УДК 681.3.06

Паттерн *State Machine*. Внедрение. Сравнение с другими подходами

Н. Н. Шамгунов,
аспирант
Г. А. Корнеев,
аспирант
А. А. Шалыто,
д-р техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Описывается внедрение паттерна State Machine, предложенного авторами, при проектировании системы управления потоками (thread), осуществляющими асинхронные запросы к базе данных. Выполнено сравнение реализации с использованием предлагаемого паттерна с реализациями на основе флагов и SWITCH-*технологии.*

State Machine pattern, recently proposed by authors, has been applied in design and development of the subsystem of connection threads control. Comparison with traditional approach and SWITCH-technology provided.

Введение

В программировании часто возникает потребность в объектах, изменяющих свое поведение в зависимости от состояния. Обычно поведение таких объектов описывается при помощи конечных автоматов. Существуют различные паттерны проектирования для реализации указанных объектов, описанные, например, в работах [1, 2]. В большинстве из этих паттернов или автоматы реализуются неэффективно, или сильно затруднено повторное использование их компонентов. Эти недостатки устранены в предложенном авторами паттерне *State Machine* [3].

В процессе разработки программного обеспечения в компании *Транзас* [4] используются паттерны проектирования [1]. При этом до последнего времени для объектов, изменяющих свое поведение в зависимости от состояния, применялся паттерн *State*.

Кроме того, отметим, что для успешной разработки крупных проектов необходимо систематически производить улучшение существующего кода — рефакторинг [5], который требуется для того, чтобы целостность программной системы всегда находилась на приемлемом уровне. Одним из методов рефакторинга является метод, названный «*Replace Type Code with State*», идея которого состоит в замене условной логики паттерном *State*.

В настоящей статье предлагается условную логику заменять паттерном *State Machine*, который, как было показано в работе [3], обладает определенными преимуществами по сравнению с паттерном *State*. Это продемонстрировано при проектировании системы управления потоками, осуществляющими асинхронные запросы к базе данных. Выполнено сравнение предложенной реализации с другими подходами — программированием с использованием флагов и *SWITCH*-технологии.

1. Область внедрения

Группа компаний *Transas* была основана в 1990 году в Санкт-Петербурге. Название *Transas* означает TRANsport SAfety Systems (Системы Безопасности на Транспорте). Успешно работая уже более 10 лет, *Transas* является одним из ведущих производителей высокотехнологичных продуктов, пользующихся спросом во всем мире.

1.1. Система *Navi Harbour*

Система *Navi Habour* [7] разрабатывается в *департаменте береговых систем* [6] компании *Transas* и является системой управления движением судов (*СУДС*). Этот продукт установлен в портах многих стран мира, включая Россию. На рис. 1 приведена структурная схема системы *Navi Harbour*.

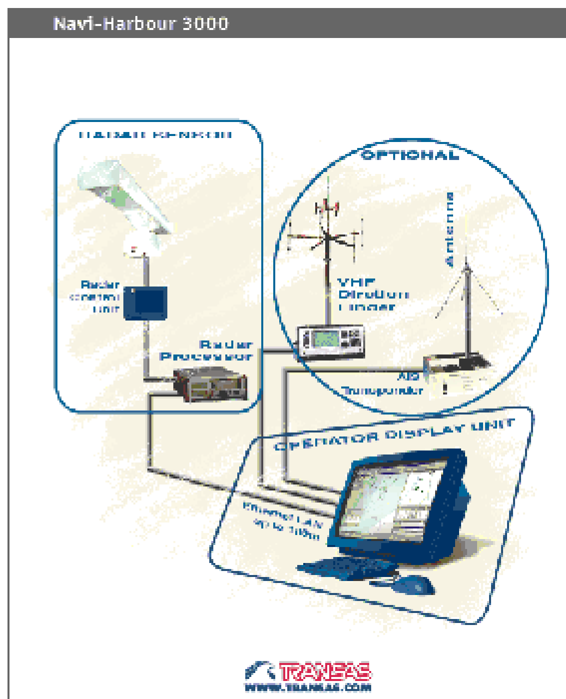

Рис. 1. Структурная схема системы *Navi Harbour*

На этом рисунке показано, что к операторскому дисплейному модулю (*ОДМ*) подключается радар и ряд дополнительных устройств (необязательных — *optional*), таких как, например, телекамеры и сенсоры погодных условий. На каждом дисплейном модуле отображается морская карта акватории, собираются данные устройств, выводятся различные тревоги (например, предупреждения столкновений, сообщения об ошибках устройств), а также другая информация, имеющая отношение к безопасности движения.

Одна из составляющих системы *Navi Harbour* — база данных *СУДС*, которая не представлена на рис. 1. В статье демонстрируется эффективность применения паттерна *State Machine* на примере использования его при разработке менеджера потоков в указанной базе данных.

1.2. База данных *СУДС*

Рассматриваемая база данных предназначена для учета судов и связанной с ними информации, такой как, например, судозаходы, журнал погоды, вахтенный журнал. База данных *СУДС* представляет собой набор устанавливаемых на различных компьютерах и взаимодействующих между собой компонентов. Схема развертывания базы данных *СУДС* приведена на рис. 2. Для упрощения схемы не показаны связи между дисплейными модулями.

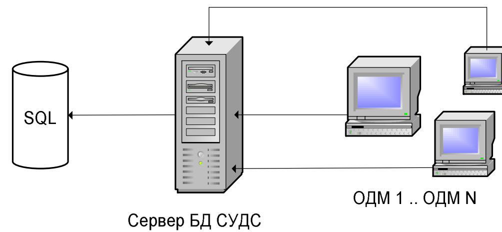

Рис. 2. Схема развертывания базы данных СУДС

В качестве хранилища данных в системе используется *Microsoft SQL Server*, который может быть размещен отдельно или на одном компьютере с сервером базы данных *СУДС*. На компьютерах *ОДМ1* … *ОДМN* установлена клиентская часть базы данных, которая подключается к системе посредством технологии *COM*. Клиентская часть взаимодействует с серверной посредством технологии *DCOM*.

Типичное окно клиента базы данных приведено на рис. 3.

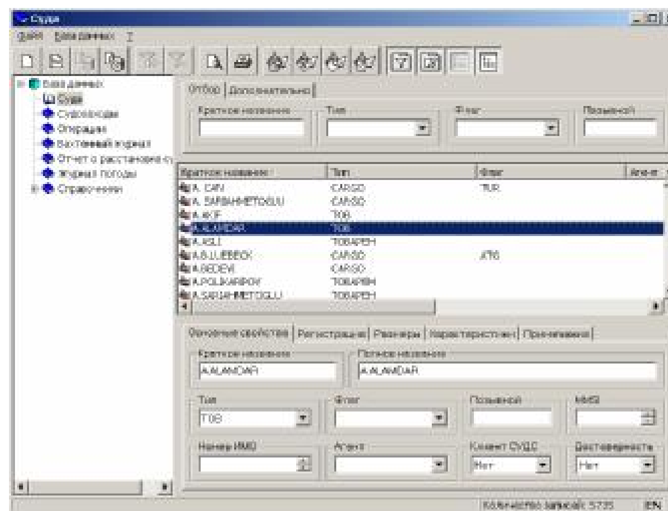

Рис. 3. Окно клиента базы данных

В этом окне производится просмотр, фильтрация и редактирование данных. При этом работа с сервером базы данных выполняется асинхронно — используются потоки (*threads*), управление которыми и является предметом внедрения предложенного паттерна.

2. Постановка задачи

Одним из основных требований к клиентской части базы данных *СУДС* является способность выполнять асинхронные запросы к ее серверной части. Это позволяет не блокировать работу системы *Navi Harbour* во время выполнения запроса, что очень важно для непрерывного мониторинга перемещений судов. Схема взаимодействия клиентской и серверной части приведена на рис. 4.

Серверная часть содержит компонент `VTSDB`, реализующий интерфейс `VTSDB.Application`, через который производится взаимодействие клиентской части с серверной. Компонент `DBWindow` создает потоки, каждый из которых содержит указатель на этот интерфейс.

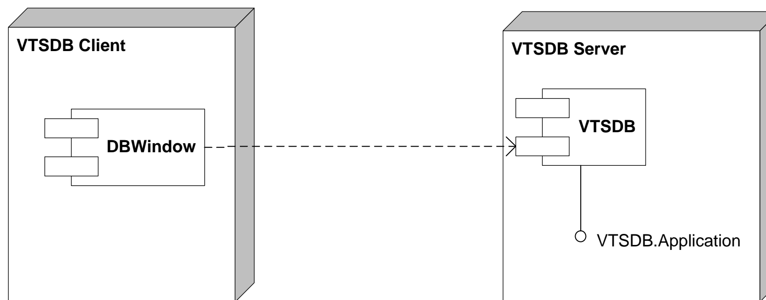

Рис. 4. Схема взаимодействия клиентской и серверной частей

Выполнение асинхронных запросов реализовано в клиентской части базы данных, посредством потоков соединения (*connection threads*). Потоки, хранящие соединение с серверной частью базы данных, создаются для каждого окна. В настоящей работе ставится задача разработки фабрики потоков — класса `ThreadFactory`, предназначенного для создания и уничтожения потоков соединения. Как будет показано в разд. 3, поведение этого класса варьируется в зависимости от состояния. Поэтому для его реализации будем применять паттерн *State Machine* [3]. Отметим, что в этой работе примеры, иллюстрирующие применение предложенного паттерна, реализованы на языке *Java*. Ввиду того, что структура паттерна не зависит от языка программирования, а также в связи с тем, что в компании *Транзас* в основном используется язык программирования *C++* [8], разрабатываемая фабрика потоков также реализован на этом языке использованием библиотеки *Boost* [9].

3. Применение паттерна *State Machine* для проектирования класса *ThreadFactory*

Покажем, как применять предлагаемый паттерн на примере проектирования класса `ThreadFactory`, введенного в разд. 2.

Реализация класса `ThreadFactory` и связанных с ним классов и интерфейсов произведена в пространстве имен `ThreadManagement`.

3.1. Формализация постановки задачи

Класс `ThreadFactory` управляет созданием и уничтожением потоков соединения. Опишем класс, инкапсулирующий поток соединения, который назовем `ConnectionThread`. Этот класс уведомляет о своей успешной инициализации (создании *DCOM*-объекта) или о завершении потока через интерфейс `IThreadNotify`. Указатель на этот интерфейс передается в конструктор класса `ConnectionThread`. Этот интерфейс, выглядит следующим образом:

```
#pragma once

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadNotify
{
  virtual void Started(ConnectionThread * threadPtr) = 0;
  virtual void Stopped(ConnectionThread * threadPtr) = 0;
};

}
```

При инициализации объект `ConnectionThread` вызывает метод `Started`, а при завершении — метод `Stopped` и передает в них указатель на себя.

Класс `ThreadFactory` должен обеспечить:

- корректное создание и уничтожение потоков соединений;
- изоляцию клиента от получения указателя на неинициализированный объект потока;

- создание или удаление не более одного потока соединения одновременно.

Интерфейс этого класса имеет следующий вид:

```
#pragma once

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadFactory
{
  virtual void Request() = 0;
  virtual void Destroy(ConnectionThread * threadPtr) = 0;
  virtual void Cancel() = 0;
};

}
```

Метод `Request` запрашивает поток соединения, метод `Cancel` отменяет запрос на создание потока соединения, а метод `Destroy` — удаляет поток соединения. Уведомления клиента класса `ThreadFactory` о создании или уничтожении потока соединения производится через интерфейс `IThreadRequest`:

```
#pragma once

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadRequest
{
  virtual void Created(ConnectionThread * threadPtr) = 0;
  virtual void Deleted() = 0;
  virtual void Canceled() = 0;
};

}
```

Метод `Created` вызывается при создании объекта соединения, в него передается указатель на инициализированный поток. Метод `Canceled` вызывается при успешной отмене запроса на создание соединения, а метод `Deleted` — при удалении объекта соединения.

Поведение класса `ThreadFactory` зависит от внутреннего состояния. Например, метод `Cancel` приводит к отмене создания потока только в том случае, если объект класса `ThreadFactory` создал объект потока и ожидает завершения его инициализации. Поэтому поведение этого класса можно описывать с помощью конечного автомата. Для его реализации решено применить паттерн *State Machine*, в котором класс `ThreadFactory` является контекстом. Контекст является единственным классом, входящим в паттерн, который доступен пользователю.

3.2. Проектирование автомата *ThreadFactory*

Выделим четыре управляющих состояния:

- *Idle* — ожидание запроса на создание или удаление потока соединения;
- *Creating* — поток создан и ожидается завершение его инициализации;
- *Destroing* — выполнен запрос на уничтожение потока и ожидается завершения потока;
- *Cancelling* — инициализация потока отменена и ожидается завершения потока.

Названия состояний соответствуют именам классов состояний. Классы генерируют следующие события:

- *Idle*: `CREATED` — объект потока создан, `STOP_SENDED` — объекту потока отправлен запрос на остановку;
- *Creating*: `INITIALIZED` — поток инициализирован; `CANCEL_SENDED` — объекту потока отправлен запрос на отмену инициализации и остановку;
- *Destroing*: `DESTROYED` — поток уничтожен;
- *Cancelling*: `CANCELLED` — отмена создания потока.

На рис. 5 приведен описывающий поведение класса `ThreadFactory` граф переходов вида, используемого в предлагаемом подходе [3].

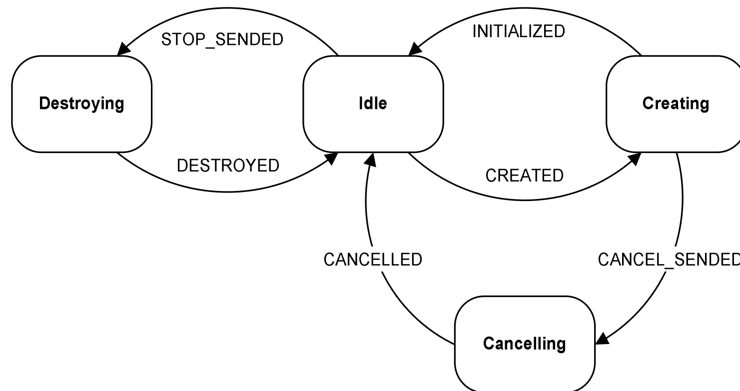

Рис. 5. Граф переходов, описывающий поведение класса **ThreadFactory**

3.3. Диаграмма классов реализации автомата *ThreadFactory*

Как было описано в разд. 3.1, класс `ThreadFactory` должен реализовывать интерфейс `IThreadFactory`. Для получения уведомлений от создаваемого потока класс `ThreadFactory` должен реализовывать интерфейс `IThreadNotify`.

Таким образом, интерфейсом автомата является интерфейс, расширяющий интерфейсы `IThreadFactory` и `IThreadNotify`:

```
#pragma once

#include ".\IThreadFactory.h"
#include ".\IThreadNotify.h"

namespace ThreadManagement
{

struct IThreadFactoryAI : public IThreadFactory, public IThreadNotify
{
};

}
```

Таким образом, в пространстве имен `ThreadManagement` находятся следующие классы и интерфейсы:

- `ConnectionThread` — класс потока соединения;
- `IThreadRequest` — интерфейс уведомления о создании и уничтожении объектов потока;
- `IThreadFactory` — интерфейс фабрики потоков;
- `IThreadNotify` — интерфейс для уведомления о создании/уничтожении потока;
- `IThreadFactoryAI` — интерфейс автомата. Наследуется от интерфейсов `IThreadFactory` и `IThreadNotify`;
- `ThreadFactory` — контекст. Реализуют интерфейс автомата `IThreadFactoryAI`, наследуясь от класса `StateMachine::AutomatonBase<IThreadFactoryAI>`;
- `Idle`, `Creating`, `Destroying`, `Cancelling` — классы состояний. Реализуют интерфейс автомата `IThreadFactoryAI`, наследуясь от класса `StateMachine::StateBase<IThreadFactoryAI>`.

Отметим, что в пространство имен `ThreadManagement`, наряду с классами паттерна *State Machine*, входят также связанные с ними классы и интерфейсы, такие как `ConnectionThread` и `IThreadRequest`. Диаграмма классов реализации автомата `ThreadFactory`, приведена на рис. 6.

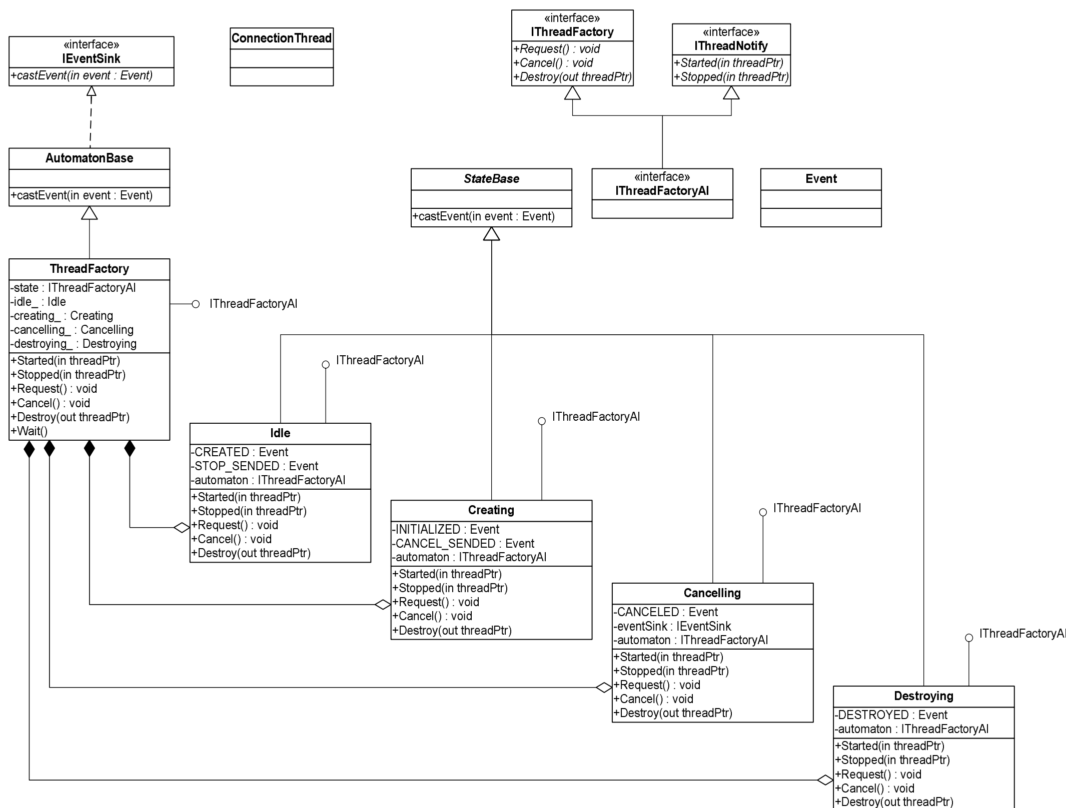

Рис. 6. Диаграмма классов реализации автомата **ThreadFactory**

Отметим, что для того, чтобы не загромождать рисунок, тот факт, что контекст и классы состояний реализуют интерфейс `IThreadFactoryAI`, изображен на рис. 6 в виде линий с кружками. Это заменяет пунктирные линии, которыми реализация интерфейса изображалась в структуре паттерна *State Machine*, приведенной в работе [3].

3.4. Реализация контекста автомата *ThreadFactory*

В соответствии с паттерном *State Machine* логика переходов задается в конструкторе контекста — классе `ThreadFactory`. В этот класс также добавлен метод `Wait`, предназначенный для ожидания завершения текущей операции (например, ожидания удаления потока).

Все методы, входящие в интерфейс автомата, реализуется однотипно: защита от многопоточного доступа (`CSingleLock locker(&lock_, TRUE)`) и делегирование операции текущему объекту состояния (например, для метода `Request` — `statePtr_->Request()`).

Приведем реализацию класса `ThreadFactory` — контекста. Отметим, что этот класс является наследником класса `AutomatonBase<IThreadFactoryAI>`, в котором реализована инфраструктура для выполнения переходов и протоколирования.

При инициализации контекст передает в конструктор своего базового класса `AutomatonBase<IThreadFactoryAI>` ссылку на начальное состояние — `idle_`. При этом вместо имени типа `AutomatonBase<IThreadFactoryAI>` используется имя `AB`, которое является его синонимом.

Переходы заданы в конструкторе класса `ThreadFactory` при помощи вызовов метода `AddTransion`. Этот метод также реализован в классе `AutomatonBase` с использованием структуры данных `map`, входящей стандартную библиотеку *C++*.

Заголовочный файл для класса `ThreadFactory`:

```
#pragma once

#include <afxmt.h>

#include "StateMachine\AutomatonBase.h"
#include ".\IThreadFactoryAI.h"
```

```
#include ".\Idle.h"
#include ".\Creating.h"
#include ".\Canceling.h"
#include ".\Destroying.h"

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadRequest;
class ThreadFactory : public StateMachine::AutomatonBase<IThreadFactoryAI>
{
public:
  ThreadFactory(IThreadRequest & request);
  ~ThreadFactory(void);

  // Public Interface
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
  virtual void Started(ConnectionThread * threadPtr);
  virtual void Stopped(ConnectionThread * threadPtr);

  // Wait for current action finish
  void Wait();

private:
  CCriticalSection lock_;
  CEvent requestEvent_;
  IThreadRequest & request_;
  ConnectionThread * threadPtr_;

  // States
  Idle idle_;
  Creating creating_;
  Canceling canceling_;
  Destroying destroying_;
 };

}
```

Исполняемый файл для класса ThreadFactory:

```
#include "stdafx.h"

#include "boost\bind.hpp"
#include "StateMachine\Event.h"
#include "Logger.h"
#include ".\ThreadFactory.h"
#include ".\ConnectionThread.h"
#include ".\IThreadRequest.h"

namespace ThreadManagement
{

// This warning is "'this' : used in base member initializer list"
// There is no error here cause state classes demand reference to
// AutomatonBase<AI> which is already initialized.
#pragma warning(disable:4355)

ThreadFactory::ThreadFactory(IThreadRequest & request)
: AB(idle_),
  requestEvent_(TRUE, TRUE),
  request_(request),
  threadPtr_(NULL),
  idle_(*this, request, threadPtr_, requestEvent_),
  creating_(*this, request, threadPtr_, requestEvent_),
  canceling_(*this, request, threadPtr_, requestEvent_),
```

```
  destroying_(*this, request, threadPtr_, requestEvent_)
{
  AddTransition(idle_, Idle::CREATED, creating_);
  AddTransition(idle_, Idle::STOP_SENDED, destroying_);
  AddTransition(creating_, Creating::INITIALIZED, idle_);
  AddTransition(creating_, Creating::CANCEL_SENDED, canceling_);
  AddTransition(canceling_, Canceling::CANCELED, idle_);
  AddTransition(destroying_, Destroying::DESTROYED, idle_);
}

#pragma warning(default:4355)

ThreadFactory::~ThreadFactory(void)
{
  ASSERT(threadPtr_ == NULL);
}

void ThreadFactory::Request()
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Request();
}

void ThreadFactory::Cancel()
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Cancel();
}

void ThreadFactory::Destroy(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Destroy(threadPtr);
}

void ThreadFactory::Started(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Started(threadPtr);
}

void ThreadFactory::Stopped(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  statePtr_->Stopped(threadPtr);
}

void ThreadFactory::Wait()
{
  ::WaitForSingleObject(requestEvent_, INFINITE);
}

}
```

Методы, образующие интерфейс автомата, могут вызываться из различных потоков. При этом методы интерфейса `IThreadFactory` вызываются из основного потока, а методы интерфейса `IThreadNotify` — из потоков, создаваемых фабрикой. Поскольку при вызове метода автомата состояние может измениться, для обеспечения потоковой безопасности в контексте производится блокировка на основе критической секции `CSingleLock locker(&lock_, TRUE)`. Отметим, что такое использование критических секций позволяет не задумываться о потоковой безопасности в классах состояний.

Макросы утверждений (`ASSERT` и `VERIFY`) [8] используются в программе для проверки ее целостности в режиме отладки.

3.5. Пример реализации класса состояния автомата *ThreadFactory*

Приведем в качестве примера реализацию одного из классов состояний `Idle`. Этот класс наследуется от класса `StateBase<IThreadFactoryAI>`. При инициализации в него передается ссылка на контекст и имя состояния — «`Idle`». При этом вместо имени типа `StateBase<IThreadFactoryAI>` используется имя `SB`, которое являющееся его синонимом.

Заголовочный файл для класса `Idle`:

```
#pragma once

#include "StateMachine\StateBase.h"
#include "StateMachine\Event.h"
#include ".\IThreadFactoryAI.h"
#include ".\IThreadRequest.h"

#include <afxmt.h>

namespace ThreadManagement
{

class ConnectionThread;

class Idle : public StateMachine::StateBase<IThreadFactoryAI>
{
public:
  // Events
  static StateMachine::Event CREATED;
  static StateMachine::Event STOP_SENDED;

  Idle(AB & automaton, IThreadRequest & request, ConnectionThread * &
         threadPtr, CEvent & requestEvent);

  // IThreadFactory
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
  void Started(ConnectionThread * threadPtr);
  void Stopped(ConnectionThread * threadPtr);

private:
  IThreadRequest & request_;
  ConnectionThread * & threadPtr_;
  CEvent & requestEvent_;
};

}
```

Исполняемый файл для класса `Idle`:

```
#include "stdafx.h"

#include "ConnectionThread.h"
#include "Logger.h"

#include ".\Idle.h"
#include ".\exceptions.h"

#include "StateMachine\AutomatonBase.h"

namespace ThreadManagement
{

Idle::Idle(AB & automaton, IThreadRequest & request, ConnectionThread * &
         threadPtr, CEvent & requestEvent)
: SB(automaton, "Idle"),
  request_(request),
  threadPtr_(threadPtr),
```

```
  requestEvent_(requestEvent)
{
}

// IThreadFactory
void Idle::Request()
{
  ASSERT(threadPtr_ == NULL);
  Logger::Instance().Log("state Idle", "Trying to create new thread");
  threadPtr_ = new ConnectionThread(automaton_);
  VERIFY(threadPtr_->CreateThread());
  automaton_.CastEvent(CREATED);
  VERIFY(requestEvent_.ResetEvent());
}

void Idle::Cancel()
{
  throw CancelFailedException();
}

void Idle::Destroy(ConnectionThread * threadPtr)
{
  ASSERT(threadPtr_ == NULL);
  Logger::Instance().Log("state Idle", "Trying to destroy thread");
  threadPtr_ = threadPtr;
  threadPtr->Stop();
  automaton_.CastEvent(STOP_SENDED);
  VERIFY(requestEvent_.ResetEvent());
}

// IThreadNotify
void Idle::Started(ConnectionThread * threadPtr)
{
  ASSERT(FALSE);
}

void Idle::Stopped(ConnectionThread * threadPtr)
{
  ASSERT(FALSE);
}

StateMachine::Event Idle::CREATED("CREATED");
StateMachine::Event Idle::STOP_SENDED("STOP_SENDED");

}
```

Отметим, что для использования объекта синхронизации `CEvent` подключается заголовочный файл `<afxmt.h>`.

Завершая описание реализации класса `ThreadFactory` отметим, что предложенный подход позволяет реализовать этот класс без условных операторов, что является признаком хорошего проектирования [5].

Отметим, что классы состояний не «знают» друг о друге. Это дает возможность, в общем случае, производить наследование от этих классов для последующего использования в других автоматах.

4. Приложение, визуализирующее работу класса *ThreadFactory*

Для визуализации и тестирования класса `ThreadFactory` разработано приложение *Thread Manager*, которое позволяет производить запросы к классу `ThreadFactory`. Главное окно этого приложения приведено на рис. 6.

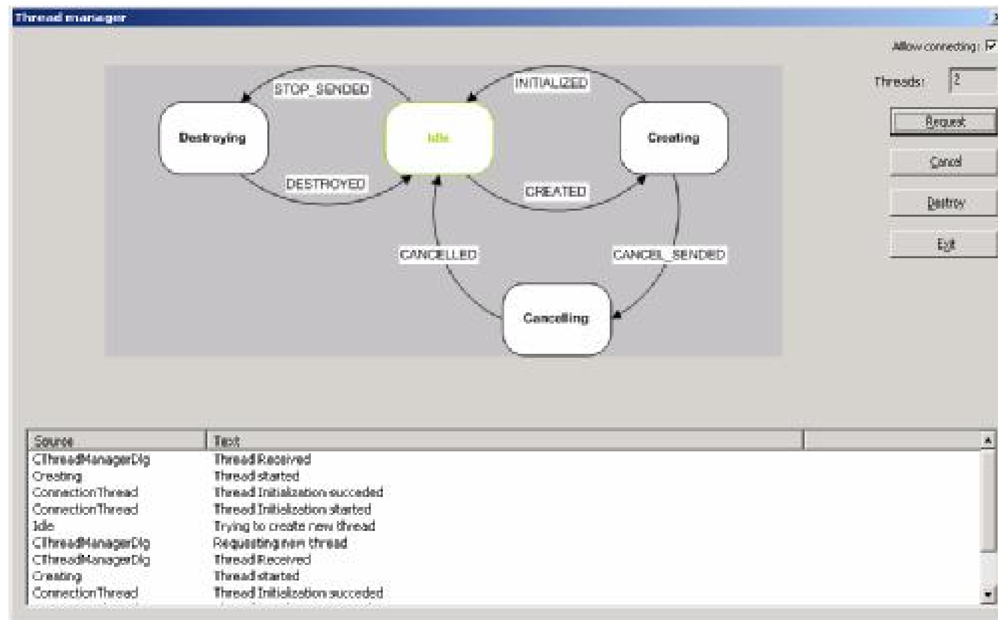

Рис. 7. Главное окно приложения *Thread Manager*

При нажатии на кнопки *Request*, *Cancel* и *Destroy* вызываются соответствующие методы класса `ThreadFactory`. Флажок *Allow connecting* эмулирует наличие/отсутствие сетевого соединения с серверной частью базы данных. В центральной части окна отображается граф переходов, визуализирующий текущее состояние автомата. В нижней части главного окна приложения отображается протокол работы программы. Программа завершается при нажатии на кнопку *Exit*.

5. Сравнение реализации класса *ThreadFactory* на основе паттерна *State Machine* и традиционного подхода

В разд. 3 описано проектирование и реализация класса `ThreadFactory` на основе паттерна *State Machine*.

При этом отметим, что обычно программисты выбирают другой способ реализации подобного класса — с использованием флагов. В этом случае код класса `ThreadFactory` выглядит иначе.

Заголовочный файл для класса `ThreadFactory`:

```
#pragma once

#include <set>
#include <afxmt.h>

#include "StateMachine\AutomatonBase.h"
#include ".\IThreadFactory.h"
#include ".\IThreadNotify.h"

namespace ThreadManagement
{

class ConnectionThread;
struct IThreadRequest;
class ThreadFactory : public IThreadFactory, public IThreadNotify
{
public:
  ThreadFactory(IThreadRequest & request);
  ~ThreadFactory(void);

  // IThreadFactory
  void Request();
  void Cancel();
```

```
    void Destroy(ConnectionThread * threadPtr);

    // IThreadNotify
    virtual void Started(ConnectionThread * threadPtr);
    virtual void Stopped(ConnectionThread * threadPtr);

    // Wait for current action finish
    void Wait();

  private:
    CCriticalSection lock_;
    CEvent requestEvent_;
    IThreadRequest & request_;
    ConnectionThread * threadPtr_;

    bool cancel_;
    bool destroy_;
  };

  }
```

Исполняемый файл для класса `ThreadFactory`:

```
  #include "stdafx.h"

  #include "boost\bind.hpp"
  #include "StateMachine\Event.h"
  #include "Logger.h"
  #include ".\FactoryOld.h"
  #include ".\ConnectionThread.h"
  #include ".\IThreadRequest.h"
  #include ".\Exceptions.h"

  namespace ThreadManagement
  {

  ThreadFactory::ThreadFactory(IThreadRequest & request)
  : requestEvent_(TRUE, TRUE),
    request_(request),
    threadPtr_(NULL),
    cancel_(false),
    destroy_(false)
  {
  }

  ThreadFactory::~ThreadFactory(void)
  {
    ASSERT(threadPtr_ == NULL);
  }

  void ThreadFactory::Request()
  {
    CSingleLock locker(&lock_, TRUE);
    if (threadPtr_ != NULL) throw BusyException();

    Logger::Instance().Log("ThreadFactory", "Trying to create new thread");
    threadPtr_ = new ConnectionThread(*this);
    VERIFY(threadPtr_->CreateThread());
    VERIFY(requestEvent_.ResetEvent());
  }

  void ThreadFactory::Cancel()
  {
    CSingleLock locker(&lock_, TRUE);
    if (cancel_ || destroy_ || threadPtr_ == NULL) throw CancelFailedException();

    cancel_ = true;
    threadPtr_->Stop();
  }
```

```
void ThreadFactory::Destroy(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  if (threadPtr_ != NULL) throw BusyException();
  threadPtr_ = threadPtr;
  destroy_ = true;
  threadPtr_->Stop();
}

void ThreadFactory::Started(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  if (cancel_) threadPtr_->Stop();
  else
  {
    ConnectionThread * tPtr = threadPtr_;
    threadPtr_ = NULL;
    request_.Created(tPtr);
    requestEvent_.SetEvent();
  }
}

void ThreadFactory::Stopped(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = NULL;
  if (cancel_)
  {
    Logger::Instance().Log("ThreadFactory", "Thread stopped. Mode cancelling");
    cancel_ = false;
    request_.Canceled();
  }
  else if (destroy_)
  {
    Logger::Instance().Log("ThreadFactory", "Thread stopped. Mode destroying");

    destroy_ = false;
    request_.Deleted();
  }
  VERIFY(requestEvent_.SetEvent());
}

void ThreadFactory::Wait()
{
  ::WaitForSingleObject(requestEvent_, INFINITE);
}

}
```

В приведенном коде присутствуют условные операторы, анализирующие значения флагов (логические переменные `cancel_` и `destroy_`). Такой код в работе [5] назван плохим, поскольку в разных методах класса эти флаги по-разному влияют на его поведение.

6. Сравнение реализации класса *ThreadFactory* на основе паттерна *State Machine* и *SWITCH*-технологии

В этом разделе опишем еще один способ проектирования класса `ThreadFactory` — на основе *SWITCH*-технологии. При этом отметим, что используется, так называемое, *оборачивание* автомата классом, примененное, например, в работе [10].

Схема связей автомата `ThreadFactory`, определяющая его интерфейс, приведена на рис. 8.

Рис. 8. Схема связей автомата **ThreadFactory**

Отметим, что для каждого метода интерфейса `IThreadFactoryAI` формируется определенное входное воздействие — событие. Например, методу `Request` соответствует событие `e0`.

Выходные воздействия реализованы с помощью закрытых методов класса `ThreadFactory`. Например, выходное воздействие `z0` создает поток соединения.

На рис. 9 приведен граф переходов структурного автомата, построенного на основе *SWITCH*-технологии.

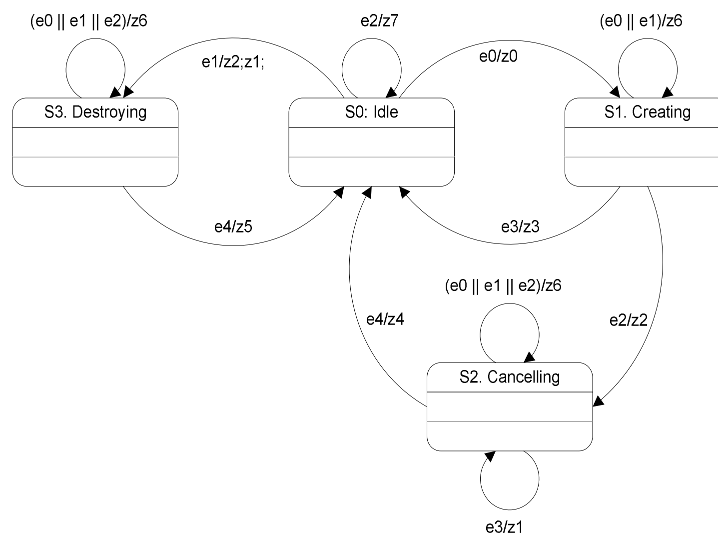

Рис. 9. Граф переходов структурного автомата

Приведем код класса `ThreadFactory`, реализующий описанный граф переходов.

Заголовочный файл для класса `ThreadFactory`:

```
#pragma once

#include <map>
#include <afxmt.h>

#include "boost\signal.hpp"
#include ".\IThreadFactory.h"
#include ".\IThreadNotify.h"

namespace ThreadManagement
{
```

```
class ConnectionThread;
struct IThreadRequest;
class ThreadFactory : public IThreadFactory, public IThreadNotify
{
public:
  ThreadFactory(IThreadRequest & request);
  ~ThreadFactory(void);

  // Add log listener
  template <class Func>
  void AddLogFunction(Func const & t)
  {
    log_.connect(t);
  }

  // Add State Change listener
  template <class Func>
  void AddStateChangeListener(Func const & t)
  {
    stateChange_.connect(t);
  }

  // IThreadFactory
  void Request();
  void Cancel();
  void Destroy(ConnectionThread * threadPtr);

  // IThreadNotify
  virtual void Started(ConnectionThread * threadPtr);
  virtual void Stopped(ConnectionThread * threadPtr);

  // Wait for current action finish
  void Wait();

private:
  CCriticalSection lock_;
  CEvent requestEvent_;
  IThreadRequest & request_;
  ConnectionThread * threadPtr_;

  enum States
  {
    s0, // Idle
    s1, // Creating
    s2, // Cancelling
    s3  // Destroying
  };

  States y;

  enum Events
  {
    e0, // Request
    e1, // Destroy
    e2, // Cancel
    e3, // Started
    e4  // Stopped
  };

  void A(Events e);

  void z0(); // Create new thread
  void z1(); // Destroy thread
  void z2(); // Reset event

  void z3(); // Notify created
  void z4(); // Notify cancelled
  void z5(); // Notify destroyed
```

```
  void z6(); // Throwing BusyException
  void z7(); // Throwing CancelFailedException

  std::map<States, std::string> stateNames_;
  boost::signal<void (char const *)> log_;
  boost::signal<void (char const *)> stateChange_;

  void Log(char const * message);
};

}
```

Исполняемый файл для класса `ThreadFactory`:

```
#include "StdAfx.h"

#include "Logger.h"
#include ".\FactorySwitch.h"
#include ".\ConnectionThread.h"
#include ".\IThreadRequest.h"
#include ".\Exceptions.h"

namespace ThreadManagement
{

ThreadFactory::ThreadFactory(IThreadRequest & request)
: requestEvent_(TRUE, TRUE),
  request_(request),
  threadPtr_(NULL),
  y(s0)
{
  stateNames_.insert(std::make_pair(s0, "Idle"));
  stateNames_.insert(std::make_pair(s1, "Creating"));
  stateNames_.insert(std::make_pair(s2, "Canceling"));
  stateNames_.insert(std::make_pair(s3, "Destroying"));
}

ThreadFactory::~ThreadFactory(void)
{
  ASSERT(threadPtr_ == NULL);
}

void ThreadFactory::Request()
{
  CSingleLock locker(&lock_, TRUE);
  A(e0);
}

void ThreadFactory::Destroy(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = threadPtr;
  A(e1);
}

void ThreadFactory::Cancel()
{
  CSingleLock locker(&lock_, TRUE);
  A(e2);
}

void ThreadFactory::Started(ConnectionThread * threadPtr)
{
  CSingleLock locker(&lock_, TRUE);
  threadPtr_ = threadPtr;
  A(e3);
}

void ThreadFactory::Stopped(ConnectionThread * threadPtr)
{
```

```
    CSingleLock locker(&lock_, TRUE);
    threadPtr_ = threadPtr;
    A(e4);
}

void ThreadFactory::A(Events e)
{
    States yOld = y;
    switch (y)
    {
    case s0: // Idle
        {
            if (e == e0) {z0(); y = s1;}
            else if (e == e1) {z2(); z1(); y = s3;}
            else if (e == e2) z7();
            else ASSERT(FALSE);
        }
        break;
    case s1: // Requesting
        {
            if (e == e0 || e == e1) z6();
            else if (e == e2) {z2(); y = s2;}
            else if (e == e3) {z3(); y = s0;}
            else ASSERT(FALSE);
        }
        break;
    case s2: // Cancelling
        {
            if (e == e0 || e == e1 || e == e2) z6();
            else if (e == e3) z1();
            else if (e == e4) {z4(); y = s0;}
        }
        break;
    case s3: // Destroying
        {
            if (e == e0 || e == e1 || e == e2) z6();
            else if (e == e4) {z5(); y = s0;}
            else ASSERT(FALSE);
        }
        break;
    default:
        ASSERT(FALSE);
    };
    if (yOld != y) stateChange_(stateNames_[y].c_str());
}

void ThreadFactory::z0()
{
    ASSERT(threadPtr_ == NULL);
    Log("Trying to create new thread");
    threadPtr_ = new ConnectionThread(*this);
    VERIFY(threadPtr_->CreateThread());
    VERIFY(requestEvent_.ResetEvent());
}

void ThreadFactory::z1()
{
    Log("Trying to destroy thread");
    threadPtr_->Stop();
}

void ThreadFactory::z2()
{
    VERIFY(requestEvent_.ResetEvent());
}

void ThreadFactory::z3()
{
    Log("Thread started");
```

```
  ConnectionThread * tPtr = threadPtr_;
  threadPtr_ = NULL;
  request_.Created(tPtr);
  requestEvent_.SetEvent();
}

void ThreadFactory::z4()
{
  Log("Thread stopped");
  threadPtr_ = NULL;
  request_.Canceled();
  requestEvent_.SetEvent();
}

void ThreadFactory::z5()
{
  Log("Thread stopped");
  request_.Deleted();
  threadPtr_ = NULL;
  VERIFY(requestEvent_.SetEvent());
}

void ThreadFactory::z6()
{
  throw BusyException();
}

void ThreadFactory::z7()
{
  throw CancelFailedException();
}

void ThreadFactory::Log(char const * message)
{
  Logger::Instance().Log(stateNames_[y].c_str(), message);
}

void ThreadFactory::Wait()
{
  ::WaitForSingleObject(requestEvent_, INFINITE);
}

}
```

Обратим внимание, что при реализации класса `ThreadFactory` не используются глобальные переменные. В качестве возможных значений состояния используются не их номера, как в работе [10], а значения перечисляемого типа `States`, что повышает типобезопасность.

Сравним реализации на основе *SWITCH*-технологии и паттерна *State Machine*.

Достоинства реализации на основе *SWITCH*-технологии:

- в графах переходов, используемых в этой технологии, наряду с состояниями и событиями применяются также входные переменные и выходные воздействия;
- код, реализующий граф переходов, строится формально и изоморфно;
- автомат реализуется в одном классе;
- код более компактен.

В случае реализации автомата с большим количеством состояний для паттерна *State Machine* необходимо создавать по классу на каждое состояние, что может быть громоздко и трудоемко.

Недостатками реализации на основе *SWITCH*-технологии:

- монолитность — в отличие от реализации на основе паттерна *State Machine* невозможно повторно использовать составные части кода класса `ThreadFactory`;
- при необходимости добавления входных и (или) выходных воздействий могут возникать ситуации, при которых компилятор не сможет обнаружить некоторые семантические ошибки, такие как, например, несоответствие метода интерфейса класса с вызовом автомата с соответствующим событием.

Выводы

Проектирование и реализация управления потоками на основе паттерна *State Machine* были выполнены в процессе работы над проектом *Navi Harbour* в компании *Транзас*. Применение паттерна *State Machine* для разработки класса `ThreadFactory` позволило обеспечить хороший [5] дизайн программы:

- исключить условные операторы из текста программы;
- разработать классы состояний независимо друг от друга;
- обеспечить при необходимости возможность наследования от классов состояний;
- обеспечить читабельность кода, сохранив преимущества автоатного проектирования.

Приложение *Thread Manager*, предназначенное для визуализации и тестирования класса `ThreadFactory`, также разработано на языке программирования *C++* с использованием библиотек *Boost* и *MFC* [11].

Исходные и бинарные коды построенных программ доступны по адресу http://is.ifmo.ru, раздел «Статьи».

Литература

1. **Gamma E., Helm R., Johnson R., Vlissides J.** Design Patterns. MA: Addison-Wesley Professional, 2001. — 395 p. (**Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. — 368 с).
2. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns (http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf)
3. **Шамгунов Н. Н., Корнеев Г А. Шалыто А. А.** Паттерн *State Machine* для объектно-ориентированного проектирования автоматов // Информационно-управляющие системы. 2004. № 5, с.
4. **Transas** — a world-leading developer and supplier of a wide range of software and integrated solutions for the transportation industry (http://www.transas.com).
5. **Fawler M.** Refactoring. Improving the Design of Existing Code. MA: Addison-Wesley. — 1999. — 431 p. (**Фаулер М.** Рефакторинг. Улучшение существующего кода. — М.: Символ-плюс, 2003. — 432 с).
6. **Shore-base systems department** (http://www.transas.com/vts/index.asp).
7. **Vessel Traffic Services/ Navi-Harbour** (http://www.transas.com/vts/navi_harbour/index.asp).
8. **Straustrup B.** The C++ Programming Language. MA: Addison-Wesley, 2000, 957p. (**Страуструп Б.** Язык программирования С++. СПб.: Бином, 2001. — 1099 с).
9. **Boost** (http://www.boost.org)
10. **Туккель Н.И., Шалыто А.А.** Система управления танком для игры "Robocode". Вариант 1. Объектно-ориентированное программирование с явным выделением состояний. (http://is.ifmo.ru/projects/tanks/)
11. **Microsoft Foundation Classes** (http://microsoft.com)