Roman Ulan

Wzorce silników zdarzeń w C++

Część I: Wzorzec Reaktor i podstawowa implementacja

Wzorzec Reaktor to podstawowy, jeden z najprostszych, ale zarazem stosunkowo efektywny wzorzec obsługi zdarzeń. Z tego artykułu dowiesz się, jakie ma własności oraz czy i kiedy go wykorzystać już na etapie projektowania rozwiązania problemu w danej aplikacji.

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

KRÓTKA WZMIANKA O C++

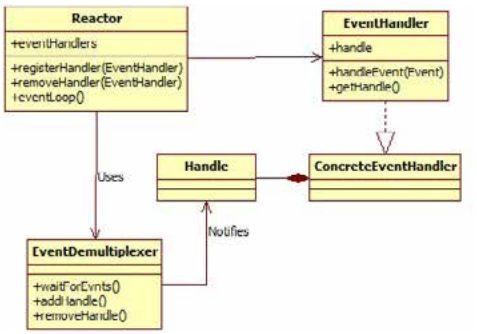
Język C++ to właściwie jeden z bardziej skomplikowanych i zarazem jeden z najbardziej elastycznych języków. Należy pamiętać o tym , że C++ jest generalnie kwalifikowany do języków niższego poziomu. Z elementami, nad którymi początkowo ciężko zapanować, należy się oswoić i nabrać w nich płynności. Koniecznie należy też się przyzwyczaić do przewrotności tego języka – weźmy przykładowo na tapetę metody `const`. Metodę deklarujemy jako „const'ową" po to, by zabronić zmiany składowych klasy, w której ta metoda jest zadeklarowana. Ale, ale... gdyby jednak była konieczność w metodzie zmiany jednej ze składowych klasy i nie wolno byłoby też zmienić sygnatury tej metody?! Nieważne, jak często mogłoby się to zdażyć, jaki byłby koszt zmiany sygnatury, twórcy języka dali taką możliwość, wprowadzając słówko kluczowe `mutable`. Nie chcę być źle zrozumiany – nie twierdzę, że `mutable` jest przyczyną wszelkiego zła, ale fakt jest jeden: należy rozumieć, po co jest, by umiejętnie go użyć, i to sprostowanie odnosi się do każdej takiej składowej języka.

WZORZEC REAKTOR W TEORII

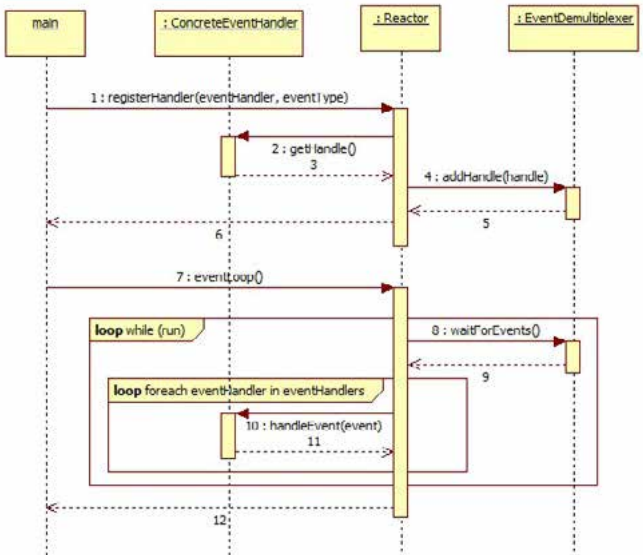
Wzorzec ten służy do rozpropagowania pewnego zdarzenia/zdarzeń ze źródła zdarzeń, którym może być jeden lub wiele kanałów (prościej mówiąc: z pewnego demultipleksera event'ów).

Często w dokumentacji technicznej występuje jako jeden z elemetów języka wzorców (Pattern Language) przedstawiającego opartą na wzorcach architekturę rozwiązania jakiegoś problemu (np.: serwer HTTP), gdzie najczęściej stanowi podstawę obsługi żądań wpadających z zewnątrz.

Wzorzec ten świetnie prezentuje się przy pomocy dwóch schematów UML:



Rysunek 1. Diagram klas dla wzorca Reaktor



Rysunek 2. Diagram interakcji dla wzorca Reaktor

Czytelnicy, którzy czytali książkę Gangu Czterech pt.„Design Patterns: Elements of Reusable Object-Oriented Software", słusznie zauważają podobieństwo do wzorca Observer. Bądź co bądź Reaktor to rozszerzony odpowiednik obserwatora, pochodzący z serii książek POSA (Pattern Oriented Software Architecture).

Najważniejszą własnością omawianego wzorca, z punktu widzenia asynchronicznej obsługi zdarzeń, jest fakt, że pobrane/otrzymane zdarzenia ze źródła są obsługiwane w tym samym wątku sekwencyjnie (jeden po drugim) w zarejestrowanych, odpowiadającym im (lub wszystkich) obiektach zwanych popularnie handler'ami.

PRZYKŁAD

Zacznijmy od klasycznego już przykładu: program serwerowy, do którego podłączają się programy klienckie (np. telnet), wysyłają wiadomość i otrzymują zwrotnie echo.

Demultipleksacja zdarzeń jest oparta o *epoll* (młodszy brat poll'a i select'a). Przyjrzyjmy się bliżej głównej klasie:

Listing 1. Implementacja klas bazowych – podstawa wzorca

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

  public:
    explicit EventHandler(int);
    virtual ~EventHandler();
  private:
    EventHandler(const EventHandler&);
    EventHandler& operator=(const EventHandler&);
```

```
  public:
    int getHandle() const;
    virtual void handleEvent(uint32_t) = 0;

  protected:
    int m_fd;
  }; //class EventHandler

public:
  EpollReactor();
  ~EpollReactor();

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

public:
  void registerHandler(EventHandler::Ptr);
  void removeHandler(int);
  void eventLoop() const;

private:
  typedef ::std::map<int, EventHandler::Ptr> t_handlers;
  int m_epollFd;
  int m_maxEvents;
  t_handlers m_handlers;
}; //class EpollReactor

EpollReactor::EpollReactor()
  : m_epollFd(-1), m_maxEvents(100)
{
  m_epollFd = ::epoll_create(m_maxEvents);
  if (m_epollFd < 0)
  {
    throw ::std::runtime_error("Epoll create failed");
  }
}

EpollReactor::~EpollReactor()
{
  close(m_epollFd);
}

void EpollReactor::registerHandler(EventHandler::Ptr
p_eventHandler)
{
  int fd = p_eventHandler->getHandle();
  int option = EPOLL_CTL_ADD;

  epoll_event e;
  ::memset(&e, 0, sizeof(e));
  e.data.fd = fd;
  e.events  |= EPOLLIN  |  EPOLLRDHUP
            | EPOLLERR  |  EPOLLHUP;

  if (::epoll_ctl(m_epollFd, option, fd, &e) < 0)
  {
    throw ::std::runtime_error("Add handler to epoll failed");
  }

  m_handlers[fd] = p_eventHandler;
}

void EpollReactor::removeHandler(int p_fd)
{
  t_handlers::iterator i = m_handlers.find(p_fd);

  if (i == m_handlers.end())
  {
    throw ::std::runtime_error("Handler not found");
  }

  if (::epoll_ctl(m_epollFd, EPOLL_CTL_DEL, p_fd, 0) < 0)
  {
    m_handlers.erase(i);
    throw ::std::runtime_error("Cannot remove fd from epoll");
  }

  m_handlers.erase(i);
}

void EpollReactor::eventLoop() const
{
  while (1)
  {
    ::boost::scoped_array<epoll_event> events(new
    epoll_event[m_maxEvents]);
    ::memset(events.get(), 0, m_maxEvents * sizeof (epoll_event));
    int eventsLen = 0;

    eventsLen = ::epoll_wait(m_epollFd, events.get(), m_maxEvents, -1);
    if (eventsLen < 0)
    {
      throw ::std::runtime_error("Epoll wait failed");
    }

    for (int i = 0; i < eventsLen; ++i)
    {
      t_handlers::const_iterator ih = m_handlers.find(events[i].
      data.fd);
      if (ih != m_handlers.end())
      {
        ih->second->handleEvent(events[i].events);
      }
      else
      {
        throw ::std::runtime_error("Epoll returned event for \
                                    unfounded handler");
      }
    }
  } //while (1)
}

EpollReactor::EventHandler::EventHandler(int p_fd)
  : m_fd(p_fd)
{
  if (p_fd < 0)
  {
    throw ::std::runtime_error("Invalid file descriptor");
  }
}

EpollReactor::EventHandler::~EventHandler()
{
}

int EpollReactor::EventHandler::getHandle() const
{
  return m_fd;
}
```

Kluczowe elementy implementacji

Na początek drobne sprostowanie odnośnie deklaracji klasy `EventHandler`: deklarowanie klas w klasie nie jest profesjonalną praktyką, ale w tym wypadku jest to zasadne z uwagi na to, że:

1. nazwa klasy wewnętrznej jest dość popularna, a związek klasy reaktora z interfejsem do obsługi zdarzeń (`EpollReactor::EventHandler`) jest oczywisty,
2. jest to mała, interfejsowa klasa.

Powyższa implementacja wzorca Reaktor opata jest o idiom RAII (*Resource Acquisition Is Initialization*). Ma to istotną zaletę - obiekt po konstrukcji jest od razu gotowy do użycia, bez konieczności wołania na nim metod typu `initialize`, a wszystkie zasoby reaktora zostają sprzątnięte w czasie destrukcji. Używając takiego obiektu, spokojnie możemy skupić się na logice funkcjonalności, a nie dodatkowo na spójności stanu niskopoziomowych detali.

Bardzo ciekawym elementem tego kodu jest wykorzystanie shared_ptr z otwartej biblioteki boost, której spora część została wciągnięta do C++11. Shared_ptr to „smart pointer" bazujący na mechanizmie zliczania referencji. To mocne narzędzie, które należy używać rozważnie, czai się tu bowiem problem cyklicznych referencji, o którym tu nie piszę, więc zainteresowanych odsyłam do literatury.

Kolejne dwa punkty powiązane z poprzednim oraz ze sobą to wirtualny destruktor klasy `EventHandler` i kontener składujący wspomniane wyżej smartowne wskaźniki do konkretnych instancji obsługujących zdarzenia.

Użycie shared_ptr powoduje, że:

- Reaktor nie musi być jedynym, który trzyma odniesienia do EH (może go dzielić z innymi), ale póki EH jest w reaktorze zarejestrowany (istnieje w mapie składującej `m_handlers`), żyć będzie,
- gdy reaktor usunie go (destrukcja całego reaktora lub jawne odrejestrowanie EH) z mapy i to było ostatnie takie odniesienie (`shared_ptr`), to uruchomi jego destrukcję - i tu potrzebny jest wirtualny destruktor, gdyż formalnie typ wskaźnika jest `EventHandler`! Dzięki wirtualnemu destruktorowi C++„połapie" się, że za danym wskazaniem stoi typ konkretny, i odpali jego destruktor.

Kolejnymi istotnymi punktami są:

- jednoargumentowy konstruktor klasy `EventHandler` zadeklarowany jako „explicit", by uniknąć niejawnych przekształceń typów (jeśli ta problematyka nie jest Ci znana, zdecydowanie zachęcam do poczytania odpowiedniej literatury, np. „C++ Bardziej Efektywny" Scott'a Meyers'a);
- publiczny zakaz kopiowania obu klas – w przypadku reaktora jego kopio-

wanie jest po prostu niebezpieczne, a w przypadku EH uzasadnieniem są podpunkty uzasadnienia użycia `boost::shared_ptr`.

Oczywiście można się spierać, co lepsze: mapa czy wektor par, a może lista dla składowania odniesień do poszczególnych *EH* (składowa klasy Reaktor `m_handlers`) – ale to detal na optymalizację, a jedna z reguł dobrych praktyk mówi: unikaj przedwczesnej optymalizacji!

Sercem reaktora jest metoda `eventLoop`, w której dokonuje się oczekiwanie na zdarzenia w epoll'u (funkcja `epoll_wait`), a gdy ten zwróci zestaw zdarzeń, szukamy dlań zarejestrowanych obiektów obsługi i propagujemy im ich event'y.

Aplikacja docelowa

Teraz już kod aplikacji docelowej:

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

```
class AcceptorEH : public EpollReactor::EventHandler
{
public:
  AcceptorEH(int, EpollReactor&);
  virtual ~AcceptorEH();

public:
  virtual void handleEvent(uint32_t);

private:
  int m_port;
  struct sockaddr m_address;
  socklen_t m_addressLength;
  EpollReactor& m_reactor;
};

AcceptorEH::AcceptorEH(int p_port, EpollReactor& p_reactor)
  : EventHandler(0), m_port(p_port), m_reactor(p_reactor)
{
  m_fd = ::socket(AF_INET, SOCK_STREAM, 0);
  if (m_fd < 0)
  {
    throw ::std::runtime_error("Cannot create acceptor socket");
  }

  int flags = fcntl(m_fd, F_GETFL, 0);
  fcntl(m_fd, F_SETFL, flags | O_NONBLOCK);

  struct sockaddr_in addr;
  ::memset(&addr, 0, sizeof(addr));

  addr.sin_family = AF_INET;
  addr.sin_port = htons(p_port);
  addr.sin_addr.s_addr = INADDR_ANY;

  ::memcpy(&m_address, &addr, sizeof(addr));
  m_addressLength = sizeof addr;

  if (::bind(m_fd, &m_address, m_addressLength) < 0) {
    throw ::std::runtime_error("Cannot bind acceptor socket");
  }

  if (::listen(m_fd, 1) != 0) {
    throw ::std::runtime_error("Cannot listen on acceptor
    socket");
  }
}

AcceptorEH::~AcceptorEH()
{
  ::close(m_fd);
}

void AcceptorEH::handleEvent(uint32_t p_event)
{
  if (p_event & EPOLLIN)
  {
    struct sockaddr addr;
    ::memset(&addr, 0, sizeof(addr));
    socklen_t addrLen = sizeof(addr);
    int fd = ::accept(m_fd, &addr, &addrLen);
    if (fd < 0)
    {
      throw ::std::runtime_error("Cannot accept incoming
      connection");
    }
    EpollReactor::EventHandler::Ptr eh(new EchoResponderEH(fd));
    m_reactor.registerHandler(eh);
  }
  else
  {
    throw ::std::runtime_error("Bad event for acceptor");
  }
}

class EchoResponderEH : public EpollReactor::EventHandler
{
public:
  explicit EchoResponderEH(int);
  virtual ~EchoResponderEH();

public:
  virtual void handleEvent(uint32_t);
};

EchoResponderEH::EchoResponderEH(int p_fd)
  : EventHandler(p_fd)
{
}

EchoResponderEH::~EchoResponderEH()
{
  ::close(m_fd);
}

void EchoResponderEH::handleEvent(uint32_t p_event)
{
  if (p_event & EPOLLIN)
  {
    char msg[2048];
    ::memset(msg, 0, 2048);
    int r = ::read(m_fd, msg, 2048);
    if (r < 0)
    {
      throw ::std::runtime_error("Cannot read");
    }

    int w = ::write(m_fd, msg, r);
    if (w < 0)
    {
      throw ::std::runtime_error("Cannot write");
    }
  }
  else
  {
    throw ::std::runtime_error("Bad event for for responder");
  }
}

class KeyboardEH : public EpollReactor::EventHandler
{
public:
  KeyboardEH(const ::std::string&);
  virtual ~KeyboardEH();

public:
  virtual void handleEvent(uint32_t);

private:
  ::std::ofstream m_file;
};


KeyboardEH::KeyboardEH(const ::std::string& p_fileName)
  : EventHandler(0), 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::handleEvent(uint32_t p_event)
{
  if (p_event & EPOLLIN)
  {
    char toRead[1024];
    ::memset(toRead, 0, 1024);

    ssize_t r = read(0, toRead, 1024);
    if (r < 0)
    {
      throw ::std::runtime_error("Cannot read from keyb fd");
    }
```

```
    m_file << toRead;

    if (::boost::istarts_with(toRead, "exit"))
    {
      throw ::std::runtime_error("exit");
    }
  }
  else
  {
    throw ::std::runtime_error("Bad event for keyb fd");
  }
}
```

Klasa `AcceptorEH` jest EventHandlerem implementowanym podobnie jak reaktor w oparciu o RAII:

- w konstruktorze otwiera gniazdo nasłuchujące na nadchodzące połączenia
- w destruktorze zamyka je.

Metoda `AcceptorEH::handleEvent` obsługuje zdarzenia (dwa rodzaje) na gnieździe serwerowym, gdzie w przypadku spodziewanego, poprawnego zdarzenia EPOLLIN - nadchodzące połączenie akceptuje je i tworzy dla niego `EchoResponderEH` i rejestruje go w reaktorze.

Klasa `EchoResponderEH` to najprostrza klasa – obsługuje wiadomości klientów i wysyła echo (w rozumieniu architektonicznym jest to implementacja protokołu warstwy aplikacji!).

Klasa `KeyboardEH` to klasa obsługi zdarzeń na standardowym wejściu (klawiatura).

Kwintesencja implementacji

Teraz czas na kwintesencję, czyli implementację funkcji `main`:

Listing 3. Implementacja funkcji main

```
int main(int, char**)
{
  try
  {
    EpollReactor reactor;
    EpollReactor::EventHandler::Ptr ehKeyb(new KeyboardEH("log.
    txt"));
    EpollReactor::EventHandler::Ptr ehAcceptor(new
    AcceptorEH(5050, reactor));

    reactor.registerHandler(ehKeyb);
    reactor.registerHandler(ehAcceptor);

    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;
}
```

Dzięki poprawnej implementacji wzorca główna funkcja programu jest (moim subiektywnym zdaniem) prosta i przejrzysta:

- utworzenie (na stosie) obiektu reaktora
- utworzenie (w stercie) obiektu obsługi zdarzeń standardowego wejścia
- utworzenie (w stercie) obiektu obsługi zdarzeń gniazda serwerowego
- rejestracja obu EH w reaktorze
- wystartowanie reaktora
- całość otoczona zbiorczym try-catch dla złapania wyjątków, które stanowią podstawę obsługi błędów (dość ubogiej :)) tej aplikacji.

WNIOSKI

Mocne strony

- Podany przykład jawnie pokazuje możliwość wykorzytania wzorca Reaktor w rzeczywistych projektach – klasy `AcceptorEH` i `EchoResponderEH` w praktyce mogą implementować obsługę dowolnie zmyślnego protokołu warstwy aplikacji lub zupełnie innych zdarzeń gniazdo-pochodnych dla naszego Reaktora.
- Dzięki implementacji wzorca bez nagięć dokumentowanie takiego kodu sprowadza się do opisu wzorca i opisu klas konkretnych.
- Moim subiektywnym zdaniem kod wygląda czysto i przejrzyście, a zastosowanie kilku trików powoduje, że kod jest bezpieczny od strony wycieków pamięci (zmory programowania w C++).

Słabe strony

- Obecny stan obsługi błędów oczywiście pozostawia sporo do życzenia, choć w tym prostym przykładzie w 100% się sprawdza.
- Spostrzegawczy czytelnicy zauważyli, że rejestracja i derejestracja EH nie jest bezpieczna wielowątkowo – ale proszę zwrócić uwagę, że Reaktor jest jedno-wątkowy! Choć fakt: zawsze w innych warunkach inne wątki mogłyby chcieć coś wrzucać i usuwać z Reaktora…
- Ciężko przewidzieć zachowanie tego Reaktora, gdybyśmy na początku nie zarejestrowali żadnego EH, a wystartowali jego pętlę obsługi zdarzeń.
- Pokrycie testami tego kodu wymaga tzw. mockowania natywnych wywołań systemowych, co nie należy do przyjemności.

ZAKOŃCZENIE

Silniki zdarzeń stanowią podstawę większości aplikacji „backend'owych" i nie tylko. To od nich w największej mierze zależy wydajność całej aplikacji i niekiedy całego systemu. Charakterystyka tego problemu jest bardzo złożona, a jednym z powodów tego jest fakt, iż w dużej mierze zakrawa na tematykę przetwarzania równoległego i wszystkich trudności z tym powiązanych (takich jak wyścigi do zasobów, problemy głodu czy tzw. „deadlock").

Zainteresowanych zachęcam koniecznie do pobrania kodu (link w ramce „W sieci") i sprawdzenia efektów ubocznych wynikających ze słabych stron.

Bardziej dociekliwi pewnie zastanawiają się, czy nie da się jakoś odseparować logiki samego wzorca od niskopoziomowych detali. Odpowiedź jest oczywista: jasne, że tak! To był tylko etap ekstrakcji, co prawda bardzo imperatywnego, interfejsu. W następnym artykule pokażę, jak przy pomocy paradygmatu obiektowego, kolejnych wzorców i kilku drobnych trików stworzyć bloki reużywalnego oraz podatnego na testowanie kodu: generycznego silnika zdarzeń i obiektowych komponentów komunikacji.

W sieci

- Kod źródłowy aplikacji: https://github.com/RomanUlan/ReactorBase

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".