УДК 681.3

РЕАЛИЗАЦИЯ БИБЛИОТЕКИ ИМИТАЦИОННЫХ МОДЕЛЕЙ КАК НАБОРА ОБОБЩЕННЫХ КОМПОНЕНТ

А. В. Колотаев,
ведущий программист
ОАО «Транзас технологии»

В статье рассматривается применимость методов обобщенного программирования для реализации библиотеки имитационных моделей на языке C++. Описываемый подход представляется перспективным для разработки моделей больших и сложных систем, поскольку позволяет сочетать эффективность имитационных программ с широкими возможностями повторного использования модулей.

Applicability of generic programming techniques to develop simulation models C++ library is discussed. The approach described seems to be valuable for large and complex systems development since it allows to create highly reusable modules without sacrifying their efficiency.

Введение

Представление имитационной модели в виде набора взаимодействующих компонент является общепринятой практикой и имеет большую историю (например, один из первых языков объектно-ориентированного программирования Simula-67 предоставлял средства для модульной декомпозиции моделей еще в конце 60-х годов прошлого века). Преимущества модульного подхода хорошо известны. Во-первых, это возможность отразить в коде имитационной программы структуру моделируемой системы, что, несомненно, способствует пониманию программы и облегчает таким образом ее сопровождение. Во-вторых, такой подход открывает возможности для использования одного модуля в качестве составного элемента для построения разных имитационных моделей, т. е. для повторного использования модулей; это снижает затраты на построение и отладку новых имитационных моделей. Чем сложнее конструируемая модель, тем ярче проявляются преимущества модульного подхода.

Одним из наиболее распространенных языков для создания имитационных программ сложных и больших систем (например, телекоммуникационных сетей – ТКС) является язык C++, что обусловлено в первую очередь его гибкостью и эффективностью.

Гибкость заключается в том, что, записывая имитационную программу на языке C++, программист имеет в своем распоряжении всю мощь универсального языка программирования с его развитыми механизмами построения абстракций. Это оказывается важным при работе со сложными структурами данных и нетривиальными алгоритмами, которые встречаются при моделировании таких сложных систем, как ТКС.

Под эффективностью подразумевается возможность написания на языке C++ программ, которые обладают выразительностью исходного кода (т. е. позволяют записывать решение задачи на высоком уровне абстракции) и в то же время не уступают в быстродействии аналогичным программам, написанным на языках более низкого уровня (например, C). Иными словами, язык C++ позволяет свести к минимуму расходы, связанные с введением дополнительных абстракций и записью программы на более высоком уровне (abstraction penalty). Замедление программы в несколько раз допустимо, если время ее прогона измеряется секундами. В таких случаях производительностью можно пожертвовать в пользу, например, удобства разработки модели. Однако по мере усложенения модели все более недопустимым становится наличие abstraction penalty. Время имитации больших ТКС измеряется часами, поэтому быстродействие имитационной программы играет важную роль.

Язык C++ в силу своей универсальности не предоставляет, в отличие от языков ИМ или пакетов

■ *Таблица. Библиотеки обобщенного программирования на языке C++*

Предметная область	Библиотеки
Линейные последовательности (динамический массив, список, стек, очередь, куча, дека, сбалансированное дерево) и алгоритмы над ними	STL (Standart Template Library) [3]
Алгоритмы и структуры данных вычислительной геометрии	CGAL (Computational Geometry Algorithm Library) [4]
Графы и алгоритмы на графах	BGL (Boost Graph Library) [5]
Матрицы и матричные вычисления	MTL (Matrix Template Library) [6], GMCL (Generative Matrix Computational Library) [7]
Многомерные массивы для численных методов	blitz++[8]
Итеративные численные методы	ITL++ (Iterative Template Library)[9]

ИМ, готовых средств для непосредственной записи имитационной программы. Поэтому при написании имитационной программы на C++ программисту необходимо либо самому реализовывать инфраструктуру ИМ, обеспечивающую взаимодействие компонент модели и проведение вычислительного эксперимента (например, календарь событий), а также многие вспомогательные компоненты, которые часто используются при построении моделей (например, генераторы случайных чисел с заданными законами распределения), либо прибегать к использованию этих компонент, уже кем-то реализованных и оформленных в виде библиотек.

Часто вместе с библиотекой имитационного моделирования поставляются программы, облегчающие проведение имитационных экспериментов, составляя вместе систему или пакет имитационного моделирования. Например, в состав пакета OMNeT++ [1] входят визуальный редактор моделей gned, среда проведения имитационных экспериментов tkenv, средства для анализа результатов экспериментов и т. п. Вместе с системой имитационного моделирования зачастую поставляется библиотека имитационных моделей. Например, библиотека ns-2 [2] поставляется вместе с большим числом моделей каналов связи, узлов ТКС, работающих согласно различным протоколам.

Разработчики библиотек помимо выполнения главной задачи – предоставления некоторой полезной функциональности – стремятся сделать компоненты библиотеки пригодными для использования при решении как можно большего спектра задач, расширяя таким образом область применимости библиотеки и, следовательно, ее полезность. С пригодностью компоненты к ее повторному использованию связана возможность для пользователя настраивать ее поведение для своих нужд, а также способность быть использованной в различных окружениях. Основной прием достижения этих качеств – абстрагирование, которое в упрощенном виде можно описать как вынесение из компоненты всего, что не относится к ее сути, в виде параметров компоненты. Связывая с параметрами компоненты разные значения, можно менять ее поведение. Основной метод подобной параметризации компонент в ООП – использование механизма динамического полиморфизма, поддержка которого во многих ОО языках программирования реализована через механизм виртуальных функций.

К сожалению, подобную параметризацию средствами ООП можно проводить только до определенного предела, когда накладные расходы, связанные с использованием виртуальных функций, станут неприемлемо большими. Кроме урона быстродействию выполняемой программе использование динамического полиморфизма обладает рядом отрицательных эффектов, среди которых можно указать следующее:

1) уменьшение возможностей для статического контроля типов, что отодвигает момент диагностики неправильного использования компоненты на более поздний период: с момента компиляции на момент выполнения программы;

2) необходимость наследования классов, которые предполагается использовать как параметры компоненты, от определенных базовых классов, что снижает адаптируемость компоненты.

В следующем разделе будут более детально обсуждаться препятствия на пути использования методов ООП для повышения уровня повторного использования компонент библиотеки.

Указанные проблемы успешно решаются средствами обобщенного программирования, которое поддержано в языке C++ мощным механизмом шаблонов. Библиотеки, выполненные в парадиг-

ме обобщенного программирования, сочетают широкие возможности настройки компонент под нужды пользователя, удобство их использования с эффективностью результирующего кода.

В таблице перечислен ряд предметных областей, для которых созданы и применяются библиотеки обобщенного программирования на C++.

За последнее десятилетие обобщенное программирование оформилось в поддисциплину информатики, «которая занимается поиском абстрактных представлений эффективных алгоритмов, структур данных и других понятий программного обеспечения вместе с их систематической организацией. Целью обобщенного программирования является выражение алгоритмов и структур данных в форме, которая обеспечивает легкость их адаптации, возможность взаимодействия (interoperability) между ними, а также позволяет прямое их использование при построении программного обеспечения» [10].

В обобщенном программировании конфигурирование компонент (т. е. связывание с параметрами компонент определенных значений) происходит в момент компиляции программы, а не ее выполнения, как в объектно-ориентированном программировании. При этом используются языковые механизмы (шаблоны классов и функций), имеющие другие характеристики для построения адаптируемых компонент, нежели механизмы ООП (виртуальные функции). Это определяет необходимость при проектировании в обобщенном стиле следования принципам, в значительной степени отличающимся от принципов объектно-ориентированного проектирования[1].

В данной работе рассматривается применение методов обобщенного программирования при разработке библиотеки имитационных моделей с тем, чтобы придать ей следующие качества: модульность, эффективность, ранняя диагностика неправильного использования модулей, расширяемость, адаптируемость к окружению, выразительность исходного кода. Рассмотрим наиболее важные понятия компонентного подхода к созданию имитационных моделей на примере библиотеки OMNeT++ [1].

Модель в OMNeT++ состоит из иерархически вложенных блоков-модулей, вложенность блоков при этом не ограничена, что позволяет пользователю создавать модели сложных иерархических систем.

Каждый модуль может рассматриваться как относительно независимая сущность, которая взаимодействует с другими модулями, как правило, только при помощи механизма посылки сообщений. Посылка сообщения модулем заключается в помещении объекта, представляющего сообщение, в один из выходных портов (output gate), ассоциированных с модулем. Они абстрагируют модуль от приемника сообщения, позволяя таким образом установить сообщение между любыми двумя модулями, имеющими порты. Серьезным недостатком такого подхода является невозможность выразить синтаксически (и проверить на этапе компиляции), что модуль может принимать сообщения только определенного типа.

С каждым модулем могут быть ассоциированы параметры, при помощи которых можно настраивать его поведение. Параметр может иметь тип только из предопределенного набора типов, что снижает выразительность описания модели. Параметры модуля хранятся как набор пар «ключ–значение», где ключом является строка, а значением – variant-подобный тип. Такой подход не предоставляет возможности компилятору проверить, что среди параметров модуля есть определенный параметр и он обладает нужным типом. Кроме этого доступ к параметрам сопряжен с высокими накладными расходами.

В OMNeT++ модули бывают двух типов: простые и составные. Простые модули содержат алгоритм поведения некоторого элемента модели. Составные модули агрегируют в себе другие модули (неважно, простые или составные), соединяют их входные и выходные порты, а также связывают с их параметрами определенные значения. Существенным моментом здесь является то, что простые модули берут на себя всю поведенческую составляющую модели (все поведение модели есть композиция поведения простых модулей), а составные модули – всю организационную составляющую модели (именно они задают, как модули соединяются и какие значения будут присвоены их параметрам). Моделью является составной модуль верхнего уровня, т. е. такой, который не принадлежит никакому другому модулю.

Обобщенное программирование является подходящей технологией для реализации простых модулей, поскольку позволяет создавать сильно параметризованные, обобщенные компоненты, которые не уступают в эффективности непараметризованным модулям.

К сожалению, методов обобщенного программирования недостаточно для разработки всей библиотеки имитационных моделей – кроме высокопараметризованных простых модулей необходимо предоставить средства, которые помогают пользователю отобразить замысел конструируемой модели в набор классов и их параметров, который реализует задуманную модель. Эта проблема решается средствами производящего программирования (generative programming) [14] и выходит за рамки данной статьи.

В OMNeT++ для описания составных модулей (и интерфейса простых модулей для того, чтобы они могли быть частью составного модуля) используется специальный язык NED (Network

[1] Для знакомства с ними можно порекомендовать работы [11–13].

Description Language). Компилятор с языка NED преобразует описания модулей в исходный код на C++, который после обработки компилятором C++ компонуется вместе с объектными файлами, содержащими алгоритмы модели, формируя таким образом имитационную программу. Именно здесь становится важной разница между простыми и составными модулями: простые модули лучше всего записывать на алгоритмическом языке программирования, а их конфигурирование лучше поручить специальному генератору.

Генератор составных компонент может быть реализован и в рамках языка C++ при помощи шаблонного метапрограммирования [15]; перспективность этого подхода в настоящее время исследуется автором статьи.

В данной работе представлен подход к созданию модулей имитационных моделей как обобщенных компонент, оценивается адаптируемость реализованных таким образом простых модулей, использованные техники реализации сравниваются с альтернативами.

Автором статьи разработана и развивается библиотека имитационного моделирования tksym, которая включает библиотеку имитационных моделей узлов и каналов связи ТКС, используемую для сравнительного анализа различных алгоритмов маршрутизации в ТКС. Простые модули реализованы в ней при помощи описываемых ниже техник обобщенного программирования. В настоящее время для составления из простых модулей имитационной модели программист должен написать код, явно их конфигурирующий, что весьма неудобно; поэтому очень актуальна задача автоматической генерации модели по высокоуровневой спецификации.

Сравнение механизмов статического и динамического полиморфизма в языке C++

Основной метод достижения гибкости программных продуктов – разбиение на модули, при котором стремятся, с одной стороны, ослабить зависимости между модулями (слабое зацепление между модулями), а с другой – составлять модули из сильно связанных элементов (сильная связность в модулях).

В языке C++ для уменьшения зависимости между модулями можно использовать механизм виртуальных функций (который обеспечивает полиморфизм времени выполнения, или динамический полиморфизм) или механизм шаблонов языка C++ (который обеспечивает полиморфизм времени компиляции, или статический полиморфизм). Первый подход лежит в основе объектно-ориентированного программирования, второй – в основе обобщенного программирования. Каждый из них имеет свои преимущества и недостатки, которые в зависимости от специфики разрабатываемого программного обеспечения проявляются в разной степени.

Придание гибкости программе при помощи механизма виртуальных функций часто имеет положительный эффект, что широко проиллюстрировано в литературе по объектно-ориентированному программированию. Хорошее изложение приемов решения типичных задач проектирования (паттернов проектирования) в рамках парадигмы объектно-ориентированного программирования читатель может найти в классическом труде [16].

К сожалению, есть ситуации, когда от применения механизма виртуальных функций приходится отказаться по ряду причин.

Основная причина – ухудшение производительности программы по сравнению с аналогичной по функциональности программой, но написанной без разбиения на модули и уменьшения зависимостей между модулями. Этот эффект известен как abstraction penalty – накладные расходы, связанные с введением абстракций.

Ухудшение производительности в первую очередь обусловлено неспособностью компилятора проводить оптимизации кода вокруг точки вызова виртуальной функции, поскольку он не может определить, какая именно функция будет вызвана. Кроме этого, на некоторых процессорных архитектурах косвенный вызов, к которому сводится вызов виртуальной функции, сбрасывает содержимое конвейера команд, что также наносит удар по производительности.

Урон производительности, причиняемый использованием механизма виртуальных функций, зависит от размера функций и частоты их вызова. Чем меньше время выполнения виртуальной функции и чем чаще она вызывается, тем больше урон. Например, если спроектировать библиотеку матричных вычислений так, что различные классы матриц должны переопределить виртуальную функцию доступа к элементу матрицы, объявленную в базовом для всех матриц классе абстрактной матрицы, то эффективность алгоритмов, записанных в терминах абстрактного базового класса всех матриц, может снизиться во много раз по сравнению с алгоритмами, работающими над конкретными типами матриц.

Если функция достаточно большая и не может быть встроена в точку вызова, отрицательный эффект от виртуального вызова достаточно мал, и с ним можно смириться в пользу большей гибкости программы и более быстрой ее компиляции.

При интенсивном использовании виртуальных функций иногда наблюдается тенденция к созданию большого числа маленьких объектов в свободной памяти. Если не применять оптимизированного на работу с маленькими объектами менеджера памяти, их размещение может потребовать излишнего расхода памяти и времени.

Использование абстрактных базовых классов уменьшает возможности статического контроля типов. Показательным примером может служить проектирование контейнера элементов, подразуме-

вающее при таком подходе определение абстрактного базового класса для всех классов, которые могут храниться в контейнере. Назовем этот класс `Object`, а контейнер объектов типа `Object` – `ObjectContainer`. Допустим, что пользователь желает хранить в этом контейнере только объекты определенного типа `X`. К сожалению, это намерение при использовании `ObjectContainer` никак не может быть выражено синтаксически в коде – компилятор не выдаст сообщения об ошибке при попытке вставить в `ObjectContainer` объект другого класса. Кроме этого, после извлечения элемента из контейнера необходимо делать понижающее приведение типа от `Object` к `X` и проверять его успешность, что отрицательно сказывается на выразительности кода. (В данном случае проблему можно решить введением обертки над `ObjectContainer`, которая будет гарантировать типобезопасность. Однако такое решение очень плохо масштабируется и поэтому непригодно в качестве универсального подхода).

Некоторые авторы [17] указывают еще ряд отрицательных эффектов, которые могут проявиться при попытках придания большей гибкости программной архитектуре при помощи механизма виртуальных функций, а именно: необходимость заблаговременного проектирования классов, учитывая возможные изменения в будущем (preplanning problem), введение избыточного числа дополнительных абстракций, наличие которых затрудняет понимание кода программы (confusion problem) и пр.

При использовании шаблонов языка C++ информация, которая помогает компилятору сгенерировать оптимизированный код, не теряется, и поэтому при использовании хороших оптимизирующих компиляторов получаемый код не уступает значительно по эффективности нешаблонному коду. Безопасность типов также сохраняется, поскольку реальный тип объектов известен в момент компиляции.

Слабыми сторонами шаблонного программирования на C++ считаются следующие.

1. Недостаточная спецификация синтаксических ограничений на параметры шаблона. Если при программировании с виртуальными функциями абстрактный базовый класс является тем местом, где сформулированы синтакисические требования класса к своему параметру (наличие методов с определенной сигнатурой), то аналогичного способа спецификации требований к параметрам шаблона в языке C++ нет – приходится использовать дополнительные техники вроде concept checking [18] для отражения в исходном коде требований, которым должны удовлетворять аргументы шаблона.

2. Неосторожное использование шаблонов может привести к значительному увеличению объема исполняемого кода – так называемому «разбуханию» кода. Существуют несложные приемы, позволяющие избежать подобного эффекта.

3. Шаблонная программа может компилироваться и компоноваться значительно дольше, чем аналогичная программа без использования шаблонов. Кроме этого, использование шаблонов уменьшает возможности использования раздельной компиляции.

Некоторые понятия обобщенного программирования

Важнейшим приемом обобщенного проектирования является выделение минимальных требований компоненты к своим параметрам и реализация компоненты с минимальными предположениями о своих параметрах. Чем более слабые ограничения накладываются на параметры компоненты, тем больше компонент могут стать ее аргументами. Требования усиливаются, если это позволяет сделать компоненту более эффективной. Разные алгоритмы, выполняющие одну задачу, могут сосуществовать, если они работают в различных предположениях о своих параметрах; при этом предоставляются средства автоматического выбора наиболее подходящего алгоритма под заданный набор параметров.

Наборы требований, предъявляемых обобщенной компонентой к своим параметрам, называют *концепциями*. Требования делятся на синтаксические и семантические.

Синтаксические требования могут быть проверены на этапе компиляции – при попытке использовать компонент с параметром, не удовлетворяющим его синтаксическим требованиям, будет выдана ошибка компиляции. В ООП синтаксическим требованием к параметру класса является свойство «быть наследником определенного базового класса». Это подразумевает, что в аргументе компоненты должны быть объявлены функции-члены, сигнатура которых в точности совпадает с сигнатурой функций-членов базового класса[1].

В обобщенном программировании синтаксическое требование выражается менее жестко: вместо требования совпадения сигнатуры функции-члена требуется, чтобы она имела параметры, к типам которых может быть приведен тип определенного выражения (т. е. требования приводимости).

Семантические требования не могут быть проверены на этапе компиляции (например, что некоторая последовательность отсортирована). Некоторые из них могут быть выражены в коде в виде утверждений о поведении параметра компонента, некоторые представляют собой утверждения о вычислительной сложности той или иной операции: например, на являющийся параметром шаблона контейнер может быть наложено требование выполнять вставку элемента в начало контейнера за время O(1) (этому требованию удовлетворя-

[1] Ослаблением этого правила в C++ являются ковариантные типы возвращаемых значений.

ют шаблоны классов `std::list`, `std::deque` и не удовлетворяют `std::vector`, `std::set`).

Тип, удовлетворяющий набору требований концепции, называется *моделью* этой концепции (*моделирует* концепцию). Концепция, которая расширяет набор требований другой концепции, называется ее *уточнением*.

Проектирование простого модуля в обобщенном стиле

Рассмотрим различные подходы к реализации часто встречаемой в имитационных программах модели устройства (которая является несколько упрощенным аналогом модуля `Server` системы ИМ Arena [19]), работающего по следующему алгоритму.

Когда в устройство поступает объект для обработки *e*, проверяется, не обрабатывает ли устройство другой объект. Если да, то пришедший объект *e* сохраняется в ассоциированной с устройством очереди. Если нет, вычисляется время *t*, которое займет обработка *e*, и устройство переходит в состояние обработки *e* на время *t*. После окончания обработки *e* посылается далее, устройство спрашивает у очереди очередной объект для обработки и, если таковой имеется, приступает к его обработке, если нет, то переходит в состояние ожидания.

В библиотеке tksym модель устройства реализована следующим образом:

```
template
<
   class Base,
   class World,
   class Queue,
   class ProcessingTime,
   class Sink,
   class Events
>
struct Server : Base, World, Queue, ProcessingTime, Sink, Events
{
   typedef typename Base::Entity   Entity;

   using Queue::queue;
   using Events::events;

   // "входной порт" модуля, через который
   // он получает сообщения от других модулей
   void process(Entity e)
   {
      if (being_sent_)          // если устройство занято,
           queue().push(e);     // сохраним e в очереди
      else                      // иначе
           startProcessing(e);// начнем его обработку
   }

   Entity const & beingSent() const{ return being_sent_; }

private:
   void startProcessing(Entity e)
   {
       // оценим время t, которое займет обработка e
       // и запланируем вызов метода "release" через t секунд
      World::schedule(ProcessingTime::get(e),
          boost::bind(&ResourceEx::release,this));

      // перейдем в состояние "занято"
      being_sent_ = e;

      // оповестим подписавшихся на прослушивание событий,
      // что обработка e началась
       events().OnStartProcessing(e);
   }

   void release()
   {
      // поместим обработанный объект в "выходной" порт
      Sink::process(being_sent_);

      // оповестим слушателей событий,
      //что обработка закончилась
      events().OnStopProcessing(being_sent_);

      // перейдем в состояние "ожидание"
      being_sent_ = Entity();

      // если в очереди есть еще объекты для обработки
      if (!queue().empty())
      {
           // выберем из них один
           // и приступим к его обработке
           startProcessing(queue().top());
           // удалим обрабатываемый объект из очереди
           queue().pop();
      }
   }

private:
   Entity being_sent_;
};
```

Рассмотрим характерные черты представленного дизайна класса `Server`.

Компонент наследуется от набора параметров шаблона (которые иногда называются стратегиями – policy [12]) и реализуется с использованием исключительно имен, зависимых от параметров шаблона. Это позволяет использовать класс `Server` в любом окружении – достаточно лишь связать его параметры с типами, которые удовлетворяют определенному набору требований.

Примером синтаксических требований может служить требование от параметра `Base` иметь внутренний тип `Entity`, который обладает конструктором по умолчанию, конструктором копирования и оператором приведения к типу, который может быть условием в инструкции `if`. Будем считать, что этот оператор приводит объект типа `Entity` к типу `bool`. Класс `Server` считает, что результат такого приведения равен `false` тогда и только тогда, когда `Entity` сконструирован по умолчанию (что может служить примером семантического требования). Сконструированный по умолчанию `Entity` служит для обозначения состояния устройства «свободен».

Если стратегия предоставляет несколько функций-членов, то они объединяются в один класс (будем называть такие стратегии составными). Это упрощает создание стратегий, вызовы к себе делегирующих другим классам. В нашем примере та-

кими стратегиями являются параметры Queue и Events. Если бы 4 функции-члена очереди не были агрегированы в объект, возвращаемый функцией queue(), их пришлось бы реализовывать к каждом классе, связываемом с параметром Queue.

Для доступа к методам обычных стратегий используется запись Стратегия::Метод(). Для доступа к методам составной стратегии Стратегия, метод доступа к ним стратегия() вносится в область имен модуля (при помощи using-объявления), после этого доступ к методу Стратегия::методА() осуществляется как стратегия().методА().

Альтернативный подход мог бы заключаться в доступе к методам стратегий через указатель this: this->МетодНекоторойСтратегии(). Однако при его использовании возможны неоднозначности поиска имен в базовых классах. Например, параметр Sink мог быть тоже унаследован от типа, являющегося параметром World. В этом случае два базовых класса содержали бы метод schedule, что привело бы к ошибке компиляции.

Среди параметров модуля выделяется класс Base, при помощи которого можно достичь так называемой управляемой виртуальности методов модуля [13]. Например, если бы в классе Base была объявлена виртуальная функция-член void process(Entity e), то и функция-член void Server::process(Entity e) тоже стала бы виртуальной.

Класс Server позволяет другим классам инспектировать свое состояние. Для этого он предоставляет метод, позволяющий прочитать его состояние (beingSent), а также события о изменении состояния. Слушатель этих событий задается параметром Events.

Каждый из параметров модуля может быть реализован разными способами.

Например, класс Server можно использовать с различными очередями. Они могут различаться дисциплиной обслуживания: простые очереди по принципу first come – first served (FCFSQueueHolder) или first come – last served (FCLSQueueHolder), очереди с приоритетом, для которых пользователь может указать различные предикаты, упорядочивающие объекты типа Entity (PriorityQueueHolder).

Реализация классов FCFSQueueHolder, FCLSQueueHolder и PriorityQueueHolder выглядит следующим образом:

```
template <class QueueStorage>
  struct QueueHolder
{
  typedef QueueStorage      queue_type;

  queue_type  const & queue() const { return queue_; }
  queue_type        & queue()       { return queue_; }
private:
  queue_type          queue_;
};

template <class Entity> struct FCFSQueueHolder
    : QueueHolder <std::queue<Entity> >
{};

template <class Enity>  struct FCLSQueueHolder
    : QueueHolder <std::stack<Entity> >
{};

template <class Entity, class Comparator>
    struct PriorityQueueHolder :
      QueueHolder <
            std::priority_queue<Entity, Comparator> >
{};
```

Владение очередью может быть монопольным (классы, использующие QueueHolder) или разделяемым (QueueInDerived или QueueIndirect):

```
// Предполагаем, что у Derived есть метод
// QueueType & getQueue() и QueueInDerived является
// предком (возможно, непрямым) Derived
template <class QueueType, class Derived>
  struct QueueInDerived
{
  typedef QueueType   queue_type;

  queue_type const & queue() const {
    return static_cast<Derived const *>
                              (this)->getQueue(); }
  queue_type & queue(){
    return static_cast<Derived *>
                              (this)->getQueue(); }
};

template <class QueueType, class Holder>
  struct QueueIndirect
{
  typedef QueueType   queue_type;

  void setHolder(Holder * h) { holder_ = h; }

  queue_type const & queue() const
    { return holder_->queue(); }
  queue_type         & queue()
    { return holder_->queue(); }

private:
  Holder       * holder_;
};
```

Очередь может быть бесконечной емкости (как, например, FCFSQueueHolder) или иметь фиксированную емкость. В случае, если происходит попытка вставки объекта в переполненную очередь, возможны разные стратегии: игнорирование объекта, передача очередью объекта обработчику объектов, которым не хватило в очереди места, генерация исключения – все они могут быть отражены соответствующими классами.

Требования класса Server к платформе ИМ минимальны: от шаблонного параметра World требуется только наличие функции-члена schedule, первым параметром которой является тип, представляющий время (к нему должен приводиться тип, возвращаемый функцией ProcessingTime::get()), а вторым параметром – тип, к которому можно привести функционал, имеющий ноль аргументов

(в текущей версии библиотеки tksym для этого используется тип boost::function<void ()>).

Вместо наследования модуля от стратегий можно было бы использовать агрегирование стратегий, что, однако, имеет недостатки по сравнению с наследованием по следующим показателям.

1. Экономия памяти. Каждый агрегируемый класс (даже, если он имеет нулевой размер) вносит ненулевой вклад в размер объекта класса Server. Наследование от класса нулевого размера не увеличивает размер наследника (если компилятор поддерживает оптимизацию пустого базового класса, Empty Base Class Optimization – EBCO).

2. Доступ к наследнику. Параметризация базового класса своим наследником является мощным и распространенным приемом, известным как модель странного рекуррентного шаблона (curiously recurring template pattern, CRTP)[13]. Базовый класс может получить доступ к методам своего наследника приводя указатель this к типу указателя на своего наследника. При этом не вносятся накладные расходы ни по времени выполнения, ни по используемой памяти. Примером использования такого приема может служить класс QueueInDerived. Агрегированный класс не имеет доступа к содержащему его классу («хозяину») (чтобы обойти это, мы могли бы запоминать в агрегируемом классе указатель на его «хозяина», что увеличило бы расход памяти, или при вызове методов агрегированного объекта передавать ему указатель на «хозяина», что не всегда удобно).

Альтернативой наследованию модуля от классов стратегий могло бы быть применение для класса модуля CRTP: класс Server параметризовался бы классом наследником. При помощи этого приема элегантно реализуется архитектура, основанная на примесях (mixin-based design).

Допустим, у нас есть набор классов-примесей M_1, ..., M_n, от которых наследуется некоторый класс C. Некоторые примеси зависят от других примесей. Проблема заключается в организации доступа из одной примеси к методам другой примеси. Если не использовать шаблонов, такой доступ можно реализовать, заведя виртуальный базовый класс с методами, к которым нужно организовать доступ, и наследования от него всех примесей, что является довольно хрупким решением. Другой способ – использование динамического приведения типа от указателя this примеси к нужной примеси. Недостатком такого решения является потеря статической типизации. Неудачное приведение типа может быть обнаружено только в момент выполнения программы. Параметризация классов-примесей классом C является элегантным решением этой проблемы: класс-примесь может получить доступ к методам других примесей, приведя указатель this к указателю на класс C (поскольку класс C унаследован от классов M_1, ..., M_n, то через класс C можно получить доступ к методам примесей).

В ранних версиях библиотеки tksym основным способом параметризации компонент модели было использование приема CRTP. Оказалось, что применительно к построению библиотеки моделей этот прием обладает следующими недостатками.

1. Методы базового класса могут вызывать методы производного класса, однако, к сожалению, при определении базового класса нельзя использовать типы, определяемые в производном классе. Для обхода этой проблемы типы, необходимые базовому классу, приходилось ему передавать как параметры шаблона, что в некоторых случаях загромождало исходный код.

2. Прием CRTP плохо масштабируется. По мере роста класса C увеличивается вероятность того, что найдутся две его примеси, которые содержат метод с одним и тем же именем. Если третья примесь обратится к этому методу, компилятор не сможет разрешить неоднозначность. Для разрешения этой неоднозначности может потребоваться введение прокси-классов, что при наличии в классах-примесей большого числа методов весьма неудобно.

Заключение

Архитектура рассмотренного выше класса Server является адаптацией для нужд библиотеки имитационных моделей архитектуры, основанной на стратегиях (policy-based design), описанной в работе [12].

Высокопараметризованный модуль Server, будучи инстанциирован с определенным набором параметров, не уступает по эффективности (в терминах времени выполнения и занимаемой памяти) непараметризованному монолитному аналогу при использовании современных компиляторов языка C++ (эксперимент проводился с использованием компиляторов Microsoft Visual C++ 7.1 и Intel C++ 8.0). К сожалению, за абстрактность кода и его эффективность приходится платить временем компиляции, однако есть надежда, что по мере развития технологии компиляции эта проблема станет менее острой, чем в настоящее время.

Модуль Server обладает высокой способностью к повторному использованию за счет высокой параметризации и того, что реализация класса Server не зависит от имен, независимых от параметров шаблона, т. е. в реализации класса отсутствует какая-либо информация, зависимая от контекста определения класса Server. Проблема несоответствия компоненты, предназначенной быть аргументом модуля, синтаксическим требованиям, предъявляемым модулем своему параметру, легко решается введением «обертки» над компонентом, адаптирующей его интерфейс.

Среди программистов бытует выражение, что «любую проблему программной архитектуры можно решить при помощи введения дополнительного слоя абстракции. Единственная проблема, не поддающаяся такому решению, – наличие слишком большого числа слоев абстракции». При

разработке библиотеки крайне важно сделать интерфейсы компонент и их требования к параметрам согласованными, чтобы свести к минимуму необходимость введения адаптеров. Это способствует лучшему изучению библиотеки, более глубокому ее пониманию. Несоответствие интерфейсов не является фатальным, однако избыточное введение адаптеров представляется попыткой исправить архитектурные ошибки.

При проектировании интерфейса модуля и спецификации требований на его параметры следует стремиться достичь баланса между простотой интерфейсов и тем, чтобы через них можно было получить достаточное для эффективой реализации модуля и работы с ним количество информации. Например, для того, чтобы пользователи класса `Server` могли эффективно инспектировать его состояние, класс `Server` позволяет подписаться на прием сообщений об изменении своего состояния.

Укажем направления будущих работ.

1. Разработка принципов создания составных модулей. Поскольку их основная задача – агрегировать в себе другие компоненты, соединить их друг с другом и связать с их параметрами определенные значения, представляется перспективной реализация составных модулей как классов метафункций (metafunction class) с использованием библиотеки Boost.Mpl [15].

2. Средства высокоуровневой спецификации (как в текстовом, так и в графическом виде) модели пользователем, которому достаточно функциональности существующих модулей. Средства трансляции высокоуровневого описания модели в набор классов языка C++. Средства, описывающие такому транслятору интерфейс модулей.

3. Реализация инструментов, облегчающих проведение вычислительных экспериментов. При решении этой задачи важно максимально использовать уже существующие аналогичные инструменты, программируя только переходники (bridge), обеспечивающие взаимодействие с ними модулей библиотеки.

Литература

1. **OMNeT++** Home page. http://www.omnetpp.org/
2. **The Network** Simulator - ns-2. Home page. http://www.isi.edu/nsnam/ns/
3. **Standard** Template Library Programmer's Guide http://www.sgi.com/tech/stl/
4. **Computational** Geometry Algorithm Library Home Page. http://www.cgal.org
5. **Boost** Graph Library Home Page http://www.boost.org/libs/graph/doc/table_of_contents.html
6. **Matrix** Template Library Home Page http://www.osl.iu.edu/research/mtl/
7. **Generative** Matrix Computational Library Home Page http://www-ia.tu-ilmenau.de/~czarn/gmcl/
8. **Blitz++** Home Page http://www.oonumerics.org/blitz/
9. **Iterative** Template Library Home Page http://www.osl.iu.edu/research/itl/
10. http://www.cs.rpi.edu/~musser/gp/dagstuhl/gpdag_2.html
11. **Austern M.** Generic Programming and the STL: Using and Extending the C++ Standard Template Library. – Addison-Wesley Professional; 1998. 576 p.
12. **Alexandrescu A.** Modern C++ Design: Generic Programming and Design Patterns Applied. – Addison-Wesley Professional, 2001. 352 p.
13. **Vandervoorde,D., Josuttis N.** C++ Templates: The Complete Guide. – Addison-Wesley Professional, 2002. 552 p.
14. **Czarnecki K., Eisenecker U.** Generative Programming: Methods, Techniques, and Applications. – Addison-Wesley, 1999. 864 p.
15. **Boost** Metaprogramming Library Home Page. http://www.boost.org/libs/mpl/doc/index.html
16. **Gamma E., Helm R., Johnson R., Vlissides J.** Design Pattern. – Addison-Wesley Professional; 1995. 416 p.
17. **Subject-oriented** programming Home Page. http://www.research.ibm.com/sop/
18. **The Boost** Concept Check Library. http://www.boost.org/libs/concept_check/concept_check.htm
19. http://www.arenasimulation.com/