Roman Ulan

Wzorce silników zdarzeń w C++

Część V: „Put all things together” – orkiestracja poprzednich rozwiązań w jedną konfigurowalną, skalowalną aplikację.

W poprzednich artykułach przedstawiałem kolejne wzorce, które znacznie porządkują i upraszczają implementację silników obsługi zdarzeń. Pierwszym był Reactor – prosty, jednowątkowy, drugi to ThreadPool – wielowątkowa alternatywa dla reaktora, trzeci: Half Sync/Half Async – zrównoważone połączenie obu poprzedników. Każdy z nich miał jasno nakreślony zbiór wad i zalet, na bazie których należy dobrać odpowiedni wzorzec do klasy problemu, który przyszło nam implementować. Przez klasę problemu mam na myśli nie tylko złożoność podstawowej funkcjonalności (np. protokołu, który implementujemy), ale też zbiór wymagań niefunkcjonalnych – skalę, w jaką mamy się wpasować (np. ilość obsługiwanych klientów). W tej części pokażę, jak złożyć dotychczasowe rozwiązania w jedną aplikację sterowaną prostą konfiguracją.

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.

PROXY

Na początek przyjrzyjmy się dokładniej jeszcze raz najważniejszym interfejsom, od których zależy cała aplikacja:

Listing 1. Implementacja klas wzorca

```
class Reactor
{
public:
  typedef std::shared_ptr<Reactor> Ptr;

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

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

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;

  std::mutex m_mutex;
  bool m_run;
  t_handlers m_handlers;
  EventDemultiplexer::Ptr m_eventDemultiplexer;
}; //class Reactor

class ThreadPool
{
public:
  typedef std::shared_ptr<ThreadPool> Ptr;

public:
  explicit ThreadPool(EventDemultiplexer::Ptr, size_t);
  ~ThreadPool();

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

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

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;

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

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

Patrząc na tę implementację, od razu rzuca się w oczy ogromne podobieństwo i chęć ujednolicenia przez wprowadzenie interfejsu abstrakcyjnego, z którego będą dziedziczyć nasze trzy implementacje wzorców. Jest to właściwie zasadne choćby z uwagi na powielenie kodu. Jednak cała operacja nie jest trywialna i oczywista z uwagi na sygnaturę funkcji `add`. Ta w przypadku klas: `Reactor` i `ThreadPool`, jest dokładnie taka sama, ale jeśli chodzi o `HalfSyncHalfAsync`, już nie. Jak więc można by sobie z tym poradzić? Otóż mamy mechanizm szablonów, więc możemy zrobić tak:

1. Wprowadzić szablonową klasę abstrakcyjną (np. `EventEngine`), która jako parametr szablonu będzie przyjmowała rodzaj `EventHandler`'a (lub `EventHandler`'ów w przypadku HSHA).
2. Przenieść do niej właściwie całą implementację klasy `Reactor`.
3. Dodać chronioną, czysto wirtualną metodę `processEvents`, którą wszystkie trzy klasy pochodne musiałyby zaimplementować, a jej kod wywoływany byłby w `eventLoop`, w miejscu gdzie obecnie odbywa się zasadnicza obsługa zdarzeń (wywołanie metody `handle` z odpowiednich `EventHandler`'ów).
4. Zwirtualizować przynajmniej metodę `add`, by móc ją przeładować i wyspecjalizować w klasie `HalfSyncHalfAsyc`.

Już same punkty brzmią magicznie i wydają się być strasznie pogmatwane, i tak faktycznie jest. Dziedziczenie i jednoczesna specjalizacja szablonów nie jest czymś oczywistym! Generalnie jest to możliwe z punktu widzenia składni języka C++. Patrząc jednak przez pryzmat zrozumienia takiego kodu przez nowe osoby w projekcie, czy też późniejsze problemy z utrzymaniem tak skomplikowanego rozwiązania, należy się mocno zastanowić nad wyborem takiego podejścia.

Patrząc na ten problem z punktu widzenia czystej inżynierii oprogramowania, omawiane wzorce są zupełnie niezależnymi bytami, i podchodząc do implementacji w stylu DDD (ang. *Domain Driven Design*, która kładzie ogromny nacisk na jak najwierniejsze odwzorowanie rzeczywistości w modelu obiektowym), nie ma żadnego obowiązku dokonywania tak karkołomnych przedsięwzięć.

W jaki sposób więc zaradzić sprawie? Otóż możemy posilić się wzorcem *Proxy*.

Uważni czytelnicy serii pewnie oczekują tu dwóch tradycyjnych diagramów UML (klas i sekwencji). Nic jednak bardziej mylnego, bo wzorzec ten w zasadzie nie ma przedstawiającego go UMLa, ale za to jego opis w postaci zdefiniowanych celów jest bardzo przejrzysty – zastępuje inne obiekty:

- „ciężkie" obiekty, których kopiowanie jest cenne (np. inteligentne wskaźniki)
- obiekty wymagające kontroli dostępu (uwierzytelnianie i autoryzacja)
- obiekty, których API można znacznie uprościć.

Ponieważ kod naszej „biblioteki" z implementacją silników obsługi zdarzeń (komponent `EventEngines`) jest już stabilny i krytyczny z punktu widzenia realizowanego zadania, wprowadzanie do niego zmian (zwłaszcza tak inwazyjnych jak dziedziczenie ze specjalizacją szablonów, wspomniane nieco wyżej) jest niepożądane. Postanowiłem skorzystać z dobrodziejstw, jakie daje nam *Proxy*, które nie wymaga tknięcia implementacji obiektów, dla których będzie pośredniczył w dostępie, i przygotowałem nową hierarchię klas na potrzeby ujednolicenia interfejsu do naszych rdzennych klas:

Listing 2. Implementacja klas Proxy dla klas wzorców silników zdarzeń

```
class EventEngine {
public:
  typedef std::shared_ptr<EventEngine> Ptr;
  virtual ~EventEngine() {};
  virtual void start() = 0;
  virtual void stop(bool) = 0;
};

class ReactorEE: public EventEngine {
public:
  ReactorEE(EventEngines::Reactor::Ptr);
  virtual ~ReactorEE();
  virtual void start();
  virtual void stop(bool);
  private:
  EventEngines::Reactor::Ptr m_reactor;
};

class ThreadPoolEE: public EventEngine {
public:
  ThreadPoolEE(EventEngines::ThreadPool::Ptr);
  virtual ~ThreadPoolEE();
  virtual void start();
  virtual void stop(bool);
  private:
  EventEngines::ThreadPool::Ptr m_threadPool;
};

class HalfSyncHalfAsyncEE: public EventEngine {
public:
  HalfSyncHalfAsyncEE(HSHA::Ptr);
  virtual ~HalfSyncHalfAsyncEE();
  virtual void start();
  virtual void stop(bool);
private:
  HSHA::Ptr m_hsha;
};

ReactorEE::ReactorEE(EventEngines::Reactor::Ptr p_reactor)
  : m_reactor(p_reactor)

{
}

ReactorEE::~ReactorEE()
{
}

void ReactorEE::start()
{
  m_reactor->eventLoop();
}

void ReactorEE::stop(bool p_immediately)
{
  m_reactor->interrupt(p_immediately);
}

ThreadPoolEE::ThreadPoolEE(EventEngines::ThreadPool::Ptr p_threadPool)
  : m_threadPool(p_threadPool)
{
}

ThreadPoolEE::~ThreadPoolEE()
{
}

void ThreadPoolEE::start()
{
  m_threadPool->eventLoop();
}

void ThreadPoolEE::stop(bool p_immediately)
{
  m_threadPool->interrupt(p_immediately);

}

HalfSyncHalfAsyncEE::HalfSyncHalfAsyncEE(HSHA::Ptr p_hsha)
  : m_hsha(p_hsha)
{
}
```

```
HalfSyncHalfAsyncEE::~HalfSyncHalfAsyncEE()
{
}

void HalfSyncHalfAsyncEE::start()
{
  m_hsha->eventLoop();
}

void HalfSyncHalfAsyncEE::stop(bool p_immediately)
{
  m_hsha->interrupt(p_immediately);
}
```

Uważni czytelnicy już widzą i „drapią się w głowę", pytając: dlaczego tak? Przecież to są praktycznie identyczne klasy! Można je z łatwością zastąpić szablonem.

Tak, to prawda, ale takie podejście wymusi konieczność rozumienia zależności pomiędzy obiektami i typami szablonowymi właściwie aż do funkcji `main`, gdzie tak czy inaczej będziemy musieli powielić kod.

W tym natomiast podejściu, takie rozumienie zależności i budowę odpowiedniego obiektu *Proxy* możemy wydelegować do obiektu-wzorca fabryki abstrakcji, ucinając niebezpieczną eskalację złożonych zależności:

Listing 3. Fabryka abstrakcji dla obiektów proxy

```
class EventEngineFactory {
public:
  static EventEngine::Ptr create(const Config&);
private:
  static EventEngine::Ptr createReactor(const Config&);
  static EventEngine::Ptr createTP(const Config&);
  static EventEngine::Ptr createHSHA(const Config&);
};

EventEngine::Ptr EventEngineFactory::create(const Config& p_config)
{
  EventEngine::Ptr result;
  switch (p_config.eventEngineType) {
  case Config::ThreadPool:
    result = createTP(p_config);
    break;
  case Config::HalfSyncHalfAsync:
    result = createHSHA(p_config);
    break;
  default:
  result = createReactor(p_config);
  }

  return result;
}

EventEngine::Ptr EventEngineFactory::createReactor(const Config&
p_config)
{
  Epoll::Ptr epoll(new Epoll());
  EpollED::Ptr epollED(new EpollED(epoll));
  EventEngines::Reactor::Ptr r(new EventEngines::Reactor(epollED));

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

  TcpSocket::Ptr listenerSokcet(new TcpSocket());
  ListenerES::Ptr listenerES(new ListenerES(listenerSokcet, p_
  config.port));
  EventHandler::Ptr acceptorEH(new AcceptorEH<EventEngines::React
  or>(listenerES, *r));

  r->add(keybEH);
  r->add(acceptorEH);

  return EventEngine::Ptr(new ReactorEE(r));
}

EventEngine::Ptr EventEngineFactory::createTP(const Config& p_config)
{
  Epoll::Ptr epoll(new Epoll());
  EpollED::Ptr epollED(new EpollED(epoll));
  EventEngines::ThreadPool::Ptr tp(new EventEngines::ThreadPool(e
  pollED, p_config.eventEngineSize));

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

  TcpSocket::Ptr listenerSokcet(new TcpSocket());
  ListenerES::Ptr listenerES(new ListenerES(listenerSokcet, p_
  config.port));
  EventHandler::Ptr acceptorEH(new AcceptorEH<EventEngines::Threa
  dPool>(listenerES, *tp));

  tp->add(keybEH);
  tp->add(acceptorEH);

  return EventEngine::Ptr(new ThreadPoolEE(tp));
}

EventEngine::Ptr EventEngineFactory::createHSHA(const Config& p_config)
{
  Epoll::Ptr epoll(new Epoll());
  EpollED::Ptr epollED(new EpollED(epoll));
  HSHA::Ptr hsha(new HSHA(epollED, p_config.eventEngineSize));

  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, p_
  config.port));
  SEH::Ptr acceptorSEH(new AcceptorSEH(listenerES, *hsha));
  AEH::Ptr acceptorAEH;
  HSHA::Handlers acceptorHandlers(acceptorSEH, acceptorAEH);

  hsha->add(keybHandlers);
  hsha->add(acceptorHandlers);

  return EventEngine::Ptr(new HalfSyncHalfAsyncEE(hsha));

}
```

Czytelnicy, którzy dociekliwie analizowali przykłady w kolejnych częściach serii, zauważyli z pewnością dwie rzeczy:

1. Każda z kolejnych funkcji fabrykujących kolejne silniki zdarzeń (a właściwie ich obiekt pełnomocnika) to niemalże kopia sporej części funkcji `main` z kolejnych części.
2. Pojawił się nowy byt Config – o tym za moment.

KONFIGURACJA

Ponieważ naszym aktualnym celem jest konfiguracja i skalowalność, przyszedł czas na poświęcenie uwagi temu pierwszemu.

W tej materii mamy dwa problemy: źródło konfiguracji i jego interpretacja oraz reprezentacja konfiguracji wewnątrz aplikacji. Na te dwa cele przygotowałem dwa obiekty:

Listing 3. Obiekty związane z konfiguracją aplikacji

```
struct Config {
  enum EventEngineType {
    Reactor = 0,
    ThreadPool = 1,
    HalfSyncHalfAsync = 2
  };

  EventEngineType eventEngineType;
  size_t eventEngineSize;
  int port;
};
class ConfigParser {
public:
  ConfigParser(int p_argc, char** p_argv);
  ~ConfigParser();
  Config get() const;
private:
  int m_argc;
  char** m_argv;
};

ConfigParser::ConfigParser(int p_argc, char** p_argv)
  : m_argc(p_argc), m_argv(p_argv)
{
}

ConfigParser::~ConfigParser()
{
}

Config ConfigParser::get() const
{
  Config c;
  c.eventEngineType = Config::Reactor;
  c.eventEngineSize = 1;
  c.port = 5050;

  int o;
```

```
  while((o = getopt(m_argc, m_argv, "e:s:p:")) != EOF) {
    switch (o) {
    case 'e':
      if (strcmp(optarg, "reacor") == 0)
        c.eventEngineType = Config::Reactor;
      else if (strcmp(optarg, "tp") == 0)
        c.eventEngineType = Config::ThreadPool;
      else if (strcmp(optarg, "hsha") == 0)
        c.eventEngineType = Config::HalfSyncHalfAsync;
      else
        throw std::runtime_error("Not supported engine type");
      break;
    case 's':
      c.eventEngineSize = atoi(optarg);
      break;
    case 'p':
      c.port = atoi(optarg);
    }
  }
  return c;
}
```

Struktura Config to najzwyczajniejsza w świecie struktura danych przechowująca dane dotyczące tego, co jest konfigurowalne w naszej aplikacji:

- `eventEngineType` – rodzaj silnika zdarzeń
- `eventEngineSize` – rozmiar puli wątków dla silników TP i HSHA
- `port` – numer portu, na którym nasz serwer echo oczekuje na zgłoszenie się nowych klientów.

`ConfigParser` to implementacja idiomu *Parser*, która interpretuje źródło konfiguracji (linia argumentów wywołania aplikacji) i przekształca je na rozumianą w logice aplikacyjnej strukturę danych Config. Można by się tu zastanowić nad wykorzystaniem kompleksowego wzorca *Interpreter*, lecz w tym wypadku złożoność wyrażeń jest właściwie równa zeru, ograniczyłem się zatem do prostszej postaci.

Main

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

Listing 4. Implementacja funkcji main

```
int main(int argc, char** argv)
{
  try
  {
    ConfigParser cp(argc, argv);
    Config c = cp.get();
    EventEngine::Ptr ee = EventEngineFactory::create(c);
    if (ee) {
      ee->start();
    }
  }
  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;
}
```

Tym razem to jeszcze prostsza funkcja, skupiająca się na scaleniu całej logiki domenowej: pobranie konfiguracji, stworzenie najważniejszych obiektów i wystartowanie silnika zdarzeń.

WNIOSKI

Mocne strony

- Dzięki wykorzystaniu odpowiednich wzorców i odpowiedniej segregacji komponentów na wcześniejszych etapach spokojnie można było dołożyć nową funkcjonalność, a właściwie złączyć kilka rozwiązań w jedno, bez dotykania rdzennych części aplikacji (implementacje poszczególnych silników zdarzeń).
- Brak konieczności ingerencji w kod oparty o natywne, bardzo podatne na błędy API C.
- Dzięki nienaruszeniu bazowej architektury obiektowej utrzymaliśmy proste zależności między obiektami i poziom testowalności kodu.
- Aplikację można dostosować do dowolnych wymagań niefunkcjonalnych (skalowanie) bez zmiany kodu C++, a jedynie linią argumentów wywołania programu.
- Kod utrzymał czytelność i intuicyjność (choć wymaga to znajomości wzorców i ich mechaniki!).

Słabe strony

- Słusznym jest zarzut, że takie podejście do tak prostego zadania (serwer echo) jest przesadzone, niemniej jednak jest to tylko pokaz koncepcji, w którą bardzo łatwo wpleść dowolny protokół powyżej warstwy czwartej modelu OSI (np. HTTP).
- Nadal stan obsługi błędów pozostawia sporo do życzenia, choć nadal w 100% się sprawdza.

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 przedstawioną aplikację w uprząż testową okraszoną pętlami ciągłej integracji.

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 z części IV:
  https://github.com/RomanUlan/HalfSyncHalfAsync
- Kod źródłowy aplikacji prezentowanej w niniejszej części:
  https://github.com/RomanUlan/AllInOne

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".