Roman Ulan

Wzorce silników zdarzeń w C++

Część IV: Wzorzec Half Sync/Half Async – obsługa częściowo synchroniczna (reaktywna) i częściowo asynchroniczna.

W poprzednim artykule przedstawiłem wzorzec Thread Pool („Programista" 10/2014), który jest skalowalną, wielowątkową alternatywą dla wzorca Reactor. To zupełnie asynchroniczne podejście w przypadku TP wymagało oczywiście drobnych zmian w kodzie aplikacyjnym. W niniejszym przedstawię wzorzec będący mieszanką obu podejść omawianych uprzednio.

WZORZEC HALF SYNC/HALF ASYNC (HSHA)

Zacznijmy od dwóch diagramów (patrz: Rysunki 1 i 2), które pozwolą nam zrozumieć podstawę nowego rozwiązania.

Uważni czytelnicy serii na pewno już wyobrażają sobie implementację, więc z marszu cytuję kod nowego silnika zdarzeń i obiektów z nim powiązanych:

Listing 1. Implementacja klas wzorca

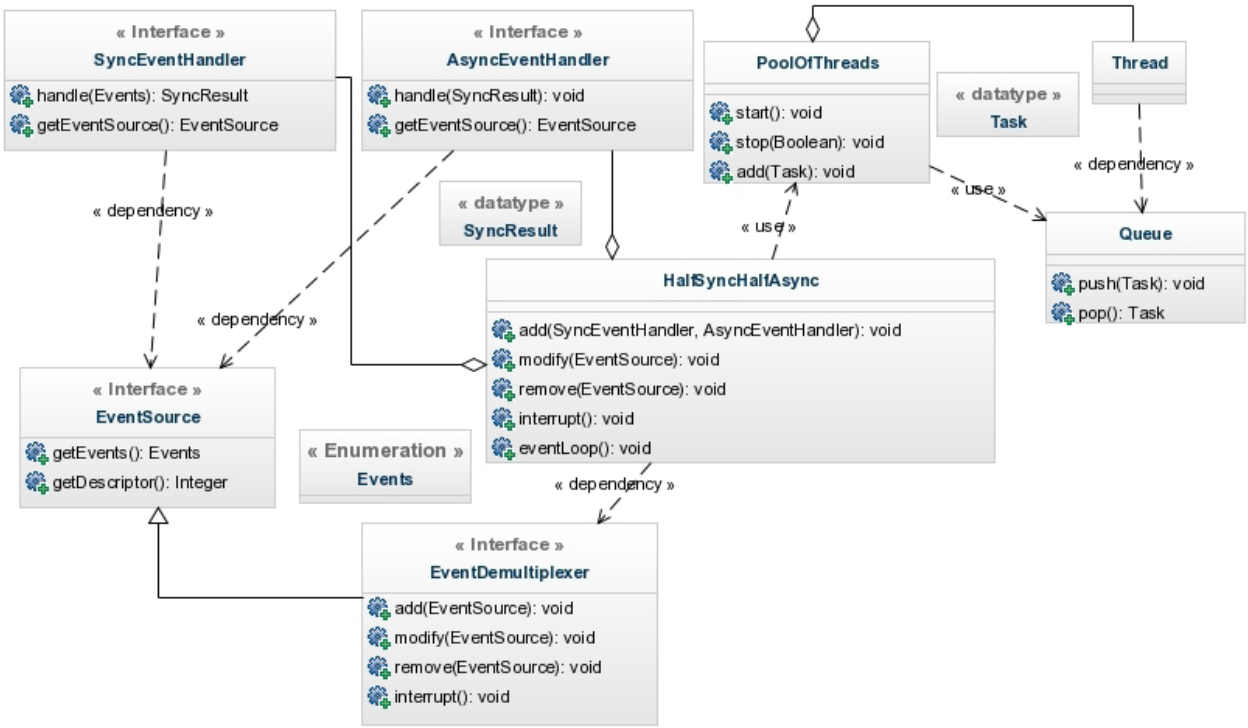
```
template <class SyncResult>
class SyncEventHandler
{
public:
  typedef typename std::shared_ptr<SyncEventHandler<SyncResult> > Ptr;

public:
  virtual ~SyncEventHandler();

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

protected:
  explicit SyncEventHandler(EventSource::Ptr);
  SyncEventHandler(const SyncEventHandler&);
  SyncEventHandler& operator=(const SyncEventHandler&);

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

template <class SyncResult>
SyncEventHandler<SyncResult>::SyncEventHandler(EventSource::Ptr p_es)
  : m_eventSource(p_es)
{
}

template <class SyncResult>
SyncEventHandler<SyncResult>::~SyncEventHandler()
{
}

template <class SyncResult>
EventSource::Ptr SyncEventHandler<SyncResult>::getEventSource()
const
{
  return m_eventSource;
}

template <class SyncResult>
EventSource::EventTypes SyncEventHandler<SyncResult>::getEventTy
pes() const
{
  return m_eventSource->getEventTypes();
}
```

O serii „Wzorce silników zdarzeń"

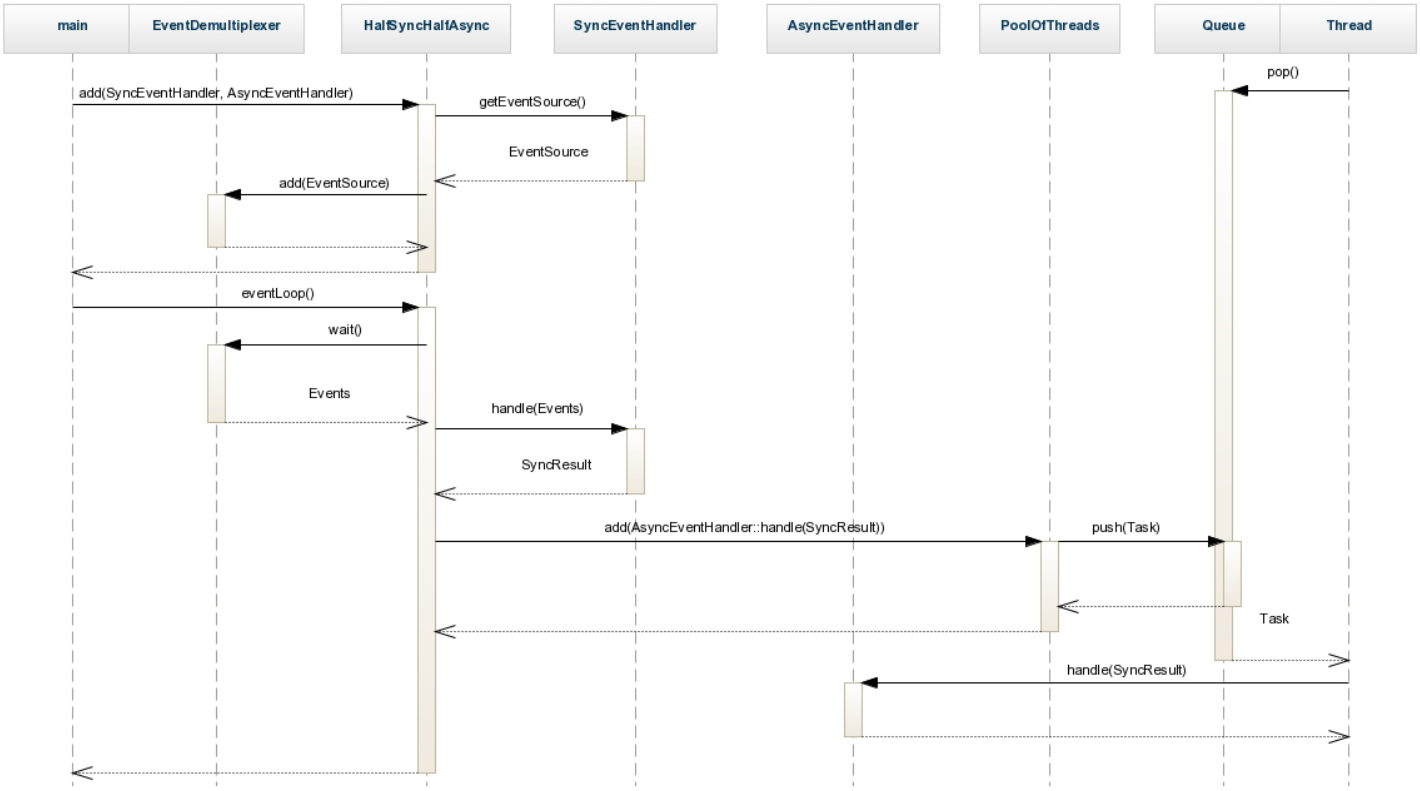
Intencją serii „Wzorce silników zdarzeń" jest dostarczenie usystematyzowanej wiedzy bazowej początkującym programistom, projektantom i architektom. Przez projektanta lub architekta rozumiem tutaj rolę pełnioną w projekcie.

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.

```
}

template <class SyncResult>
class AsyncEventHandler
{
public:
  typedef typename std::shared_ptr<AsyncEventHandler<SyncResult> > Ptr;

public:
  virtual ~AsyncEventHandler();

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

protected:
  explicit AsyncEventHandler(EventSource::Ptr);
  AsyncEventHandler(const AsyncEventHandler&);
  AsyncEventHandler& operator=(const AsyncEventHandler&);

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

template <class SyncResult>
AsyncEventHandler<SyncResult>::AsyncEventHandler(EventSource::P
tr p_es)
  : m_eventSource(p_es)
{
}

template <class SyncResult>
AsyncEventHandler<SyncResult>::~AsyncEventHandler()
{
}

template <class SyncResult>
EventSource::Ptr AsyncEventHandler<SyncResult>::getEventSource()
const
{
  return m_eventSource;
}
```



Rysunek 1. Diagram klas wzorca Half Sync/Half Async



Rysunek 2. Diagram sekwencji wzorca Half Sync/Half Async

```
template <class SyncResult>
EventSource::EventTypes AsyncEventHandler<SyncResult>::getEventTypes() const
{
  return m_eventSource->getEventTypes();
}

template <class SyncResult>
class HalfSyncHalfAsync
{
public:
  typedef typename std::shared_ptr<HalfSyncHalfAsync<SyncResult
  > > Ptr;
  typedef typename std::pair<typename SyncEventHandler<SyncResult>
  ::Ptr, typename AsyncEventHandler<SyncResult>::Ptr> Handlers;

public:
  HalfSyncHalfAsync(EventDemultiplexer::Ptr, size_t);
  virtual ~HalfSyncHalfAsync();

  void add(const Handlers&);
  void modify(EventSource::Ptr);
  void remove(EventSource::Descriptor);
  void eventLoop();
  void interrupt(bool);

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

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

  std::mutex m_mutex;
  bool m_run;
  t_handlers m_handlers;
  EventDemultiplexer::Ptr m_eventDemultiplexer;
  Threading::ThreadPool m_threadPool;
}; //class HalfSyncHalfAsync

template <class SyncResult>
HalfSyncHalfAsync<SyncResult>::HalfSyncHalfAsync(EventDemultiplexer::Ptr p_ed, size_t p_tpSize)
  : m_run(false), m_eventDemultiplexer(p_ed), m_threadPool(p_tpSize)
{
  m_threadPool.start();
}

template <class SyncResult>
HalfSyncHalfAsync<SyncResult>::~HalfSyncHalfAsync()
{
  m_threadPool.stop(true);
}

template <class SyncResult>
void HalfSyncHalfAsync<SyncResult>::add(const Handlers&
p_handlers)
{
  EventSource::Ptr es = p_handlers.first->getEventSource();
  EventSource::Descriptor syncD = es->getDescriptor();
  if (p_handlers.second && (syncD != p_handlers.
  second->getEventSource()->getDescriptor()))
  {
    throw std::runtime_error("Event handlers descriptors don't match");

  }

  { //m_mutex lock scope begin
    std::unique_lock<std::mutex> lock(m_mutex);
    m_eventDemultiplexer->add(es);
    m_handlers.insert(std::make_pair(syncD, p_handlers));
  } //m_mutex lock scope end
}

template <class SyncResult>
void HalfSyncHalfAsync<SyncResult>::modify(EventSource::Ptr
p_eventSource)
{
  std::unique_lock<std::mutex> lock(m_mutex);
  m_eventDemultiplexer->modify(p_eventSource);
}

template <class SyncResult>
void HalfSyncHalfAsync<SyncResult>::remove(EventSource::Descriptor p_descriptor)
{
  { //m_mutex lock scope begin
    std::unique_lock<std::mutex> lock(m_mutex);
    auto i = m_handlers.find(p_descriptor);
    if (i != m_handlers.end())
    {
      m_eventDemultiplexer->remove(p_descriptor);
      m_handlers.erase(i);
    }
  } //m_mutex lock scope end
}

template <class SyncResult>
void HalfSyncHalfAsync<SyncResult>::eventLoop()
{
  m_run = true;
  while (m_run)
  {
    EventDemultiplexer::Events events;
    m_eventDemultiplexer->wait(events);

    t_toHandles toHandles;
    { //m_mutex lock scope begin
      std::unique_lock<std::mutex> lock(m_mutex);
      for (auto i = events.begin(); i != events.end(); ++i)
      {
        auto 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");
        }
      }
    } //m_mutex lock scope end

    for (auto i = toHandles.begin(); i != toHandles.end(); ++i)
    {
      auto aeh = i->first.second;
      SyncResult res = i->first.first->handle(i->second);
      if (aeh)
      {
        Threading::ThreadPool::Task t = [aeh, res](void)->void {
        aeh->handle(res); };
        m_threadPool.add(t);
      }
    }
  } //while (1)
}

template <class SyncResult>
void HalfSyncHalfAsync<SyncResult>::interrupt(bool p_immediately)
{
  m_run = false;
  m_eventDemultiplexer->interrupt();
}
```

Pierwszą rzeczą, która rzuca się w oczy, jest oczywiście użycie szablonów. Rodzi się więc pytanie: dlaczego silimy się tu na statyczny polimorfizm, jeśli do tej pory radziliśmy sobie przy użyciu dziedziczenia? Odpowiedź jest dość trywialna: z lenistwa i chęci uniknięcia zbędnych problemów. Konkretnie: szablony w najprostszy sposób pozwalają nam na implementowanie szkieletów zachowań na rzecz obiektów posiadających operatory, funkcje i typy wykorzystywane w danym szkielecie funkcjonalnym. Innymi słowy mówiąc, schemat piszemy funkcyjnie, niezależnie od typu danych, a ten specyfikujemy dopiero podczas użycia.

Ponieważ w naszej implementacji oddzielamy logikę wzorca od logiki domenowej, kod wzorca musi być jak najbardziej generyczny, a to znaczy, że nie powinniśmy narzucać typu, jaki będą wymieniać między sobą `SyncEventHandler` i `AsyncEventHandler`. Oczywiście można by to uzyskać poprzez wprowadzenia pustej klasy interfejsowej, lecz to dołożyłoby niepotrzebnego rzutowania i konieczność użycia wskaźników, lub użyć typu void*, który niesie ze sobą te same problemy i dodatkowo wprowadza problem odpowiedzialności za zwalnianie pamięci pod takim wskazaniem. Dlatego moim zdaniem dla tak prostych klas interfejsowych ich specyfikacja w sposób szablonowy

daje sporo udogodnień pomimo nie najpiękniejszej składni i konieczności przeniesienia tego szablonu na klasę samego silnika (`HalfSyncHalfAsync`).

MECHANIKA WZORCA

Wzorzec generalnie, jak już sama nazwa wskazuje, dzieli się na dwie części: synchroniczną (reaktywną) i asynchroniczną (wielowątkową).

Część reaktywna to częściowa obsługa zdarzenia przez `SyncEventHandler` w głównym wątku silnika, który odbiera zdarzenia z demultipleksera.

Rezultat tego wstępnego przetwarzania oraz odpowiedni `AsyncEventHandler` jest delegowany jako zadanie do puli wątków w postaci funkcji (w tej implementacji zapisanej jako wyrażenie lambda).

Część wielowątkowa to znana już z poprzedniej części cyklu pula wątków sensu stricto, której kodu już tu nie będę omawiał, a zainteresowanych odsyłam do kodu oraz lektury poprzednich części.

ZMIANY W IO

IO to dokładnie ten sam komponent z poprzedniej części, rozszerzony wówczas o asynchroniczne przerwanie blokującej metody oczekiwania na zdarzenia oraz obsługę zdarzeń w trybie „one shot" i „edge triggered" (dla odświeżenia pamięci ponownie odsyłam do cz. III serii).

ZMIANY W KOMPONENCIE APLIKACJI DOCELOWEJ

Teraz już główne zmiany w kodzie aplikacji docelowej:

Listing 2. Implementacja klas aplikacyjnych (obsługujących logikę protokołu warstwy aplikacji i logiki aplikacji)

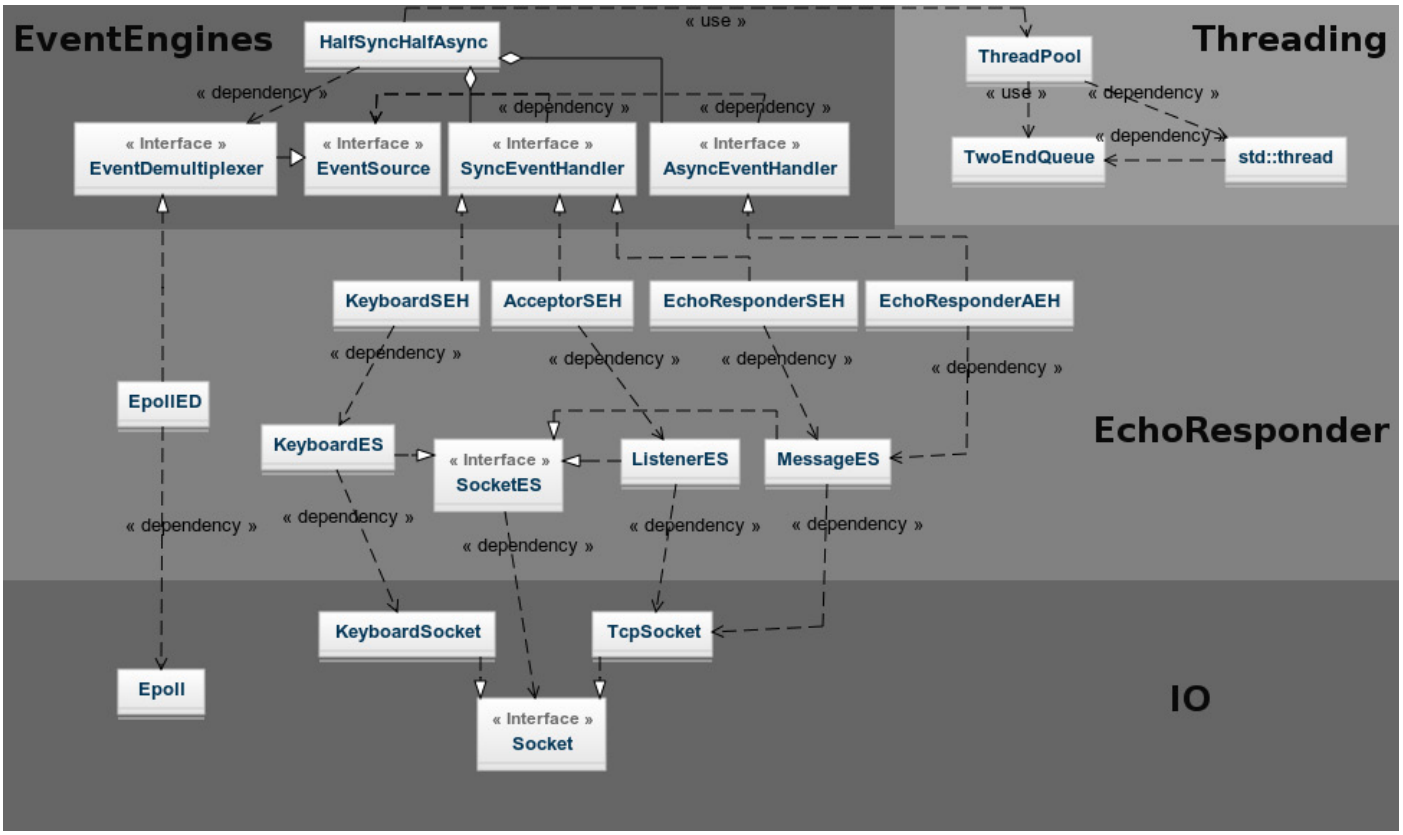
```
typedef HalfSyncHalfAsync<std::string> HSHA;
typedef SyncEventHandler<std::string> SEH;
typedef AsyncEventHandler<std::string> AEH;

class AcceptorSEH: public SEH
{
public:
  AcceptorSEH(ListenerES::Ptr, HSHA&);
  virtual ~AcceptorSEH();

public:
  virtual std::string handle(const EventSource::EventTypes&);

private:
  HSHA& m_hsha;
};

AcceptorSEH::AcceptorSEH(ListenerES::Ptr p_listenerES, HSHA&
p_hsha)
  : SyncEventHandler(p_listenerES)
  , m_hsha(p_hsha)
{
}

AcceptorSEH::~AcceptorSEH()
{
}

std::string AcceptorSEH::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 =
    std::dynamic_pointer_cast<ListenerES>(m_eventSource);
    MessageES::Ptr msgES = listenerES->accept();
    EchoResponderSEH::Ptr erSEH(new EchoResponderSEH(msgES,
    m_hsha));
    EchoResponderAEH::Ptr erAEH(new EchoResponderAEH(msgES,
    m_hsha));
    m_hsha.add(HSHA::Handlers(erSEH, erAEH));
  }
  else
  {
    throw std::runtime_error("Bad event for for acceptor");
  }

  return std::string();
}

class KeyboardSEH : public SEH
{
public:
  KeyboardSEH(const std::string&, KeyboardES::Ptr);
  virtual ~KeyboardSEH();

public:
  virtual std::string handle(const EventSource::EventTypes&);

private:
  std::ofstream m_file;
};

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

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

std::string KeyboardSEH::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) )
  {
    KeyboardES::Ptr keyboardES =
    std::dynamic_pointer_cast<KeyboardES>(m_eventSource);
    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");
  }

  return std::string();
}

class EchoResponderSEH: public SEH
{
public:
  EchoResponderSEH(MessageES::Ptr, HSHA&);
  virtual ~EchoResponderSEH();

public:
  virtual std::string handle(const EventSource::EventTypes&);

private:
  HSHA& m_hsha;
};

EchoResponderSEH::EchoResponderSEH(MessageES::Ptr p_messageES,
HSHA& p_hsha)
  : SyncEventHandler(p_messageES), m_hsha(p_hsha)
{
}

EchoResponderSEH::~EchoResponderSEH()
{
}

std::string EchoResponderSEH::handle(const
EventSource::EventTypes& p_eventTypes)
```

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

  return data;
}

class EchoResponderAEH: public AEH
{
public:
  EchoResponderAEH(MessageES::Ptr, HSHA&);
  virtual ~EchoResponderAEH();

public:
  virtual void handle(const std::string&);

private:
  HSHA& m_hsha;
};

EchoResponderAEH::EchoResponderAEH(MessageES::Ptr p_socket, HSHA&
p_hsha)
  : AsyncEventHandler(p_socket), m_hsha(p_hsha)
{
}

EchoResponderAEH::~EchoResponderAEH()
{
}

void EchoResponderAEH::handle(const std::string& p_message)
{
  MessageES::Ptr mES =
  std::dynamic_pointer_cast<MessageES>(m_eventSource);

  try
  {
    mES->getSocket()->write(p_message);
  }
  catch (const std::exception& e)
  {
    m_hsha.remove(mES->getDescriptor());
  }
}
```

Całość zależności pomiędzy zacytowanymi klasami prezentuje diagram widoczny na Rysunku 3:

Na początek klasa `AcceptorSEH`. Implementuje ona synchroniczną część akceptacji nowego klienta. Jest to właściwie ta sama klasa co `AcceptorEH` z poprzednich części naszej serii, rozszerzona o zwracany pusty łańcuch znaków z metody `handle`. Ponieważ całość akceptacji nowego połączenia odbywa się w tym miejscu, klasa `AcceptorAEH` (czyli asynchroniczna część w/w akceptacji) jest zbędna. W przyszłości można by dołożyć ją na okoliczność np. implementacji nawiązania połączenia SSL.

`KeyboardSEH` to synchroniczna obsługa gniazda klawiatury. Podobnie jak w przypadku gniazda serwerowego (akceptacji połączeń) jest to właściwie kopia `KeyboardEH` z poprzednich części cyklu, gdzie metoda `handle` zawiera komplet – odczytanie bufora klawiatury i zalogowanie go do pliku, a zwraca pusty `std::string`. Całość sprawia, że i w tym wypadku klasa `KeyboardAEH` nie jest potrzebna. Oczywiście faktem jest, że np. sam zapis do pliku mógłby się odbyć już asynchronicznie, ale uwaga:



Rysunek 3. Diagram zależności pomiędzy wszystkimi klasami projektu

- będzie to wymagało wprowadzenia sekcji krytycznej na dostępie do pliku (co zredukowałoby korzyści z wielowątkowości dla tego gniazda właściwie do zera),
- stracimy wówczas pewność co do kolejności wpisów w logu.

Z uwagi na te dwa znaczące problemy zdecydowałem arbitralnie, że obsługę gniazda klawiatury lepiej jest zostawić reaktywną.

`EchoResponderSEH` – synchroniczna część obsługi żądania echo. Metoda `handle` odczytuje wiadomość z gniazda klienckiego i zwraca ją jako rezultat synchronicznej obsługi zdarzenia, co automatycznie zostanie przekazane przez nasz silnik HSHA do `EchoResponderAEH`, a ściślej mówiąc, do jego metody `handle`, jako argument, która odeśle wiadomość zwrotną echo.

Main

Tradycyjnie w na końcu prezentuję zwieńczenie naszych starań, czyli treść funkcji `main`:

Listing 3. Implementacja funkcji main

```
int main(int, char**)
{
  try
  {
    Epoll::Ptr epoll(new Epoll());
    EpollED::Ptr epollED(new EpollED(epoll));
    HSHA hsha(epollED, 1);

    KeyboardSocket::Ptr keybSocket(new KeyboardSocket());
    KeyboardES::Ptr keybES(new KeyboardES(keybSocket));
    SEH::Ptr keybSEH(new KeyboardSEH("log.txt", keybES));
    AEH::Ptr keybAEH;
    HSHA::Handlers keybHandlers(keybSEH, keybAEH);

    TcpSocket::Ptr listenerSokcet(new TcpSocket());
    ListenerES::Ptr listenerES(new ListenerES(listenerSokcet,
    5050));
    SEH::Ptr acceptorSEH(new AcceptorSEH(listenerES, hsha));
    AEH::Ptr acceptorAEH;
    HSHA::Handlers acceptorHandlers(acceptorSEH, acceptorAEH);

    hsha.add(keybHandlers);
    hsha.add(acceptorHandlers);

    hsha.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;
}
```

To już klasyka – zwięzła funkcja jedynie z:

- fabrykacją jedynie najważniejszych obiektów: obiekty obsługujące zdarzenia gniazda klawiatury i serwera (oraz ich gniazda same w sobie) oraz obiekt silnika;
- uruchomieniem pętli obsługi zdarzeń silnika HSHA.

Zwróćmy dodatkowo uwagę, że dzięki reaktywnemu podejściu do obsługi zdarzeń gniazda serwerowego i klawiatury, aplikacja nadal może być zamykana bezpiecznie przez luźne rzucenie wyjątku odpowiedniej treści tak jak to miało miejsce w przypadku reaktora.

WNIOSKI

Mocne strony

- Po raz drugi dzięki dobrze posegregowanej logice domenowej i aplikacyjnej poważna zmiana (znów zmienia się model wielowątkowy) nie jest inwazyjna – jest wręcz skromna.
- Brak konieczności ingerencji w kod oparty o natywne, bardzo podatne na błędy API C.
- Ponieważ lwia część pracy (odpowiadanie echo) jest wykonywana wielowątkowo, aplikację nadal można spokojnie wyskalować podobnie jak przy okazji wzorca Thread Pool.
- Dzięki nienaruszeniu bazowej architektury obiektowej utrzymaliśmy proste zależności między obiektami i poziom testowalności kodu.
- Kod (przynajmniej w porównaniu do poprzedniej wersji) tylko nieznacznie stracił na czytelności i intuicyjności (choć to absolutnie dostarcza znajomość mechaniki wzorca!).

Słabe strony

- Może się wydawać, że takie podejście do tak prostego zadania (serwer echo) jest nieco przesadzone, niemniej jednak jest to tylko pokaz koncepcji, w którą bardzo łatwo wszczepić dowolny protokół powyżej warstwy czwartej modelu OSI (np. HTTP).
- Nadal stan obsługi błędów pozostawia sporo do życzenia, choć nadal w 100% się sprawdza.
- Rozmiar TP będącej częścią silnika HSHA jest podany na sztywno, a mógłby być przynajmniej wczytywany z linii argumentów wywołania programu.

ZAKOŃCZENIE

Tradycyjnie zachęcam do zajrzenia do ramki „W sieci", gdzie znajdują się linki do całości kodów cytowanych w tej i poprzednich częściach: pobierajcie, testujcie i debugujcie, a przede wszystkim spróbujcie dokonać analizy porównawczej wszystkich rozwiązań podanych do tej pory w różnych dziedzinach.

W następnym artykule włożę omówione trzy rozwiązania w jedną, konfigurowalną aplikację, korzystając przy tym z innych przydatnych wzorców.

W sieci

- Kod źródłowy aplikacji z części I:
  https://github.com/RomanUlan/ReactorBase
- Kody źródłowe aplikacji z części II:
  https://github.com/RomanUlan/ReactorInheritance
  https://github.com/RomanUlan/ReactorTemplates
- Kody źródłowe aplikacji z części III:
  https://github.com/RomanUlan/ThreadPool
- Kod źródłowy aplikacji prezentowanej w niniejszej części:
  https://github.com/RomanUlan/HalfSyncHalfAsync

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".