający właściwie funkcjonalność 1:1 w stosunku do oryginalnej (strukturalnej, pochodzącej z języka C), z tą różnicą, że jest ona zamknięta w obiekt posiadający własne, zawsze przewidywalne API, pozwalające zachować spójność stanu obiektu. API `Epoll` używa własnej uobiektowionej maski zdarzeń (`Epoll::EventTypes`) oraz jawnie nawiązuje do powiązanego obiektu tego samego komponentu warstwy domenowej (`IO`): `Socket`.

Klasy gniazd

To zestaw obiektów różnych typów gniazd.

Listing 2. Deklaracja klas gniazd

```
class Socket
{
public:
  typedef boost::shared_ptr<Socket> Ptr;
  typedef int Descriptor;

public:
  virtual ~Socket();
```

```
public:
  Descriptor getDescriptor() const;
  void setNonBlocking() const;
  void read(std::string&) const;
  void write(const std::string&) const;

protected:
  Socket(int, int, int);
  explicit Socket(int);
  Socket(const Socket&);
  Socket& operator=(const Socket&);

protected:
  Descriptor m_fd;
};

class KeyboardSocket: public Socket
{
public:
  typedef boost::shared_ptr<KeyboardSocket> Ptr;

public:
  KeyboardSocket();
  virtual ~KeyboardSocket();
};

class TcpSocket: public Socket
{
public:
  typedef boost::shared_ptr<TcpSocket> Ptr;

public:
  TcpSocket();
  virtual ~TcpSocket();

public:
  void bind(int);
  void listen() const;
  TcpSocket::Ptr accept() const;

private:
  TcpSocket(int, const struct sockaddr&, socklen_t);

private:
  struct sockaddr m_address;
  socklen_t m_addressLength;
};
```

Klasa `Socket` to abstrakcja (obiekt, którego w rzeczywistości utworzyć się nie da z uwagi na brak konstruktora publicznego i brak metod/funkcji fabrykujących) udostępniająca podstawowy zestaw funkcjonalności gniazd: przestawienie w tryb nieblokujący, czytanie i pisanie z/do gniazda.

Klasa `KeyboardSocket` to uszczegółowienie klasy `Socket` – gniazda, które posiada pre-definiowany deskryptor o wartości 0 (konstruktor klasy `KeyboardScoket` buduje bazę – `Socket` – z deskryptorem (`m_fd`) równym 0).

`TcpSocket` to kolejne *RAII*, gniazdo (`Socket`) specyficzne dla protokołu *TCP/IP*, rozszerza `Socket` o: możliwość związania gniazda z portem (`bind`), nasłuchiwanie na nadchodzące połączenia (`listen`) oraz akceptację nowych połączeń (`accept`). Oczywistym brakiem jest tutaj `connect` – czyli brak możliwości nawiązania połączenia, ale w tym przykładzie jest to całkowicie zbędne.

Każdy z w/w obiektów posiada własny typ automatycznego wskaźnika (`Ptr`), w razie gdyby wystąpiła konieczność dzielenia tego obiektu pomiędzy innymi.

Myślę, że tak przygotowany zestaw obiektów może stanowić solidną podstawę dla kolejnych kroków naszego design'u i implementacji.

REACTOR W GENERYCZNEJ POSTACI

Skoro wcześniej rozłożyliśmy aplikację na warstwy i określiliśmy złożoność obiektową wzorca, czas pokazać obiekty komponentu `EventEngines`, w którym znajduje się abstrakcyjna implementacja wzorca.

Listing 3. Klasy komponentu `EventEngines`

```
class EventSource
{
public:
  typedef boost::shared_ptr<EventSource> Ptr;
  typedef int Descriptor;
  typedef int EventType;
  typedef std::set<EventType> EventTypes;

public:
  virtual ~EventSource();

  virtual Descriptor getDescriptor() const = 0;
  void setEventTypes(const EventTypes&);
  const EventTypes& getEventTypes() const;

protected:
  explicit EventSource(const EventTypes&);
  EventSource(const EventSource&);
  EventSource& operator=(const EventSource&);

protected:
  EventTypes m_eventTypes;
}; //class EventSource

class EventDemultiplexer
{
public:
  typedef boost::shared_ptr<EventDemultiplexer> Ptr;
  struct Event
  {
    EventSource::Descriptor descriptor;
    EventSource::EventTypes eventTypes;
  };
  typedef std::vector<Event> Events;

public:
  virtual ~EventDemultiplexer();

  virtual void add(EventSource::Ptr) = 0;
  virtual void modify(EventSource::Ptr) = 0;
  virtual void remove(EventSource::Descriptor) = 0;
  virtual void wait(Events&) = 0;

protected:
  EventDemultiplexer();
  EventDemultiplexer(const EventDemultiplexer&);
  EventDemultiplexer& operator=(const EventDemultiplexer&);
}; //class EventDemultiplexer

class EventHandler
{
public:
  typedef ::boost::shared_ptr<EventHandler> Ptr;

public:
  virtual ~EventHandler();

public:
  virtual void handle(const EventSource::EventTypes&) = 0;
  EventSource::Ptr getEventSource() const;
  EventSource::EventTypes getEventTypes() const;

protected:
  explicit EventHandler(EventSource::Ptr);

private:
  EventHandler(const EventHandler&);
  EventHandler& operator=(const EventHandler&);

protected:
  EventSource::Ptr m_eventSource;
};//class EventHandler

class Reactor
{
public:
  typedef boost::shared_ptr<Reactor> Ptr;

public:
  explicit Reactor(EventDemultiplexer::Ptr);
  ~Reactor();

  void add(EventHandler::Ptr);
  void remove(EventSource::Descriptor);
  void eventLoop();

private:
  Reactor(const Reactor&);
  Reactor& operator=(const Reactor&);

private:
  typedef std::map<EventSource::Descriptor, EventHandler::Ptr>
  t_handlers;
  typedef std::pair<EventHandler::Ptr, EventSource::EventTypes>
  t_toHandle;
  typedef std::vector<t_toHandle> t_toHandles;
```

```
  boost::mutex m_mutex;
  t_handlers m_handlers;
  EventDemultiplexer::Ptr m_eventDemultiplexer;
}; //class Reactor

Reactor::Reactor(EventDemultiplexer::Ptr p_eventDemultiplexer)
  : m_eventDemultiplexer(p_eventDemultiplexer)
{
}

Reactor::~Reactor()
{
}

void Reactor::add(EventHandler::Ptr p_eventHandler)
{
  boost::mutex::scoped_lock lock(m_mutex);
  m_eventDemultiplexer->add(p_eventHandler->getEventSource());
  m_handlers.insert(std::make_pair(p_eventHandler-
  >getEventSource()->getDescriptor(), p_eventHandler));
}

void Reactor::remove(EventSource::Descriptor p_descriptor)
{
  boost::mutex::scoped_lock lock(m_mutex);
  t_handlers::iterator i = m_handlers.find(p_descriptor);
  if (i != m_handlers.end())
  {
    m_handlers.erase(i);
    m_eventDemultiplexer->remove(p_descriptor);
  }
}

void Reactor::eventLoop()
{
  while (1)
  {
    EventDemultiplexer::Events events;
    m_eventDemultiplexer->wait(events);

    t_toHandles toHandles;
    {//scope of m_mutex lock begin
      boost::mutex::scoped_lock lock(m_mutex);
      for (EventDemultiplexer::Events::iterator i = events.begin();
      i < events.end(); ++i)
      {
        t_handlers::const_iterator ih = m_handlers.find(i->descriptor);
        if (ih != m_handlers.end())
        {
          toHandles.push_back(t_toHandle(ih->second, i->eventTypes));
        }
        else
        {
          throw std::runtime_error("EventDemultiplexer returned
          event for \
            unfounded handler");
        }
      }
    }//scope of m_mutex lock end

    for (t_toHandles::iterator i = toHandles.begin(); i !=
    toHandles.end(); ++i)
    {
      i->first->handle(i->second);
    }
  } //while (1)
}
```

Zaczynając (znów) „od szczegółu do ogółu", najbardziej niskopoziomową klasą, a właściwie interfejsem jest `EventSource`. To interfejs dla przyszłych, rozmaitych źródeł zdarzeń. Cechą charakterystyczną `EventSource` (bardzo ważną) jest `Descriptor` – unikalny identyfikator, dzięki któremu możemy znaleźć konkretne źródło zdarzeń (można to porównać ze skrótem takim jak np. *MD5*). Dlaczego `Descriptor` jest typu `int`? Bo to dość uniwersalny typ, a jego zbieżność z typem deskryptora gniazda jest zupełnie przypadkowa ;-). Drugą ważną składową źródła zdarzeń jest, a właściwie są typy zdarzeń, jakie dane źródło będzie zgłaszać: `EventType` (typ pojedynczego zdarzenia), `EventTypes` (zestaw zdarzeń) oraz `m_eventTypes` (typy zdarzeń instancji źródła zdarzeń). Typy zdarzeń również są w postaci `int`, z uwagi na to, że to dość uniwersalny typ, a jego znaczenie będzie nadawać konkretna implementacja.

Ostanim kluczowym elementem `EventDescriptor` jest abstrakcyjna metoda `getDescriptor`. To ważne, bo jedynie konkretne źródło zdarzeń będzie w stanie, w swój unikalny sposób, wytworzyć/podać skrót do swojej instancji. To też potwierdza przypadkowość zbieżności typów deskryptorów w `Socket` i `EventSource`.

`EventDemultiplexer` to interfejs definiujący publiczne API każdego przyszłego, konkretnego demultipleksera zdarzeń. Do każdego ED można dodawać źródła zdarzeń, modyfikować je lub usuwać. Zwróćmy uwagę, że pojawia się tu pierwsza inwersja kontroli: `EventDemultiplexer` przyjmuje odniesienia do pełnych obiektów `EventSource` i sam winien pobrać typy zdarzeń, jakie dane źródło będzie zgłaszać. Oczywiście można tu dodatkowo użyć wzorca *Template Method*, by inwersji kontroli stało się zadość już na poziomie tej klasy, lecz nie jest to konieczne.

`EventHandler` to niemalże kopia interfejsu z poprzedniego artykułu dla klas odpowiadających za obsługę zdarzeń. W odróżnieniu od poprzedniej implementacji jest jednak ściśle związany i jest współwłaścicielem `EventSource` (patrz: konstruktor `EventHandler` oraz składowa `m_eventSource`).

Aktualny `Reactor` to podrasowana wersja poprzedniego, wzbogacona o zabezpieczenia wielowątkowe oraz udoskonalonymi możliwościami usuwania `EventHandler`'ów. Implementuje wzorzec *Monitor*, który zakłada, że wywołania metod publicznych są okraszone wspólnym muteksem, lecz w odróżnieniu od wzrorca *Thread-safe Interface* nakazuje wyjście z sekcji krytycznej, gdy sterowanie wychodzi poza obiekt monitora (czyli gdy *Monitor* woła metody innych obiektów). Jest to ważne z dwóch powodów:

1. do/z reaktora możemy dokładać, usuwać zupełnie asynchronicznie `EventHandler`'y;
2. `EventHandler` może stwierdzić, że dane źródło „wyczerpało się", i usunąć swoje odniesienie z reaktora.

Kwintesencją tego wywodu jest struktura metody `Reactor::eventLoop`:

1. wywołanie (raczej) blokującej metody oczekiwania na zdarzenia z obiektu `EventDemultiplexer`
2. po wyjściu z `m_eventDemultiplexer->wait`, reaktor wchodzi w sekcję krytyczną i przygotowuje listę zdarzeń i obiektów ich obsługi
3. wychodzi z sekcji krytycznej i uruchamia obsługę zdarzeń.

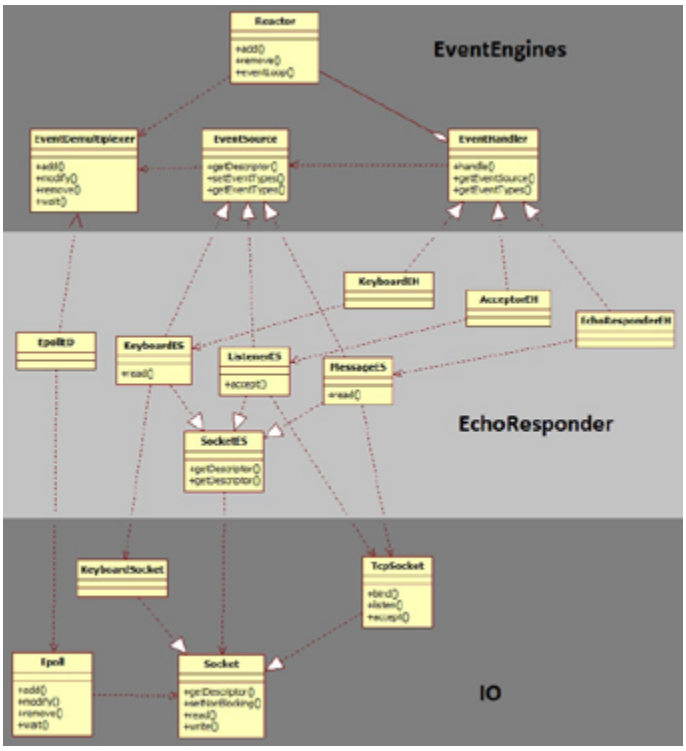
Tak przygotowana pętla obsługi zdarzeń zapewnia możliwość dodawania/usuwania `EventHandler`'ów zarówno podczas oczekiwania na zdarzenia, jak również podczas obsługi zdarzeń. Zwróćmy uwagę, że taką możliwość musi również udostępniać `EventDemultiplexer`.

APLIKACJA W KOŃCOWEJ POSTACI

Czas poskładać wcześniej przygotowane klocki. Aby dobrze powiązać obiekty, trafnym sposobem jest opisać ich naturę. Jak? Zwyczajnie, w sposób leksykalny:

- źródłami zdarzeń są gniazda, gniazda nie wiedzą nic o źródłach zdarzeń, więc za późno na dziedziczenie, możemy za to stworzyć obiekt `SocketES`, który będzie implementował interfejs źródła zdarzeń i posiadał gniazdo;
- źródła zdarzeń są trzy: dwa gniazda *TCP* (gniazdo nasłuchujące połączeń i gniazdo komunikacji z klientem) oraz gniazdo klawiatury, ich bazą jest `Socket`, możemy więc utworzyć specyficzne źródła zdarzeń dziedziczące po `SocketES`: `KeyboardES`, `ListenerES` i `MessageES`;
- demultiplekserem zdarzeń gniazd jest `Epoll`, `Epoll` nie wie nic o `EventSource`, znów za późno na dziedziczenie, więc podobnie jak w przypadku gniazd zróbmy obiekt `EpollED`, dziedziczący interfejs `EventDemultiplexer` i posiadający `Epoll`
- każdy `EventHandler` jest związany z konkretnym `EventSource`, koniecznym więc jest posiadanie osobnych EH dla każdego z typów źródeł, czyli: `KeyboardEH`, `AcceptorEH`, `EchoResponderEH`.

Tak opisaną rzeczywistość śmiało możemy przedstawić na diagramie klas:



Rysunek 4. Diagram klas całego projektu

Aplikacja docelowa

Teraz już kod aplikacji docelowej:

Listing 4. Implementacja klas aplikacyjnych (obsługujących logikę protokołu warstwy aplikacji i logiki aplikacji)

```
class SocketES: public EventSource
{
public:
  typedef boost::shared_ptr<SocketES> Ptr;

public:
  virtual ~SocketES();

  Descriptor getDescriptor() const;
  Socket::Ptr getSocket() const;

protected:
  explicit SocketES(Socket::Ptr);

protected:
  Socket::Ptr m_socket;
};

SocketES::SocketES(Socket::Ptr p_socket)
 : EventSource(EventTypes()), m_socket(p_socket)
{
}

SocketES::~SocketES()
{
}

EventSource::Descriptor SocketES::getDescriptor() const
{
  return m_socket->getDescriptor();
}

Socket::Ptr SocketES::getSocket() const
{
  return m_socket;
}

class KeyboardES: public SocketES
{
public:
  typedef boost::shared_ptr<KeyboardES> Ptr;

public:
  explicit KeyboardES(KeyboardSocket::Ptr);
  virtual ~KeyboardES();

  void read(std::string&);
};

KeyboardES::KeyboardES(KeyboardSocket::Ptr p_keybSocket)
  : SocketES(p_keybSocket)
{
  m_eventTypes.insert(Epoll::EventType::Error);
  m_eventTypes.insert(Epoll::EventType::In);

  m_socket->setNonBlocking();
}

KeyboardES::~KeyboardES()
{
}

void KeyboardES::read(std::string& p_data)
{
  m_socket->read(p_data);
}

class ListenerES: public SocketES
{
public:
  typedef boost::shared_ptr<ListenerES> Ptr;

public:
  ListenerES(TcpSocket::Ptr, int);
  virtual ~ListenerES();

  MessageES::Ptr accept() const;
};

ListenerES::ListenerES(TcpSocket::Ptr p_tcpSocket, int p_port)
  : SocketES(p_tcpSocket)
{
  m_eventTypes.insert(Epoll::EventType::Error);
  m_eventTypes.insert(Epoll::EventType::In);
  m_eventTypes.insert(Epoll::EventType::Hup);
  m_eventTypes.insert(Epoll::EventType::RdHup);

  p_tcpSocket->setNonBlocking();
  p_tcpSocket->bind(p_port);
  p_tcpSocket->listen();
}

ListenerES::~ListenerES()
{
}

MessageES::Ptr ListenerES::accept() const
{
  TcpSocket::Ptr tcpSocket =
  boost::dynamic_pointer_cast<TcpSocket>(m_socket);
  TcpSocket::Ptr accepted = tcpSocket->accept();
  return MessageES::Ptr(new MessageES(accepted));
}

class MessageES: public SocketES
{
public:
  typedef boost::shared_ptr<MessageES> Ptr;

public:
  explicit MessageES(TcpSocket::Ptr);
  virtual ~MessageES();

  void read(std::string&) const;
};

MessageES::MessageES(TcpSocket::Ptr p_tcpSocket)
  : SocketES(p_tcpSocket)
{
  m_eventTypes.insert(Epoll::EventType::Error);
  m_eventTypes.insert(Epoll::EventType::In);
  m_eventTypes.insert(Epoll::EventType::Hup);
  m_eventTypes.insert(Epoll::EventType::RdHup);

  m_socket->setNonBlocking();
}

MessageES::~MessageES()
{
}

void MessageES::read(std::string& p_message) const
{
  m_socket->read(p_message);
}

class KeyboardEH : public EventHandler
{
public:
  KeyboardEH(const std::string&, KeyboardES::Ptr);
```

```
  virtual ~KeyboardEH();

public:
  virtual void handle(const EventSource::EventTypes&);

private:
  std::ofstream m_file;
};

KeyboardEH::KeyboardEH(const std::string& p_fileName,
KeyboardES::Ptr p_kES)
  : EventHandler(p_kES), m_file(p_fileName.c_str(),
  std::ofstream::out)
{
  if (!m_file.is_open())
  {
    throw std::runtime_error("Cannot open file");
  }
}

KeyboardEH::~KeyboardEH()
{
  m_file.close();
}

void KeyboardEH::handle(const EventSource::EventTypes&
p_eventTypes)
{
  EventSource::EventTypes::const_iterator iIn = p_eventTypes.
  find(Epoll::EventType::In);
  if ( (iIn != p_eventTypes.end()) && (p_eventTypes.size() == 1) )
  {
    SocketES::Ptr socketES =
    boost::dynamic_pointer_cast<SocketES>(m_eventSource);
    KeyboardES::Ptr keyboardES =
    boost::dynamic_pointer_cast<KeyboardES>(socketES);
    std::string data;
    keyboardES->read(data);
    m_file << data;

    if (boost::istarts_with(data, "exit"))
    {
      throw std::runtime_error("exit");
    }
  }
  else
  {
    throw std::runtime_error("Bad event for for acceptor");
  }
}

class AcceptorEH : public EventHandler
{
public:
  AcceptorEH(ListenerES::Ptr, Reactor&);
  virtual ~AcceptorEH();

public:
  virtual void handle(const EventSource::EventTypes&);

private:
  Reactor& m_reactor;
};

AcceptorEH::AcceptorEH(ListenerES::Ptr p_listenerES, Reactor&
p_reactor)
  : EventHandler(p_listenerES)
  , m_reactor(p_reactor)
{
}

AcceptorEH::~AcceptorEH()
{
}

void AcceptorEH::handle(const EventSource::EventTypes&
p_eventTypes)
{
  EventSource::EventTypes::const_iterator iIn = p_eventTypes.
  find(Epoll::EventType::In);
  if ( (iIn != p_eventTypes.end()) && (p_eventTypes.size() == 1) )
  {
    ListenerES::Ptr listenerES =
    boost::dynamic_pointer_cast<ListenerES>(m_eventSource);
    MessageES::Ptr msgES = listenerES->accept();
    EchoResponderEH::Ptr erEH(new EchoResponderEH(msgES,
    m_reactor));
    m_reactor.add(erEH);
  }
  else
  {
    throw std::runtime_error("Bad event for for acceptor");
  }
}
```

```
class EchoResponderEH : public EventHandler
{
public:
  EchoResponderEH(MessageES::Ptr, Reactor&);
  virtual ~EchoResponderEH();

public:
  virtual void handle(const EventSource::EventTypes&);

private:
  Reactor& m_reactor;
};

EchoResponderEH::EchoResponderEH(MessageES::Ptr p_messageES,
Reactor& p_reactor)
  : EventHandler(p_messageES), m_reactor(p_reactor)
{
}

EchoResponderEH::~EchoResponderEH()
{
}

void EchoResponderEH::handle(const EventSource::EventTypes&
p_eventTypes)
{
  EventSource::EventTypes::const_iterator iIn = p_eventTypes.
  find(Epoll::EventType::In);
  if ( (iIn != p_eventTypes.end()) && (p_eventTypes.size() == 1) )
  {
    MessageES::Ptr mES =
    boost::dynamic_pointer_cast<MessageES>(m_eventSource);
    std::string data;
    while (!boost::ends_with(data, "\n"))
    {
      std::string part;
      mES->read(part);
      data.append(part);
    }
    mES->getSocket()->write(data);
  }
  else
  {
    m_reactor.remove(m_eventSource->getDescriptor());
  }
}
```

W stosunku do poprzedniej implementacji, w obecnej doszedł nam zestaw klas implementujących konkretne źródła zdarzeń:

- `SocketES` – podstawa źródeł zdarzeń, których pochodzeniem jest typ ogólny `Socket`
- `KeyboardES`, czyli źródło zdarzeń pochodzących ze standardowego wejścia
- `ListenerES` – źródło zdarzeń nadchodzących połączeń
- `MessageES` – zdarzenia informujące o nadchodzących wiadomościach.

Obiekty obsługi zdarzeń (*EH) to właściwie kopie tych z pierwszej części artykułu, z dwiema różnicami:

1. nie używają już natywnych gniazd, lecz współpracują z obiektami źródeł zdarzeń (zakres ich odpowiedzialności się nie zmienił – np. `EchoResponderEH` nadal w sposób kompletny implementuje protokół warstwy aplikacji);
2. `EchoResponderEH` wyrejestrowuje się z reaktora, gdy jego `MessageES` zgłasza błąd.

Wszystkie w/w klasy wykorzystują trzy pryncypały:

1. wstrzykiwanie zależności (np. gniazdo podawane w konstruktorze), co znacznie ułatwia testowanie jednostkowe i modułowe;
2. inwersje kontroli (m.in. zależność jest ustawiana zgodnie z potrzebami przez obiekt zależny, np. ustawianie gniazda w tryb nieblokujący);
3. specyfikacja – samookreślenie – ustawienie stanu początkowego obiektu konkretnego (specyfikacja typów zdarzeń `m_eventTypes`).

Main

Teraz czas na zwieńczenie implementacji, czyli na funkcję `main`:

Listing 5. Implementacja funkcji `main`

```
int main(int, char**)
{
  try
  {
    Epoll::Ptr epoll(new Epoll());
    EpollED::Ptr epollED(new EpollED(epoll));
    Reactor reactor(epollED);

    KeyboardSocket::Ptr keybSocket(new KeyboardSocket());
    KeyboardES::Ptr keybES(new KeyboardES(keybSocket));
    EventHandler::Ptr keybEH(new KeyboardEH("log.txt", keybES));

    TcpSocket::Ptr listenerSokcet(new TcpSocket());
    ListenerES::Ptr listenerES(new ListenerES(listenerSokcet,
    5050));
    EventHandler::Ptr acceptorEH(new AcceptorEH(listenerES,
    reactor));

    reactor.add(keybEH);
    reactor.add(acceptorEH);

    reactor.eventLoop();
  }
  catch (const std::runtime_error& rte)
  {
    std::cout << "Runtime exception: " << rte.what() << std::endl;
  }
  catch (const std::exception& e)
  {
    std::cout << "STD exception: " << e.what() << std::endl;
  }

  return 0;
}
```

W stosunku do poprzedniczki z pierwszego artykułu, obecna funkcja `main` jest jedynie wzbogacona o fabrykację obiektów zależnych (zaznaczone na czerwono).

Jedynym szczegółem, który wymaga wyjaśnienia, jest sposób tworzenia instancji reaktora i sposób przekazywania jego odniesienia do innych obiektów (`AcceptorEH` i `EchoResponderEH`). `Reactor` jest zwykłym obiektem stworzonym na stosie, a odniesienie do niego jest propagowane jako klasyczna referencja. Odpowiedź na pytanie, skąd ta różnica, wynika z problematyki używania `boost::shared_ptr`, a konkretnie z problemu cyklicznych referencji, o którym wspominałem w poprzednim artykule. Mówiąc wprost, jeśli dwa obiekty posiadają do siebie nawzajem odniesienia bazujące na mechanizmie zliczania referencji (a tak właśnie działa `boos::shared_ptr`), to ich licznik referencji nie może spaść do zera bez „manualnych" interwencji.

WNIOSKI

Mocne strony

- Separacja – odseparowanie logiki domenowej od aplikacyjnej odsłoniło spore pole manewru na czas rozwoju i utrzymania projektu. Uniwersalność komponentów warstwy domenowej pozwala na re-używanie ich przy realizacji innych potencjalnych celów.
- Wstrzykiwanie zależności otworzyło kod na testowanie jednostkowe i modułowe w różnych konfiguracjach (np. możemy testować aplikację z oryginalnym komponentem `EventEngines`, lecz z *mock'owym* komponentem `IO`).
- Proste zależności między obiektami.
- Proste obiekty, ze zredukowaną do jednego elementu listą odpowiedzialności.
- Moim (subiektywnym) zdaniem kod nie stracił na przejrzystości i czytelności.

Słabe strony

- Stan obsługi błędów nadal pozostawia sporo do życzenia, choć nadal w 100% się sprawdza.
- Wydajność kodu niestety nieco się obniżyła z uwagi na przyrost obiektów pośredniczących do wywołań natywnych oraz dziedziczenie.
- Większa granulacja obiektów zwiększa nieco czas, który trzeba poświęcić na zrozumienie kodu.

ZAKOŃCZENIE

Tradycyjnie, zainteresowanych zachęcam do pobrania kodu (link w ramce „W sieci") i jego przetestowania. W repozytorium znajduje się alternatywna implementacja oparta o szablony, która w porównaniu do prezentowanej tutaj cechuje się nieco wyższą wydajnością, okupioną trochę dłuższym czasem kompilacji oraz słabszą czytelnością z uwagi na skomplikowaną składnię szablonów w C++.

W następnym artykule do naszego projektu dołożę kolejny z ważnych wzorców silników zdarzeń i postaram się go równie dokładnie opisać.

W sieci

- Kod źródłowy aplikacji z I części artykułu: https://github.com/RomanUlan/ReactorBase
- Kod źródłowy prezentowanej aplikacji: https://github.com/RomanUlan/ReactorInheritance
- Kod źródłowy implementacji alternatywnej: https://github.com/RomanUlan/ReactorTemplates

Roman Ulan roman.ulan@gmail.com

Senior Software Architect w Tieto Poland Sp. z o.o. Trener C++ w Bottega IT Solutions. Specjalizuje się w rozwiązaniach backend'owych w języku C++. Entuzjasta czystego i testowalnego kodu osadzonego w przemyślanej architekturze opartej na sprawdzonych wzorcach. Interesuje go programowanie równoległe i rozproszone oraz zagadnienia „software design".