Тимкин В. Н.

Имитационное моделирование на языке Delphi

# Введение

Имитационное моделирование функционирования процессов и систем может выполняться с использованием различных подходов. Рассмотрим некоторые из них.

Программная модель процесса или системы представлена рядом параметров, изменение которых во времени характеризует состояние моделируемого объекта. *Дискретное* моделирование предполагает, что эти параметры меняются в определенные моменты времени, причем изменение происходит мгновенно. Между этими изменениями все параметры системы сохраняют постоянные значения. При *непрерывном* моделировании предполагается, что параметры системы меняются непрерывно. Фактически же моделирующая программа вычисляет их значения в определенные моменты времени, пользуясь для этого уравнениями, позволяющими вычислять состояние моделируемой системы в любой момент времени, или алгоритмами численного интегрирования.

*Процессно-ориентированное* моделирование подразумевает построение модели как совокупности параллельно работающих процессов. При исполнении процессов меняется состояние модели. Однако процессы управляют не только параметрами, моделирующими систему, но и порядком исполнения друг друга. Процессы могут приостанавливать и возобновлять свое исполнение, планировать возобновление друг друга на определенные моменты времени, создавать новые процессы и уничтожать завершенные.

При *событийном* моделировании программа состоит из ряда процедур обработки событий, которые происходят в определенные моменты времени. При обработке каждого события определенным образом меняется состояние системы. Процедуры обработки событий могут планировать новые события на какие-то последующие моменты времени.

Моделирование, *ориентированное на состояния*, подразумевает наличие ряда условий, которые отслеживаются системой моделирования. При наступлении некоторого условия (например, равенства некоторых переменных, описывающих состояние), порождается событие, которое обрабатывается соответствующей процедурой. Как правило, такой подход к моделированию используется при построении непрерывных моделей.

Разные способы имитационного моделирования не исключают друг друга и могут сочетаться и взаимно дополнять друг друга в рамках одной модели. Так, событийные подходы широко используются в процессно-ориентированном моделировании, а непрерывная имитация может сочетаться с элементами дискретной.

В данной работе рассматривается построение дискретных процессно-ориентированных моделей с использованием разработанной автором библиотеки классов на языке Delphi. Эта библиотека, а также рассматриваемые здесь примеры размещены в подпапках папки Modeling на учебном сетевом диске. Все классы библиотеки для более простого использования размещены в одном модуле, который называется USimulation. Этот модуль размещен в одноименной папке. В дальнейшем планируется дополнение библиотеки средствами непрерывного моделирования.

При создании данной работы преследовались две основные цели. Первая цель достаточно очевидна: познакомить студентов с основными понятиями, подходами и приемами, используемыми в имитационном моделировании. Вторая цель, однако, не менее важна: на практических реальных примерах изучить основные технологии объектно-ориентированного программирования. Кроме того, автор надеется, что приведенные программы могут служить наглядным примером такого важного, но практически не поддающегося описанию понятия, как хороший стиль программирования (елинственный способ освоения которого — знакомство с примерами).

Первые две главы носят описательно-справочный характер. При первом чтении с ними нужно хотя бы бегло ознакомиться, чтобы ориентироваться в основных понятиях имитационного моделирования, а в дальнейшем обращаться к ним для получения детальной информации по используемым средствам. Третья глава содержит ряд примеров построения имитации. В четвертой главе описываются дополнительные возможности имитации и ряд примеров с их использованием. В пятой главе содержатся задания и упражнения для самостоятельной работы.

# 1. Базовые классы

Сначала рассмотрим классы, которые являются базовыми инструментами и призваны выполнять некоторые вспомогательные функции. Все последующие классы моделирования активно используют эти средства. К рассматриваемой категории относятся классы для построения списков, сопрограмм, генерации случайных последовательностей и сбора статистики. В тексте описывается только содержимое открытой (**public**) и защищенной (**protected**) частей классов, которые используются при построении программ. Закрытые (**private**) части классов не описываются, поскольку они предназначены для реализации внутренних механизмов работы классов, и не могут быть использованы при разработке собственных программ. Фрагменты определения класса, явно не рассматриваемые, не будем включать в листинг. Таким образом, реальное определение класса не всегда в точности соответствует тому тексту, который приведен здесь. Кроме того, при рассмотрении свойств (**property**) классов будем указывать только ключевые слова **read** и/или **write**, указывающие на возможность их чтения и записи, и не указывать, при обращении к какому полю или методу производится чтение/запись свойства.

## 1.1. Списки

Список является одной из фундаментальнейших динамических структур. В моделировании списки используются для представления, в частности, очередей ожидания.

Общим базовым классом для всех объектов, входящих в состав списочных структур, является класс TLinkage (связная ячейка):

TLinkage = **class**

**protected**

FPrev, FNext : TLinkage;

**public**

**constructor** Create;

**function** Next : TLink;

**function** Prev : TLink;

**end**;

Ячейка содержит два поля связи FPrev и FNext, указывающие соответственно на предыдущую и последующую ячейки списка. Методы Prev и Next возвращают те же указатели, за исключением случая, когда они указывают на заголовочную ячейку списка, представленную классом TList. В последнем случае эти методы возвращают пустой указатель (**nil**). Конструктор Create при создании просто помещает в оба поля связи пустые указатели.

Более сложные функции внутренней ячейки списка выполняет производный класс TLink:

TLink = **class**(TLinkage)

**public**

**destructor** Destroy; **override**;

**procedure** Remove;

**procedure** InsertBefore(L : TLinkage);

**procedure** InsertAfter(L : TLinkage);

**procedure** InsertLast(L : TList);

**procedure** InsertFirst(L : TList);

**procedure** Insert(L : TList); **overload**;

**procedure** Insert(L : TList; Order : TcompareFunction); **overload**;

**end**;

Методы InsertBefore и InsertAfter позволяют помещать ячейку в список перед или после указанной. Методы InsertFirst и InsertLast помещают ячейку в начало или конец указанного списка соответственно. Метод Remove исключает ячейку из текущего списка. Попутно заметим, что перед добавлением с использованием методов Insert\*\*\* ячейка предварительно исключается из того списка, в котором она в данный момент находится, поэтому исключать ее в явном виде нет необходимости. Деструктор перед удалением ячейки также исключает ее из текущего списка.

Две разновидности метода Insert позволяют вставить ячейку в список L, местоположение ячейки в котором задается функцией сравнения. Эта функция может быть встроена в список; в этом случае следует пользоваться вариантом метода Insert с одним параметром. Если функция сравнения для списка не задана, ячейка помещается в конец списка, то есть, в этом случае вызов lnk.Insert(L) эквивалентен lnk.InsertLast(L). Вариант метода Insert с двумя параметрами позволяет указать произвольную функцию сравнения.

Сама функция сравнения должна быть определена в соответствии с прототипом:

TCompareFunc = **function**(A, B : TLink) : Boolean;

Эта функция принимает в качестве параметров две списочные ячейки и возвращает значение истина (True), если ячейка A должна располагаться в списке непосредственно перед ячейкой B. Алгоритм вставки производит перебор ячеек списка, начиная с головной, и для каждой ячейки вызывает функцию сравнения в виде Comp(A, B), где A — добавляемая ячейка, а B — очередная ячейка списка. Как только результат функции будет равен True, ячейка вставляется в список перед ячейкой B. Если для всех ячеек списка результат функции равен False, ячейка вставляется в конец списка.

Приведем примеры построения функции сравнения. Предположим, что в список помещаются объекты класса TMyLink, который является производным от класса TLink. Упорядочение производится в соответствии со значением свойства (поля, функции) Rank числового типа.

**function** Comp1(A, B : TLink) : Boolean;

**begin**

Result := (A **as** TMyLink).Rank < (B **as** TMyLink).Rank;

**end**;

**function** Comp2(A, B : TLink) : Boolean;

**begin**

Result := (A **as** TMyLink).Rank <= (B **as** TMyLink).Rank;

**end**;

**function** Comp3(A, B : TLink) : Boolean;

**begin**

Result := (A **as** TMyLink).Rank > (B **as** TMyLink).Rank;

**end**;

При использовании функции Comp1 ячейка помещается перед первой ячейкой, значение свойства Rank которой меньше, чем у текущей, то есть, после всех ячеек, у которых значение этого свойства больше или равно его значению текущей ячейки. Если эта функция используется всегда при формировании списка (например, если эта функция задана в качестве функции сравнения списка), то его ячейки будут упорядочены по возрастанию значения свойства Rank. Ячейки, у которых значения этого свойства равны, будут выстроены в порядке поступления (так называемая дисциплина FIFO — first in, first out).

Функция Comp2 также выстраивает ячейки списка в порядке возрастания значения свойства Rank. Однако в этом случае новая ячейка будет добавляться перед всеми ячейками, у которых значение свойства Rank равно его значению у новой ячейки. Таким образом, ячейки с равными значениями свойства Rank будут выстроены в порядке, обратном порядку поступления (дисциплина LIFO — last in, first out).

При использовании функции Comp3 ячейки списка будут упорядочены по убыванию значения атрибута Rank, при этом ячейки с равным значением этого атрибута упорядочены в соответствии с дисциплиной FIFO.

Заметим еще раз, что возможны два способа использования функции сравнения. Первый способ подразумевает задание функции сравнения при создании списка (см. далее). В этом случае метод Insert с одним параметром каждый раз будет вставлять ячейку в соответствии с этой функцией. Второй способ — указать функцию сравнения вторым параметром метода Insert. В этом случае заданная для списка в целом функция сравнения не принимается во внимание.

Смешивать два способа использования функции сравнения, а также сочетать различные функции сравнения и смешивать использование методов Insert с InsertFirst/InsertLast следует крайне осторожно. Дело в том, что при неконтролируемом использовании таких сочетаний порядок ячеек в списке может стать непредсказуемым. Поэтому сформулируем две основные стратегии построения списков, дающие предсказуемые результаты.

Первая стратегия не использует функцию сравнения списка, а сам список формируется при помощи методов InsertFirst/InsertLast (в зависимости от того, какую из дисциплин построения списка — LIFO или FIFO — требуется обеспечить). Как вариант, можно пользоваться методом Insert с одним параметром — в отсутствие функции сравнения он эквивалентен методу InsertLast. В отдельных случаях, когда ячейку надо вставить в середину списка, можно воспользоваться методом Insert с указанием функции сравнения.

Вторая стратегия подразумевает задание функции сравнения для списка. В этом случае все ячейки помещаются в список только методом Insert в одним параметром. Методы InsertFirst, InsertLast и Insert с двумя параметрами не используются.

Для представления списка в целом используется класс TList, который, по существу, является заголовочной ячейкой списка. Этот класс также является производным от TLinkage и содержит дополнительные средства, необходимые для организации работы списков:

TList = **class**(TLinkage)

**public**

LengthStat : TIntervalStatistics;

WaitStat : TStatistics;

**constructor** Create; **overload**;

**constructor** Create(SimTime : Double); **overload**;

**constructor** Create(Order : TCompareFunction); **overload**;

**constructor** Create(Order : TCompareFunction; SimTime : Double); **overload**;

**destructor** Destroy; **override**;

**function** First : TLink;

**function** Last : TLink;

**function** Empty : Boolean;

**function** Size : Integer;

**procedure** Clear;

**procedure** StopStat; **overload**;

**procedure** StopStat(NewTime : Double); **overload**;

**procedure** ClearStat; **overload**;

**procedure** ClearStat(NewTime : Double); **overload**;

**property** Order : TCompareFunc **write**;

**end**;

Заголовочная ячейка содержит два поля, отвечающих за сбор статистики: LengthStat собирает статистику по длине списка, а WaitStat — по времени нахождения ячеек в списке. Методы StopStat корректируют данные статистики к текущему времени, а методы ClearStat очищают статистику. Подробности сбора статистики см. в параграфе 1.5.

Методы First и Last возвращают указатели на первую и последнюю ячейки списка соответственно. Метод Empty возвращает истинный результат, если в списке в данный момент нет ни одной ячейки. Метод Size возвращает текущее число ячеек в списке. Метод Clear очищает список, исключая из него каждую ячейку и удаляя соответствующий объект.

Конструктор Create существует в четырех вариантах: без параметров, с одним параметром (два варианта) и с двумя параметрами. Параметр SimTime позволяет указать имитационное время создания списка с целью обеспечения правильного функционирования статистики. Если этот параметр не указан, в качестве этого времени используется текущее имитационное время в момент создания списка. Параметр Order позволяет указать функцию сравнения для вставки ячеек в список. Если этот параметр не указан, никакая функция сравнения не используется, и метод ячейки Insert без параметров будет помещать ее в конец списка. Деструктор Destroy перед удалением списка удаляет все находящиеся в нем объекты.

Свойство Order позволяет задать функцию сравнения для уже построенного списка. Однако сделать это можно только, во-первых, для пустого списка, а во-вторых, если функция сравнения ранее не была задана. Следовательно, основное применение этого свойства состоит в задании функции сравнения сразу после создания списка. Обычно это делается для списков, автоматически создаваемых при создании других объектов (например, ресурсов и затворов). Если хотя бы одно из указанных условий не выполняется, попытка присваивания свойству Order порождает исключение.

## 1.2. Сопрограммы

Основными базовыми средствами построения параллельно исполняемых процессов являются *сопрограммы*. Сопрограммы отличаются от подпрограмм равноправным характером своего взаимодействия. Обращение к сопрограмме приостанавливает ранее исполнявшуюся сопрограмму (как и в случае подпрограмм) и возобновляет работу вызванной сопрограммы с того места, где она была ранее остановлена (или с начала, если она еще не была запущена).

Таким образом, сопрограммы позволяют попеременно исполнять отдельные шаги своего алгоритма, передавая управление друг другу по мере необходимости. Фактически, сопрограмма — это инструмент создания так называемой *кооперативной многозадачности*. Такая многозадачность, в отличие от применяемой в операционных системах *вытесняющей* многозадачности, не требует вмешательства диспетчера и планировщика операционной системы, требует гораздо меньше ресурсов и позволяет точно управлять тем, в какие моменты сопрограммы приостанавливают и возобновляют свое исполнение.

Основной недостаток сопрограмм и кооперативной многозадачности заключается в том, что их использование требует от программиста строгого выполнения некоторых правил построения кода. Если эти правила не соблюдать, программа либо перестает работать, либо ее работа становится неустойчивой. Однако в рассматриваемом контексте сопрограммы — достаточно удобный инструмент, поскольку позволяют описывать правила функционирования моделируемой системы с использованием всех приемов программирования и любых алгоритмических структур.

Сопрограммы строятся на основе класса TRunningObject:

TRunningObject = **class**(TLink)

**protected**

Owner : TRunningObject;

**procedure** Execute; **virtual**; **abstract**;

**public**

**constructor** Create;

**destructor** Destroy; **override**;

**property** Terminated : Boolean **read** FTerminated;

**end**;

В защищенной (**protected**) части класса имеется поле Owner и абстрактный метод Execute. В поле Owner хранится указатель на объект-владелец сопрограммы. Владельцем является либо сопрограмма, при исполнении которой (в дальнейшем будем говорить: в *контексте* которой) была создана данная сопрограмма, либо **nil**, если сопрограмма была создана в контексте исполнения главного потока команд программы. Главным потоком считается поток команд, исполняющихся сразу после запуска программы. Программы, построенные без использования сопрограмм и/или потоков, состоят только из главного потока команд. Соответственно, при переключении сопрограмм указание **nil** в качестве следующей сопрограммы означает переход к главному потоку команд программы.

Абстрактный метод Execute следует переопределить в производном классе и реализовать в нем алгоритм работы сопрограммы. Другими словами, именно в этом методе определяется, какие действия выполняет та или иная сопрограмма. Этот метод определен в защищенной части класса, поскольку, с одной стороны, любой производный класс должен переопределять его, а в другой стороны, другие классы не должны напрямую обращаться к этому методу.

Открытая (**public**) часть определения класса содержит конструктор Create, деструктор Destroy и свойство Terminated, предназначенное только для чтения. В это свойство помещается значение True, когда сопрограмма завершила свою работу. Это делается автоматически после завершения метода Execute.

Переключение процессов производится с помощью двух глобальных функций:

**procedure** SwitchTo(Proc : TRunningObject);

**procedure** Detach;

Процедура SwitchTo приостанавливает текущий исполняющийся поток и передает управление потоку команд, соответствующему объекту-сопрограмме. Процедура Detach (открепить) передает управление сопрограмме, являющейся владельцем текущей (что может быть выражено также посредством вызова SwitchTo(Owner)). Вызов SwitchTo(**nil**) производит переключение к главному потоку команд программы. Заметим, что эти процедуры являются именно глобальными процедурами, а не методами какого-либо класса. Текущий исполняемый поток отслеживается автоматически.

Вернемся к методу Execute класса TRunningObject. Рассмотрим некоторые особенности его переопределения в производных классах. Пусть, к примеру, производный класс называется TMyProcess. Тогда его определение в простейшем варианте выглядит так:

TMyProcess = **class**(TRunningObject)

**protected**

**procedure** Execute; **override**;

**end**;

В этом определении указывается, что метод Execute *перекрывает* метод базового класса (**override**). Сама структура определения этого метода приведена ниже:

**procedure** TMyProcess.Execute;

**begin**

*// Инициализация процесса, создание необходимых объектов и т. п.*

…

Detach;

*// Выполнение алгоритма сопрограммы*

…

Detach;

**end**;

Как видно из приведенного примера, текст процедуры фактически включает две части: инициализацию и собственно алгоритм работы. Инициализация завершается вызовом Detach. Этот вызов фактически завершает исполнение конструктора в контексте потока, создавшего данную сопрограмму. После того как сопрограмме будет передано управление, выполнение метода Execute продолжается после вызова Detach. Переключение сопрограмм в процессе исполнения, как правило, производится с помощью процедуры SwitchTo. Завершается исполнение сопрограммы последним вызовом Detach. Заметим, что попытка вызова сопрограммы, закончившей свою работу, приведет к ошибке времени исполнения программы. Поэтому использование сопрограмм, в частности, требует тщательного контроля их состояния (это одно из упоминавшихся правил, за исполнением которых приходится следить разработчику). Естественно, при необходимости в производном классе можно к имеющимся методам добавлять свои и вызывать их из метода Execute.

Если возникает потребность в создании нового конструктора (например, если требуется при создании объекта указывать какие-то параметры), то *последней* строкой конструктора *обязательно* должен быть вызов унаследованного конструктора:

**constructor** TMyProcess.Create(*параметры*);

**begin**

*// Инициалиация объекта в соответствии с переданными параметрами*

…

**inherited** Create;

**end**;

Если последнее правило не будет соблюдено, объект будет неработоспособен.

Сопрограммы позволяют организовать псевдопараллельную работу нескольких потоков исполнения, однако класс TRunningObject не очень удобен для построения программ, поскольку предлагает лишь базовые возможности и вынуждает программиста следить за многими подробностями работы программы. Поэтому для построения моделей мы не будем непосредственно его использовать, а воспользуемся более мощным и удобным производным классом TProcess, который рассмотрим в главе 2.

## 1.3. Примеры использования сопрограмм

Рассмотрим два примера построения сопрограмм: первый пример совсем простой, второй несколько сложнее. Оба примера строятся как консольные приложения.

*Пример 1.1*. Построим 3 одинаковых сопрограммы, которые поочередно выводят на экран числа от 1 до 5, сопровождая его именем подпрограммы. Это логично сделать, переключая задачи в циклическом порядке. Для этого следует в каждом объекте-сопрограмме иметь указатель на ту сопрограмму, которая должна исполняться следующей. Кроме того, в нем также следует хранить имя текущей сопрограммы. Текст модуля, в котором определяется класс сопрограммы, предельно прост:

**unit** UCoroutine;

**interface**

**uses** USimulation;

**type**

TCoroutine = **class**(TRunningObject)

**public**

NextCor : TRunningObject;

Name : **string**;

**protected**

**procedure** Execute; **override**;

**end**;

**implementation**

*{ TCoroutine }*

**procedure** TCoroutine.Execute;

**var**

i : Integer;

**begin**

Detach;

**for** i := 1 **to** 5 **do**

**begin**

WriteLn('Coroutine ', Name, '; i = ', i);

SwitchTo(NextCor);

**end**;

Detach;

**end**;

**end**.

В определении класса поле NextCor содержит ссылку на объект-сопрограмму, исполняемый следующим, а поле Name содержит имя данного объекта. Переопределенный метод Execute построен по упомянутым выше правилам. Поскольку никакой специальной инициализации объекту не требуется, метод начинается сразу с вызова Detach. После него следует исполняющая часть алгоритма, который представляет собой цикл. В теле цикла выводится на экран сообщение, предусмотренное формулировкой задачи, после чего управление передается следующей сопрограмме. Второй вызов Detach возобновляет работу главного потока команд (он производится из той сопрограммы, которая первой завершит свой цикл), что в данном случае ведет к завершению программы.

К сожалению, консольные программы, разработанные в Delphi (при разработке примеров использовалась версия 7), не учитывают различия кодировок русских букв для DOS и Windows. Поэтому мы вынужденно формулируем все выводимые на экран сообщения по-английски.

Текст главного модуля программы также предельно прост. В нем создаются объекты-сопрограммы, задаются значения их свойств, после чего запускается первая сопрограмма (corA). После завершения сопрограмм основная программа продолжает свою работу, что в данном случае означает ожидание нажатия пользователем клавиши Enter и завершение программы.

**var**

corA, corB, corC : TCoroutine;

**begin**

corA := TCoroutine.Create;

corB := TCoroutine.Create;

corC := TCoroutine.Create;

corA.Name := 'A';

corB.Name := 'B';

corC.Name := 'C';

corA.NextCor := corB;

corB.NextCor := corC;

corC.NextCor := corA;

SwitchTo(corA);

ReadLn;

**end**.

После выполнения программы в консольном окне получаем следующее содержимое:

Coroutine A; i = 1

Coroutine B; i = 1

Coroutine C; i = 1

Coroutine A; i = 2

Coroutine B; i = 2

Coroutine C; i = 2

Coroutine A; i = 3

Coroutine B; i = 3

Coroutine C; i = 3

Coroutine A; i = 4

Coroutine B; i = 4

Coroutine C; i = 4

Coroutine A; i = 5

Coroutine B; i = 5

Coroutine C; i = 5

Полный текст проекта находится в папке 1.3 Coroutines.

*Пример 1.2*. Этот пример взят из [1]. Рассмотрим классическую задачу расстановки ферзей: на шахматной доске размером N × N требуется расставить N ферзей так, чтобы ни один из них не угрожал никакому другому (напомним, что, по правилам игры в шахматы, ферзь угрожает всем полям, расположенным по отношению к нему на одной вертикали, горизонтали и по обеим диагоналям). Расстановку, удовлетворяющую сформулированным условиям, будем называть *безопасной*. Требуется получить все возможные безопасные расстановки (без учета эквивалентности расстановок, получаемых друг из друга посредством зеркального отражения по любому из направлений и поворота на 90 градусов). Классическое решение этой задачи подразумевает использование рекурсивных процедур (подпрограмм). Рассмотрим, как можно решить эту же задачу с использованием сопрограмм.

Сначала выберем способ представления информации о расположении ферзей на доске. Поскольку требуется разместить N ферзей на доске размером N × N, то очевидно, что каждый ферзь должен располагаться на отдельной вертикали. И наоборот, на каждой вертикали размещается ровно по одному ферзю. Положение ферзя на вертикали представлено номером поля, считая снизу. Информация о расстановке ферзей, таким образом, может быть представлена последовательностью из N вертикальных координат ферзей. Так, к примеру, при N = 5 расстановка может отображаться следующим образом:

1 3 5 2 4

Эта запись обозначает, что ферзи расставлены в полях (1, 1), (2, 3), (3, 5), (4, 2) и (5, 4). Здесь для удобства мы обозначаем вертикали доски не буквами, как это принято в шахматах, а номерами. Данную расстановку изобразим в виде таблицы:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | \* |  |  |
|  |  |  |  | \* |
|  | \* |  |  |  |
|  |  |  | \* |  |
| \* |  |  |  |  |

Легко проверить, что данная расстановка удовлетворяет формулировке задачи.

Представим программу в виде совокупности N сопрограмм. Каждая из них имитирует процесс поиска ферзем безопасного поля в отведенной для него вертикали. Номер вертикали, занимаемой каждым ферзем, задается при его создании и в процессе поиска не меняется. Номер горизонтали ферзь выбирает в процессе поиска, просматривая все возможные варианты от 1 до N. Если процесс поиска оказался успешным, то есть, безопасное поле (будем называть его также свободным) было найдено, этот ферзь предлагает следующему ферзю найти место для него. Если поиск завершился неудачно, то есть, ферзь достиг последней горизонтали и не нашел безопасного поля, то он предлагает предыдущему ферзю попробовать другое расположение. Если текущий ферзь является последним, то есть, номер его вертикали равен N, то при нахождении безопасного поля он выполняет вывод полной информации о расстановке ферзей, а затем продолжает поиск, поскольку по условию задачи требуется найти все возможные расстановки. Если текущий ферзь является первым, то есть, номер его вертикали равен 1, то после перебора всех вариантов он возобновляет главный поток приложения, тем самым сообщая, что поиск закончен.

Запишем обсужденный алгоритм действий ферзя в виде класса TQueen, производного от TRunningObject. Заметим, что, поскольку при создании ферзя требуется задать ему номер вертикали, в данном классе, кроме переопределенного метода Execute, будет присутствовать конструктор:

**type**

TQueen = **class**(TRunningObject)

**public**

Col, Row : Byte;

**constructor** Create(ACol : Byte);

**protected**

**procedure** Execute; **override**;

**end**;

Как видно из этого листинга, в объекте класса присутствуют поля Col и Row, отвечающие за текущие номера вертикали и горизонтали, занимаемых ферзем, соответственно. При этом номер вертикали в процессе работы ферзя не изменяется и задается при его создании как параметр конструктора ACol. Номер горизонтали ферзь изменяет в процессе поиска свободного поля. Сами сопрограммы, отвечающие за работу ферзей, собраны в массив:

**const**

N = 5; *// для примера*

**var**

Queens : **array** [1 .. N] **of** TQueen;

Тексты рассмотренных процедур приведены ниже:

**constructor** TQueen.Create(ACol : Byte);

**begin**

*// Установка номера вертикали*

Col := ACol;

*// Поиск начинается с 1-й горизонтали*

Row := 1;

*// Требуется по правилам разработки сопрограмм*

**inherited** Create;

**end**;

**procedure** TQueen.Execute;

**begin**

*// Вся инициализация произведена в конструкторе*

Detach;

*// «Бесконечный» цикл*

**while** True **do**

**begin**

*// Если выбранное поле свободно*

**if** IsFree(Col, Row) **then**

**begin**

*// Занять его*

MakeOccupied(Col, Row);

*// Если ферзь не последний*

**if** Col < N **then**

*// Передать управление следующему*

SwitchTo(Queens[Col + 1])

**else**

*// Последний ферзь выводит на экран текущее состояние расстановки*

Remember;

*// Освободить поле для поиска других вариантов*

MakeFree(Col, Row);

**end**; *// if IsFree*

*// Перейти на следующую горизонталь*

Inc(Row);

*// Если варианты исчерпаны*

**if** Row > N **then**

**begin**

*// Начать сначала*

Row := 1;

**if** Col = 1 **then**

*// Первый ферзь передает управление главному потоку программы*

Detach

**else**

*// Остальные — обращаются к предыдущим ферзям*

*// для продолжения поиска*

SwitchTo(Queens[Col - 1]);

**end**; *// if Row > N*

**end**; *// while True*

**end**;

В этом алгоритме используется несколько вспомогательных процедур, которые позволяют анализировать состояние игрового поля, а также ставить и удалять ферзя на указанном поле. Функция IsFree(Col, Row) возвращает значение True, если указанное поле свободно, то есть, ни один из предыдущих ферзей ему не угрожает. Функция MakeOccupied(Col, Row) ставит ферзя на заданное поле, а функция MakeFree(Col, Row) снимает ферзя с поля. Логика взаимодействия ферзей такова, что при работе любого ферзя все предыдущие ферзи установлены на какие-либо поля, а все последующие не установлены ни на какие.

Рассмотрим распространенный вариант реализации этих процедур. Поскольку каждый установленный на доску ферзь занимает одну горизонталь и две диагонали, проверку занятости поля удобно выполнить, проверив занятость горизонтали и двух диагоналей, на которых располагается поле. При установке ферзя соответствующие горизонталь и диагонали помечаются как занятые, а при его снятии — как свободные. Информацию о занятости горизонталей и диагоналей удобно представлять в виде массивов значений логического типа (Boolean), при этом значение True соответствует занятости горизонтали или диагонали, а значение False — ее свободности. Горизонтали в массиве индексируются, очевидно, их номерами. Индексы диагоналей одного направления являются суммами номеров вертикали и горизонтали (это диагонали, параллельные диагонали, идущей из левого верхнего угла доски в правый нижний). Очевидно, значения этих индексов находятся в интервале от 2 до 2 \* N. Индексы диагоналей другого направления представляют собой разности номеров вертикали и горизонтали. Они находятся в границах от –N + 1 до N – 1. Ниже приводятся листинги процедур, выполняющих указанные действия:

**var**

Rows : **array** [1 .. N] **of** Boolean;

DiagsUp : **array** [-N + 1 .. N - 1] **of** Boolean;

DiagsDown : **array** [2 .. 2 \* N] **of** Boolean;

**function** IsFree(Col, Row : Byte) : Boolean;

**begin**

Result := **not** Rows[Row] **and** **not** DiagsUp[Col - Row] **and**

**not** DiagsDown[Col + Row];

**end**;

**procedure** MakeOccupied(Col, Row : Byte);

**begin**

Rows[Row] := True;

DiagsUp[Col - Row] := True;

DiagsDown[Col + Row] := True;

**end**;

**procedure** MakeFree(Col, Row : Byte);

**begin**

Rows[Row] := False;

DiagsUp[Col - Row] := False;

DiagsDown[Col + Row] := False;

**end**;

Процедура Remember предназначена для вывода на экран или какой-то иной обработки (запоминания и т. п.) полученной безопасной конфигурации. В простейшем варианте она просто отображает последовательность горизонталей, занятых ферзями, как это указывалось ранее:

**procedure** Remember;

**var**

i : Integer;

**begin**

**for** i := 1 **to** N **do**

Write(Queens[i].Row, ' ');

WriteLn;

**end**;

Однако разработанных классов и процедур еще недостаточно для работы программы. Необходимо еще создать массив сопрограмм-ферзей, запустить выполнение, а по его завершении удалить созданные объекты. Все это можно сделать в главной программе, однако это не самый элегантный вариант. Гораздо лучше предоставить эти действия отдельной сопрограмме. Данный подход, кстати, будет широко использоваться в дальнейшем, при разработке программ имитационного моделирования, и там он является единственно возможным. Определение сопрограммы, управляющей всем процессом поиска, достаточно просто:

**type**

TQueensSimulation = **class**(TRunningObject)

**protected**

**procedure** Execute; **override**;

**end**;

. . .

**procedure** TQueensSimulation.Execute;

**var**

i : Integer;

**begin**

*// Отдельной инициализации не требуется*

Detach;

*// Создать все сопрограммы*

**for** i := 1 **to** N **do**

Queens[i] := TQueen.Create(i);

*// Запустить первую*

SwitchTo(Queens[1]);

*// После окончания все удалить*

**for** i :=1 **to** N **do**

Queens[i].Free;

Detach;

**end**;

Тогда текст главной программы будет выглядеть предельно просто:

**var**

Sim : TQueensSimulation;

**begin**

*// Создать управляющую сопрограмму*

Sim := TQueensSimulation.Create;

*// Запустить ее*

SwitchTo(Sim);

*// После завершения удалить*

Sim.Free;

WriteLn('Done');

ReadLn;

**end**.

При N = 5 программа выдает следующий результат:

1 3 5 2 4

1 4 2 5 3

2 4 1 3 5

2 5 3 1 4

3 1 4 2 5

3 5 2 4 1

4 1 3 5 2

4 2 5 3 1

5 2 4 1 3

5 3 1 4 2

Done

Убедитесь самостоятельно в том, что все эти комбинации действительно безопасны. Проверить, что этот список представляет собой полный набор возможных вариантов, существенно сложнее. Однако вы можете проверить любую другую комбинацию, не входящую в приведенный набор, и убедиться, что она не является безопасной. Полный текст программы поиска расстановки ферзей находится в папке 1.3 Queens.

## 1.4. Случайные величины

В имитационном моделировании часто моделируются *стохастические* системы и процессы. Для таких систем характерно то, что временн***о***е распределение последовательности событий точно задать невозможно. Другими словами, в функционировании такой системы существенную роль играют случайные факторы. Эти факторы моделируются с помощью последовательностей *псевдослучайных* величин, то есть величин, порядок появления которых невозможно описать простыми соотношениями, и их можно с некоторой долей условности считать случайными. Важной характеристикой последовательности случайных (и, соответственно, псевдослучайных) величин является их *распределение*, описывающее вероятности принятия величиной того или иного значения. Для генерации последовательностей псевдослучайных величин, распределенных в соответствии с различными законами распределения, используется класс TRandom:

TRandom = **class**

**public**

**constructor** Create; **overload**;

**constructor** Create(Seed : Integer); **overload**;

**function** Draw(A : Double) : Boolean;

**function** NextInt : Integer; **overload**;

**function** NextInt(High : Integer) : Integer; **overload**;

**function** NextInt(Low, High : Integer) : Integer; **overload**;

**function** NextFloat : Double;

**function** Uniform(A, B : Double) : Double;

**function** TableIndex(Table : **array** **of** Double) : Integer;

**function** Normal(Mean, Sigma : Double) : Double;

**function** PSNorm(Mean, Sigma : Double; Count : Integer) : Double;

**function** Triangular(A, Moda, B : Double) : Double;

**function** LogNormal(Mean, Sigma : Double) : Double;

**function** Exponential(Mean : Double) : Double;

**function** Poisson(Lambda : Double) : Integer;

**function** NegExp(Mean : Double) : Double;

**function** Erlang(Mean : Double; K : Integer) : Double;

**function** Gamma(Beta, Alpha : Double) : Double;

**function** Beta(Alpha, Betta : Double) : Double; **overload**;

**function** Beta(Min, Mean, Max, Sigma : Double) : Double; **overload**;

**function** Weibull(Beta, Alpha : Double) : Double;

**end**;

Рассмотрим открытую часть класса. Конструктор Create с одним параметром задает начальное значение, исходя из которого рассчитываются все последующие значения. Этот конструктор удобно использовать для того, чтобы зафиксировать последовательность порождаемых псевдослучайных величин. Конструктор Create без параметров запускает последовательность псевдослучайных чисел, исходя из текущего показания системных часов, с использованием стандартной процедуры Randomize. В этом случае при каждом запуске программы получаются разные последовательности значений. Использование нескольких объектов класса TRandom позволяет использовать в программе несколько независимых потоков псевдослучайных значений, а также фиксировать некоторые потоки при многократных запусках.

Остальные методы класса позволяют получать очередные псевдослучайные значения, распределенные в соответствии с разными законами. Метод Draw возвращает значение True с вероятностью A (0 < A < 1). Разные варианты метода NextInt возвращают равномерно распределенные целочисленные значения. Вариант без параметров дает значения от 0 до 231 – 1. Вариант с одним параметром дает значения от 0 до High – 1 (то есть, не включая указанную верхнюю границу). Вариант с двумя параметрами дает значение в интервале от Low до High – 1. Метод NextFloat дает равномерно распределенное вещественное значение в интервале [0, 1). Метод Uniform возвращает равномерно распределенное значение в интервале [A, B).

Метод TableIndex возвращает целочисленное значение с вероятностью, соответствующей значениям, указанным в параметре-массиве Table. Этот массив должен содержать значения от 0 до 1, упорядоченные по возрастанию. Эти значения определяют границы вероятности получения того или иного значения. Так, к примеру, если в качестве параметра передан массив [0.2, 0.5, 0.6, 0.9], то вероятность получить результат 0 равна 0,2, значения 1 — 0,3, 2 — 0,1, 3 — 0,3, 4 — 0,1. Из примера видно, что максимально возможное значение результата равно числу элементов в массиве, а вероятность получения каждого значения равна разности соседних элементов в массиве (при этом подразумевается, что массив с краев неявно дополняется значениями 0 и 1). Результат, полученный от метода TableIndex, удобно использовать в качестве индекса для выбора значения из массива, содержащего произвольные значения.

Метод Normal возвращает нормально распределенную величину с математическим ожиданием Mean и стандартным отклонением Sigma. Для получения результата в этом методе используется стандартный алгоритм преобразования равномерно распределенной последовательности в нормально распределенную, в котором значения порождаются парами. Метод PSNorm также дает нормально распределенную величину, но для ее получения используется метод суммирования Count равномерно распределенных значений. Для получения распределения, близкого к нормальному, следует задавать достаточно большие значения Count (обычно в диапазоне 10—20).

Метод Triangular дает треугольно распределенную величину в интервале от A до B с модой (то есть, значением, где функция плотности вероятности имеет максимум) Moda. Метод LogNormal дает логнормально распределенную величину, то есть, величину, логарифм которой распределен нормально.

Метод Exponential дает экспоненциально распределенную положительную величину с математическим ожиданием Mean и равным ему стандартным отклонением. Метод NegExp также дает экспоненциально распределенную величину, только в качестве параметра указывается величина, обратная математическому ожиданию. Другими словами, вызов r.NegExp(A) эквивалентен вызову r.Exponential(1 / A). Выбор одного из этих двух методов обычно диктуется смыслом исходных данных: если задана величина математического ожидания интервала между событиями, удобнее пользоваться методом Exponential, а если задано среднее количество событий в единицу времени (интенсивность событий), удобнее использовать NegExp.

Метод Poisson дает неотрицательную целочисленную величину, распределенную по закону Пуассона с заданным параметром математического ожидания, который равен дисперсии.

Метод Erlang дает величину, представляющую собой сумму K экспоненциально распределенных величин с математическим ожиданием Mean для каждой из них. Полученная величина имеет математическое ожидание K \* Mean и дисперсию K \* Mean2.

Методы Gamma, Beta и Weibull возвращают величины, распределенные в соответствии с гамма-, бета-распределением и распределением Вейбулла соответственно. В дальнейших примерах они не используются, поэтому их подробное рассмотрение опускаем.

Рассмотрим области применимости популярных распределений. *Равномерное* распределение обычно используется, если информация о моделируемой величине минимальна: известны только возможные ее границы. Если есть информация о *моде*, то есть о том, какое значение величины появляется чаще других, удобно использовать *треугольное* распределение. Для моделирования *промежутков* времени между равномерно распределенными по времени событиями (то есть, если известно, что в единицу времени в среднем происходит одно и то же количество событий) используется *экспоненциальное* распределение. *Нормальное* распределение используется для моделирования продолжительности действия, когда известна его средняя продолжительность и стандартное отклонение. Распределение *Пуассона* используется, если требуется смоделировать количество событий за некоторый фиксированный промежуток времени при условии, что эти события в среднем распределены равномерно. Распределение *Эрланга* используется, если есть основания считать, что моделируемое действие состоит из нескольких этапов, каждый из которых имеет экспоненциально распределенную продолжительность. Другой случай применения этого распределения — моделирование интервала времени между не соседними событиями, то есть, событиями, которые следуют не сразу друг за другом, а, например, через одно, через два и т. д.

Иногда требуется ограничить получаемое значение, то есть, обеспечить, чтобы значение, получаемое из генератора, лежало в заданных границах. Один из способов сделать это — воспользоваться следующими функциями:

**function** Min(a, b : Double) : Double; **overload**;

**function** Max(a, b : Double) : Double; **overload**;

**function** Min(a, b : Integer) : Integer; **overload**;

**function** Max(a, b : Integer) : Integer; **overload**;

Результат этих функций вычисляется предельно просто: это меньшее или большее из двух значений параметров соответственно. Тогда, например, если требуется, чтобы нормально распределенная величина была неотрицательной, можно обеспечить это при помощи конструкции Max(rnd.Normal(1, 0.2), 0). Если же требуется ограничить величину с двух сторон, следует воспользоваться обеими этими функциями: Min(Max(rnd.Normal(0.5, 0.2), 0), 1). Функции Min и Max определены в двух вариантах: для целочисленных и вещественных параметров.

## 1.5. Классы сбора статистики и гистограмм

Процесс имитации характеризуется большим объемом числовых значений, составить по которым общее представление о ее ходе практически невозможно в силу большого количества величин. Гораздо удобнее анализировать результаты имитации по статистическим данным, собранным по этим величинам. Рассмотрим здесь основные классы, предназначенные для сбора статистической информации. Специализированные классы сбора статистики, работающие с действиями, ресурсами и затворами, будут рассмотрены далее при обсуждении соответствующих классов.

Класс TStatistics предназначен для сбора *точечной* статистики по отдельным числовым величинам:

TStatistics = **class**

**public**

Count : Integer;

**constructor** Create;

**procedure** AddData(NewX : Double);

**procedure** Clear;

**function** Mean : Double;

**function** Deviation : Double;

**function** Disperse : Double;

**property** Min : Double **read**;

**property** Max : Double **read**;

**property** Count : Integer **read**;

**end**;

Рассмотрим открытую часть класса. Назначение конструктора Create совершенно традиционно и не нуждается в пояснениях. Метод AddData добавляет к статистике новую величину. Метод Clear очищает все накопленные данные и подготавливает объект к сбору новых данных (фактически, приводит его в исходное состояние). Методы Mean, Deviation и Disperse позволяют получить среднее значение, стандартное отклонение и дисперсию по накопленным данным соответственно. Если с момента создания или очистки объекта никакие данные не добавлялись, во избежание порождения исключения эти методы дают результат, равный нулю. Свойства Min и Max дают минимальное и максимальное из добавленных значений соответственно. Аналогично предыдущим методам, если никаких данных не было добавлено, эти свойства дают нулевые значения. Свойство Count дает общее количество добавленных в статистику значений с момента создания или последней очистки.

Для сбора статистики о временн***ы***х интервалах между определенными событиями используется похожий на предыдущий класс TTimeBetStatistics:

TTimeBetStatistics = **class**

**public**

**constructor** Create;

**procedure** AddData; **overload**;

**procedure** AddData(NewTime : Double); **overload**;

**procedure** Clear;

**function** Mean : Double;

**function** Deviation : Double;

**function** Disperse : Double;

**property** Min : Double **read**;

**property** Max : Double **read**;

**property** Count : Integer **read**;

**end**;

Методы и свойства данного класса аналогичны классу TStatistics, однако предметом изучения здесь являются интервалы времени между событиями. Метод AddData с одним параметром добавляет к статистике интервал времени между предыдущим добавленным временем и текущим временем, указанным в качестве параметра (на самом деле, можно добавлять в статистику величины произвольной природы, однако будут учитываться разности между соседними значениями). Это время запоминается в объекте и впоследствии используется для вычисления следующего интервала времени при добавлении следующего значения. Метод AddData без параметров учитывает в качестве текущего времени имитационное время текущего процесса (см. параграф 2.2). Если данный метод вызван в отсутствие текущего процесса, генерируется исключение.

Сразу после создания или очистки статистики свойство Count равно –1, а после добавления одного значения — нулю. Эта особенность класса отражает тот факт, что количество интервалов времени между событиями на 1 меньше числа событий.

Учет статистики по величинам, меняющимся во времени (*интервальной* статистики), производится в классе TIntervalStatistics. Исходя из общих положений дискретного моделирования, методы данного класса предполагают, что учитываемые величины сохраняют свое значение постоянным между моментами их изменения.

TIntervalStatistics = **class**

**public**

**constructor** Create(InitX : Double); **overload**;

**constructor** Create(InitX, InitTime : Double); **overload**;

**procedure** AddData(NewX : Double); **overload**;

**procedure** AddData(NewX, NewTime : Double); **overload**;

**procedure** StopStat; **overload**;

**procedure** StopStat(NewTime : Double); **overload**;

**procedure** Clear; **overload**;

**procedure** Clear(NewTime : Double); **overload**;

**function** Mean : Double;

**function** Deviation : Double;

**function** Disperse : Double;

**property** Min : Double **read**;

**property** Max : Double **read**;

**property** Current : Double **read**;

**property** TotalTime : Double **read**;

**end**;

Конструктор Create и методы AddData, StopStat, Clear могут вызываться в двух вариантах с различным числом параметров. Во всех случаях последним параметром, который может быть опущен, является имитационное время, которому соответствует изменение величины. Если этот параметр не указан, всегда подразумевается текущее значение имитационного времени.

В конструкторе Create, кроме (необязательного) указания начального момента времени сбора статистики, задается начальное значение исследуемой величины. В методе AddData указывается новое значение величины, которое будет сохраняться неизменным до следующего изменения. Последнее заданное значение величины может быть получено при помощи свойства Current. Метод Clear очищает накопленные данные и приводит объект в начальное состояние, но с двумя отличиями: во-первых, начальным временем сбора статистики является момент времени, когда была произведена очистка; во-вторых, начальным значением величины является то значение, которое было занесено в объект статистики последним перед очисткой.

Метод StopStat производит коррекцию статистики к текущему времени в следующем смысле: время последнего изменения величины устанавливается равным заданному времени, в то время как значение исследуемой величины не изменяется. Это действие требуется выполнять каждый раз перед отображением статистической информации, чтобы учесть промежуток времени, истекший с момента последнего изменения до текущего времени. Свойство TotalTime дает общий промежуток времени, в течение которого происходит сбор статистики, то есть, разность между последним временем изменения величины и временем создания объекта или последней очистки.

В ряде случаев требуется получить не просто статистические характеристики набора данных, а более подробную информацию, а именно, какое количество значений попадает в тот или иной интервал. Это можно сделать с помощью объектов-*гистограмм*. Общим базовым классом для гистограмм является абстрактный класс THistogram:

THistogram = **class**

**public**

**constructor** Create;

**procedure** Clear; **virtual**; **abstract**;

**procedure** AddData(d : Double); **virtual**; **abstract**;

**property** Count[i : Integer] : Integer **read**;

**property** Percent[i : Integer] : Double **read**;

**property** CumulativeCount[i : Integer] : Integer **read**;

**property** CumulativePercent[i : Integer] : Double **read**;

**property** LowerBound[i : Integer] : Double **read** **write**;

**property** UpperBound[i : Integer] : Double **read** **write**;

**property** IntervalCount : Integer **read** **write**;

**property** TotalCount : Integer **read**;

**end**;

Поскольку класс THistogram является абстрактным, рассматривать его конструктор не имеет смысла. Метод Clear очищает все данные гистограммы и приводит ее в начальное состояние. Метод AddData добавляет к гистограмме новые данные.

Учет значений в гистограмме производится по интервалам. При этом два крайних интервала являются полубесконечными, остальные — конечными. Если *a*0, *a*1, …, *an* — граничные значения интервалов, то сами интервалы выглядят так: (–∞, *a*0), [*a*0, *a*1), …, [*an*, +∞). Отсюда видно, что граничное значение всегда располагается на левом краю соответствующего интервала. Свойство гистограммы IntervalCount задает число внутренних (конечных) интервалов. С учетом двух крайних интервалов их общее число получается на 2 больше. Нумерация интервалов производится следующим образом: номер 0 имеет левый полубесконечный интервал, номера от 1 до IntervalCount — внутренние конечные интервалы, а номер IntervalCount + 1 — правый полубесконечный интервал.

Свойства-массивы позволяют получить информацию по каждому интервалу в отдельности. Count[i] дает общее количество значений, попавшее внутрь интервала номер i. Percent[i] дает долю количества значений в данном интервале от общего количества значений, накопленных в гистограмме. CumulativeCount[i] и CumulativePercent[i] дают сумму предыдущих значений по интервалам от 0 до i включительно. LowerBound[i] и UpperBound[i] дают значения нижней и верхней границы указанного интервала соответственно (снова напомним, что значение UpperBound[i] на самом деле принадлежит интервалу номер i + 1). Естественно, UpperBound[i] = LowerBound[i + 1] для всех i от 0 до IntervalCount. Для крайних «бесконечных» границ задаются очень большие отрицательное и положительное значения: –10300 и 10300 соответственно.

Свойство TotalCount дает общее количество значений, накопленных в гистограмме. Несмотря на то, что для свойств LowerBound, UpperBound и IntervalCount предусмотрены методы записи, имеющиеся в данный момент конкретные классы гистограмм эти возможности не используют. При записи значений в эти свойства ничего не происходит.

Поскольку класс THistogram абстрактный, создавать объекты этого класса невозможно. В программах следует использовать производные конкретные классы, построенные на его базе. Первый из них, TUniformHistogram, подразумевает равную длину внутренних интервалов:

TUniformHistogram = **class**(THistogram)

**public**

**constructor** Create(ALow, AStep : Double; AIntervalCount : Integer);

**procedure** Clear; **override**;

**procedure** AddData(d : Double); **override**;

**end**;

В конструкторе класса указываются следующие параметры: нижняя граница первого внутреннего интервала, ширина каждого из интервалов и их количество. К примеру, оператор:

Hist := TUniformHistogram.Create(0, 1, 20);

создаст объект-гистограмму, содержащую 20 интервалов шириной в 1 единицу каждый, причем первый интервал начинается с 0. Таким образом, граничными значениями интервалов гистограммы будут числа: 0, 1, …, 20.

Второй из производных классов — TArrayHistogram:

TArrayHistogram = **class**(THistogram)

**public**

**constructor** Create(ABounds : **array** **of** Double);

**procedure** Clear; **override**;

**procedure** AddData(d : Double); **override**;

**end**;

Границы внутренних интервалов задаются в произвольном числовом массиве. Элементы этого массива должны быть упорядочены по возрастанию. Число внутренних интервалов, естественно, на 1 меньше, чем число элементов массива. При использовании гистограмм желательно задавать число внутренних интервалов не меньше 4-5 и не больше 20-30. Слишком маленькое число интервалов не позволить в полной мере оценить распределение изучаемой величины (что является основной целью использования гистограммы), а слишком большое — сделает собранный материал труднообозримым.

## 1.6. Отображение статистики и гистограмм

Объекты статистики и гистограмм накапливают данные и выдают результаты, которые, разумеется, нуждаются в отображении на экране. Конечно, можно для этого воспользоваться стандартными процедурами и объектами вывода. Однако для еще б***о***льшего удобства в библиотеке предусмотрено несколько процедур для отображения статистики и гистограмм. Поскольку приложения могут разрабатываться для работы как в консольном, так и в оконном режиме, имеется два набора таких процедур.

Рассмотрим основные процедуры отображения статистики для консольного режима. Они все называются WriteStat и различаются типом параметра-статистики:

**procedure** WriteStat(Header : **string**; Stat : TStatistics); **overload**;

**procedure** WriteStat(Header : **string**; Stat : TIntervalStatistics); **overload**;

**procedure** WriteStat(Header : **string**; Stat : TTimeBetStatistics); **overload**;

**procedure** WriteStat(Header : **string**; Queue : TList); **overload**;

Первым параметром всех этих процедур является строка-заголовок, которая выводится перед остальными данными. Процедуры для параметров типа TStatistics и TTimeBetStatistics выводят следующую информацию: среднее значение, стандартное отклонение, минимальное и максимальное значения и общее количество значений. Эти данные связаны поясняющим текстом (по указанным выше причинам, англоязычным). Процедура для параметра типа TIntervalStatistics вместо количества значений выводит общее время сбора статистики и текущее значение наблюдаемой величины.

Для вывода статистики о работе списка предусмотрен отдельный вариант процедуры. Конечно, можно воспользоваться ранее указанными процедурами и отобразить отдельно статистику по длине списка и времени ожидания ячеек в нем, однако в большинстве случаев эта информация избыточна. Отдельная версия процедуры WriteStat для списка выводит следующую информацию: среднюю длину списка со стандартным отклонением, максимальную длину за время наблюдения и текущую длину, а также среднее время ожидания.

Рассмотрим некоторые примеры отображения статистики. Пример вывода точечной статистики:

Clients in bank time:

Average = 9.009 +- 5.777

Min = 2.507, max = 25.827

Total = 100 values

Пример вывода статистики списка:

Clients queue:

Average length = 1.024 +- 1.443

Max = 6, current = 0

Average waiting time = 5.190

Кроме этих вариантов, имеются процедуры вывода для других видов статистики и объектов моделирования. Они будут рассмотрены далее в соответствующих параграфах.

Процедура WriteHist предназначена для вывода на консольный экран гистограммы:

**procedure** WriteHist(Header : **string**; Hist : THistogram);

Эта процедура выводит на экран заголовок, а затем для каждого интервала: нижнюю и верхнюю границы, количество значений, попавшее в интервал, долю значений в интервале от общего их количества, и накопленную долю количества значений от начального интервала до текущего. Кроме того, для каждого интервала визуально отображается примерная доля количества значений и накопленная доля. Пример вывода гистограммы:

In bank time histogram:

-INF - 2.00 : 0 ( 0.00%), 0.00%

2.00 - 4.00 : 16 (16.00%), 16.00% \*\*\*\*\*\*

4.00 - 6.00 : 26 (26.00%), 42.00% \*\*\*\*\*\*\*\*\*\* O

6.00 - 8.00 : 14 (14.00%), 56.00% \*\*\*\*\*\* O

8.00 - 10.00 : 11 (11.00%), 67.00% \*\*\*\* O

10.00 - 12.00 : 10 (10.00%), 77.00% \*\*\*\* O

12.00 - 14.00 : 4 ( 4.00%), 81.00% \*\* O

14.00 - 16.00 : 4 ( 4.00%), 85.00% \*\* O

16.00 - 18.00 : 4 ( 4.00%), 89.00% \*\* O

18.00 - 20.00 : 4 ( 4.00%), 93.00% \*\* O

20.00 - 22.00 : 2 ( 2.00%), 95.00% \* O

22.00 - 24.00 : 4 ( 4.00%), 99.00% \*\* O

24.00 - 26.00 : 1 ( 1.00%), 100.00% O

26.00 - 28.00 : 0 ( 0.00%), 100.00% O

28.00 - +INF : 0 ( 0.00%), 100.00% O

Как видно из приведенного примера, визуальное отображение распределения весьма приблизительное, однако оно позволяет примерно оценить его особенности.

Пример проекта, в котором демонстрируется использование генератора случайных чисел, объектов сбора статистики и гистограмм и вывода их на консольный экран, находится в папке 1.6 Random.

Отображение объектов статистики в оконных приложениях выполняется с помощью объекта табличной сетки (TStringGrid). Для этого используются несколько разновидностей процедуры ShowStat:

**procedure** ShowStat(Grid : TStringGrid; **const** Headers : **array** **of** **string**;

**const** Stat : **array** **of** TStatistics); **overload**;

**procedure** ShowStat(Grid : TStringGrid; **const** Headers : **array** **of** **string**;

**const** Stat : **array** **of** TIntervalStatistics); **overload**;

**procedure** ShowStat(Grid : TStringGrid; **const** Headers : **array** **of** **string**;

**const** Stat : **array** **of** TTimeBetStatistics); **overload**;

**procedure** ShowStat(Grid : TStringGrid; **const** Headers : **array** **of** **string**;

**const** Queue : **array** **of** TList); **overload**;

Каждая из этих процедур используется для отображения в одной табличной сетке результатов сразу нескольких объектов статистики одного типа, указанных в массиве. Первым параметром является объект-сетка, вторым — массив строк, являющихся пояснительными надписями к каждой статистике, третьим — массив объектов статистики (или списков). Объект-сетка при разработке должен быть настроен следующим образом: число строк (свойство объекта RowCount) на 1 больше числа отображаемых статистик, число столбцов (ColCount) равно 7 для интервальной статистики и 6 для остальных. Желательно выделить по одному фиксированному столбцу и фиксированной строке (FixedColCount, FixedRowCount).

При запуске программы (лучше всего в процедуре события Create) следует настроить ширины столбцов всех статистик, пользуясь свойством ColWidths[i] объекта. Конкретные ширины столбцов лучше всего подбирать опытным путем исходя из выбранного шрифта. Так, к примеру, при использовании шрифта Tahoma размером 12 пунктов для статистики очереди следующие значения близки к оптимальным (в порядке возрастания индекса): 90, 70, 100, 80, 60, 80. Они задаются фрагментом кода, подобным следующему:

**procedure** TfrFlowLineVis.FormCreate(Sender: TObject);

**begin**

…

sgQueues.ColWidths[0] := 90;

sgQueues.ColWidths[1] := 70;

sgQueues.ColWidths[2] := 100;

sgQueues.ColWidths[3] := 80;

sgQueues.ColWidths[4] := 60;

sgQueues.ColWidths[5] := 80;

…

**end**;

Ссылки на сами объекты статистики должны быть помещены в массив сразу же после создания имитации:

**var**

QueStat : **array** [0 .. 2] **of** TList;

…

QueStat[0] := sim.Queue1;

QueStat[1] := sim.Queue2;

QueStat[2] := sim.Calendar;

Собственно отображение массива статистик производится посредством вызова процедуры ShowStat:

ShowStat(sgQueues, ['Очередь 1', 'Очередь 2', 'Календарь'], QueStat);

Как видно из этого примера, массив поясняющих строк можно создать непосредственно при вызове процедуры. К сожалению, сделать то же самое с массивом статистик нельзя (по крайней мере, в версии Delphi 7). Это создает некоторые неудобства, связанные с необходимостью уже указанного определения массива и его заполнения.

Пример отображения статистики:



Для отображения гистограмм используется иной подход. Во-первых, в этом случае следует использовать похожий, но другой объект TDrawGrid. Следует задать число его столбцов равным 6, при этом удобно выделить 2 фиксированных столбца. Число строк должно быть на 3 больше количества внутренних интервалов гистограммы.

Во-вторых, само отображение выполняется в процедуре события OnDrawCell данного объекта. Эта процедура содержит обращение к процедуре DrawHistCell:

**procedure** TfrFlowLineVis.dgHistogramDrawCell(Sender: TObject;

ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);

**begin**

**if** sim <> **nil** **then**

DrawHistCell(dgHistogram, ACol, ARow, Rect, sim.TimeHist);

**end**;

Как видно из примера, первым параметром процедуре передается объект-сетка, в котором происходит отображение. Далее ей передаются параметры процедуры события ACol, ARow, Rect (без изменения). Последний параметр – отображаемый объект-гистограмма.

Непосредственно отображение содержимого гистограммы инициируется посредством перерисовки сетки:

dgHistogram.Repaint;

Также полезно определить процедуру для события OnTopLeftChanged:

**procedure** TfrFlowLineVis.dgHistogramTopLeftChanged(Sender: TObject);

**begin**

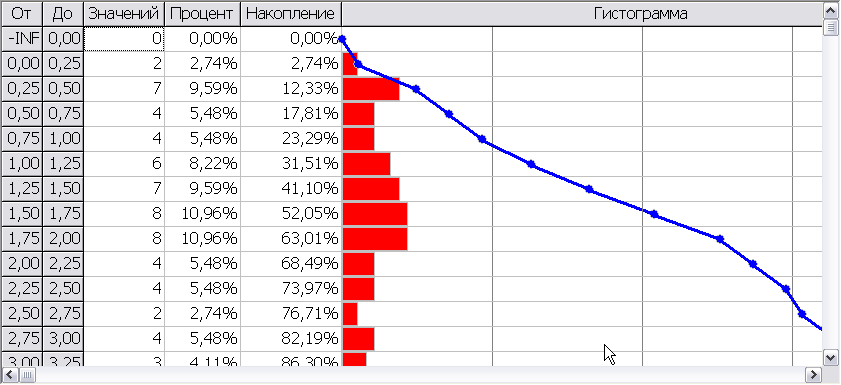
**if** sim <> **nil** **then**

dgHistogram.Repaint;

**end**;

Эта процедура вызывает перерисовку сетки при ее прокрутке в любом направлении. Если эту процедуру не определить, то в некоторых случаях содержимое отдельных ячеек сетки после прокрутки может отображаться некорректно вплоть до следующей перерисовки. Текст процедуры не нуждается в комментариях. Его можно просто переносить в другие проекты, не забывая, естественно, указать действительное имя объекта-сетки.

Пример отображения гистограммы:



# 2. Процессы

Основным элементом имитации является процесс. В объектах-процессах кодируются алгоритмы работы элементов имитации. Все классы процессов являются производными от класса TProcess, который, в свою очередь, является производным от класса сопрограмм TRunningObject. Сам класс TProcess является абстрактным, и объекты этого класса никогда не создаются. Вместо этого следует определять производные классы, в которых реализуются конкретные алгоритмы работы.

## 2.1. Классы процессов

Класс TProcess содержит большое количество полей и методов, поэтому будем рассматривать его по частям. Основные составляющие класса приведены ниже:

TProcess = **class**(TRunningObject)

**protected**

Parent : TSimulation;

**procedure** Init; **virtual**;

**procedure** Execute; **override**;

**procedure** RunProcess; **virtual**; **abstract**;

…

**public**

**constructor** Create;

**destructor** Destroy; **override**;

…

**end**;

Основной метод, с которым приходится иметь дело — RunProcess. Этот метод должен быть определен в любом производном процессе. Именно в этом методе кодируется алгоритм работы процесса. В простейшем случае производный класс содержит только этот переопределенный метод. При автоматическом создании заготовки метода в производном классе Delphi вставляет строку:

**inherited**;

Эту строку следует удалять, потому что попытка обращения к абстрактному методу генерирует исключение.

Метод Init следует использовать, если до запуска процесса в нем следует произвести некоторые дополнительные действия, например, создать какие-либо объекты, инициализировать поля и т. д. Первой строкой этого метода будет:

**inherited**;

В отличие от предыдущего метода, эта строка должна присутствовать обязательно, и она обязательно должна быть первой строкой метода.

Действия по инициализации процесса можно также выполнить, переопределяя конструктор Create, однако лучше использовать конструктор только в тех случаях, когда для создания процесса требуются некоторые параметры. И лучше ограничиться в конструкторе только тем, что записать эти параметры в соответствующих полях класса, а остальную инициализацию выполнять все же в методе Init. Это связано с тем, что в процессе имеется достаточно сложная поддерживающая инфраструктура, которая при работе конструктора еще не создана. А к начала работы метода Init она уже полностью построена. Если процесс имеет свой конструктор, его последней (и только последней!) строкой должна быть:

**inherited** Create;

Деструктор Destroy следует переопределять, если в процессе созданы некоторые объекты, нуждающиеся в удалении при удалении процесса. Аналогично, последней строкой этого метода должна быть строка

**inherited**;

Поле Parent содержит ссылку на процесс имитации (см. параграф 2.7), в рамках которой функционирует данный процесс. Метод Execute переопределен по сравнению с базовым классом TRunningObject. Ни в одном из производных классов он не должен переопределяться и вызываться.

## 2.2. Имитационное время и календарь процессов

В имитационном моделировании существенную роль играет время. В дискретном моделировании используется особое дискретно изменяющееся *имитационное* время. Это время доступно всем процессам посредством метода SimTime:

**function** SimTime : Double;

Этот метод не имеет параметров, поэтому обращение к нему выглядит как обращение к переменной. Однако присваивание ему нового значения времени, разумеется, запрещено. Управлять этим временем могут только предопределенные методы, отвечающие за процесс имитации, которые программист не может изменить.

Имитационное время имеет весьма косвенное отношение к реальному времени. Скорее, это некий независимый глобальный параметр, управляющий процессом имитации. Имитационное время измеряется в условных единицах времени, размер которых не ограничен ничем (кроме, естественно, того, чтобы быть положительными и конечными). Единица имитационного времени может соответствовать и секунде, и минут, и году…

Имитационное время в процессе моделирования изменяется *дискретно*, при смене состояний модели. Шаг изменения имитационного времени также произвольный — от 0 до любого конечного положительного значения. Нулевой шаг изменения обозначает, что на один и тот же момент времени намечается несколько действий по изменению состояния системы. Отрицательный шаг, то есть, обратный ход времени, в процессе имитации невозможен.

Несмотря на отмеченную независимость имитационного времени от реального, иногда возникает потребность их связать, например, чтобы отображать ход развития моделируемого процесса в реальном времени или в любом ином временн***о***м масштабе. Это возможно при использовании специального процесса-*визуализатора*, работа которого рассматривается в параграфе 2.8.

Следующим важным понятием имитационного моделирования является *событие*. Под событием будем понимать возобновление (или начало) работы того или иного процесса. Каждое событие происходит в некоторый заранее определенный момент имитационного времени. События исполняются в порядке возрастания (точнее, неубывания) их имитационного времени.

Последовательность возникновения событий управляется специальным списком — *календарем*. Этот список состоит из особых ячеек — *уведомлений о событиях*, являющихся экземплярами класса TEventNotice, производного от TLink. Подробно структуру этого класса рассматривать не будем, отметим лишь, что в нем указано имитационное время события и ссылка на событие. Список упорядочен по значениям имитационного времени. В процессе исполнения имитации календарь может дополняться планируемыми событиями, а обработанные события из него исключаются. При переходе к обработке нового события имитационное время устанавливается в значение, находящееся в его записи уведомления.

Каждый процесс в течение имитации может находиться в следующих четырех состояниях:

* *активный* процесс — это процесс, который исполняется в данный момент времени. Такой процесс всегда ровно один, и ему соответствует первая запись календаря;
* *приостановленный* процесс — это процесс, ожидающий возобновления, для которого в календаре имеется соответствующая запись (естественно, не первая);
* *пассивный* процесс — процесс, ожидающий возобновления, но не представленный в календаре;
* *завершенный* процесс — процесс, алгоритм работы которого завершен. Попытка возобновления его работы ведет к ошибке. В календаре также отсутствуют записи, ссылающиеся на такие процессы.

Информацию о состоянии процесса можно получить с помощью методов:

**function** Idle : Boolean;

**function** EventTime : Double;

Метод Idle возвращает значение True, если процесс находится в пассивном или завершенном состоянии, то есть ему не соответствует никакая запись календаря. Отличить одно от другого можно с помощью свойства Terminated, унаследованного от класса сопрограмм. Метод EventTime возвращает имитационное время события, связанного с данным процессом. Естественно, это возможно только для активного или приостановленного процесса. Если выполняется попытка получить время события для пассивного процесса, возвращается невозможное значение –10300.

Еще одна важная величина — *время запуска* процесса — записывается в соответствующее его поле:

StartingTime : Double;

В это поле записывается текущее имитационное время непосредственно перед запуском метода RunProcess. То есть, это время соответствует времени не создания процесса, а его запуска.

## 2.3. Планирование работы процессов

С помощью специальных методов процессы могут планировать работу как свою, так и других процессов. При выполнении планирования могут создаваться или удаляться записи календаря событий, либо существующие записи могут перемещаться на другие места (разумеется, в силу упорядоченности календаря последнее возможно только при изменении времени наступления соответствующего события). Создание новых записей календаря на время, меньшее, чем текущее имитационное время, невозможно. Если предпринимается попытка это сделать, планируемое время заменяется текущим.

В классе TProcess имеется довольно много методов, позволяющих планировать события:

**public**

**procedure** Activate; **overload**;

**procedure** ActivateAt(t : Double); **overload**;

**procedure** ActivatePriorAt(t : Double); **overload**;

**procedure** ActivateDelay(t : Double); **overload**;

**procedure** ActivatePriorDelay(t : Double); **overload**;

**procedure** ActivateAfter(p : TProcess); **overload**;

**procedure** ActivateBefore(p : TProcess); **overload**;

**procedure** Reactivate; **overload**;

**procedure** ReactivateAt(t : Double); **overload**;

**procedure** ReactivatePriorAt(t : Double); **overload**;

**procedure** ReactivateDelay(t : Double); **overload**;

**procedure** ReactivatePriorDelay(t : Double); **overload**;

**procedure** ReactivateAfter(p : TProcess); **overload**;

**procedure** ReactivateBefore(p : TProcess); **overload**;

**procedure** Passivate;

**procedure** Hold(t : Double);

**protected**

**procedure** Activate(**const** Procs : **array** **of** TProcess); **overload**;

**procedure** Activate(Procs : TList); **overload**;

**procedure** ActivateAt(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivateAt(Procs : TList; t : Double); **overload**;

**procedure** ActivatePriorAt(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivatePriorAt(Procs : TList; t : Double); **overload**;

**procedure** ActivateDelay(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivateDelay(Procs : TList; t : Double); **overload**;

**procedure** ActivatePriorDelay(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivatePriorDelay(Procs : TList; t : Double); **overload**;

**procedure** ActivateAfter(**const** Procs : **array** **of** TProcess;

p : TProcess); **overload**;

**procedure** ActivateAfter(Procs : TList; p : TProcess); **overload**;

**procedure** ActivateBefore(**const** Procs : **array** **of** TProcess;

p : TProcess); **overload**;

**procedure** ActivateBefore(Procs : TList; p : TProcess); **overload**;

**procedure** ActivateAll(**const** Procs : **array** **of** TProcess); **overload**;

**procedure** ActivateAll(Procs : TList); **overload**;

**procedure** ActivateAllAt(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivateAllAt(Procs : TList; t : Double); **overload**;

**procedure** ActivateAllPriorAt(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivateAllPriorAt(Procs : TList; t : Double); **overload**;

**procedure** ActivateAllDelay(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivateAllDelay(Procs : TList; t : Double); **overload**;

**procedure** ActivateAllPriorDelay(**const** Procs : **array** **of** TProcess;

t : Double); **overload**;

**procedure** ActivateAllPriorDelay(Procs : TList; t : Double); **overload**;

**procedure** ActivateAllAfter(**const** Procs : **array** **of** TProcess;

p : TProcess); **overload**;

**procedure** ActivateAllAfter(Procs : TList; p : TProcess); **overload**;

**procedure** ActivateAllBefore(**const** Procs : **array** **of** TProcess;

p : TProcess); **overload**;

**procedure** ActivateAllBefore(Procs : TList; p : TProcess); **overload**;

**procedure** Reactivate(Proc : TProcess); **overload**;

**procedure** ReactivateAt(Proc : TProcess; t : Double); **overload**;

**procedure** ReactivatePriorAt(Proc : TProcess; t : Double); **overload**;

**procedure** ReactivateDelay(Proc : TProcess; t : Double); **overload**;

**procedure** ReactivatePriorDelay(Proc : TProcess; t : Double); **overload**;

**procedure** ReactivateAfter(Proc, p : TProcess); **overload**;

**procedure** ReactivateBefore(Proc, p : TProcess); **overload**;

Сначала рассмотрим методы, объявленные в открытой части класса. Первую их группу составляют методы Activate\*\*\* без параметров или с одним параметром. Эти методы *активируют*, то есть переводят в приостановленное или активное состояние объект-процесс, для которого они вызываются. Если процесс уже находится в активном или приостановленном состоянии, эти методы не производят никаких действий. Разные методы позволяют разными способами указать положение в календаре уведомления о событии для процесса.

Метод Activate создает событие для процесса *первым* элементом календаря, при этом текущий активный процесс приостанавливается, а активируемый процесс переходит в активное состояние, то есть возобновляет (или начинает) свою работу.

Метод ActivateAt указывает время, на которое планируется активация процесса. Если в календаре уже есть запланированные события на то же самое время, новая запись встанет после последней из них. Метод ActivatePriorAt выполняет аналогичное действие, но запись о событии помещает перед всеми записями с тем же имитационным временем события. Если в качестве его параметра указать текущее имитационное время, активный процесс будет приостановлен, передав управление активируемому.

Методы ActivateDelay и ActivatePriorDelay активируют процесс на время, б***о***льшее текущего имитационного времени на заданный промежуток. Фактически, вызов proc.ActivateDelay(dt) эквивалентен вызову proc.ActivateAt(SimTime + dt). Разница между методами ActivateDelay и ActivatePriorDelay та же, что и между предыдущими.

Метод ActivateAfter помещает запись уведомления о событии в календарь непосредственно *после* записи для процесса, указанного параметром, и задает этому событию то же время. Аналогично, метод ActivateBefore создает уведомление о событии с тем же временем, что и для процесса-параметра, и помещает его *перед* событием указанного процесса. В случае, когда процесс активируется перед текущим активным, активный процесс приостанавливается, и активированный начинает свое исполнение. Фактически, вызов proc.Activate эквивалентен вызову proc.ActivateBefore(CurrentProc), где CurrentProc — ссылка на текущий активный процесс.

Заметим, что вызовы proc.ActivateDelay(0) и proc.ActivateAfter(Self) дают очень похожий результат: в обоих случаях процесс proc активируется при текущем значении имитационного времени. Однако между ними есть разница: если после текущего активного процесса на то же самое время запланированы еще некоторые события, то первый вызов поместит новую запись о событии после них всех, а второй — непосредственно после текущей записи, то есть, перед ними всеми.

Методы группы Reactivate\*\*\* для пассивных процессов выполняют те же действия, что и аналогичные методы предыдущей группы. Фактически, если процесс пассивен, производится вызов соответствующего метода из группы Activate\*\*\*. Если же процесс активен или приостановлен, его запись уведомления о событии перемещается в календаре на новое место, соответствующее новому времени события. При этом, если первой становится другая запись уведомления (то есть, если либо другая запись встает на первое место, либо текущая первая запись перемещается на другое место), происходит переключение процессов: текущий активный процесс приостанавливается, а процесс, чья запись оказалась первой, возобновляется.

Метод Passivate переводит указанный процесс в *пассивное* состояние, то есть, его запись уведомления о событии исключается из календаря. Если в пассивное состояние переводится активный процесс, активируется процесс, расположенный в календаре следом за текущим. Метод Hold планирует возобновление процесса на время, отстоящее от текущего имитационного времени на заданный промежуток. Вызов proc.Hold(dt) эквивалентен вызову proc.ReactivateDelay(dt). Вызов этого метода для активного процесса приостанавливает его исполнение на заданный промежуток имитационного времени.

Нелогичный, на первый взгляд, вызов Hold(0) на самом деле тоже имеет смысл. При выполнении такого вызова уведомление о событии для текущего процесса устанавливается после всех других событий, назначенных на это же время. То есть, такой вызов позволяет активировать все ожидающие процессы, запланированные на это время, после чего продолжить работу текущего процесса.

Методы Activate\*\*\* из защищенной части класса позволяют указывать наборы активируемых процессов. Набор можно указывать либо в виде массива процессов, либо в виде списка. Каждый из этих методов просматривает процессы в порядке их нахождения в наборе и активирует *первый* из них, находящийся в пассивном состоянии. Если элемент списка не является процессом, он игнорируется.

Особенностью этих методов является то, что вызывать их можно только из методов активного процесса. То есть, к примеру, вызов proc.ActivateDelay(ProcList, dt) запрещен. Нарушение этого правила порождает исключение. Таким способом можно активировать и единственный процесс, если поместить его в автоматически создаваемый массив: ActivateDelay([proc], dt).

Если набор процессов задается в виде массива, он должен быть определен как массив объектов класса TProcess, а не того класса, экземплярами которого они в действительности являются. Это связано с особенностями передачи параметров-массивов в языке Delphi (по крайней мере, до версии 7). Такое объявление массива никак не сказывается на корректности работы процессов, однако в некоторых случаях при обращении к такому процессу может потребоваться приведение ссылки на него к действительному классу с помощью операции **as**.

Методы ActivateAll\*\*\* также позволяют указывать наборы активируемых процессов. Их отличие о предыдущих состоит в том, что активируются *все* процессы из набора, находящиеся в пассивном состоянии. При этом для всех активированных процессов задается одно и то же время события, а их порядок в календаре совпадает с их взаимным порядком в исходном наборе. Эти методы, как и предыдущие, должны вызываться только из активного процесса.

Методы Reactivate\*\*\* из защищенной части класса предоставляют альтернативный способ вызова одноименных открытых методов. Так, к примеру, вызов ReactivateAt(proc, t) эквивалентен вызову proc.ReactivateAt(t). Но, как и в предыдущих случаях, вызывать их можно только из активного процесса.

## 2.4. Создание процессов

Само по себе создание процессов не представляет сложности. Для этого следует воспользоваться либо стандартным конструктором Create, либо конструктором с параметрами, указанным в определении класса процесса. Однако создание процесса еще не инициирует его исполнения. Чтобы процесс начал работу, нужно поместить в календарь запись события для его активации. Это можно сделать, например, следующим способом:

Proc := TMyProcess.Create;

Proc.ActivateDelay(0);

В приведенном примере вновь созданный процесс активируется в тот же момент имитационного времени, когда он был создан. Можно объединить эти два действия в одном операторе:

TMyProcess.Create.ActivateDelay(0);

Правда, в отличие от предыдущего примера, ссылка на вновь созданный процесс не сохраняется ни в какой переменной. Как обеспечить корректное удаление объекта-процесса после его завершения, рассматривается в следующем параграфе.

В задачах имитации часто возникает потребность в создании ряда однотипных процессов через некоторые промежутки времени. Это удобно делать с помощью двух типичных способов.

Первый способ предусматривает создание нового процесса сразу же после запуска текущего процесса и его активацию через некоторое время. В этом случае начальная часть основного метода выглядит так:

**procedure** TPiece.RunProcess;

**var**

*// Необходимые локальные переменные*

**begin**

*// Запланировать начало исполнения следующего процесса*

TPiece.Create.ActivateDelay(rndPiece.Exponential(MeanPieceInterval));

*// Алгоритм работы процесса*

**end**;

Второй способ подразумевает наличие специального процесса — *генератора*. Алгоритм генератора предусматривает циклически происходящее создание нового процесса с немедленной активацией, за которым следует задержка на время, предусмотренное формулировкой задачи. Класс процесса-генератора определяется просто:

TGenerator = **class**(TProcess)

**protected**

**procedure** RunProcess; **override**;

**end**;

Алгоритм работы генератора также чрезвычайно прост:

**procedure** TGenerator.RunProcess;

**begin**

**while** True **do**

**begin**

ClearFinished;

*// Создать новый процесс и поместить его в систему*

TCustomer.Create.ActivateDelay(0);

*// Подождать перед созданием следующего*

Hold(rndCust.Exponential(MeanCustInterval));

**end**;

**end**;

В данном примере используется «бесконечный» цикл, следовательно, завершение процесса должно выполняться какими-то иными методами. Обычно эта задача возлагается на процесс-имитацию. В то же время цикл создания процессов может быть ограничен каким-то условием или счетчиком, а также иметь более сложную структуру.

Пояснение метода ClearFinished отложим до следующего параграфа.

## 2.5. Завершение процессов

Одним из важнейших правил при работе с объектами является то, что любой объект, созданный в программе, после окончания работы с ним необходимо удалять, освобождая занятую им память. Если этого не сделать, неиспользуемый объект будет продолжать занимать выделенный ему объем памяти, а обратиться к нему и использовать в любом другом качестве будет невозможно. Когда в памяти оказывается много не удаленных объектов, возникает ситуация, называемая *утечкой памяти*, когда программа занимает большой объем памяти, а воспользоваться ей не может. Утечки памяти могут ощутимо негативно повлиять на общую производительность системы.

При работе с одиночными объектами отслеживать их создание и удаление сравнительно просто. Однако в задачах имитации приходится иметь дело с большим количеством работающих объектов-процессов (в приводимых ниже примерах их количество исчисляется сотнями и даже тысячами). Если не предпринимать специальных усилий по автоматическому удалению завершенных процессов, это грозит значительными утечками памяти. Ситуация усугубляется за счет того, что каждый процесс занимает значительный объем памяти для организации собственного стека вызова процедур (в нынешнем варианте это 64 Кбайта для каждого процесса).

Для более удобной и предсказуемой работы вся ответственность за удаление процессов возлагается на процесс имитации, подробно рассматриваемый далее в параграфе 2.7.

Заметим, что удалять приходится не только завершенные процессы. Дело в том, что при завершении имитации в ней могут еще остаться приостановленные и пассивные процессы. Естественно, поскольку имитация завершена, дальнейшая потребность в них отпадает, и их следует удалять, как и завершенные процессы. Попутно отметим, что удаление процесса, не закончившего работу своего основного метода RunProcess, не представляет проблем: он просто выполнит не все действия, предусмотренные алгоритмом.

Чтобы разобраться с механизмом удаления процессов, проведем следующую классификацию процессов по возможности доступа к ним:

* *статические* процессы — это процессы, прямые ссылки на которые включены в класс имитации. Удаление таких процессов не составляет труда: достаточно написать соответствующие операторы в деструкторе имитации;
* *связанные* процессы — это пассивные процессы, включенные в один из списков, ссылка на которые имеется в процессе имитации или в одном из его объектов-полей. Удаление таких процессов также не вызывает проблем: при удалении списка в деструкторе имитации все процессы, находящиеся в нем, удаляются автоматически;
* *свободные* процессы — это процессы, на которые нет прямых ссылок, находящиеся в активном или приостановленном состоянии;
* *завершенные* процессы.

Как видно из этой классификации, наибольшую проблему представляют свободные и завершенные процессы. Для их автоматического учета в классе имитации TSimulation (см. параграф 2.7) предусмотрены два списка:

TSimulation = **class**(TProcess)

**protected**

RunningObjects : TList;

FinishedObjects : TList;

**end**;

Список RunningObjects предназначен для учета имеющихся в системе свободных процессов, а список FinishedObjects — для завершенных. Процесс включается в список посредством исполнения своих методов:

**procedure** StartRunning;

**procedure** Finish;

Поскольку включение элемента в список означает автоматическое его исключение из того, в котором он ранее находился (если таковой был), метод StartRunning можно использовать для одновременного извлечения объекта из списка, в котором он находился (например, очереди ожидания). Фактически, любой процесс, на который в процессе имитации нет прямой ссылки, должен извлекать себя из списка именно таким образом. Что касается процессов, на которые ссылки имеются, то они могут для этой цели пользоваться как методом StartRunning, так и унаследованным от класса TLink методом Remove. Кроме того, любой процесс, на который нет прямой ссылки, должен завершать свою работу вызовом метода Finish.

При выполнении деструктора имитации производится автоматическое удаление всех процессов, входящих в указанные списки. Однако может возникнуть проблема при удалении статических процессов, которые оказались включены в любой из этих списков (а также в любой другой список). Как уже упоминалось, при удалении списка удаляются все включенные в него объекты. Однако если после этого попытаться удалить объект через прямую ссылку на него, возникнет исключение, поскольку дважды удалить один и тот же объект невозможно.

Чтобы устранить эту проблему, в деструкторе имитации следует придерживаться следующего правила. Сначала следует выполнять удаление всех статических объектов. Если такой объект при этом оказался включен в какой-либо список, то его деструктор автоматически исключает его из списка. Следовательно, при последующем удалении списка повторной попытки удаления объекта уже не произойдет. Это относится не только к собственно спискам, то есть объектам класса TList, но и другим объектам, в состав которых входят списки: ресурсам и затворам. Взаимный порядок удаления списков и объектов, их содержащих, не имеет значения, потому что объект может входить в состав не более чем одного списка. Удаление списков RunningObjects и FinishedObjects производится автоматически в деструкторе, унаследованном от класса TSimulation, который должен вызываться последним в деструкторе производного объекта-симуляции.

Указанные меры, конечно, значительно облегчают процесс автоматического удаления объектов. Однако остается еще одна проблема. Если пользоваться только этими средствами, то удаление всех завершенных процессов откладывается до окончания процесса имитации. При этом все завершенные процессы суммарно будут занимать значительный объем памяти. Чтобы избежать таких затрат, желательно периодически удалять все завершенные к данному моменту процессы. Для этого в классе TProcess имеется метод:

**procedure** ClearFinished;

Этот метод удаляет все процессы из списка FinishedObjects активной имитации. Для регулярной очистки этого списка следует предусмотреть периодический вызов этого метода в каком-либо из процессов (или даже не в одном). Удобный вариант — выполнять очистку в процессе-генераторе, как было показано в последнем примере предыдущего параграфа. В этом случае перед созданием нового процесса предварительно удаляются уже завершенные. Этот вариант удобен тем, что общее количество завершенных процессов почти равно количеству созданных (на самом деле, несколько меньше — на количество процессов, находящихся в состоянии исполнения). Следовательно, заканчиваться процессы будут в среднем примерно с такой же периодичностью, как и создавались, а это значит, что между каждой парой создаваемых процессов успевает завершиться немного процессов (в среднем меньше 1), и суммарный объем занятой ими памяти будет невелик.

## 2.6. Типовые роли процессов в имитации

Как отмечалось выше, алгоритм метода RunProcess может быть произвольным, с применением любых алгоритмических конструкций. Однако, как и в традиционном программировании, процесс разработки будет более простым, целенаправленным и предсказуемым, если использовать некоторые типовые приемы. Использование этих приемов основано на роли, которую тот или иной процесс играет в имитации.

Первая типовая роль процесса — *компонент*. Процессы-компоненты, как правило, составляют абсолютное большинство всех процессов имитации. Они периодически, один за другим, создаются, выполняют предусмотренные алгоритмом действия и завершают свою работу. Как правило, процессы-компоненты не бывают статическими. В течение времени своего исполнения они находятся то в связанном, то в свободном состоянии, и в конце алгоритма завершаются.

Действия процессов часто имитируются с помощью метода Hold. То есть, процесс, приступая к действию, вызывает этот метод, что вызывает приостановку процесса на заданный промежуток имитационного времени. Когда этот интервал времени истекает, процесс возобновляет свою работу, и это считается окончанием действия. Такое действие будем называть *обычным* или *самостоятельным*. По окончании действия процесс может произвести некоторые глобальные изменения в системе, что соответствует общему подходу дискретной имитации — любые изменения в системе происходят дискретно в отдельные моменты времени.

Если процесс по каким-либо причинам не может немедленно приступить к исполнению действия, он должен перейти в состояние ожидания. Это происходит посредством постановки процесса в некоторую очередь (список) ожидания и его перевода в пассивное состояние, что происходит при исполнении метода Wait:

**procedure** Wait(l : TList);

Те же самые действия нетрудно определить непосредственно. И действительно, определение этого метода выглядит следующим образом:

**procedure** TProcess.Wait(l: TList);

**begin**

Insert(l);

Passivate;

**end**;

Однако нередко приходится формулировать более сложные алгоритмы ожидания. Например, если требуется ожидать до тех пор, пока значение некоторой переменной x не станет положительным, можно написать такой фрагмент кода:

Insert(Queue);

**while** x <= 0 **do**

Passivate;

Этот фрагмент кода предусматривает, что даже если данный процесс будет в течение времени ожидания активирован, он все равно не продолжит свою работу до тех пор, пока не будет выполняться указанное условие.

Автоматизировать процесс ожидания и возобновления процесса можно с помощью ресурсов и затворов, о чем будет рассказано в главе 4.

Заметим также, что посредством постановки в очередь ожидания можно реализовать *действия с неопределенной продолжительностью*, то есть, действия, в момент начала которых неизвестна их длительность. В случае выполнения такого действия процесс просто встает в очередь ожидания (как вариант — записывает ссылку на себя в какую-то статическую переменную и переходит в пассивное состояние). Окончание действия определяется каким-то друним процессом, который возобновляет процесс, выполняющий действие (или все такие процессы) с помощью методов семейства Activate\*\*\* или ActivateAll\*\*\*.

Другой способ выполнения действия предполагает в нем участие другого процесса — *обслуживающего*. Соответственно, данную разновидность действия назовем *обслуживающим*. Типичные варианты построения обслуживающих процессов обсудим несколько ниже. Здесь лишь заметим, что процесс-компонент для выполнения обслуживающего действия встает в очередь ожидания и активирует обслуживающий процесс:

Server.ActivateDelay(0);

Wait(Queue);

Если обслуживающий процесс в данный момент пассивен, он будет активирован и приступит к исполнению действия. Обычно он берет на обслуживание первый компонент из очереди (не обязательно таковым окажется именно тот, который его активировал), выполняет с ним действие, после чего активирует компонент, предоставляя ему возможность дальнейшего исполнения. Таким образом, в процессе-компоненте обслуживающее действие никак не формулируется, все выполняет обслуживающий процесс.

Как обсуждалось в параграфе 2.4, для регулярного создания процессов-компонентов удобно использовать два типовых приема. Первый состоит в том, что каждый запущенный компонент создает следующий компонент и активирует его с некоторой задержкой. Второй прием предполагает наличие отдельного процесса-генератора. Собственно, *генератор* и является второй типовой ролью процесса в имитации. Основные приемы построения его алгоритма обсуждались ранее.

Третья роль — уже упоминавшийся *обслуживающий* процесс. Как правило, этот процесс ориентирован на выполнение большого числа обслуживающих действий. Поэтому его алгоритм является циклическим. Рассмотрим некоторые его типовые составляющие на следующем примере:

**procedure** TDevice.RunProcess;

**var**

Piece : TComponent;

par : TMySimulation;

**begin**

par := Parent **as** TMySimulation;

**while** True **do**

**begin**

*// Ожидать появления компонентов в очереди*

**while** par.Queue.Empty **do**

Passivate;

*// Извлечь первый компонент*

Piece := par.Queue.First **as** TComponent;

Piece.StartRunning;

*// Выполнить обслуживание*

Hold(rndDevice.Exponential(MeanDeviceTime));

*// Освободить компонент, предоставив ему возможность продолжить работу*

Piece.ActivateDelay(0);

**end**;

**end**;

Первая строка алгоритма записывает в переменную par ссылку на процесс имитации для сокращения последующих обращений к ее компонентам. Сам алгоритм представляет собой «бесконечный» цикл. Его тело начинается с ожидания появления в очереди хотя бы одного компонента. Как только очередь становится непустой, внутренний цикл ожидания заканчивается, и процесс извлекает из очереди компонент, находящийся в ней на первом месте. Затем имитируется действие обслуживания посредством выполнения задержки. По завершении обслуживания процесс-компонент активируется, чтобы он мог продолжить свою работу. Если данное действие является последним в списке действий, предусмотренных для компонента, можно его поместить в очередь завершенных (Piece.Finish) или совсем удалить (Piece.Free).

Если по завершении обслуживания процесс не может сразу освободить компонент (например, если моделируется ситуация, когда данный компонент не может быть передан на следующее обслуживающее устройство по причине его поломки или занятости), то он *блокируется*. Блокировка моделируется аналогичным циклом ожидания по условию блокировки. По завершении цикла ожидания процесс активирует обслуживаемый компонент.

Естественно, такое разделение процессов по ролям не является абсолютным. Нетрудно придумать ситуации, когда один процесс на разных этапах своего исполнения выполняет разные роли. Можно придумать и другие типовые модели поведения процессов. Кроме того, существует альтернативный вариант обслуживающим действиям: аналогичные ситуации можно имитировать и без обслуживающих процессов, пользуясь ресурсами и затворами, рассматриваемыми в главе 4.

Еще одна возможность состоит в том, что компоненты имитации в ряде случаев не обязательно выполнять как процессы. Это возможно, если процесс сам по себе не выполняет никаких действий (фактически, если в его алгоритме не предусматривается никаких задержек, и все время своего существования компонент проводит, ожидая обслуживания в какой-либо очереди или обслуживаясь другим процессом). В таких случаях оказывается достаточным определить класс, производный от TLink, в котором предусмотреть поля для размещения необходимой информации. В еще более простых случаях, когда эта информация не нужна, можно непосредственно пользоваться объектами класса TLink, или вообще не создавать компонентов и использовать целочисленные счетчики для учета их количества.

## 2.7. Процесс имитации

Важнейшую роль в выполнении имитации играет процесс-*имитация*. Он создает все необходимые для выполнения имитации объекты, инициирует выполнение процессов, обеспечивает сбор статистики, завершение имитации и удаление объектов после этого. В отличие от остальных процессов, процесс-имитация всегда ровно один (впрочем, возможно наличие вложенных имитаций, когда некоторая имитация исполняется как составная часть процесса другой имитации; однако в данной версии библиотеки классов такая возможность отсутствует). Процесс-имитация строится как объект класса, производного от класса TSimulation, который, в свою очередь, является производным от класса TProcess:

TSimulation = **class**(TProcess)

**public**

Calendar : TList;

**constructor** Create;

**destructor** Destroy; **override**;

**function** SimTime : Double; **override**;

**procedure** StopStat; **virtual**;

**procedure** ClearStat; **virtual**;

**property** LastCleared : Double **read**;

**protected**

RunningObjects : TList;

FinishedObjects : TList;

**procedure** RunSimulation; **virtual**; **abstract**;

**procedure** Init; **override**;

**procedure** Finalize; **virtual**;

**end**;

Класс TSimulation является абстрактным, поэтому создать его экземпляры невозможно. Любая программа имитации должна построить производный от него класс, в котором переопределить основные методы.

Сначала рассмотрим содержимое защищенной части класса. Списки RunningObjects и FinishedObjects уже обсуждались в параграфе 2.5. Абстрактный метод RunSimulation должен быть переопределен в производном классе. В нем реализуется алгоритм работы имитации подобно тому, как в классах-процессах переопределяется метод RunProcess. Метод Init, как правило, тоже переопределяется. В нем записывается создание компонентов класса, необходимых для работы имитации. Метод Finalize переопределяется, если нужно выполнить некоторые завершающие действия после остановки имитации (но до исполнения деструктора).

В открытой части класса находится список Calendar, который представляет собой подробно рассмотренный ранее календарь событий. Конструктор Create переопределяется, если при создании имитации требуется указать некоторые параметры, влияющие на ее работу. В переопределяемом деструкторе производного класса следует записать удаление объектов, созданных при запуске имитации. Требования по размещению в конструкторе, деструкторе и методе Init вызова унаследованного метода те же, что и для процессов.

Метод StopStat переопределяется, чтобы скорректировать к текущему имитационному времени объекты-статистики, имеющиеся в имитации. Метод ClearStat следует переопределить, чтобы очистить содержимое объектов-статистик. Этот метод запоминает время очистки статистики в свойстве, которое называется LastCleared.

Метод SimTime в комментариях уже не нуждается. Собственно, именно к нему и обращаются одноименные методы остальных процессов.

Рассмотрим достаточно типовой пример построения имитации. Предположим, в имитации имеется одна очередь и один обслуживающий процесс, а также один объект интервальной статистики. Компоненты создаются генератором. Описания классов для процессов приводить не будем.

Вначале определим класс имитации:

TModel = **class**(TSimulation)

**public**

Queue : TList;

Stat : TIntervalStatistics;

Generator : TGenerator;

Server : TServer;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

В методах Init и Destroy определим создание и удаление элементов имитации:

**procedure** TModel.Init;

**begin**

**inherited**;

Queue := TList.Create;

Stat := TIntervalStatistics.Create(0);

Generator := TGenerator.Create;

Server := TServer.Create;

**end**;

**destructor** TModel.Destroy;

**begin**

Server.Free;

Generator.Free;

Stat.Free;

Queue.Free;

**inherited**;

**end**;

В имитации есть два объекта, подлежащих коррекции статистики: это собственно статистика Stat и очередь Queue. Этот факт находит отражение в процедуре StopStat:

**procedure** TModel.StopStat;

**begin**

**inherited**;

Queue.StopStat(SimTime);

Stat.StopStat(SimTime);

**end**;

Собственно выполнение имитации состоит в том, что активируется генератор, после чего процесс имитации приостанавливается на время, необходимое для ее исполнения. По истечении этого времени корректируется статистика:

**procedure** TModel.RunSimulation;

**begin**

Generator.ActivateDelay(0);

Hold(SimulationTime);

StopStat;

**end**;

Этот вариант подразумевает заранее заданное время окончания имитации. Возможны и другие варианты, например, когда через систему пройдет заданное число компонентов. В этом случае процесс имитации должен приостановиться с помощью метода Passivate, а один из процессов, входящих в имитацию, обнаружив условие, соответствующее окончанию имитации, должен возобновить ее посредством вызова par.ActivateDelay(0).

Работа главной программы с имитацией организована просто:

mdl := TModel.Create;

SwitchTo(mdl);

*// Отображение статистики по завершении имитации*

mdl.Free;

Однократное исполнение процесса имитации называется *прогоном*. Все объекты имитации предназначены для однократного прогона, который начинается со значения имитационного времени, равного 0. Если требуется получить результаты серии из нескольких прогонов, для каждого из них следует заново создавать процесс имитации и запускать его, а после сбора результатов — удалять. Если по условиям задачи требуется начинать имитацию не с нулевого значения имитационного времени, а с какого-либо другого (обязательно положительного), в ее алгоритме следует предусмотреть пропуск заданного времени перед началом имитационных действий (например, с помощью метода Hold).

## 2.8. Визуализация имитации

При обычном исполнении имитационного прогона основной поток программы создает объект-имитацию, передает ему управление и получает его обратно только по завершении прогона. Вся информация, которую можно при этом извлечь — это статистика, собранная в процессе прогона имитации. Однако в ряде случаев бывает полезно (да и просто интересно) отслеживать исполнение имитации непосредственно в ее процессе. При этом, в частности, удается обнаруживать ошибки в формулировках алгоритма, которые не видны при изучении статистических результатов прогона. Возможность наблюдения за ходом имитации присутствует в классе TSimulation и называется *визуализацией*.

Возможность визуализации поддерживается с помощью специального процесса — *визуализатора*. Этот процесс является объектом класса TVisualizator, производного от TProcess. Его работа состоит в том, что он активируется с заданным интервалом имитационного времени и передает управление основному потоку программы. Программа, в свою очередь, должна при этом собрать информацию о состоянии имитации в текущий момент времени, каким-то образом отобразить ее и вернуть управление визуализатору. После возобновления он планирует свою активацию через заданный промежуток времени и снова переходит в состояние ожидания.

Процесс-визуализатор создается в методе Init имитации с помощью метода:

**procedure** MakeVisualizator(dt : Double);

Параметром этой процедуры задается промежуток между моментами срабатывания. Ссылка на созданный в процедуре инициализации визуализатор записывается в поле процесса-имитации:

Visualizator : TVisualizator;

Этот процесс автоматически активируется при запуске имитации. Если процесс визуализации создается позже, программист сам должен позаботиться о его активации. Если процедура MakeVisualizator вызывается, когда процесс-визуализатор уже существует, новый процесс не создается, а вместо этого изменяется его шаг времени срабатывания.

В самом классе TVisualizator интерес представляет единственное поле:

DeltaT : Double;

Это поле, как нетрудно догадаться, как раз и определяет интервал имитационного времени срабатывания визуализатора.

Главный поток программы возвращает управление имитации посредством вызова глобальной процедуры:

**procedure** RunSimulation(sim : TSimulation);

Визуализация при разработке консольных приложений затруднена, поскольку в таких приложениях отсутствуют удобные средства оперативного отображения информации. Гораздо удобнее выполнять визуализацию при разработке оконных приложений. Для этого удобно использовать объект-таймер, устанавливая масштаб соответствия имитационного и реального времени: шаг имитационного времени визуализатора соответствует интервалу реального времени таймера.

В процедуре события таймера следует предусмотреть следующие основные действия: коррекцию статистики имитации к текущему времени, отображение статистики и иных данных о состоянии имитации на визуальных элементах формы (отображение статистики и гистограмм в табличных сетках рассматривалось ранее, некоторые другие приемы отображения рассмотрим далее), проверка завершения имитации, и, в зависимости от ее результатов, либо передачу управления имитации для продолжения прогона, либо остановку таймера. В общем, структуру этой процедуры можно описать следующим образом:

**procedure** TfrMySim.tmSimTimer(Sender: TObject);

**begin**

*// Коррекция статистики*

sim.StopStat;

*// Отображение состояния имитации*

lbState.Caption := sim.StateValue;

*// Отображение статистики*

ShowStat(sgStat, ['Статистика'], sim.Stat);

*// Отображение гистограммы*

dgHist.Repaint;

*// Проверка завершения имитации*

**if** sim.Terminated **then**

*// Если завершена — отключить таймер*

tmSim.Enabled := False

**else**

*// Если нет, продолжить имитацию*

RunSimulation(sim);

**end**;

Для удобства многократных прогонов имитации в программе следует также предусмотреть создание имитации и его запуск с одновременным включением таймера. Удобно это привязать к событию для кнопки с надписью (кто бы мог подумать!) «Пуск»:

**procedure** TfrMySim.btStartClick(Sender: TObject);

**begin**

*// Если процесс визуализации уже исполняется*

**if** tmSim.Enabled **then**

*// Остановить его*

tmSim.Enabled := False

**else**

**begin**

*// Удалить предыдущую имитацию (если есть)*

sim.Free;

*// Включить таймер*

tmSim.Enabled := True;

*// Создать новую имитацию*

sim := TMySimulation.Create;

*// Запустить ее*

SwitchTo(sim);

**end**;

**end**;

В вышеприведенном тексте пояснений требует, пожалуй, только удаление предыдущей имитации перед созданием новой. Сама необходимость удаления имитации вопросов не вызывает, но почему только перед созданием новой? Дело в том, что, если процесс имитации будет удален непосредственно после своего завершения, последующие попытки обращения к завершенной и удаленной имитации для отображения ее состоянию могут привести к многократным исключениям. Поэтому остановленную имитацию следует сохранять до тех пор, пока в ней окончательно не исчезнет необходимость. А произойдет это только при запуске новой имитации.

Что касается удаления несуществующей имитации перед запуском первого прогона, то, как известно, стандартный метод всех объектов Free корректно обрабатывает случай, когда ссылка на объект содержит значение **nil**. Следовательно, надо позаботиться, чтобы изначально при запуске программы в ней находилось именно это значение. Это условие будет выполнено автоматически, если ссылка на процесс имитации будет глобальной статической переменной — такие переменные получают нулевые значения при запуске программы.

Отображение состояния имитации проще всего выполнять с помощью объектов-надписей (Label), как было показано в вышеприведенном примере. Для этого можно воспользоваться стандартными функциями преобразования числовых данных в текст IntToStr и FloatToStr. Однако в ряде случаев можно воспользоваться более наглядным представлением.

Если требуется отобразить целочисленную величину, принимающую не слишком большие значения, например, длину очереди, можно воспользоваться функцией:

**function** Chars(Count : Integer; Ch : Char) : **string**;

Эта функция строит строку, состоящую из одинаковых символов. Первый параметр указывает количество символов, а второй — сам этот символ. Например, отобразить длину очереди в виде цепочки звездочек можно следующим образом:

lbQueueLen.Caption := Chars(sim.Queue.Size, ‘\*’);

В зависимости от предполагаемых целей следует выбрать для надписи выравнивание по левому или правому краю, а ее свойство AutoSize следует отключить (то есть, задать равным False).

Еще два приема, которые удобно использовать при отображении состояния действий, рассмотрим в следующем параграфе.

## 2.9. Сбор и отображение статистики для действий

Как уже было отмечено ранее, процессы, составляющие имитацию, в ходе своей работы выполняют те или иные действия, которые моделируются посредством задержки исполнения процесса с помощью метода Hold. В ряде случаев бывает желательно собрать статистику по действиям, исполняемым процессами, в частности, по количеству процессов, одновременно исполняющих то или иное действие, по доле времени, занятой исполнением действия и т. п. Поскольку действия делятся на обычные (самостоятельные) и обслуживающие, в библиотеке предусмотрены два класса сбора статистики по действиям: TActionStatistics и TServiceStatistics. Понятие действия применительно к сбору статистики трактуется более широко: под действием подразумевается просто промежуток времени с момента, отмеченного как начало действия, до момента, отмеченного как его конец.

Рассмотрим класс TActionStatistics:

TActionStatistics = **class**

**public**

**constructor** Create(InitX : Integer; InitTime : Double); **overload**;

**constructor** Create(InitX : Integer); **overload**;

**constructor** Create; **overload**;

**procedure** Start(NewTime : Double); **overload**;

**procedure** Start; **overload**;

**procedure** Finish(NewTime : Double); **overload**;

**procedure** Finish; **overload**;

**procedure** StopStat(NewTime : Double); **overload**;

**procedure** StopStat; **overload**;

**procedure** Clear(NewTime : Double); **overload**;

**procedure** Clear; **overload**;

**function** Mean : Double;

**function** Deviation : Double;

**function** Disperse : Double;

**property** Max : Integer **read**;

**property** Finished : Integer **read**;

**property** Running : Integer **read**;

**property** TotalTime : Double **read**;

**end**;

Многие компоненты этого класса аналогичны компонентам ранее рассмотренных классов сбора статистики, поэтому ограничимся их кратким обзором. В конструкторе с двумя параметрами указывается начальное количество процессов, выполняющих действие и время, начиная с которого ведется сбор данных. В конструкторе с одним параметром предполагается, что сбор начинается с нулевого имитационного времени, а конструктор без параметров при этом еще и предполагает, что в момент начала сбора данных ни один процесс не выполняет действие.

Методы Start и Finish сообщают объекту сбора статистики о том, что некоторый процесс начинает или заканчивает действие соответственно. Все процессы, выполняющие одно действие, считаются равноправными, и никак не различаются.

Методы и свойства Mean, Deviation, Disperse, Max выдают статистику по количеству процессов, одновременно исполнявших действие. Свойство Finished указывает, сколько действий было завершено, а свойство Running — сколько исполняется в данный момент.

Остальные методы и свойства аналогичны ранее рассмотренным у других объектов сбора статистики.

Кроме его прямого назначения, объект TActionStatistics можно трактовать и как «умный» счетчик, который автоматически собирает интервальную статистику по своим значениям (правда, минимальное значение напрямую получить нельзя). При этом методы Start и Finish увеличивают и уменьшают его значение, а свойство Running позволяет узнать текущее значение. Особо отметим, что значение счетчика может быть даже отрицательным.

Класс TServiceStatistics выглядит следующим образом:

TServiceStatistics = **class**

**public**

**constructor** Create(Devices, InitUtil : Integer;

InitTime : Double); **overload**;

**constructor** Create(Devices : Integer); **overload**;

**procedure** Start(NewTime : Double); **overload**;

**procedure** Start; **overload**;

**procedure** Finish(NewTime : Double); **overload**;

**procedure** Finish; **overload**;

**procedure** StartBlock(NewTime : Double); **overload**;

**procedure** StartBlock; **overload**;

**procedure** FinishBlock(NewTime : Double); **overload**;

**procedure** FinishBlock; **overload**;

**procedure** StopStat(NewTime : Double); **overload**;

**procedure** StopStat; **overload**;

**procedure** Clear(NewTime : Double); **overload**;

**procedure** Clear; **overload**;

**function** Mean : Double;

**function** MeanBlockage : Double;

**function** Deviation : Double;

**function** Disperse : Double;

**property** MaxBusy : Integer **read**;

**property** MinBusy : Integer **read**;

**property** Finished : Integer **read**;

**property** Running : Integer **read**;

**property** MaxIdleTime : Double **read**;

**property** MaxBusyTime : Double **read**;

**property** Devices : Integer **read**;

**property** TotalTime : Double **read**;

**end**;

При построении класса статистики для обслуживающих действий имеется в виду, что количество обслуживающих процессов, выполняющих данное действие (в данном контексте они называется устройствами), фиксировано. Это количество и задается первым параметром конструктора Create. В конструкторе с тремя параметрами, кроме того, указывается исходная занятость устройств и время начала сбора статистики.

Методы StartBlock и FinishBlock позволяют учитывать блокировку устройств — ситуации, когда устройство не может продолжить работу, потому что дальнейшее продвижение обслуживаемого процесса по каким-либо причинам временно невозможно. Свойство MeanBlockage дает усредненное по времени количество заблокированных устройств.

Отметим, что, с точки зрения сбора статистики, занятость и блокировка устройства не связаны между собой и учитываются отдельно. Считать время блокировки как занятое или свободное — зависит от конкретной постановки задачи.

Свойства MaxBusy и MinBusy дают максимальное количество устройств, одновременно выполнявших данное действие. Они аналогичны свойствам Max и Min ранее рассмотренных объектов сбора статистики.

Свойства MaxIdleTime и MaxBusyTime дают максимальные непрерывные интервалы времени, в течение которых ни одно из устройств не работало, либо работало как минимум одно устройство, соответственно. В случае если устройств, выполняющих данное действие, больше одного, эти результаты, как правило, не имеют практической пользы.

Свойство Devices дает количество устройств, которые могут выполнять данное действие (то значение, которое было указано в конструкторе метода).

Основных различий между этими классами сбора статистики два. Во-первых, класс TServiceStatistics используется для сбора данных о действиях, которые могут выполнять одновременно ограниченное число устройств. Класс TActionStatistics такого ограничения не подразумевает. Второе отличие состоит в том, что класс TServiceStatistics собирает более подробную статистику — в частности, по блокировке действий.

Для отображения этих разновидностей статистики предусмотрены варианты процедуры WriteStat и ShowStat, рассмотренных ранее. Они отображают следующие статистические результаты. Для объекта TActionStatistics выводятся: среднее количество одновременно исполняемых действий и его стандартное отклонение, максимальное количество одновременно исполняемых действий, текущее количество исполняемых действий, а также количество завершенных действий. Для объекта TServiceStatistics это следующие данные: количество устройств, средняя занятость действия со стандартным отклонением, текущее количество исполняемых действий и количество завершенных действий, среднее по времени количество блокированных процессов. Кроме того, для действий, выполняемых одиночным устройством, выводится максимальные непрерывные времена простоя и занятости, а для действий, которые выполняются более чем одним устройством, выводится максимальное за время наблюдения количество свободных и занятых устройств.

Пример вывода статистики по обслуживающему действию (одно устройство):

Worker 1:

Device count = 1

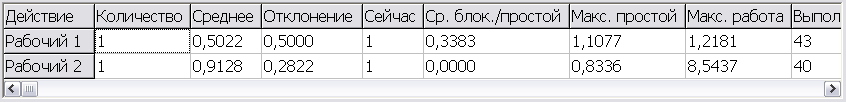
Average usage = 0.445 +- 0.497

Current running = 1, completed = 523

Average blockage = 0.467

Max idle time = 3.388, max busy time = 3.492

Пример отображения двух статистик по обслуживающим действиям, каждое с одним устройством:



При визуализации обслуживающих действий следует отображать их состояние. Это удобно делать, обращаясь к соответствующей статистике, и, в зависимости от ее состояния, помещать в надпись-индикатор то или иное содержимое. Например, если отображать неработающее устройство пустой надписью, а работающее — содержимым «(\*)» (пользуясь звездочкой для обозначения наличия компонента, как в очереди), то можно это выполнить следующим образом:

**if** sim.Stat.Running > 0 **then**

lbDevice.Caption := ‘(\*)’

**else**

lbDevice.Caption := ‘’;

Если действие выполняют несколько устройств, можно усложнить предыдущий фрагмент, чтобы проиллюстрировать количество занятых устройств:

**if** sim.Stat.Running > 0 **then**

lbDevice.Caption := ‘(‘ + Chars(sim.Stat.Running, ‘\*’) + ‘)’

**else**

lbDevice.Caption := ‘’;

Среднюю загруженность устройства можно проиллюстрировать с помощью шкалы, выполненной из двух объектов-фигур (Shape). Для этого следует задать им разный цвет и одинаковый размер и разместить их в одном месте формы друг над другом. Предположим, фигура на переднем плане называется shDeviceScale, а на заднем — shDeviceScaleBack. Тогда отображение загруженности в виде шкалы можно выполнить следующим образом:

shDeviceScale.Width := Round(sim.Stat.Mean \* shDeviceScaleBack.Width);

Если же требуется отобразить загруженность для действия, которое выполняют несколько устройств, эту строку следует несколько усложнить:

shDeviceScale.Width := Round(sim.Stat.Mean \* shDeviceScaleBack.Width /

sim.Stat.Devices);

Можно пойти еще дальше и добавить индикатор блокировки процесса. Для этого надо разместить там же еще и третью такую же фигуру, только третьего цвета. Предположим, она будет называться shDeviceBlockage, а заблокированное время учитывается как незанятое. Тогда управлять ее расположением будут следующие операторы:

shDeviceBlockage.Width := Round(sim.Stat.MeanBlockage \*

shDeviceScaleBack.Width);

shDeviceBlockage.Left := shDeviceScale.Left + shDeviceScale.Width;

# 3. Примеры моделей

Рассмотрим несколько примеров построения моделей. Постановки задач в основном взяты из работы [2], однако в ней используются другие подходы для их решения.

## 3.1. Модель банка

Первая модель является простейшим примером *задачи массового обслуживания*. большинство последующих примеров (но не все) также будут разновидностями этой задачи.

*Постановка задачи.* В банке имеется одна касса. Посетители приходят в банк со случайным интервалом времени, экспоненциально распределенным с математическим ожиданием 5 минут. Кассир обслуживает клиента за время, равномерно распределенное в интервале от 2 до 6 минут. Если кассир занят в момент прихода клиента, он становится в очередь. Требуется выполнить имитацию обслуживания 100 клиентов и получить следующие результаты:

* статистику и гистограмму по времени нахождения клиентов в банке;
* статистику по очереди в кассу (по длине очереди и времени ожидания в ней);
* статистику по загруженности кассира;
* количество клиентов, которые не ожидали в очереди.

Очевидно, банк — далеко не единственная возможность приложения данной задачи. Это могут быть и покупатели в магазине, и детали в очереди на обработку, и автомобили в очереди на ремонт, и многое другое.

### 3.1.1. Исходная задача

Будем рассматривать решение задачи непосредственно как процесс построения приложения. Создайте в Delphi новый проект консольного приложения, сохраните его в отдельную папку и назовите, например, Bank. Для размещения определения модели добавьте к проекту новый модуль и сохраните его под именем UBank. Следующим действием добавьте к проекту модуль (именно добавьте существующий, а не создавайте новый!) USimulation, размещенный в одноименной папке (можно его скопировать в папку вашего проекта, но это не обязательно). Модульная структура проекта готова. Не забудьте на этом этапе полностью сохранить проект.

Для того чтобы в модуле UBank можно было пользоваться средствами моделирования, в начало его интерфейсной части (сразу после строки **interface**) следует добавить строку:

**uses** USimulation;

Теперь приступим к формулировке модели в модуле UBank. Сначала определим, какие процессы нам потребуются в имитации. Очевидно, посетители банка будут представлены процессами-компонентами. Для их периодического порождения удобно использовать процесс-генератор. Действия кассира будет имитировать обслуживающий процесс. Создадим заготовки классов процессов в интерфейсном разделе модуля:

**type**

*// Класс TClient – процесс, моделирующий клиента банка*

TClient = **class**(TProcess)

**protected**

**procedure** RunProcess; **override**;

**end**;

*// Класс TClientGenerator – процесс, порождающий клиентов банка*

TClientGenerator = **class**(TProcess)

**protected**

**procedure** RunProcess; **override**;

**end**;

*// Класс TCashman – процесс, моделирующий работу кассира*

TCashman = **class**(TProcess)

**protected**

**procedure** RunProcess; **override**;

**end**;

Как видно, структура определений этих классов предельно проста и однотипна: в каждом из них переопределяется только основной метод RunProcess. Никаких других средств в этих классах нам пока не потребовалось (и в данном примере не потребуется).

Следом создадим определение класса имитации. В отличие от процессов, он должен поддерживать некоторую инфраструктуру для обеспечения работы имитации, поэтому в нем обязательно должны быть дополнительные компоненты. Перечислим их. Во-первых, это два статических процесса: генератор клиентов и кассир. Эти два процесса удобно делать статическими, потому что они существуют все время, пока существует имитация. Во-вторых, это очередь клиентов, которая будет представлена списком. В-третьих, это объекты для сбора статистики и гистограмм: точечная статистика и гистограмма по времени пребывания клиентов в банке, статистика по обслуживающим действиям кассира, счетчик клиентов, обслуженных без ожидания в очереди. Что касается последнего счетчика, то, поскольку в формулировке задачи требуется узнать просто количество таких клиентов, достаточно простой целочисленной переменной. Что касается методов, переопределяемых в этом классе, это должны быть, как уже упоминалось: RunSimulation, Init, StopStat и деструктор Destroy. Поскольку в данном примере не предполагается наличие каких-либо параметров при создании имитации, определять собственный конструктор нет надобности. В итоге определение класса имитации принимает следующий вид:

*// Класс TBankSimulation - моделирование работы банка*

TBankSimulation = **class**(TSimulation)

**public**

*// Кассир*

Cashman : TCashman;

*// Генератор клиентов*

Generator : TClientGenerator;

*// Очередь ожидания*

Queue : TList;

*// Статистика и гистограмма по времени пребывания клиентов в банке*

InBankTime : TStatistics;

InBankHist : THistogram;

*// Статистика по занятости кассира*

CashStat : TServiceStatistics;

*// Счетчик клиентов, обслуженных без ожидания*

NotWaited : Integer;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** RunSimulation; **override**;

**procedure** Init; **override**;

**end**;

Следующим шагом является создание заготовок методов всех классов. Delphi позволяет сделать это автоматически. Для этого нужно, щелкая правой кнопкой внутри определений каждого класса, из контекстного меню выбирать команду Complete class at cursor (Завершить определение класса, указанного мышью) или воспользоваться указанной в том же меню комбинацией клавиш (Shift + Ctrl + С).

Следующим шагом является определение глобальных объектов и параметров имитации. Это: датчики случайных чисел для моделирования интервалов между прибытием клиентов и времени обслуживания кассира, максимальное количество обслуживаемых клиентов, средний промежуток времени между прибытием клиентов, границы времени обслуживания и параметры гистограммы. Попутно заметим, что естественным масштабом имитационного времени для данной задачи будет следующий: 1 единица времени = 1 минута. Внесем следующие определения в интерфейсную часть модуля после определений классов:

**var**

*// Датчики случайных чисел*

rndClient : TRandom;

rndCashman : TRandom;

*// Количество обслуживаемых клиентов*

MaxClientCount : Integer = 100;

*// Средний интервал между прибытием клиентов*

MeanClientInterval : Double = 5;

*// Границы времени обслуживания*

MinCashTime : Double = 2;

MaxCashTime : Double = 6;

*// Параметры гистограммы*

HistMin : Double = 2;

HistStep : Double = 2;

HistStepCount : Integer = 20;

Что касается указанных числовых параметров, то их можно было бы определить и как константы (**const**). Однако указанный способ определения предпочтительнее с той точки зрения, что эти параметры легко могут быть изменены в программе, и тогда их можно вводить как исходные данные или вычислять исходя из любых соображений.

Теперь приступаем к формулировке алгоритмов процессов. Алгоритм генератора простой и очевидный, он, по сути, повторяет то, что приводилось в параграфе 2.4. Единственное отличие состоит в том, что, поскольку в формулировке задачи явно указано количество обслуживаемых клиентов, в нем используется цикл со счетчиком:

**procedure** TClientGenerator.RunProcess;

**var**

i : Integer;

**begin**

**for** i := 1 **to** MaxClientCount **do**

**begin**

ClearFinished;

TClient.Create.ActivateDelay(0);

Hold(rndClient.Exponential(MeanClientInterval));

**end**;

**end**;

Действия процесса-клиента также не отличаются сложностью. Клиент, войдя в банк, встает в очередь, предварительно активировав кассира (если кассир свободен, он сразу же приступит к обслуживанию данного клиента; если же он занят, это действие никак на него не повлияет). Поскольку предполагается использование обслуживающего действия, исполняемого кассиром, то от клиента на этом этапе также ничего не требуется, и он продолжит работу только после того как кассир закончит его обслуживание. После этого клиент покидает банк, что моделируется постановкой в очередь завершенных процессов:

**procedure** TClient.RunProcess;

**var**

par : TBankSimulation;

**begin**

par := Parent **as** TBankSimulation;

*// Активировать кассу*

par.Cashman.ActivateDelay(0);

*// Встать в очередь и ждать обслуживания*

Wait(par.Queue);

*// Встать в очередь завершенных процессов*

Finish;

**end**;

Наиболее сложную работу выполняет кассир. Он циклически выполняет следующие действия. Сначала кассир дожидается, когда в очереди появится хотя бы один клиент. Дождавшись, когда очередь станет не пустой, кассир извлекает клиента из очереди (при этом клиент из разряда связанных процессов переходит в разряд свободных). Далее кассир сравнивает текущее время с временем прибытия клиента. Если они равны, следует учесть данного клиента как такого, которому не пришлось ожидать в очереди. Затем выполняется действие, которое моделируется задержкой согласно формулировке задачи и сбором статистики действия. Статистика собирается уведомлением о начале действия перед задержкой и об окончании после нее. После выполнения обслуживания кассир вычисляет время, которое клиент провел в банке, и учитывает его в соответствующей статистике и гистограмме. Затем клиент активируется, чтобы продолжить свою работу (как вы помните, в данном примере он просто сразу завершает работу). И наконец, проверив количество выполненных действий обслуживания, кассир принимает решение о продолжении или окончании имитации. Если имитацию следует закончить, это моделируется возобновлением процесса имитации, который, собственно, и заканчивает работу. Ниже приводится листинг процедуры:

**procedure** TCashman.RunProcess;

**var**

Client : TClient;

InTime : Double;

par : TBankSimulation;

**begin**

par := Parent **as** TBankSimulation;

**while** True **do**

**begin**

*// Если очередь пуста, ждать прибытия клиента*

**while** par.Queue.Empty **do**

Passivate;

*// Извлечь первого клиента из очереди*

Client := par.Queue.First **as** TClient;

Client.StartRunning;

*// Если клиент не ждал, учесть его*

**if** Client.StartingTime = SimTime **then**

Inc(par.NotWaited);

*// Выполнить обслуживание*

par.CashStat.Start(SimTime);

Hold(rndCashman.Uniform(MinCashTime, MaxCashTime));

par.CashStat.Finish(SimTime);

*// Учесть полное время пребывания в банке*

InTime := SimTime - Client.StartingTime;

par.InBankTime.AddData(InTime);

par.InBankHist.AddData(InTime);

*// Возобновить клиента, дав ему возможность закончить работу*

Client.ActivateDelay(0);

*// Если все клиенты обслужены, завершить работу*

**if** par.CashStat.Finished = MaxClientCount **then**

par.ActivateDelay(0);

**end**;

**end**;

Методы процесса имитации достаточно просты и построены в соответствии с ранее описанными правилами. В деструкторе удаляются все созданные объекты, в методе Init они создаются, в методе StopStat производится коррекция статистик очереди и занятости кассира (статистика по времени пребывания клиентов в банке, как и любая точечная статистика, в коррекции не нуждается). Алгоритм выполнения имитации в методе RunSimulation предполагает активацию генератора для запуска процесса создания клиентов, остановку имитации на неопределенное время (поскольку решение о ее завершении принимает кассир) и окончательную коррекцию статистики:

**destructor** TBankSimulation.Destroy;

**begin**

Cashman.Free;

Generator.Free;

InBankTime.Free;

InBankHist.Free;

CashStat.Free;

Queue.Free;

**inherited**;

**end**;

**procedure** TBankSimulation.Init;

**begin**

**inherited**;

Queue := TList.Create;

Cashman := TCashman.Create;

Generator := TClientGenerator.Create;

InBankTime := TStatistics.Create;

CashStat := TServiceStatistics.Create(1);

InBankHist := TUniformHistogram.Create(HistMin, HistStep, HistStepCount);

NotWaited := 0;

**end**;

**procedure** TBankSimulation.RunSimulation;

**begin**

*// Запустить процесс создания клиентов*

Generator.ActivateDelay(0);

*// Ждать конца имитации*

Passivate;

StopStat;

**end**;

**procedure** TBankSimulation.StopStat;

**begin**

**inherited**;

Queue.StopStat(SimTime);

CashStat.StopStat(SimTime);

**end**;

На этом процесс создания имитации завершен. Следующим шагом является разработка главной программы. В ее функции входит создание объектов датчиков случайных чисел, процесса имитации, запуск имитации и вывод собранной статистики. Ее текст выглядит следующим образом:

**var**

bnk : TBankSimulation;

*// Моделирование работы банка с одним кассиром*

**begin**

*{ TODO -oUser -cConsole Main : Insert code here }*

rndClient := TRandom.Create;

rndCashman := TRandom.Create;

MaxClientCount := 100;

*// Создание и запуск имитации*

bnk := TBankSimulation.Create;

SwitchTo(bnk);

**with** bnk **do**

**begin**

*// Вывод статистики*

WriteLn('Finished at ', bnk.SimTime : 6 : 3);

WriteLn;

WriteStat('Clients in bank time:', InBankTime);

WriteLn;

WriteStat('Cash usage:', CashStat);

WriteLn;

WriteStat('Clients queue:', Queue);

WriteLn;

WriteStat('System calendar:', Calendar);

WriteLn;

WriteLn('Not waited: ', NotWaited);

WriteLn;

WriteHist('In bank time histogram:', InBankHist);

**end**;

bnk.Free;

ReadLn;

**end**.

Результаты имитации выводятся в следующем порядке: время завершения имитации, статистика по времени пребывания клиентов в банке, статистика по занятости кассира, статистика по очереди ожидания, статистика системного календаря, количество клиентов, обслуженных без ожидания и гистограмма по времени пребывания клиентов в банке. При одном из прогонов был получен следующий результат:

Finished at 504.220

Clients in bank time:

Average = 10.893 +- 6.043

Min = 2.402, max = 27.895

Total = 100 values

Cash usage:

Device count = 1

Average usage = 0.806 +- 0.396

Current running = 0, completed = 100

Average blockage = 0.000

Max idle time = 17.502, max busy time = 181.878

Clients queue:

Average length = 1.355 +- 1.584

Max = 7, current = 0

Average waiting time = 6.831

System calendar:

Average length = 1.788 +- 0.408

Max = 2, current = 1

Average waiting time = 2.162

Not waited: 15

In bank time histogram:

-INF - 2.00 : 0 ( 0.00%), 0.00%

2.00 - 4.00 : 6 ( 6.00%), 6.00% \*\*

4.00 - 6.00 : 21 (21.00%), 27.00% \*\*\*\*\*\*\*\* O

6.00 - 8.00 : 10 (10.00%), 37.00% \*\*\*\* O

8.00 - 10.00 : 15 (15.00%), 52.00% \*\*\*\*\*\* O

10.00 - 12.00 : 15 (15.00%), 67.00% \*\*\*\*\*\* O

12.00 - 14.00 : 8 ( 8.00%), 75.00% \*\*\* O

14.00 - 16.00 : 4 ( 4.00%), 79.00% \*\* O

16.00 - 18.00 : 8 ( 8.00%), 87.00% \*\*\* O

18.00 - 20.00 : 1 ( 1.00%), 88.00% O

20.00 - 22.00 : 5 ( 5.00%), 93.00% \*\* O

22.00 - 24.00 : 4 ( 4.00%), 97.00% \*\* O

24.00 - 26.00 : 1 ( 1.00%), 98.00% O

26.00 - 28.00 : 2 ( 2.00%), 100.00% \* O

28.00 - 30.00 : 0 ( 0.00%), 100.00% O

30.00 - 32.00 : 0 ( 0.00%), 100.00% O

32.00 - 34.00 : 0 ( 0.00%), 100.00% O

34.00 - 36.00 : 0 ( 0.00%), 100.00% O

36.00 - 38.00 : 0 ( 0.00%), 100.00% O

38.00 - 40.00 : 0 ( 0.00%), 100.00% O

40.00 - 42.00 : 0 ( 0.00%), 100.00% O

42.00 - +INF : 0 ( 0.00%), 100.00% O

Интересно обсудить статистические результаты по календарю событий. Здесь показано, что максимальная длина календаря равнялась 2, а средняя — несколько меньше этого значения. То есть, в календаре находилось не более двух процессов. Что это за процессы? Очевидно, это, во-первых, генератор клиентов, а во-вторых, кассир. Но как же тогда с самими клиентами? Они же тоже активируются, а это значит, что их записи уведомлений о событии тоже должны попадать в календарь.

Дело вот в чем. Каждый клиент активируется за время своего существования два раза: при создании и после завершения обслуживания. В обоих случаях он активируется с нулевой задержкой и, исполнив необходимые действия, переходит в пассивное состояние. То есть, несмотря на то, что запись уведомления о событии действительно добавляется в календарь, она находится в нем в течение нулевого промежутка имитационного времени. А значит, она не оказывает влияния на интервальную статистику по длине календаря.

Иное дело — точечная статистика по среднему времени ожидания в календаре. В этом случае процессы-клиенты со своим нулевым временем ожидания вносят существенный вклад в общий результат. Оценим его. На каждого клиента приходится одно ожидание генератора со средним временем 5 минут, одно ожидание кассира со средним временем 4 минуты и два ожидания самого клиента с нулевым временем. Таким образом, среднее время ожидания в календаре должно составить примерно 2,25 минуты, что близко к полученному результату.

Заметим, что при выполнении разных прогонов могут получаться разные результаты, порой существенно различающиеся. Так, например, среднее время нахождения клиентов в банке в результате нескольких прогонов, проведенных автором, оказывалось в диапазоне от 7 до 22 минут. Это означает, что однократный прогон, как правило, не позволяет получить достаточно полных результатов. Для их получения требуется провести достаточно много прогонов и обобщить их результаты. Подробнее эта тема будет рассмотрена в параграфе 3.2 при изучении следующей модели.

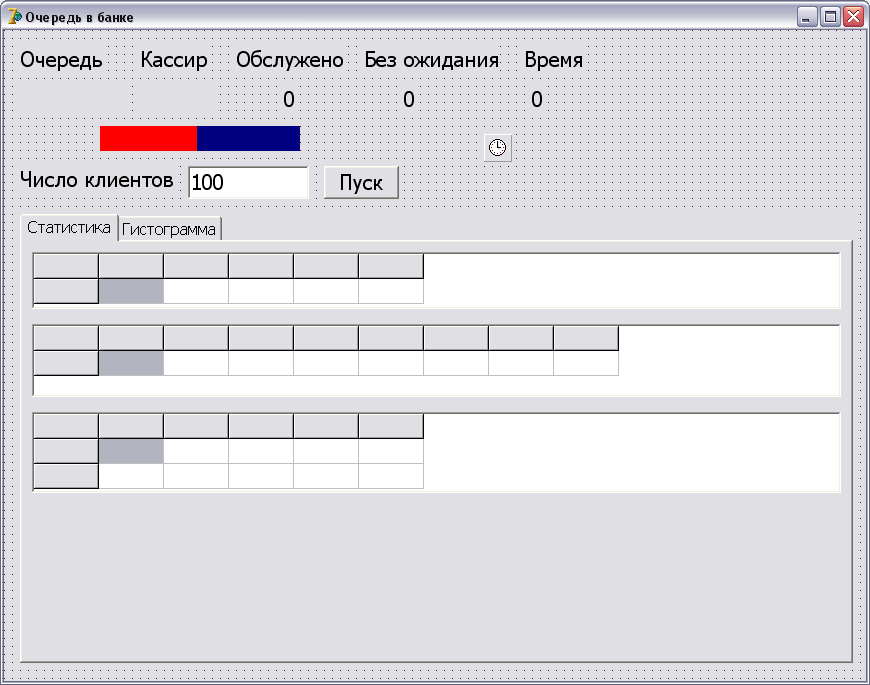
Полный текст программы находится в папке 3.1.1 Bank.

### 3.1.2. Визуализация

Теперь добавим к созданной имитации возможности визуализации. Создайте в Delphi новый проект, на этот раз это будет обычное оконное приложение. Сохраните проект в отдельную папку, модуль формы назовите, например, UBankVis, а сам проект — BankVis (как и в предыдущем и последующих проектах, указание имен носит рекомендательный характер; однако если вы будете давать объектам другие имена, вам предстоит самостоятельно менять текст программы, чтобы сохранить соответствие имен и объектов). Скопируйте из папки предыдущего проекта и добавьте к новому проекту файл модуля UBank (его надо именно скопировать, потому что в его текст придется внести небольшое дополнение) и добавьте к проекту, как и раньше, модуль USimulation (его копировать, как и в предыдущем проекте, не обязательно). Модульная структура проекта готова.

Следующий шаг — разработка интерфейса. Как было написано в параграфе 1.6, для отображения статистики и гистограмм удобно использовать табличные сетки. Однако визуализация будет нагляднее и эффектнее, если, кроме них, отображать основные параметры имитации, такие как длину очереди, состояние кассира и т. п. Как указывалось в параграфе 2.8, это легко сделать при помощи объектов-надписей. Поскольку табличные сетки занимают на форме довольно много места, разместим их на многостраничной панели с закладками: на одной закладке — таблицы для отображения статистики, на другой — сетка для построения гистограммы.

Приведем примерный вид интерфейса:



Набор основных объектов интерфейса и некоторые их свойства перечислены в таблице. В ней не приведены поясняющие надписи (Label), их размещение очевидно, а имена можно оставить стандартные. На многостраничной панели следует создать две страницы и задать им подходящие надписи и имена (поскольку в программе прямое обращение к страницам не производится, выбор имен остается на ваше усмотрение). Три табличных сетки для отображения статистики размещаются на первой странице панели, а четвертая, в которой изображается гистограмма — на второй странице.

| Тип объекта | Имя | Описание | Свойства | |
| --- | --- | --- | --- | --- |
| Form | frBankVis | Основная форма приложения | Caption | Очередь в банке |
| Font | Tahoma, 16 pt |
| BorderStyle | bsSingle |
| BorderIcons | biMaximize = False |
| Label | lbQueue | Надпись для отображения очереди ожидания | AutoSize | False |
| Alignment | taRightJustify |
| Label | lbCashman | Надпись для отображения состояния кассира | AutoSize | False |
| Alignment | taCenter |
| Label | lbServiced | Надпись для отображения количества обслуженных клиентов | Caption | 0 |
| Label | lbNoWait | Надпись для отображения количества клиентов, обслуженных без ожидания в очереди | Caption | 0 |
| Label | lbSimTime | Надпись для отображения текущего имитационного времени | Caption | 0 |
| Timer | tmBank | Таймер, управляющий ходом визуализации | Interval | 100 |
| Enabled | False |
| Shape | shCashBack | Фон шкалы загруженности кассира | Brush.Color | clNavy |
| Width | 200 |
| Shape | shCashBusy | Шкала загруженности кассира | Brush.Color | clRed |
| Width | 200 |
| Edir | edClientCount | Поле ввода количества обслуживаемых клиентов | Text | 100 |
| Button | btStart | Кнопка для запуска и остановки имитации | Caption | Пуск |
| PageControl | pgStats | Многостраничная панель для отображения статистики и гистограмм | Font | Tahoma, 12 pt |
| StringGrid | sgTimeStat | Табличная сетка для отображения статистики по времени нахождения клиентов в банке | ColCount | 6 |
| RowCount | 2 |
| StringGrid | sgCashStat | Табличная сетка для отображения статистики по обслуживающему действию кассира | ColCount | 9 |
| RowCount | 2 |
| StringGrid | sgQueueStat | Табличная сетка для отображения статистики по очереди ожидания и календарю событий | ColCount | 6 |
| RowCount | 3 |
| DrawGrid | dgHistogram | Табличная сетка для отображения гистограммы | ColCount | 6 |
| FixedCols | 2 |

Теперь приступим к формированию текста. Сначала определим масштаб соответствия имитационного и реального времени. Как следует из постановки задачи, в имитации создается 100 клиентов со средним интервалом между ними 5 единиц времени (минут). Следовательно, имитация завершается примерно за 500 единиц времени, что подтверждается в процессе нескольких прогонов предыдущей модели. Если принять масштаб 1 секунда = 5 единицам имитационного времени, то длительность имитации будет составлять примерно 100 секунд, а средний интервал между прибытием клиентов в банк будет равен 1 секунде. Это достаточно удобный масштаб времени.

Поскольку интервал события срабатывания таймера выбран равным 100, то он будет срабатывать 10 раз в секунду, что делает визуализацию достаточно динамичной. Это значит, что за один шаг визуализации должно проходить 0,5 единиц имитационного времени. Чтобы это указать, в список параметров модуля UBank добавим еще одну переменную:

*// Временной шаг визуализации имитации*

VisTimeStep : Double = 0.5;

А в метод Init класса имитации следует добавить последней строку, создающую объект-визуализатор:

MakeVisualizator(VisTimeStep);

На этом все изменения, вносимые в модуль модели, закончены. Как видите, изменения в ней, связанные с добавлением визуализации, минимальны. То есть, текст программы модели практически не зависит от наличия или отсутствия ее визуализации. Это позволяет легко отделить разработку модели от ее визуализации и сначала, к примеру, разработать и отладить модель в консольном режиме, а потом выполнить ее визуализацию.

Теперь приступим к разработке текста модуля основной формы приложения. Сначала добавим к нему ссылки на модули имитации и глобальные переменные. Их следует разместить в начале раздела реализации модуля, непосредственно после слова **implementation** (в листинге приведены некоторые фрагменты текста, автоматически создаваемые в заготовке модуля, для облегчения привязки):

**implementation**

**uses** UBank, USimulation;

*{$R \*.dfm}*

**var**

bnk : TBankSimulation = **nil**;

Queue : **array** [0 .. 1] **of** TList;

LastFinished : Integer = 0;

Переменная bnk будет содержать ссылку на процесс имитации, массив Queue — ссылки на два списка для отображения статистики: очередь ожидания и системный календарь. переменная LastFinished будет использоваться для запоминания последнего числа обслуженных клиентов. Эта переменная нужна, чтобы сделать отображение гистограммы более комфортным. Действительно, если при каждом срабатывании таймера (10 раз в секунду) гистограмма будет заново перерисовываться, это приведет к ее ощутимому мерцанию, что негативно скажется на визуальном восприятии гистограммы. Мы же будем перерисовывать гистограмму только в том случае, если общее количество обслуженных клиентов изменилось по сравнению с предыдущим состоянием. Чтобы запоминать это количество, как раз и используется переменная LastFinished.

В коде программы нужно реализовать процедуры следующих событий: щелчка по кнопке Пуск, срабатывания таймера, создания формы, прорисовки ячейки табличной сетки для гистограммы и изменения положения сетки гистограммы. Тексты этих процедур соответствуют общим принципам, изложенным в параграфах 2.8 и 2.9, поэтому приведем их без пояснений (кроме, разумеется, комментариев, вставленных в текст программы):

**procedure** TfrBank.btStartClick(Sender: TObject);

**begin**

*// Кнопка Пуск*

**if** tmBank.Enabled **then**

*// Если имитация выполняется, остановить ее*

tmBank.Enabled := False

**else**

**begin**

*// Начало новой имитации*

MaxClientCount := StrToInt(edClientCount.Text);

tmBank.Enabled := True;

*// Создать имитацию*

bnk.Free;

bnk := TBankSimulation.Create;

*// Заполнить массив списков для отображения статистики*

Queue[0] := bnk.Queue;

Queue[1] := bnk.Calendar;

LastFinished := 0;

dgHistogram.RowCount := bnk.InBankHist.IntervalCount + 3;

*// Запустить имитацию*

SwitchTo(bnk);

**end**;

**end**;

**procedure** TfrBank.tmBankTimer(Sender: TObject);

**begin**

*// Таймер*

bnk.StopStat;

*// Отобразить состояние имитации*

lbQueue.Caption := Chars(bnk.Queue.Size, '\*');

**if** bnk.Cashman.Idle **then**

lbCashman.Caption := ''

**else**

lbCashman.Caption := '(\*)';

lbSimTime.Caption := Format('%1.0f', [bnk.SimTime]);

lbServiced.Caption := IntToStr(bnk.InBankTime.Count);

lbNoWait.Caption := IntToStr(bnk.NotWaited);

*// Отобразить статистики*

ShowStat(sgCashStat, ['Касса'], bnk.CashStat);

ShowStat(sgQueueStat, ['Очередь', 'Календарь'], Queue);

ShowStat(sgTimeStat, ['Время в банке'], bnk.InBankTime);

*// Если количество обслуженных изменилось*

**if** bnk.InBankHist.TotalCount > LastFinished **then**

**begin**

*// Перерисовать гистограмму*

LastFinished := bnk.InBankHist.TotalCount;

dgHistogram.Repaint;

**end**;

*// Отобразить шкалу*

shCashBusy.Width := Round(shCashBack.Width \* bnk.CashStat.Mean);

*// Если имитация завершена*

**if** bnk.Terminated **then**

*// Остановить таймер*

tmBank.Enabled := False

**else**

*// Иначе - выполнить следующий фрагмент имитации*

RunSimulation(bnk);

**end**;

**procedure** TfrBank.FormCreate(Sender: TObject);

**begin**

*// Создание датчиков случайных чисел*

rndClient := TRandom.Create;

rndCashman := TRandom.Create;

*// Настройка ширины столбцов табличных сеток*

sgQueueStat.ColWidths[0] := 90;

sgQueueStat.ColWidths[1] := 70;

*// Остальные строки пропущены*

*// …*

dgHistogram.ColWidths[5] := 600;

**end**;

**procedure** TfrBank.dgHistogramDrawCell(Sender: TObject;

ACol, ARow: Integer; Rect: TRect; State: TGridDrawState);

**begin**

**if** bnk <> **nil** **then**

DrawHistCell(dgHistogram, ACol, ARow, Rect, bnk.InBankHist);

**end**;

**procedure** TfrBank.dgHistogramTopLeftChanged(Sender: TObject);

**begin**

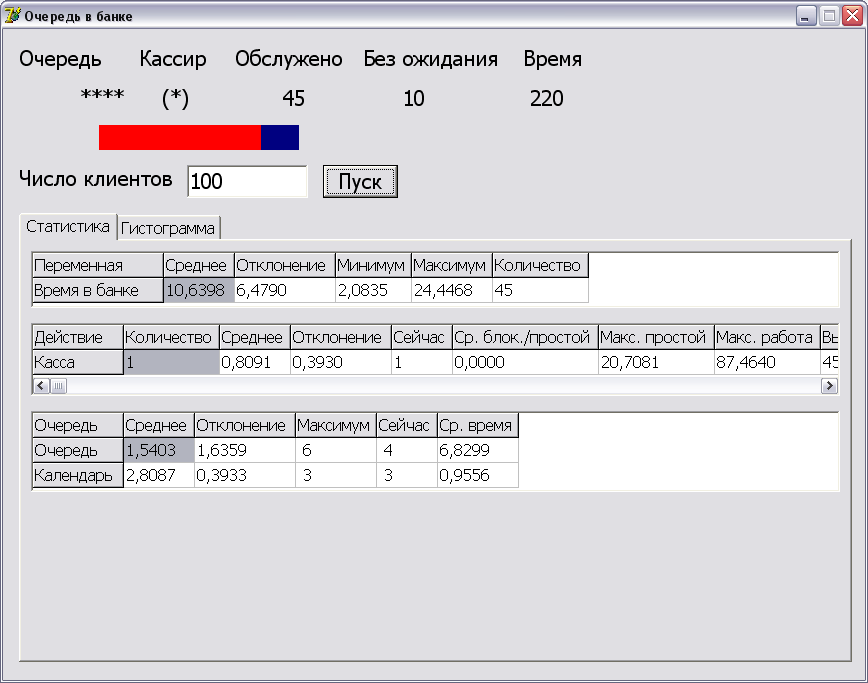
**if** bnk <> **nil** **then**

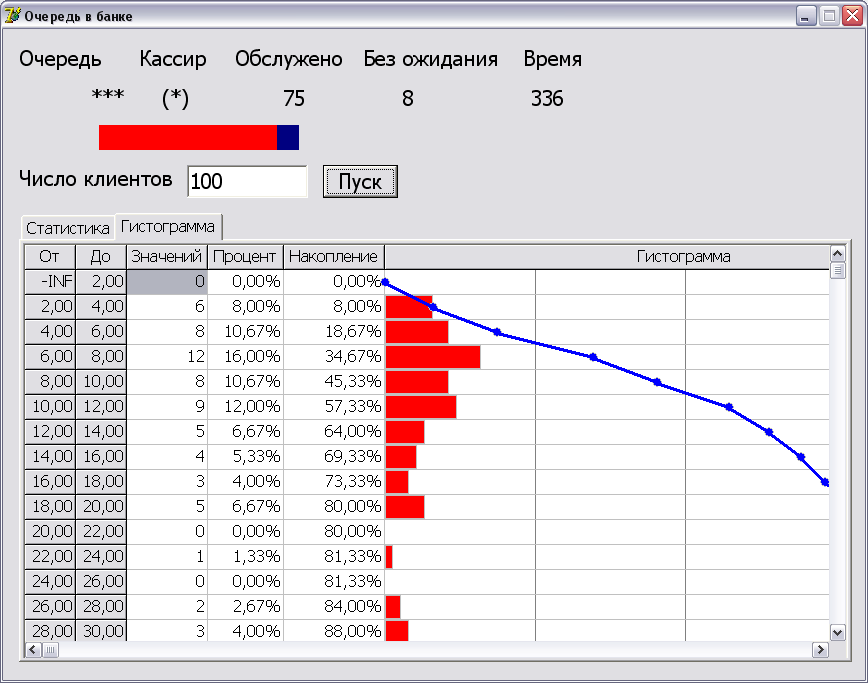
dgHistogram.Repaint;

**end**;

В событии формы Create находится много однотипных строк, задающих ширины столбцов табличных сеток. Чтобы не загромождать текст большим набором одинаковых строк, они исключены из приведенного здесь листинга.

Приведем два примера изображения формы в процессе прогона имитации. В одном примере открыта страница панели с изображением таблиц статистики, а в другом — с гистограммой:





По результатам данной имитации заметим, что средняя и максимальная длина календаря по сравнению с консольной версией увеличилась на единицу. Это вполне объяснимо, поскольку в системе появился еще один процесс — визуализатор. Что же касается среднего времени ожидания в календаре, сделайте самостоятельные оценки этого времени с учетом работы визуализатора и сравните их с имеющимися результатами.

Полный текст данного проекта находится в папке 3.1.2 Bank Visual.

### 3.1.3. Использование нескольких обслуживающих устройств

Внесем некоторые изменения и усложнения в исходную постановку задачи.

Во-первых, в банке теперь работают два кассира. Время обслуживания у каждого из них распределено равномерно в интервале от 6 до 12 минут. Первый клиент, ожидающий в очереди, поступает на обслуживание к первому освободившемуся кассиру.

Во-вторых, в момент начала имитации перед каждым из кассиров уже находится по одному клиенту, и еще два клиента ожидают в очереди. Поступление следующего клиента запланировано на момент времени, равный 5 минутам.

В-третьих, длина очереди ограничена 10 человеками. Если при поступлении очередного клиента в очереди уже находится 10 человек, он покидает банк без обслуживания. Такая ситуация называется *отказом в обслуживании*. Требуется, помимо прочего, подсчитать, сколько поступивших клиентов не было обслужено.

В-четвертых, имитация заканчивается, когда заканчивается обслуживание сотого клиента, даже если в очереди еще есть клиенты. Таким образом, количество поступивших в систему клиентов может оказаться больше числа обслуженных. Причем не только по причине наличия не обслуженных клиентов в очереди, но и за счет клиентов, которым было отказано в обслуживании. Это, в частности, скажется на построении алгоритма генератора клиентов: теперь в нем уже нельзя применять цикл со счетчиком.

Поскольку разработка приложения производится в соответствии с теми же общими принципами, что и в предыдущем примере, мы уже не будем полностью описывать весь процесс разработки. Сосредоточимся только на тех фрагментах программы, которые существенно отличаются от предыдущего проекта.

Начнем с определений классов. Класс процесса-клиента должен допускать возможность его принудительной постановки на обслуживание кассиру или в очередь. Чем отличается клиент, принудительно включенный в систему, от всех остальных? Тем, что ему не приходится выполнять действий по постановке в очередь. Это относится как к клиентам, поставленным на обслуживание в начале имитации, так и к клиентам, просто поставленным в очередь. Следовательно, клиент должен иметь возможность каким-то образом узнать, был ли он помещен в систему принудительно или обычным порядком. Это удобно сделать с помощью поля Inserted, имеющего логический тип. Значение True в нем означает, что клиент был помещен в систему принудительно при запуске имитации.

Поскольку процесс-клиент содержит поле, значение которого должно задаваться при его создании, следует определить конструктор с параметром. Тогда создание объекта-клиента при запуске имитации производится с помощью вызова TClient.Create(True), а остальных клиентов — вызовом TClient.Create(False). Однако чтобы свести к минимуму исправления в алгоритме процесса-генератора, удобно определить еще и конструктор без параметров, вызов которого эквивалентен вызову конструктора с параметром False. Таким образом, получаем следующее определение класса клиента:

TClient = **class**(TProcess)

**public**

Inserted : Boolean;

**constructor** Create; **overload**;

**constructor** Create(AInserted : Boolean); **overload**;

**protected**

**procedure** RunProcess; **override**;

**end**;

Теперь обратимся к классу кассира. Необходимо каким-то образом при запуске имитации передать ему того клиента, с которым он будет работать первым. Удобнее всего сделать это, превратив локальную переменную его алгоритма Client в поле класса. Поскольку в классе появилось поле, необходимо предусмотреть в нем конструктор, при вызове которого и задается клиент, с которым начинает работу кассир. Если при создании процесса кассира передать параметром **nil**, то он начнет свою работу в обычном режиме, с ожидания клиента из очереди. Это удобно на тот случай, если в дальнейшем формулировка задачи будет меняться. Ниже приводится определение класса:

TCashman = **class**(TProcess)

**public**

Client : TClient;

**constructor** Create(AClient : TClient);

**protected**

**procedure** RunProcess; **override**;

**end**;

В классе имитации следует заменить одиночный процесс Cashman на массив процессов Cashmen, причем определить его следует как массив объектов TProcess, а не TCashman. Причины этого обсуждались в параграфе 2.3. Кроме того, следует добавить счетчик количества не обслуженных клиентов. Вот определение класса:

TBankSimulation = **class**(TSimulation)

**public**

Cashmen : **array** **of** TProcess;

Generator : TClientGenerator;

InBankTime : TStatistics;

InBankHist : THistogram;

CashStat : TServiceStatistics;

Queue : TList;

NotWaited : Integer;

NotServiced : Integer;

**destructor** Destroy; **override**;

**protected**

**procedure** RunSimulation; **override**;

**procedure** Init; **override**;

**end**;

Алгоритм работы клиента содержит два изменения по сравнению с исходным вариантом. Во-первых, если клиент был внедрен в систему принудительно, он не встает в очередь, а сразу переходит в режим ожидания обслуживания. Во-вторых, если при попытке постановки в очередь она заполнена до максимального значения, он увеличивает счетчик не обслуженных клиентов и завершает работу. Вот текст алгоритма:

**procedure** TClient.RunProcess;

**var**

par : TBankSimulation;

**begin**

par := Parent **as** TBankSimulation;

*// Если клиент был внедрен в систему принудительно*

**if** Inserted **then**

**begin**

*// Сразу ожидать обслуживания*

Passivate;

**end**

*// Встать в очередь*

**else** **if** par.Queue.Size < MaxQueueLength **then**

**begin**

Insert(par.Queue);

*// Активировать кассиров*

ActivateDelay(par.Cashmen, 0);

*// Ждать обслуживания*

Passivate;

**end**

**else**

*// Очередь заполнена - увеличить счетчик отказов*

Inc(par.NotServiced);

*// Встать в очередь завершенных процессов*

Finish;

**end**;

Алгоритм генератора отличается тем, что, во-первых, в нем задается задержка перед появлением первого клиента, а во-вторых, вместо цикла со счетчиком используется «бесконечный» цикл:

**procedure** TClientGenerator.RunProcess;

**begin**

*// Подождать перед появлением первого клиента*

Hold(FirstClientArrival);

**while** True **do**

**begin**

ClearFinished;

TClient.Create.ActivateDelay(0);

Hold(rndClient.Exponential(MeanClientInterval));

**end**;

**end**;

Теперь обратимся к алгоритму работы кассира. Напомним, что в новой постановке задачи с самого начала работы кассира один клиент находится у него на обслуживании, ссылка на него записана в его поле Client. Кассир может сразу начать обслуживание клиента, однако клиент еще должен запомнить время начала своей работы. Чтобы он смог это сделать, его надо активировать. Как указывалось ранее, такой клиент после своего запуска немедленно переходит в режим ожидания обслуживания. Поэтому после его активации процесс кассира должен выполнить задержку на нулевой интервал имитационного времени и начать обслуживание клиента. Поскольку такой вариант принятия клиента на обслуживание применяется только для первого клиента, это несколько меняет логику выполнения процесса: первый клиент принимается на обслуживание до начала цикла, в теле цикла выполняется обслуживание клиента, и уже потом из очереди извлекается следующий клиент. В целях сохранения совместимости с предыдущим вариантом проекта можно предусмотреть возможность двух вариантов принятия первого клиента на обслуживание: если клиент задан непосредственно, принять его, как только что было описано, а если нет, то, как и раньше, ожидать его появления в очереди. Однако такая возможность дополнительно усложнила бы алгоритм, и сейчас мы ее реализовать не будем. Как и раньше, если обслужено заданное число клиентов, активируется процесс имитации, который завершает свою работу. В итоге алгоритм работы кассира описывается следующей процедурой:

**procedure** TCashman.RunProcess;

**var**

InTime : Double;

par : TBankSimulation;

**begin**

par := Parent **as** TBankSimulation;

*// Передать управление клиенту, чтобы он запомнил время запуска*

Client.StartRunning;

Client.ActivateDelay(0);

Hold(0);

*// Приступить к работе с ним*

**while** True **do**

**begin**

*// Если клиент не ждал, учесть его*

**if** Client.StartingTime = SimTime **then**

Inc(par.NotWaited);

*// Выполнить обслуживание*

par.CashStat.Start(SimTime);

Hold(rndCashman.Uniform(MinCashTime, MaxCashTime));

par.CashStat.Finish(SimTime);

*// Учесть полное время пребывания в банке*

InTime := SimTime - Client.StartingTime;

par.InBankTime.AddData(InTime);

par.InBankHist.AddData(InTime);

*// Возобновить процесс клиента, дав ему возможность закончить работу*

Client.ActivateDelay(0);

*// Если достигнуто предельное число клиентов, завершить работу*

**if** par.CashStat.Finished = MaxClientCount **then**

**begin**

par.ActivateDelay(0);

Passivate;

**end**;

*// Если очередь пуста, ждать прибытия клиента*

**while** par.Queue.Empty **do**

Passivate;

*// Извлечь первого клиента из очереди*

Client := par.Queue.First **as** TClient;

Client.StartRunning;

**end**;

**end**;

Остается описать изменения, вносимые в процесс имитации. В методе Init дополнения связаны с наличием массива процессов-кассиров и процессов-клиентов, принудительно внедренных в систему. Обратите также внимание на установку размера динамического массива и заполнение его ссылками на создаваемые объекты:

SetLength(Cashmen, CashCount);

**for** i := 0 **to** CashCount - 1 **do**

Cashmen[i] := TCashman.Create(TClient.Create(True));

**for** i := 1 **to** StartClientNum **do**

TClient.Create(True).Insert(Queue);

Что касается основного алгоритма имитации, то он, как и в предыдущем случае, не представляет собой ничего сложного. Единственное дополнение состоит в том, что надо в самом начале имитации активировать всех кассиров, а также всех клиентов, изначально внедренных в систему, чтобы они запомнили время начала работы:

**procedure** TBankSimulation.RunSimulation;

**begin**

*// Запустить процесс создания клиентов*

Generator.ActivateDelay(0);

*// Активировать кассиров*

ActivateAllDelay(Cashmen, 0);

*// Активировать клиентов, принудительно поставленных в очередь,*

*// чтобы они запомнили время начала работы*

ActivateAllDelay(Queue, 0);

*// Ждать конца имитации*

Passivate;

StopStat;

**end**;

Текст главного модуля программы обсуждать не будем. Он, очевидно, почти повторяет тот, что был в предыдущем проекте, с теми дополнениями, которые соответствуют изменениям в постановке задачи.

Результат одного из прогонов представлен ниже:

Finished at 517.089

Clients in bank time:

Average = 22.368 +- 10.755

Min = 6.536, max = 48.608

Total = 100 values

Cash usage:

Device count = 2

Average usage = 1.825 +- 0.463

Current running = 1, completed = 100

Average blockage = 0.000

Max idle devices = 2, max busy devices = 2

Clients queue:

Average length = 2.502 +- 2.385

Max = 9, current = 0

Average waiting time = 13.070

System calendar:

Average length = 2.825 +- 0.463

Max = 3, current = 3

Average waiting time = 3.458

Not waited: 20

Not serviced: 0

In bank time histogram:

-INF - 6.00 : 0 ( 0.00%), 0.00%

6.00 - 8.00 : 3 ( 3.00%), 3.00% \*

8.00 - 10.00 : 7 ( 7.00%), 10.00% \*\*\*O

10.00 - 12.00 : 16 (16.00%), 26.00% \*\*\*\*\*\* O

12.00 - 14.00 : 6 ( 6.00%), 32.00% \*\* O

14.00 - 16.00 : 2 ( 2.00%), 34.00% \* O

16.00 - 18.00 : 4 ( 4.00%), 38.00% \*\* O

18.00 - 20.00 : 7 ( 7.00%), 45.00% \*\*\* O

20.00 - 22.00 : 10 (10.00%), 55.00% \*\*\*\* O

22.00 - 24.00 : 5 ( 5.00%), 60.00% \*\* O

24.00 - 26.00 : 3 ( 3.00%), 63.00% \* O

26.00 - 28.00 : 8 ( 8.00%), 71.00% \*\*\* O

28.00 - 30.00 : 6 ( 6.00%), 77.00% \*\* O

30.00 - 32.00 : 3 ( 3.00%), 80.00% \* O

32.00 - 34.00 : 1 ( 1.00%), 81.00% O

34.00 - 36.00 : 8 ( 8.00%), 89.00% \*\*\* O

36.00 - 38.00 : 2 ( 2.00%), 91.00% \* O

38.00 - 40.00 : 3 ( 3.00%), 94.00% \* O

40.00 - 42.00 : 0 ( 0.00%), 94.00% O

42.00 - 44.00 : 2 ( 2.00%), 96.00% \* O

44.00 - 46.00 : 2 ( 2.00%), 98.00% \* O

46.00 - +INF : 2 ( 2.00%), 100.00% \* O

Полный текст программы находится в папке 3.1.3 Bank 2, а ее визуализированного варианта — в папке 3.1.3 Bank 2 Visual.

### 3.1.4. Имитация без использования процессов

Выше мы рассмотрели технологию построения моделирующих программ на основе процессов. Однако заметим, что процесс не всегда является единственно возможным способом представления того или иного объекта, участвующего в имитации. Действительно, какие действия выполняет процесс-клиент в уже рассмотренном примере? Он встает на обслуживание в очередь, а потом в очередь завершенных процессов. Плюс к тому перед началом своей работы он запоминает время запуска (которое в данном случае совпадает с временем включения объекта в систему). Вот, собственно, и все.

Но те же самые действия вполне можно поручить процессам, обслуживающим его: генератору и кассиру. Тогда необходимость в клиенте как в процессе полностью отпадает, и клиента можно представлять просто ячейкой списка, в которой хранится его время прибытия. Это упростит программу и снизит ее затраты ресурсов компьютера, в первую очередь, памяти. Однако действия, которые раньше выполнялись автоматически или с помощью методов класса TProcess, теперь придется формулировать вручную. Впрочем, они не представляют сложности.

Итак, теперь класс клиента является не процессом, а просто ячейкой списка:

TClient = **class**(TLink)

**public**

StartingTime : Double;

**end**;

Естественно, теперь говорить о каком бы то ни было алгоритме работы клиента не приходится. Его просто нет.

В алгоритм генератора придется внести дополнения, чтобы помечать время создания клиента и помещать его в список ожидания:

**procedure** TClientGenerator.RunProcess;

**var**

i : Integer;

par : TBankSimulation;

cl : TClient;

**begin**

par := Parent **as** TBankSimulation;

**for** i := 1 **to** MaxClientCount **do**

**begin**

ClearFinished;

*// Создать клиента*

cl := TClient.Create;

*// Отметить время создания*

cl.StartingTime := SimTime;

*// Включить клиента в систему*

cl.Insert(par.Queue);

*// Активировать кассира*

par.Cashman.ActivateDelay(0);

Hold(rndClient.Exponential(MeanClientInterval));

**end**;

**end**;

Изменения в алгоритме кассира касаются извлечения клиента из очереди и завершения его обслуживания. Приведем только измененные фрагменты, остальной текст не меняется:

*// Извлечь первого клиента из очереди*

Client := par.Queue.First **as** TClient;

Client.Insert(par.RunningObjects);

…

*// Завершить работу клиента*

Client.Insert(par.FinishedObjects);

Полный текст программы с визуализацией находится в папке 3.1.4 Bank No Processes.

## 3.2. Модель магазина

*Постановка задачи.* Следующая задача является развитием предыдущей. Перенесем действующих лиц в магазин самообслуживания. Покупатели прибывают в магазин с экспоненциально распределенным промежутком с математическим ожиданием 2 минуты. Покупатель, прибыв в магазин, производит выбор покупок, на что затрачивает время, равномерно распределенное в интервале от 2 до 12 минут. За это время он выбирает от 2 до 15 (включительно) покупок с равномерным распределением. После выбора покупок покупатель встает в очередь к кассе. Кассир обслуживает покупателя за время, распределенное в соответствии с распределением Эрланга с параметрами, равными 0,2 минуты и количество покупок. Другими словами, время обслуживания складывается из нескольких промежутков времени, каждый из которых имеет экспоненциально распределенное значение с математическим ожиданием, равным 0,2, а количество промежутков равно числу покупок. Требуется провести имитацию в течение 8 часов (480 минут) и собрать статистику по времени нахождения покупателей в магазине (от прибытия до завершения обслуживания), очереди покупателей, загруженности кассира и количеству покупателей в торговом зале, выбирающих покупки.

### 3.2.1. Исходная задача

Сначала, как и в предыдущей задаче, построим определения классов процессов. Классы генератора и кассира, как и в предыдущей задаче (в ее исходной постановке), не содержат никаких дополнительных полей, поэтому повторно приводить такие же определения здесь не будем. Класс покупателя отличается от клиента банка тем, что у каждого покупателя имеется свое число покупок, которое должно быть известно кассиру. Следовательно, поместим это значение в открытое поле покупателя. В классе имитации дополнительно следует добавить статистику по нахождению покупателей в торговом зале. Какого типа должна быть эта статистика? Поскольку выбор покупок фактически является действием покупателя, которое он совершает перед постановкой в очередь, воспользуемся для этого статистикой по действиям. Зато, в отличие от предыдущей задачи, в условии не требуется получать гистограмму, поэтому соответствующее поле из класса исключим. Получаются такие определения классов покупателя и имитации:

*// Класс TCustomer - покупатель*

TCustomer = **class**(TProcess)

**public**

*// Количество покупок*

BuysCount : Integer;

**protected**

**procedure** RunProcess; **override**;

**end**;

*// Класс TShop - имитация работы магазина*

TShop = **class**(TSimulation)

**public**

Queue : TList;

Generator : TGenerator;

Cash : TCashman;

CashStat : TServiceStatistics;

TimeStat : TStatistics;

*// Статистика по нахождению покупателей в торговом зале*

PeopleStat : TActionStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

Алгоритм работы покупателя отличается от алгоритма клиента в модели банка по следующим двум параметрам. Во-первых, покупатель должен определить количество своих покупок. Во-вторых, покупатель должен выполнить действие выбора покупок в торговом зале со сбором соответствующей статистики. С учетом этих различий алгоритм работы покупателя выглядит следующим образом:

**procedure** TCustomer.RunProcess;

**var**

par : TShop;

**begin**

par := Parent **as** TShop;

StartRunning;

*// Задать число покупок*

BuysCount := rndService.NextInt(MinBuysCount, MaxBuysCount);

*// Выбор покупок*

par.PeopleStat.Start(SimTime);

Hold(rndService.Uniform(MinShoppingTime, MaxShoppingTime));

par.PeopleStat.Finish(SimTime);

*// Активировать кассира*

par.Cash.ActivateDelay(0);

*// Встать в очередь к кассе*

Wait(par.Queue);

*// По окончании обслуживания завершить работу*

Finish;

**end**;

Алгоритм работы кассира в этой задаче практически не отличается от предыдущей задачи. Однако для внесения разнообразия построим его по иной схеме. В предыдущей задаче во внешнем бесконечном цикле присутствовал внутренний цикл ожидания, который переводил покупателя в пассивное состояние, пока очередь была пуста. Здесь построим алгоритм так, что внутренний цикл будет проводить обслуживание очередного покупателя до тех пор, пока в очереди есть хотя бы один покупатель. Перестроенный по этому правилу алгоритм выглядит следующим образом:

**procedure** TCashman.RunProcess;

**var**

Cust : TCustomer;

par : TShop;

**begin**

par := Parent **as** TShop;

**while** True **do**

**begin**

**while** **not** par.Queue.Empty **do**

**begin**

*// Обратиться к очередному покупателю*

Cust := par.Queue.First **as** TCustomer;

*// Извлечь из очереди*

Cust.StartRunning;

*// Кассир занят*

par.CashStat.Start(SimTime);

*// Рассчитать покупателя*

Hold(rndService.Erlang(TimePerBuy, Cust.BuysCount));

*// Покупатель обслужен - кассир свободен*

par.CashStat.Finish(SimTime);

par.TimeStat.AddData(SimTime - Cust.StartingTime);

*// Активировать его завершение*

Cust.ActivateDelay(0);

**end**;

*// Ждать очередного покупателя*

Passivate;

**end**;

**end**;

Генератор покупателей в этой задаче отличается только типом цикла. Поскольку конкретное число покупателей не задано, используется бесконечный цикл, а прерван он будет с окончанием всей имитации:

**procedure** TGenerator.RunProcess;

**begin**

**while** True **do**

**begin**

ClearFinished;

*// Создать нового покупателя и поместить его в систему*

TCustomer.Create.ActivateDelay(0);

*// Подождать перед созданием следующего*

Hold(rndCust.Exponential(MeanCustInterval));

**end**;

**end**;

Алгоритм имитации также предельно простой. В нем активируется генератор, потом имитация приостанавливается на заданное время, по истечении которого корректирует статистику и завершает работу:

**procedure** TShop.RunSimulation;

**begin**

*// Запустить генератор*

Generator.ActivateDelay(0);

*// Ждать окончания моделирования*

Hold(SimulationTime);

*// Закончить статистику*

StopStat;

**end**;

Остальные методы, а также текст главного модуля программы приводить здесь не будем, поскольку они не содержат в себе ничего нового и построены по той же схеме, что и в предыдущей задаче. Один из прогонов имитации выдал результат:

Cash statistics:

Device count = 1

Average usage = 0.823 +- 0.382

Current running = 1, completed = 235

Average blockage = 0.000

Max idle time = 7.379, max busy time = 120.763

In system time statistics:

Average = 13.013 +- 4.869

Min = 2.820, max = 25.266

Total = 235 values

People in shop statistics:

Average = 3.541 +- 2.102

Max = 11

Current running = 0, completed = 239

Customers queue statistics:

Average length = 2.098 +- 2.143

Max = 10, current = 3

Average waiting time = 4.249

Calendar statistics:

Average length = 6.364 +- 2.133

Max = 14, current = 3

Average waiting time = 2.101

Полный текст программы находится в папке 3.2.1 Shop. В визуализированной версии этой программы нет никаких особенностей, заслуживающих отдельного обсуждения. Ее текст находится в папке 3.2.1 Shop Visual.

### 3.2.2. Многократные прогоны

Как уже можно было заметить из приведенных примеров, результаты отдельных прогонов могут ощутимо различаться. Однако это противоречит одной из основных целей моделирования: получить достоверные данные о поведении реальной системы. Действительно, какие выводы можно сделать о моделируемой системе или процессе, если при каждом запуске получаются разные результаты? Чтобы модель была более достоверной и, следовательно, практически полезной, следует каким-то образом обеспечить, чтобы можно было более определенно судить о результатах моделирования в целом. Одним из простых способов решения этой проблемы является многократный прогон модели.

Суть этого способа состоит в том, что производятся многократные прогоны модели, и собирается статистика по интересующим нас результатам. Не вдаваясь в подробности математической статистики, заметим, что для получения вполне достоверной информации достаточно сравнительно небольшого числа прогонов, исчисляемого несколькими сотнями, что вполне под силу даже компьютерам небольшой мощности.

Построим программу, выполняющую многократные прогоны рассмотренной модели магазина самообслуживания. Сначала выберем данные, по которым будет собираться статистика. Для рассматриваемой задачи вполне разумный выбор включает в себя следующие данные: средняя занятость кассира, среднее время нахождения покупателей в магазине, среднее и максимальное количество покупателей в торговом зале, максимальная длина очереди, среднее время ожидания в очереди. Кроме того, построим гистограмму по распределению средних времен нахождения покупателей в магазине.

Для сбора всех этих данных нам потребуется создать соответствующие объекты. Заметим, что это будут совсем не те объекты сбора статистики, что уже имеются в классе имитации. Имеющиеся объекты статистики в данном случае будут источником исходных данных для сбора обобщенной статистики. Новые объекты будут собирать точечную статистику, поскольку в данном случае речь идет о наборе результатов независимых прогонов. Таким образом, набор глобальных переменных не нуждается в дополнительных пояснениях:

**var**

Shop : TShop;

CashUsageStat : TStatistics;

TimeStat : TStatistics;

InShopStat : TStatistics;

InShopMaxStat : TStatistics;

MaxQueueLenStat : TStatistics;

WaitStat : TStatistics;

TimeHist : TUniformHistogram;

RunsCount : Integer = 400;

HistMin : Double = 8;

HistStep : Double = 1;

HistStepCount : Integer = 20;

Что касается непосредственно самой модели, то она не нуждается в каких-либо изменениях, и файл соответствующего модуля (для консольного варианта) можно просто скопировать в папку нового проекта или даже включить в состав проекта без копирования. Разумеется, модуль USimulation тоже следует включить в состав проекта.

Сам алгоритм программы предельно прост. В начале создаются все необходимые объекты сбора статистики, затем в цикле создается и выполняется очередной прогон имитации, его данные собираются. После завершения всех прогонов выводятся результаты собранной статистики. Поскольку процесс выполнения многократных прогонов может занять ощутимое время (от нескольких секунд до нескольких минут), то, чтобы у пользователя не возникло ощущения, что программа «зависла», на каждом 10-м прогоне на экран выводится точка:

CashUsageStat := TStatistics.Create;

TimeStat := TStatistics.Create;

InShopStat := TStatistics.Create;

InShopMaxStat := TStatistics.Create;

MaxQueueLenStat := TStatistics.Create;

WaitStat := TStatistics.Create;

TimeHist := TUniformHistogram.Create(HistMin, HistStep, HistStepCount);

**for** i := 1 **to** RunsCount **do**

**begin**

*// Создать имитацию*

Shop := TShop.Create;

*// Запуск*

SwitchTo(Shop);

*// Сбор статистики*

CashUsageStat.AddData(Shop.CashStat.Mean);

TimeStat.AddData(Shop.TimeStat.Mean);

InShopStat.AddData(Shop.PeopleStat.Mean);

InShopMaxStat.AddData(Shop.PeopleStat.Max);

MaxQueueLenStat.AddData(Shop.Queue.LengthStat.Max);

WaitStat.AddData(Shop.Queue.WaitStat.Mean);

TimeHist.AddData(Shop.TimeStat.Mean);

Shop.Free;

**if** i **mod** 10 = 0 **then**

Write('.');

**end**;

WriteLn;

WriteStat('Cash usage statistics:', CashUsageStat);

WriteLn;

WriteStat('In system time statistics:', TimeStat);

WriteLn;

WriteStat('People in shop statistics:', InShopStat);

WriteLn;

WriteStat('Max people in shop statistics:', InShopMaxStat);

WriteLn;

WriteStat('Max queue length statistics:', MaxQueueLenStat);

WriteLn;

WriteStat('Waiting time statistics:', WaitStat);

WriteLn;

WriteHist('Mean in system time histogram:', TimeHist);

Результаты, полученные при выполнении одной из серий прогонов, приведены ниже:

........................................

Cash usage statistics:

Average = 0.826 +- 0.060

Min = 0.636, max = 0.980

Total = 400 values

In system time statistics:

Average = 14.127 +- 3.051

Min = 9.708, max = 30.580

Total = 400 values

People in shop statistics:

Average = 3.469 +- 0.258

Min = 2.725, max = 4.259

Total = 400 values

Max people in shop statistics:

Average = 10.080 +- 1.305

Min = 7.000, max = 14.000

Total = 400 values

Max queue length statistics:

Average = 11.457 +- 3.958

Min = 4.000, max = 27.000

Total = 400 values

Waiting time statistics:

Average = 5.448 +- 3.025

Min = 1.176, max = 21.596

Total = 400 values

Mean in system time histogram:

-INF - 8.00 : 0 ( 0.00%), 0.00%

8.00 - 9.00 : 0 ( 0.00%), 0.00%

9.00 - 10.00 : 1 ( 0.25%), 0.25%

10.00 - 11.00 : 23 ( 5.75%), 6.00% \*\*

11.00 - 12.00 : 59 (14.75%), 20.75% \*\*\*\*\*\* O

12.00 - 13.00 : 77 (19.25%), 40.00% \*\*\*\*\*\*\*\* O

13.00 - 14.00 : 82 (20.50%), 60.50% \*\*\*\*\*\*\*\* O

14.00 - 15.00 : 67 (16.75%), 77.25% \*\*\*\*\*\*\* O

15.00 - 16.00 : 24 ( 6.00%), 83.25% \*\* O

16.00 - 17.00 : 16 ( 4.00%), 87.25% \*\* O

17.00 - 18.00 : 10 ( 2.50%), 89.75% \* O

18.00 - 19.00 : 12 ( 3.00%), 92.75% \* O

19.00 - 20.00 : 8 ( 2.00%), 94.75% \* O

20.00 - 21.00 : 7 ( 1.75%), 96.50% \* O

21.00 - 22.00 : 3 ( 0.75%), 97.25% O

22.00 - 23.00 : 2 ( 0.50%), 97.75% O

23.00 - 24.00 : 1 ( 0.25%), 98.00% O

24.00 - 25.00 : 0 ( 0.00%), 98.00% O

25.00 - 26.00 : 1 ( 0.25%), 98.25% O

26.00 - 27.00 : 3 ( 0.75%), 99.00% O

27.00 - 28.00 : 2 ( 0.50%), 99.50% O

28.00 - +INF : 2 ( 0.50%), 100.00% O

Интересно оценить полученные результаты. Так, видно, что средняя загруженность кассира, равно как и среднее количество покупателей в торговом заде мало меняются от прогона к прогону — стандартное отклонение не превышает 10% от среднего значения. Немного уступает в этом отношении значение максимального количества покупателей в торговом зале — стандартное отклонение составляет около 13% среднего значения, а максимальное и минимальное значения различаются вдвое. Следовательно, по этим значениям можно делать достаточно уверенные прогнозы. А вот, к примеру, величина среднего времени ожидания в очереди имеет очень сильный разброс значений: стандартное отклонение превышает половину среднего значения, а крайние значения различаются почти в 20 раз. Следовательно, для этой величины сколько-нибудь точные прогнозы невозможны.

Приведенный способ сбора данных по многократным прогонам не является единственно возможным. Данные по ряду изучаемых величин можно собирать непосредственно в процессе всех прогонов. В этом случае соответствующие объекты сбора статистики должны быть определены как глобальные и исключены из класса имитации. Соответственно, их создание и удаление возлагается на главную программу.

Полный текст программы находится в папке 3.2.2 Shop Multi.

При выполнении многократных прогонов в оконном приложении исчезает необходимость в визуализации. Действительно, если однократный прогон в таком режиме занимает не одну минуту, то сколько времени потребуется на выполнение хотя бы одной сотни прогонов? Поэтому модуль модели переносится из консольного варианта проекта без каких-либо дополнений. В форме приложения размещается вся необходимая инфраструктура: кнопка пуска, табличные сетки для сбора статистики, текстовые поля для ввода исходных данных.

Как и в консольном варианте, здесь также желательно информировать пользователя о ходе выполнения имитации, поэтому предусмотрим еще отображение счетчика выполненных имитаций. Однако если построить программу как консольную, с выполнением прогонов в цикле со счетчиком и отображением данных после каждого прогона, то ее результат будет виден только после завершения всех прогонов. То есть, информирование пользователя будет произведено только в самом конце работы, а вовсе не в процессе. Это связано с особенностями отображения состояния объектов управления в оконных приложениях: оно не отображается мгновенно при записи значений их свойств, а откладывается до завершения процедуры события.

Решить проблему можно, поместив создание и однократный прогон модели со сбором статистики в алгоритм процедуры события таймера. Для максимально быстрого исполнения серии прогонов следует задать таймеру значение свойства Interval, равное 1 — фактически, это означает «сработать сразу, как только это станет возможным».

Текст визуализированной программы, выполняющей много прогонов, приведен в папке 3.2.2 Shop Multi Visual.

## 3.3. Модель поточной линии

*Постановка задачи.* На поточной производственной линии находятся два рабочих, обрабатывающих поступающие изделия. Каждое изделие сначала поступает на обработку к первому рабочему, а от него — ко второму. Перед первым рабочим есть место для накопления четырех изделий, ожидающих обработки. Между первым и вторым рабочим есть место для двух изделий.

Если в момент поступления очередного изделия отсутствуют свободные места в очереди перед первым рабочим, оно снимается с обслуживания и удаляется из системы. Если в момент, когда первый рабочий заканчивает обработку очередного изделия, между ним и вторым рабочим нет свободных мест для его размещения, его работа блокируется: он прекращает работу и ждет до тех пор, пока не появится свободное место. При появлении свободного места первый рабочий помещает туда обработанное изделие и берет следующее для обработки.

Поступление изделий для обработки происходит с экспоненциально распределенным интервалом с математическим ожиданием 0,4 единицы времени. Обработка изделий рабочими имеет экспоненциально распределенную длительность с математическим ожиданием 0,25 единиц времени для первого рабочего и 0,5 единиц времени — для второго.

Требуется выполнить имитацию в течение 300 единиц времени и собрать следующую статистику: по времени пребывания изделий в системе (тех, которым не было отказано в обслуживании), по интервалу времени между отказами в обслуживании, по загруженности каждого рабочего в отдельности, по состоянию каждой из очередей в отдельности.

### 3.3.1. Блокировка

В этой задаче мы сталкиваемся с новой для нас ситуацией — *блокированием* процессов. При обнаружении наличия условия блокировки процесс должен уведомить об этом свою статистику занятости и перейти в пассивное состояние. После выхода из пассивного состояния процесс также должен уведомить об этом свою статистику занятости. Активирует заблокированный процесс какой-то другой процесс, который предположительно снимает условие блокировки. Поскольку заблокированный процесс, по идее, может активироваться разными процессами, то сам факт его активации в состоянии блокировки еще не означает выхода из него. Поэтому в заблокированном состоянии процесс должен постоянно проверять состояние блокировки. В целом, поведение заблокированного процесса кодируется алгоритмом такого рода:

**if** Условие блокировки **then**

**begin**

*// Войти в заблокированное состояние*

Stat.StartBlock(SimTime);

*// Ожидать окончания блокировки*

**while** Условие блокировки **do**

Passivate;

*// Выйти из заблокированного состояния*

Start.FinishBlock(SimTime);

**end**;

Что касается остальных составляющих задачи, то, очевидно, обрабатываемое изделие здесь является компонентом. В качестве примера рассмотрим применение здесь другого способа порождения компонентов, также обсуждавшегося в параграфе 2.4: каждый запущенный процесс-компонент создает следующий и активирует его с заданной задержкой времени (заметим, что, если использовать тот же подход, что и в предыдущих примерах, то есть с применением процесса-генератора, то компонент не обязательно представлять процессом; достаточно построить новый класс на основе ячейки списка). Ниже приводится текст алгоритма процесса изделия:

**procedure** TPiece.RunProcess;

**var**

par : TFlowLineSimulation;

**begin**

par := Parent **as** TFlowLineSimulation;

ClearFinished;

*// Запланировать прибытие следующего изделия*

TPiece.Create.ActivateDelay(rndPiece.Exponential(PieceMeanInterval));

*// Если очередь к первому рабочему месту не заполнена*

**if** par.Queue1.Size < Queue1Size **then**

**begin**

*// Активировать обслуживание*

par.Worker1.ActivateDelay(0);

*// Встать в очередь ожидания*

Wait(par.Queue1);

**end**

**else**

**begin**

*// Добавить статистику по отказам*

par.Balks.AddData(SimTime);

**end**;

Finish;

**end**;

Процесс, имитирующий первого рабочего, отличается от ранее рассмотренных обслуживающих процессов только тем, что в нем предусматривается возможность блокировки в случае, когда вторая очередь заполнена:

**procedure** TWorker1.RunProcess;

**var**

Piece : TPiece;

par : TFlowLineSimulation;

**begin**

par := Parent **as** TFlowLineSimulation;

*// Первое рабочее место*

**while** True **do**

**begin**

*// Ждать появления изделий в очереди*

**while** par.Queue1.Empty **do**

Passivate;

*// Начать операцию*

par.Stat1.Start(SimTime);

*// Извлечь очередное изделие*

Piece := par.Queue1.First **as** TPiece;

Piece.StartRunning;

*// Выполнить обслуживание*

Hold(rndWorker1.Exponential(Worker1MeanTime));

*// Закончить операцию*

par.Stat1.Finish(SimTime);

*// Если очередь ко второму рабочему месту заполнена*

**if** par.Queue2.Size >= Queue2Size **then**

**begin**

*// Заблокировать процесс*

par.Stat1.StartBlock(SimTime);

*// Ожидать наличия места в очереди ко второму рабочему месту*

**while** par.Queue2.Size >= Queue2Size **do**

Passivate;

*// Разблокировать*

par.Stat1.FinishBlock(SimTime);

**end**;

*// Поместить изделие в очередь*

Piece.Insert(par.Queue2);

*// Активировать второй процесс*

par.Worker2.ActivateDelay(0);

**end**;

**end**;

Процесс, имитирующий второго рабочего, еще проще. От ранее рассмотренных обслуживающих процессов он отличается, помимо сбора статистики, только тем, что после извлечения очередного изделия из своей очереди, активирует предыдущего рабочего — на тот случай, если он был заблокирован по причине переполненности очереди:

**procedure** TWorker2.RunProcess;

**var**

Piece : TPiece;

par : TFlowLineSimulation;

**begin**

par := Parent **as** TFlowLineSimulation;

*// Второе рабочее место*

**while** True **do**

**begin**

*// Ожидать появления изделий в очереди*

**while** par.Queue2.Empty **do**

Passivate;

*// Начать операцию*

par.Stat2.Start(SimTime);

*// Извлечь изделие*

Piece := par.Queue2.First **as** TPiece;

Piece.StartRunning;

*// Активировать первого работника на случай, если он был заблокирован*

par.Worker1.ActivateDelay(0);

*// Выполнить обслуживание*

Hold(rndWorker2.Exponential(Worker2MeanTime));

*// Зафиксировать статистику по времени пребывания в системе*

par.TimeInSystem.AddData(SimTime - Piece.StartingTime);

par.TimeHist.AddData(SimTime - Piece.StartingTime);

*// Закончить операцию*

par.Stat2.Finish(SimTime);

*// Активировать изделия для завершения работы*

Piece.ActivateDelay(0);

**end**;

**end**;

Определение класса имитации и его алгоритм построены по уже традиционной схеме и в комментариях не нуждаются:

TFlowLineSimulation = **class**(TSimulation)

**public**

*// Статистика по отказам*

Balks : TTimeBetStatistics;

*// Статистика по времени нахождения в системе*

TimeInSystem : TStatistics;

*// Гистограмма по времени нахождения в системе*

TimeHist : TUniformHistogram;

*// Первое рабочее место*

Worker1 : TWorker1;

*// Второе рабочее место*

Worker2 : TWorker2;

*// Очередь изделий, ожидающих обслуживания на первом рабочем месте*

Queue1 : TList;

*// Очередь изделий, ожидающих обслуживания на втором рабочем месте*

Queue2 : TList;

*// Статистика по первой операции*

Stat1 : TServiceStatistics;

*// Статистика по второй операции*

Stat2 : TServiceStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

**procedure** TFlowLineSimulation.RunSimulation;

**begin**

*// Поместить первое изделие в момент 0*

TPiece.Create.ActivateDelay(0);

*// Выполнение моделирования*

Hold(SimulationTime);

StopStat;

**end**;

Тексты остальных методов и главной программы здесь не приводятся, потому что они построены на тех же принципах, что и в предыдущих примерах. Один из прогонов программы дал следующие результаты:

Time in system:

Average = 3.023 +- 1.538

Min = 0.093, max = 9.734

Total = 547 values

Time between balks:

Average = 1.334 +- 3.100

Min = 0.001, max = 29.240

Total = 203 values

Worker 1:

Device count = 1

Average usage = 0.470 +- 0.499

Current running = 0, completed = 549

Average blockage = 0.420

Max idle time = 2.681, max busy time = 4.552

Worker 2:

Device count = 1

Average usage = 0.932 +- 0.252

Current running = 1, completed = 547

Average blockage = 0.000

Max idle time = 1.699, max busy time = 78.326

Queue 1:

Average length = 2.176 +- 1.459

Max = 4, current = 0

Average waiting time = 1.189

Queue 2:

Average length = 1.517 +- 0.766

Max = 2, current = 1

Average waiting time = 0.830

Calendar:

Average length = 3.402 +- 0.516

Max = 4, current = 3

Average waiting time = 0.237

Time in system histogram:

-INF - 0.00 : 0 ( 0.00%), 0.00%

0.00 - 0.50 : 8 ( 1.46%), 1.46% \*

0.50 - 1.00 : 35 ( 6.40%), 7.86% \*\*\*

1.00 - 1.50 : 49 ( 8.96%), 16.82% \*\*\*\* O

1.50 - 2.00 : 54 ( 9.87%), 26.69% \*\*\*\* O

2.00 - 2.50 : 68 (12.43%), 39.12% \*\*\*\*\* O

2.50 - 3.00 : 72 (13.16%), 52.29% \*\*\*\*\* O

3.00 - 3.50 : 69 (12.61%), 64.90% \*\*\*\*\* O

3.50 - 4.00 : 64 (11.70%), 76.60% \*\*\*\*\* O

4.00 - 4.50 : 49 ( 8.96%), 85.56% \*\*\*\* O

4.50 - 5.00 : 31 ( 5.67%), 91.22% \*\* O

5.00 - 5.50 : 17 ( 3.11%), 94.33% \* O

5.50 - 6.00 : 9 ( 1.65%), 95.98% \* O

6.00 - 6.50 : 8 ( 1.46%), 97.44% \* O

6.50 - 7.00 : 6 ( 1.10%), 98.54% O

7.00 - 7.50 : 2 ( 0.37%), 98.90% O

7.50 - 8.00 : 1 ( 0.18%), 99.09% O

8.00 - 8.50 : 0 ( 0.00%), 99.09% O

8.50 - 9.00 : 3 ( 0.55%), 99.63% O

9.00 - 9.50 : 1 ( 0.18%), 99.82% O

9.50 - 10.00 : 1 ( 0.18%), 100.00% O

10.00 - +INF : 0 ( 0.00%), 100.00% O

Полный текст проекта находится в папке 3.3.1 FlowLine.

### 3.3.2. Визуализация

Как было рассмотрено в предыдущих параграфах, процесс построения визуализации на основе готовой консольной имитации не вызывает особых затруднений. Творческий подход здесь возможен разве что в части разработки интерфейса и способов визуализации. Однако поскольку в центре нашего внимания находится программирование самой задачи моделирования, эти вопросы не представляют большого интереса. Поэтому, чтобы сделать процесс визуализации задачи более интересным, внесем в ее постановку некоторые изменения.

Предположим, следует изучить, каким образом повлияет на выполнение имитации разное распределение имеющихся шести мест ожидания изделий между двумя очередями. При этом не будем исключать и крайних вариантов, когда на ту или иную очередь не отводится мест вообще. Собственно, именно наличие этих крайностей вынуждает нас вносить в текст алгоритмов какие-либо изменения. Если бы предельный размер каждой очереди мог быть не меньше одного изделия, то вся задача свелась бы к заданию двух числовых параметров предельной длины очереди.

Рассмотрим, как поменяется поведение программы, если, к примеру, первая очередь имеет нулевую максимальную длину, или, другими словами, перед первым рабочим не предусмотрено места для размещения изделий. Если изделие прибывает в тот момент, когда рабочий свободен, оно сразу поступает к нему на обработку. Если же в момент прибытия изделия рабочий занят выполнением обслуживания или заблокирован, то происходит отказ в обслуживании, и изделие снимается с обработки. Похожие же изменения имеют место в том случае, если предельная длина второй очереди равна нулю. Разница здесь только в том, что в случае невозможности немедленной передачи изделия второму рабочему первый рабочий блокируется.

Очевидно, в алгоритме работы должен быть предусмотрен особый способ проверки условия, если предельная длина той или иной очереди равна нулю. Например, в случае, когда таковой является длина первой очереди, при поступлении изделия следует проверить состояние первого рабочего посредством обращения к его статистике действия. Однако каким образом рабочий сможет взять изделие, если он его умеет брать только из очереди? Ответ прост: поместить изделие в очередь обычным образом. Максимальная длина очереди, равная нулю, не должна нас в данном случае смущать. Дело в том, что сразу же (то есть, с нулевой задержкой) после постановки изделия в очередь активируется рабочий, который извлекает изделие из очереди. Поскольку изделие находится в очереди в течение промежутка времени нулевой длительности, этот факт не окажет никакого влияния на статистические результаты длины. Что касается статистики по времени ожидания, то все изделия будут находиться в очереди в течение нулевого промежутка времени, поэтому данная статистика интереса не представляет.

Соответствующим образом усложняется условие постановки в очередь:

*// Если очередь к первому рабочему не заполнена*

**if** ((Queue1Size = 0) **and** (par.Stat1.Running = 0) **and** (par.Stat1.Blocked = 0)) **or**

((Queue1Size > 0) **and** (par.Queue1.Size < Queue1Size)) **then**

Условие фактически состоит из двух частей. Если предельная длина очереди равна нулю, то проверяется, что первый рабочий не занят и не заблокирован. Если же длина больше нуля, то производится обычное сравнение текущей длины очереди с предельной.

Изменения, вносимые в условия в алгоритме первого рабочего, в общем, аналогичны. Разница в том, что второй рабочий не может быть заблокирован, и поэтому проверять это условие нет необходимости:

*// Если очередь ко второму рабочему заполнена*

**if** ((Queue2Size = 0) **and** (par.Stat2.Running > 0)) **or**

((Queue2Size > 0) **and** (par.Queue2.Size >= Queue2Size)) **then**

Не забудьте только, что это условие должно поменяться одновременно в двух местах: в указанном условном операторе и в цикле **while**, в котором производится ожидание завершения блокировки.

Алгоритм работы второго рабочего также потребует дополнения. Дело в том, что в случае нулевой длины второй очереди он, завершив обслуживание очередного изделия, должен каким-то образом сообщить об этом первому рабочему, чтобы тот завершил блокировку и передал ему следующее изделие. Очевидно, это можно сделать, активировав первого рабочего после завершения обслуживания изделия:

*// Активировать первого рабочего*

par.Worker1.ActivateDelay(0);

На самом деле, эта строчка уже есть в алгоритме, только она находится перед началом обслуживания. Надо просто добавить ее копию и после завершения действия.

Полный текст программы находится в папке 3.3.2 FlowLine Visual.

## 3.4. Модель линии контроля качества

*Постановка задачи.* Рассмотрим модель работы пункта контроля качества готовой продукции (например, телевизоров). Телевизоры поступают на проверку с интервалом, равномерно распределенным в диапазоне от 3,5 до 5,5 минут. Проверку параллельно выполняют двое контролеров, к которым имеется общая очередь. Каждый из них тратит на проверку очередного телевизора время, равномерно распределенное в интервале от 6 до 12 минут. С вероятностью 0,85 качество телевизора оказывается удовлетворительным, и он поступает на упаковку (фактически, выводится из системы). Если качество не удовлетворяет нормам, телевизор поступает в очередь к настройщику, который выполняет настройку телевизора за время, равномерно распределенное на интервале от 20 до 40 минут. Настроенный телевизор вновь поступает в очередь на проверку, и цикл его проверки повторяется. Количество циклов проверки и настройки не ограничено.

Требуется провести имитацию данной системы в течение 8-часового рабочего дня (480 минут) и собрать следующую статистику: по загруженности контролеров (одна статистика на обоих) и настройщика, по времени нахождения телевизора в системе, по очередям к контролерам и настройщику.

*Решение.* Очевидно, в данной задаче проверяемый телевизор является обслуживаемым компонентом. От него, в сущности, не требуется никаких действий: по прибытии он сразу встает в очередь ожидания, и дальнейший его путь включает только обслуживание и ожидание в той или иной очереди. Следовательно, для его реализации в модели нет необходимости создавать процесс. Что же касается дополнительных компонентов объекта, то потребуется запоминать время его создания для последующего сбора статистики. Таким образом, определение класса проверяемого телевизора оказывается очень простым:

TTVSet = **class**(TLink)

**public**

StartingTime : Double;

**end**;

Ненамного более сложным будет и определение класса генератора. Действия, которые он должен выполнять, уже описывались в пункте 3.1.4 (там же, напомним, обсуждались и другие особенности создания имитаций без использования процессов):

**procedure** TTVSetGenerator.RunProcess;

**var**

par : TTVControl;

tv : TTVSet;

**begin**

par := Parent **as** TTVControl;

**while** True **do**

**begin**

ClearFinished;

*// Создать новый телевизор*

tv := TTVSet.Create;

*// Зафиксировать время прибытия*

tv.StartingTime := SimTime;

*// Поместить телевизор в очередь проверки*

tv.Insert(par.InspectionQueue);

ActivateDelay(par.Inspectors, 0);

*// Подождать до следующего*

Hold(rndTVSet.Uniform(MinCreationDelay, MaxCreationDelay));

**end**;

**end**;

Работа контролера начинается с извлечения из очереди первого телевизора и выполнения его проверки. После этого с заданной в условии задачи вероятностью контролер принимает решение о том, исправен телевизор или нет. Если телевизор исправен, следует собрать статистику о его времени пребывания в системе и удалить. Если неисправен — поместить в очередь настройки:

**procedure** TInspector.RunProcess;

**var**

Piece : TTVSet;

par : TTVControl;

**begin**

*// Работа проверяющего*

par := Parent **as** TTVControl;

**while** True **do**

**begin**

*// Если нет телевизоров для проверки*

**while** par.InspectionQueue.Empty **do**

*// Встать в очередь и ждать прибытия*

Passivate;

*// Извлечь из очереди первый телевизор*

Piece := par.InspectionQueue.First **as** TTVSet;

Piece.Insert(par.RunningObjects);

*// Зафиксировать начало работы*

par.InspectorsStat.Start(SimTime);

*// Проверка*

Hold(rndInspector.Uniform(MinInspectionTime, MaxInspectionTime));

*// Зафиксировать конец работы*

par.InspectorsStat.Finish(SimTime);

*// С вероятностью NoAdjustmentProb телевизор исправен*

**if** rndInspector.Draw(NoAdjustmentProb) **then**

**begin**

*// Внести статистику о времени пребывания в системе*

par.TimeInSystemStat.AddData(SimTime - Piece.StartingTime);

*// Удалить телевизор*

Piece.Insert(par.FinishedObjects);

**end**

**else**

**begin**

*// Поместить телевизор в очередь на настройку*

Piece.Insert(par.AdjustmentQueue);

*// Дать сигнал настройщику*

par.Adjuster.ActivateDelay(0);

**end**;

**end**;

**end**;

Текст алгоритма настройщика построен по уже ставшим традиционными правилам:

**procedure** TAdjuster.RunProcess;

**var**

par : TTVControl;

Piece : TTVSet;

**begin**

*// Работа настройщика*

par := Parent **as** TTVControl;

**while** True **do**

**begin**

*// Если нет телевизоров для настройки, ожидать*

**while** par.AdjustmentQueue.Empty **do**

Passivate;

*// Извлечь первый телевизор из очереди*

Piece := par.AdjustmentQueue.First **as** TTVSet;

Piece.Insert(par.RunningObjects);

*// Зафиксировать начало работы*

par.AdjustmentStat.Start(SimTime);

*// Выполнить работу*

Hold(rndAdjuster.Uniform(MinAdjustmentTime, MaxAdjustmentTime));

*// Зафиксировать конец работы*

par.AdjustmentStat.Finish(SimTime);

*// Поместить телевизор в очередь на проверку*

Piece.Insert(par.InspectionQueue);

**end**;

**end**;

В текстах методов класса имитации и главной программы не содержится ничего принципиально нового, и приводить их здесь не будем. Полный текст программы находится в папке 3.4 TVControl.

Один из прогонов программы дал следующие результаты:

Time in system:

Average = 20.549 +- 25.395

Min = 6.413, max = 132.262

Total = 82 values

Inspectors:

Device count = 2

Average usage = 1.804 +- 0.403

Current running = 2, completed = 96

Average blockage = 0.000

Max idle devices = 2, max busy devices = 2

Adjustment:

Device count = 1

Average usage = 0.759 +- 0.428

Current running = 1, completed = 12

Average blockage = 0.000

Max idle time = 62.921, max busy time = 181.450

Inspection queue:

Average length = 0.605 +- 0.699

Max = 3, current = 1

Average waiting time = 2.953

Adjustment queue:

Average length = 0.622 +- 0.873

Max = 3, current = 1

Average waiting time = 22.636

Calendar:

Average length = 4.563 +- 0.610

Max = 5, current = 5

Average waiting time = 7.322

Поскольку перенос модели в среду визуализации уже превратился в рутинную процедуру, то для придания б***о***льшего интереса внесем в модель небольшое дополнение. А именно, будем подсчитывать количество процедур настройки, пройденных компонентами, и собирать статистику по ним, причем только для тех компонентов, которые хотя бы один раз проходили настройку.

Первое, что для этого потребуется сделать, — добавить в класс компонента поле, в котором подсчитывается количество пройденных процедур настройки:

TTVSet = **class**(TLink)

**public**

StartingTime : Double;

AdjustmentCount : Integer;

**end**;

Далее, в класс имитации следует добавить новый объект сбора статистики, не забыв добавить в метод Init его создание, а в деструктор — удаление. В процедуру создания компонентов в методе генератора следует поместить оператор, который вносит в поле подсчета количества циклов настройки исходное нулевое значение, а в процедуру контролера — операторы, которые добавляют данные в статистику. Последние два действия имеют вид:

*// Циклов настройки не было*

tv.AdjustmentCount := 0;

…

*// Внести статистику о количестве циклов настройки*

**if** Piece.AdjustmentCount > 0 **then**

par.AdjustmentCountStat.AddData(Piece.AdjustmentCount);

Полный текст программы находится в папке 3.4 TVControl Visual.

## 3.5. Модель работы по перевозке сыпучих грузов

*Постановка задачи.* Рассмотрим задачу об организации погрузки и перевозок сыпучих грузов (например, земли). Бульдозер нагребает кучу земли за время, распределенное по закону Эрланга с параметрами 4 минуты и 2. Эти кучи с помощью механических погрузчиков нагружаются на самосвалы, которые отвозят их к месту разгрузки, разгружаются и возвращаются обратно.

В наличии имеются два погрузчика, время погрузки у каждого из них распределено экспоненциально с математическим ожиданием 12 минут для одного и 14 минут для другого. Для начала работы погрузчика необходимо наличие хотя бы двух куч земли и одного самосвала, готового к погрузке. Погрузчик нагружает две кучи в самосвал, а затем в течение 5 минут готовится к следующей погрузке. Если к началу погрузки оба погрузчика свободны, работу начинает тот из них, который к этому моменту дольше простаивал (то есть, тот, который раньше закончил предыдущую погрузку). В начале работы первым начинает работу погрузчик со средним временем погрузки 12 минут.

Нагруженный самосвал едет к месту разгрузки за время, распределенное по нормальному закону с математическим ожиданием 22 минуты и стандартным отклонением 3 минуты. Разгрузка занимает время, равномерно распределенное на интервале от 2 до 6 минут. Время обратного движения распределено нормально с матожиданием 18 минут и стандартным отклонением 3 минуты. Всего имеются 4 самосвала, которые в начальный момент имитации стоят в очереди на погрузку.

Работа бульдозера продолжается в течение 8 часов (480 минут). Имитация заканчивается, когда все кучи будут погружены (кроме, возможно, одной последней), самосвалы разгрузятся и вернутся в исходное положение.

По результатам имитации требуется собрать следующую статистику: по занятости каждого погрузчика в отдельности, по очередям куч, готовых к погрузке, а также ожидающих работы погрузчиков и самосвалов.

*Решение.* Очевидно, в данной задаче роль компонентов играют перевозимые кучи земли. Также достаточно очевидно, что бульдозер играет роль генератора компонентов. Роли погрузчиков и самосвалов несколько отличаются от тех ролей, что рассматривались в предыдущих задачах. Погрузчик в основном работает как обслуживающий процесс, однако к его алгоритму добавляется дополнительное действие — подготовка к следующей погрузке в течение 5 минут. Роль самосвала в основном аналогична роли компонента: он ожидает обслуживания погрузчиком, а затем выполняет перевозку, разгрузку и возврат. Однако, в отличие от ранее рассмотренных компонентов, все самосвалы создаются при создании имитации и не уничтожаются до ее завершения.

От кучи земли как от процессов не требуется никаких действий: бульдозер помещает ее в очередь ожидания, а погрузчик потом извлекает. Вся необходимая статистика по времени ожидания собирается автоматически очередью. Поэтому можно реализовать кучу земли как простую ячейку списка:

*// Класс THeap - куча земли. По существу, просто ячейка списка*

THeap = TLink;

При извлечении объекта-кучи из очереди его можно сразу удалять, нет необходимости помещать ее ни в какие очереди завершенных объектов. Следовательно, вызов метода ClearFinished в генераторе (бульдозере) становится ненужным. Так как имитация завершается не сразу после истечения заданного времени, а продолжается еще какое-то время после окончания работы бульдозера, желательно предусмотреть какой-то признак, указывающий на то, что работа бульдозера окончена. Таким признаком у нас будет поле Finished логического (Boolean) типа в объекте-имитации. Значит, по окончании работы бульдозер должен записать в это поле значение True. Предусмотрим некоторую оптимизацию. Бульдозер активирует обслуживающий процесс (погрузчик) не после добавления каждой кучи в очередь, а только в том случае, если их количество оказалось не меньше 2. С учетом указанных особенностей алгоритм работы бульдозера выглядит следующим образом:

**procedure** TBulldozer.RunProcess;

**var**

par : TLoading;

**begin**

par := Parent **as** TLoading;

**while** SimTime <= ModelingTime **do**

**begin**

*// Создать кучу земли и поставить ее в очередь*

THeap.Create.Insert(par.HeapQueue);

*// Если куч хотя бы минимальное количество*

**if** par.HeapQueue.Size >= MinHeapQueueSize **then**

*// Активировать первый свободный погрузчик (если есть)*

ActivateDelay(par.LoadersQueue, 0);

Hold(rndBulldozer.Erlang(HeapParamInterval, HeapParamCount));

**end**;

*// Работа завершена*

par.Finished := True;

**end**;

Два погрузчика будут размещены не в массиве, как это было в предыдущих проектах с несколькими обрабатывающими устройствами, а в списке (очереди). Это связано с тем, что по условию задачи требуется, чтобы к работе каждый раз приступал тот погрузчик, который дольше всего простаивал. При использовании массива погрузчиков обеспечить выполнение этого требования затруднительно, а при использовании списка обеспечивается автоматически: погрузчик после завершения своих действий просто встает в конец очереди. Если до него в очереди уже был другой погрузчик, он первым получит управление. К тому же в этом случае происходит автоматический сбор статистики по состоянию очереди ожидания погрузчиков.

В классе погрузчика следует предусмотреть поле данных, в котором содержится среднее время выполнения погрузки. Это необходимо в связи с тем, что у каждого из погрузчиков это время различно. Кроме того, каждый погрузчик должен иметь связь со своим объектом сбора статистики. Эту связь проще всего обеспечить, поместив в класс погрузчика ссылку на соответствующий объект. Заметим, что, как и в предыдущих случаях, объект статистики создается в классе имитации, который содержит статическую ссылку на него. Поэтому объект-погрузчик не должен удалять объект статистики при своем удалении, за это отвечает класс имитации. Чтобы можно было задавать значения двух указанных полей, придется предусмотреть в классе конструктор с двумя параметрами. Сами эти поля не предназначены для доступа к ним из других объектов, поэтому определим их в закрытой (**private**) части класса. В итоге, определение класса погрузчика выглядит следующим образом:

*// Класс TLoader - погрузчик*

TLoader = **class**(TProcess)

**private**

*// Среднее время погрузки*

MeanWorkTime : Double;

*// Объект сбора статистики*

Stat : TServiceStatistics;

**public**

**constructor** Create(AStat : TServiceStatistics; AWork : Double);

**protected**

**procedure** RunProcess; **override**;

**end**;

Основную сложность при формулировке алгоритма работы погрузчика представляет собой цикл ожидания начала работы. Во-первых, в условие цикла следует проверять наличие не менее двух куч и хотя бы одного самосвала. Во-вторых, следует проверять условие завершения имитации: бульдозер закончил работу, количество куч менее двух и все четыре самосвала ожидают в очереди. Проверять при этом тот факт, что оба погрузчика находятся в очереди, не обязательно: время, которое потребуется последнему погруженному самосвалу для того, чтобы доехать до места разгрузки, разгрузиться и вернуться обратно, заведомо больше 5 минут, которые требуются последнему погрузчику, чтобы встать в очередь. К основному действию обслуживающего процесса в алгоритм добавляется еще одно: приведение в состояние готовности в течение 5 минут. В итоге, алгоритм работы погрузчика выглядит следующим образом:

**procedure** TLoader.RunProcess;

**var**

par : TLoading;

trk : TTruck;

i : Integer;

**begin**

*// Процесс, моделирующий работу погрузчика*

par := Parent **as** TLoading;

**while** True **do**

**begin**

*// Ожидать наличия не менее необходимого количества куч*

*// и свободного самосвала*

**while** (par.HeapQueue.Size < MinHeapQueueSize) **or**

par.TrucksQueue.Empty **do**

**begin**

*// Если бульдозер закончил работу,*

*// осталось меньше минимального числа куч и все самосвалы свободны*

**if** par.Finished **and** (par.HeapQueue.Size < MinHeapQueueSize) **and**

(par.TrucksQueue.Size = 4) **then**

**begin**

*// Завершить работу посредством активации родительского процесса*

par.ActivateDelay(0);

Passivate;

**end**

**else**

*// Иначе - ждать*

Passivate;

**end**;

*// Начало обслуживания*

*// Зафиксировать начало работы*

Stat.Start(SimTime);

*// Извлечь из очереди первый самосвал*

trk := par.TrucksQueue.First **as** TTruck;

trk.StartRunning;

*// Убрать кучи из начала очереди*

**for** i := 1 **to** MinHeapQueueSize **do**

par.HeapQueue.First.Free;

*// Выполнить погрузку*

Hold(rndLoader.Exponential(MeanWorkTime));

*// Закончить работу*

Stat.Finish(SimTime);

*// Активировать самосвал*

trk.ActivateDelay(0);

*// Возвращение*

Hold(LoaderReturnTime);

*// Встать в очередь ожидания на последнее место*

Insert(par.LoadersQueue);

**end**;

**end**;

Алгоритм работы самосвала устроен достаточно просто. Самосвал ожидает окончания погрузки, в это время он фактически играет роль обрабатываемого компонента. Затем он выполняет ряд самостоятельных действий, связанных с поездкой к месту разгрузки, непосредственно разгрузкой и возвратом в исходное положение. После возврата самосвал встает в очередь ожидания, и далее его работа повторяется. Попутно заметим, что при создании имитации созданные самосвалы должны быть принудительно помещены в список ожидания. Альтернативный вариант мог бы состоять в том, что самосвал встает в очередь первым своим действием, но в этом случае в классе имитации пришлось бы иметь массив статических ссылок на самосвалы и активировать их все при ее запуске. Сам класс самосвала не содержит никаких дополнительных компонентов, поэтому приведем только текст его алгоритма:

**procedure** TTruck.RunProcess;

**var**

par : TLoading;

**begin**

*// Процесс, моделирующий работу самосвала*

par := Parent **as** TLoading;

**while** True **do**

**begin**

*// Самосвал загружен после ожидания*

*// Перевозка*

Hold(rndTruck.Normal(TruckForwardMean, TruckForwardDeviation));

*// Разгрузка*

Hold(rndTruck.Uniform(TruckUnloadMin, TruckUnloadMax));

*// Возврат*

Hold(rndTruck.Normal(TruckBackMean, TruckBackDeviation));

*// Активировать свободный погрузчик*

ActivateDelay(par.LoadersQueue, 0);

*// Встать в очередь ожидания*

Wait(par.TrucksQueue);

**end**;

**end**;

Осталось определить класс имитации. В нем не содержится никаких сложностей, нуждающихся в особых комментариях. Приведем определение класса, его основной алгоритм, а также метод инициализации. В последнем обратите внимание на создание объектов погрузчиков и самосвалов и постановку их в очередь:

TLoading = **class**(TSimulation)

**public**

*// Очередь куч, готовых к перевозке*

HeapQueue : TList;

*// Очередь самосвалов, ожидающих работы*

TrucksQueue : TList;

*// Очередь погрузчиков, ожидающих работы*

LoadersQueue : TList;

*// Статистика по погрузчикам*

LoadersStat : array of TServiceStatistics;

*// Бульдозер*

Bulldozer : TBulldozer;

*// Признак завершения работы*

Finished : Boolean;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

**procedure** TLoading.RunSimulation;

**begin**

*// Запустить бульдозер*

Bulldozer.ActivateDelay(0);

*// Ожидать окончания*

Passivate;

*// Скорректировать статистики*

StopStat;

**end**;

**procedure** TLoading.Init;

**var**

i : Integer;

**begin**

**inherited**;

*// Создать списки самосвалов, погрузчиков и куч*

TrucksQueue := TList.Create;

LoadersQueue := TList.Create;

HeapQueue := TList.Create;

*// Массив статисик погрузчиков*

SetLength(LoadersStat, LoadersCount);

**for** i := 0 **to** LoadersCount - 1 **do**

LoadersStat[i] := TServiceStatistics.Create(1);

*// Создать погрузчики и поместить их в очередь*

**for** i := 0 **to** LoadersCount - 1 **do**

TLoader.Create(LoadersStat[i], LoaderTimeMean[i]).Insert(LoadersQueue);

*// Создать самосвалы и поместить их в очередь*

**for** i := 1 **to** TrucksCount **do**

TTruck.Create.Insert(TrucksQueue);

*// Создать бульдозер*

Bulldozer := TBulldozer.Create;

*// Процесс работы не окончен*

Finished := False;

**end**;

Полный текст программы находится в папке 3.5 Loading. Результаты одного из прогонов приведены ниже:

Loader 0 statistics:

Device count = 1

Average usage = 0.343 +- 0.475

Current running = 0, completed = 16

Average blockage = 0.000

Max idle time = 49.106, max busy time = 26.363

Loader 1 statistics:

Device count = 1

Average usage = 0.434 +- 0.496

Current running = 0, completed = 14

Average blockage = 0.000

Max idle time = 42.647, max busy time = 39.560

Heaps queue:

Average length = 0.997 +- 1.060

Max = 5, current = 0

Average waiting time = 8.897

Trucks queue:

Average length = 0.707 +- 0.856

Max = 4, current = 4

Average waiting time = 10.169

Loaders queue:

Average length = 2.000 +- 0.000

Max = 2, current = 2

Average waiting time = 32.964

Calendar:

Average length = 4.482 +- 1.122

Max = 7, current = 1

Average waiting time = 8.078

При разработке визуализированного варианта проекта добавим возможность иллюстрации процесса перевозки груза. А именно, будем показывать количество самосвалов, которые едут на разгрузку, разгружаются и возвращаются обратно. Естественно, для учета этих значений следует добавить соответствующие переменные. Проще всего это сделать, поместив в класс имитации целочисленные счетчики:

TLoading = **class**(TSimulation)

**public**

…

*// Количество самосвалов, отправляющихся на разгрузку*

ForwardCount : Integer;

*// Количество разгружающихся самосвалов*

UnloadCount : Integer;

*// Количество самосвалов, возвращающихся с разгрузки*

BackCount : Integer;

…

**end**;

Это потребует внесения соответствующих изменений в алгоритм работы самосвала:

**procedure** TTruck.RunProcess;

…

*// Перевозка*

Inc(par.ForwardCount);

Hold(rndTruck.Normal(TruckForwardMean, TruckForwardDeviation));

Dec(par.ForwardCount);

*// Разгрузка*

Inc(par.UnloadCount);

Hold(rndTruck.Uniform(TruckUnloadMin, TruckUnloadMax));

Dec(par.UnloadCount);

*// Возвращение*

Inc(par.BackCount);

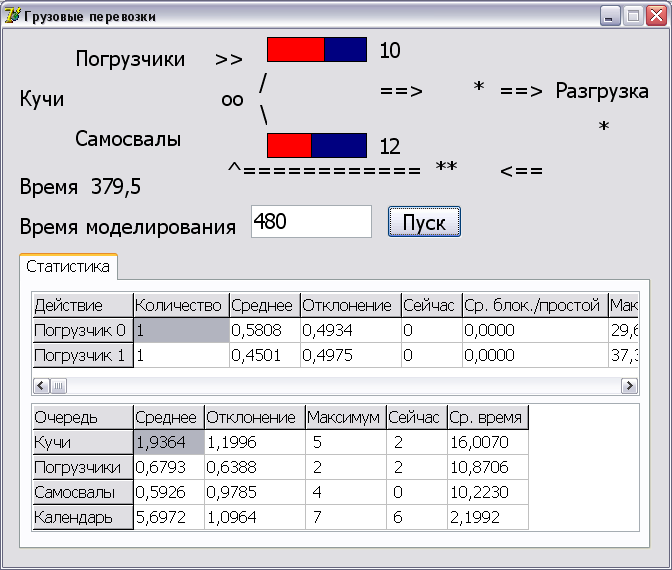
Hold(rndTruck.Normal(TruckBackMean, TruckBackDeviation));

Dec(par.BackCount);

…

**end**;

Естественно, в форме визуализируемого проекта следует предусмотреть отображение этой информации в соответствующих надписях. Полный текст программы находится в папке 3.5 Loading Visual. В этом проекте на форме также размещены элементы, помогающие делать отображаемый процесс перемещения самосвалов более наглядным:



## 3.6. Модель работы карьера

*Постановка задачи.* В карьере добывают руду три экскаватора. Каждый из них загружает руду в самосвалы, которые, в свою очередь, отвозят ее к измельчителю, разгружаются и возвращаются обратно. Каждый самосвал приписан к своему экскаватору и после разгрузки возвращается именно к нему.

Имеются самосвалы двух типов: легкие и тяжелые. Грузоподъемность легкого самосвала составляет 20 тонн. Время его погрузки распределено экспоненциально с математическим ожиданием 5 минут. Время движения пустого самосвала к экскаватору постоянно и равно 1,5 минуты. Время движения груженого самосвала от экскаватора к измельчителю на 1 минуту больше. Время разгрузки распределено экспоненциально с математическим ожиданием 2 минуты. Параметры тяжелого самосвала: грузоподъемность — 50 тонн, среднее время погрузки — 10 минут, время движения пустого самосвала — 2 минуты, груженого — на 1 минуту больше, среднее время разгрузки — 4 минуты.

Первоначально в очереди к каждому экскаватору находятся два легких самосвала, а за ними один тяжелый. При постановке в очередь к измельчителю тяжелые самосвалы имеют преимущество перед легкими: прибывший тяжелый самосвал встает в очередь перед всеми находящимися в ней легкими (но после всех тяжелых). При возврате к экскаватору самосвалы встают в очередь в порядке прибытия независимо от типа.

Требуется провести имитацию в течение 8-часового рабочего дня (480 минут) и собрать статистику по загруженности экскаваторов (каждого в отдельности) и измельчителя, по состоянию очередей к экскаваторам (каждой в отдельности) и измельчителю и по количеству самосвалов, возвращающихся после разгрузки. Имитация закачивается в заданный момент времени независимо от состояния всех ее элементов.

*Решение*. Первое отличие данной задачи от ранее рассмотренных состоит в том, что все составляющие имитации (самосвалы, экскаваторы, измельчитель) не создаются и не уничтожаются в процессе имитации. То есть, их все следует создать при инициализации, подготовить к работе и запустить имитацию, а после ее завершения удалить. Следовательно, в этом примере нет необходимости в процессе-генераторе. Второе отличие заключается в том, что каждое из идентичных устройств-экскаваторов имеет свою отдельную очередь ожидания. Это означает, что в классе имитации следует предусмотреть массив очередей наряду с массивом экскаваторов, а для каждого самосвала указывать, в какую из очередей ему следует становиться при возвращении в исходное положение. Третье отличие состоит в том, что в очередь к измельчителю самосвалы встают не в порядке поступления, как было в предыдущих задачах, а в зависимости от грузоподъемности. Это отличие потребует указать для очереди к измельчителю функцию сравнения.

Сами процессы, имитирующие работу экскаваторов и измельчителя, практически не отличаются от типичных обрабатывающих процессов. Единственное существенное отличие проявляется в том, что каждый экскаватор должен извлекать очередной самосвал из своей очереди, а также записывать статистику о своей работе в свой объект сбора статистики. Это потребует наличия ссылок на очередь и статистику в его объекте. Альтернативным вариантом было бы указание общего индекса очереди и статистики в своих массивах, однако этот вариант потребует более сложной конструкции обращения к соответствующим объектам. Что касается процесса, имитирующего работу самосвала, то его алгоритм аналогичен алгоритмам других процессов-компонентов, однако рассчитан на бесконечно повторяющееся повторение действий. В алгоритм входят два обслуживающих действия с предварительным ожиданием в очередях (погрузка и разгрузка), а также два самостоятельных действия (движение к измельчителю и обратно).

Поскольку большинство объектов (все, кроме измельчителя), составляющих имитацию, имеют дополнительные элементы, сначала приведем определения их классов:

**type**

TExcavator = **class**(TProcess)

**public**

*// Очередь самосвалов*

Queue : TList;

*// Статистика*

Stat : TServiceStatistics;

**constructor** Create(AQueue : TList; AStat : TServiceStatistics);

**protected**

**procedure** RunProcess; **override**;

**end**;

TTruck = **class**(TProcess)

**public**

*// Очередь к экскаватору*

Queue : TList;

*// Активируемый экскаватор*

Excavator : TProcess;

*// Грузоподъемность*

Tonnage : Double;

*// Время загрузки*

LoadingTime : Double;

*// Время холостого пробега*

TripTime : Double;

*// Время разгрузки*

UnloadingTime : Double;

**constructor** Create(Que : TList; Ex : TProcess;

Tng, LdTm, TrTm, UnldTm : Double);

**protected**

**procedure** RunProcess; **override**;

**end**;

TMill = **class**(TProcess)

**protected**

**procedure** RunProcess; **override**;

**end**;

TQuarry = **class**(TSimulation)

**public**

*// Очереди к экскаваторам*

ExcavatorQueue : **array** **of** TList;

*// Очереди к измельчителю*

MillQueue : TList;

*// Экскаваторы*

Excavator : **array** **of** TProcess;

*// Измельчитель*

Mill : TMill;

*// Статистика по возврату самосвалов*

ReturnStat : TActionStatistics;

*// Статистики по экскаваторам*

ExcavatorStat : **array** **of** TServiceStatistics;

*// Статистика по измельчителю*

MillStat : TServiceStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

Поскольку классы для экскаватора (TExcavator) и самосвала (TTruck) содержат параметры, влияющие на временн***ы***е характеристики их работы, в этих классах пришлось добавить конструкторы, которые только записывают эти параметры в соответствующие поля классов.

Методы RunProcess всех классов достаточно просты и не нуждаются в подробных пояснениях, помимо комментариев, размещенных непосредственно в тексте:

**procedure** TExcavator.RunProcess;

**var**

trk : TTruck;

**begin**

*// Работа экскаватора*

**while** True **do**

**begin**

*// Ожидать появления самосвалов в очереди*

**while** Queue.Empty **do**

Passivate;

*// Взять первый самосвал*

trk := Queue.First **as** TTruck;

trk.StartRunning;

*// Погрузка с записью статистики*

Stat.Start(SimTime);

Hold(rndExcavator.Exponential(trk.LoadingTime));

Stat.Finish(SimTime);

*// Отправить самосвал*

trk.ActivateDelay(0);

**end**;

**end**;

**procedure** TTruck.RunProcess;

**var**

par : TQuarry;

**begin**

par := Parent **as** TQuarry;

**while** True **do**

**begin**

*// Самосвал загружен и отправляется к измельчителю*

Hold(TripTime + 1);

*// Активировать разгрузку*

par.Mill.ActivateDelay(0);

*// Встать в очередь к измельчителю*

Wait(par.MillQueue);

*// Обратный путь с записью статистики*

par.ReturnStat.Start(SimTime);

Hold(TripTime);

par.ReturnStat.Finish(SimTime);

*// Активировать свой экскаватор*

Excavator.ActivateDelay(0);

*// Встать в очередь к своему экскаватору*

Wait(Queue);

**end**;

**end**;

**procedure** TMill.RunProcess;

**var**

par : TQuarry;

trk : TTruck;

**begin**

*// Процесс работы измельчителя*

par := Parent **as** TQuarry;

**while** True **do**

**begin**

*// Ожидать поступления самосвалов на разгрузку*

**while** par.MillQueue.Empty **do**

Passivate;

*// Извлечь первый самосвал из очереди*

trk := par.MillQueue.First **as** TTruck;

trk.StartRunning;

*// Разгрузка с записью статистики*

par.MillStat.Start(SimTime);

Hold(rndMill.Exponential(trk.UnloadingTime));

par.MillStat.Finish(SimTime);

*// Отправить самосвал в обратный путь*

trk.ActivateDelay(0);

**end**;

**end**;

Алгоритм процесса имитации не содержит никаких сложностей. Основная нагрузка ложится на метод Init, в котором следует, в частности, создать все самосвалы и расставить их по соответствующим очередям. Кроме того, при создании очереди к измельчителю следует указать способ ее упорядочения. Это делается, как указывалось в параграфе 1.1, посредством функции сравнения. Эта функция должна сравнить добавляемый в очередь самосвал с очередным самосвалом, стоящим в очереди, по грузоподъемности. Новый самосвал вставляется в очередь перед первым из имеющихся, грузоподъемность которого меньше. Приведем функцию сравнения совместно с основными алгоритмами имитации:

*// Функция сравнения для вставки в очередь ожидания разгрузки*

**function** MillQueueFunc(A, B : TLink) : Boolean;

**begin**

*// Крупнотоннажные самосвалы имеют преимущество*

Result := (A **as** TTruck).Tonnage > (B **as** TTruck).Tonnage;

**end**;

**procedure** TQuarry.Init;

**var**

i, j : Integer;

**begin**

**inherited**;

*// Создать очереди к экскаваторам*

SetLength(ExcavatorQueue, ExcavatorCount);

**for** i := 0 **to** ExcavatorCount - 1 **do**

ExcavatorQueue[i] := TList.Create;

*// Очередь к измельчителю упорядочена по грузоподъемности*

MillQueue := TList.Create(MillQueueFunc);

*// Создать объекты сбора статистики*

SetLength(ExcavatorStat, ExcavatorCount);

**for** i := 0 **to** ExcavatorCount - 1 **do**

ExcavatorStat[i] := TServiceStatistics.Create(1);

MillStat := TServiceStatistics.Create(1);

ReturnStat := TActionStatistics.Create;

*// Создать процессы*

SetLength(Excavator, ExcavatorCount);

**for** i := 0 **to** ExcavatorCount - 1 **do**

Excavator[i] := TExcavator.Create(ExcavatorQueue[i], ExcavatorStat[i]);

Mill := TMill.Create;

*// Создать грузовики и поместить их в очереди к экскаваторам*

**for** i := 0 **to** ExcavatorCount - 1 **do**

**begin**

*// Сначала легкие*

**for** j := 1 **to** InitLightTrucks **do**

TTruck.Create(ExcavatorQueue[i], Excavator[i], LightTonnage,

LightLoadingTime, LightTripTime,

LightUnloadingTime).Insert(ExcavatorQueue[i]);

*// Потом тяжелые*

**for** j := 1 **to** InitHeavyTrucks **do**

TTruck.Create(ExcavatorQueue[i], Excavator[i], HeavyTonnage,

HeavyLoadingTime, HeavyTripTime,

HeavyUnloadingTime).Insert(ExcavatorQueue[i]);

**end**;

**end**;

**procedure** TQuarry.RunSimulation;

**begin**

*// Активировать все экскаваторы*

ActivateAllDelay(Excavator, 0);

*// Ожидать окончания имитации*

Hold(SimulationTime);

StopStat;

**end**;

Полный текст программы находится в папке 3.6 Quarry.

При построении визуализированного приложения внесем в постановку задачи следующие дополнения. Во-первых, будем в очередях к экскаваторам и измельчителю не просто отображать количество самосвалов, но и показывать их типы: легкий или тяжелый. Во-вторых, будем отображать количество самосвалов, движущихся к измельчителю, раздельно по каждому экскаватору без указания их типа. Самосвалы, движущиеся обратно, будем отображать общим индикатором без разделения по принадлежности к тому или иному экскаватору и также без различия по типам. В-третьих, будем учитывать количество разгруженных самосвалов раздельно по каждому типу и подсчитывать общую массу доставленного груза.

Чтобы учесть типы самосвалов при отображении очереди, создадим отдельную процедуру, которая отображает самосвалы легкого типа одним символом, а тяжелого — другим. В рассматриваемом варианте приложения подразумевается отображение очереди самосвалов движущейся слева направо. Соответственно, первый самосвал очереди должен изображаться в самой правой позиции, то есть, последним. Процедура имеет следующий вид:

*// Отображение очереди самосвалов с использованием разных символов*

*// для самосвалов разного типа*

**function** TruckChars(Queue : TList; HeavyChar, LightChar : Char) : **string**;

**var**

Res : string;

trk : TTruck;

**begin**

Res := '';

trk := Queue.First **as** TTruck;

*// Перебор самосвалов в очереди*

**while** trk <> nil **do**

**begin**

*// Добавить к строке очередной символ*

**if** trk.Tonnage > 20 **then**

Res := HeavyChar + Res

**else**

Res := LightChar + Res;

trk := trk.Next **as** TTruck;

**end**;

Result := Res;

**end**;

Эту процедуру следует разместить в модуле главной формы программы, отвечающем за визуализацию. Вызов процедуры производится следующим образом:

lbExcavatorQueue0.Caption := TruckChars(qry.ExcavatorQueue[0], 'O', 'o');

lbExcavatorQueue1.Caption := TruckChars(qry.ExcavatorQueue[1], 'O', 'o');

lbExcavatorQueue2.Caption := TruckChars(qry.ExcavatorQueue[2], 'O', 'o');

Для отображения количества самосвалов, движущихся от экскаваторов к измельчителю и обратно, требуется информация по их количеству. Для самосвалов, двигающихся обратно, эта информация доступна из уже имеющегося объекта сбора статистики по действию возврата. Что же касается самосвалов, движущихся к измельчителю, следует добавить учет их количества раздельно для каждого экскаватора. Поскольку никакой дополнительной статистики, а также разделения по типам здесь не требуется, можно обойтись простыми целочисленными счетчиками, размещенными в массиве.

Увеличивать и уменьшать значения этих счетчиков будут сами процессы самосвалов. Для этого каждый из них должен знать, какой именно из счетчиков он должен увеличивать и уменьшать. Можно поместить в каждый процесс самосвала ссылку на «свой» элемент массива счетчиков либо его индекс в общем массиве. И тот, и другой подход потребует наличия в объекте самосвала дополнительного поля и дополнительного параметра конструктора, что дополнительно усложнит его структуру.

Однако наличие индекса счетчика в массиве позволит обращаться с помощью этого же индекса как к объектам экскаваторов, так и к их очередям. Поэтому, добавив в объект самосвала индекс экскаватора, можно избавиться от наличия в нем соответствующих ссылок и упростить как структуру класса, так и его конструктор. С другой стороны, это усложнит порядок обращения к экскаваторам и их очередям. В итоге, получаются следующие изменения в определение класса, его конструктор и алгоритм работы:

TTruck = **class**(TProcess)

**public**

*// Индекс экскаватора*

Index : Integer;

*// Грузоподъемность*

Tonnage : Double;

*// Время загрузки*

LoadingTime : Double;

*// Время холостого пробега*

TripTime : Double;

*// Время разгрузки*

UnloadingTime : Double;

**constructor** Create(Idx : Integer; Tng, LdTm, TrTm, UnldTm : Double);

**protected**

**procedure** RunProcess; **override**;

**end**;

**constructor** TTruck.Create(Idx: Integer; Tng, LdTm, TrTm, UnldTm: Double);

**begin**

Index := Idx;

Tonnage := Tng;

LoadingTime := LdTm;

TripTime := TrTm;

UnloadingTime := UnldTm;

**inherited** Create;

**end**;

**procedure** TTruck.RunProcess;

**var**

par : TQuarry;

**begin**

par := Parent **as** TQuarry;

**while** True **do**

**begin**

*// Самосвал загружен и отправляется к измельчителю*

Inc(par.ForwardTrip[Index]);

Hold(TripTime + 1);

Dec(par.ForwardTrip[Index]);

*// Активировать разгрузку*

par.Mill.ActivateDelay(0);

*// Встать в очередь к измельчителю*

Wait(par.MillQueue);

*// Обратный путь с записью статистики*

par.ReturnStat.Start(SimTime);

Hold(TripTime);

par.ReturnStat.Finish(SimTime);

*// Активировать свой экскаватор*

par.Excavator[Index].ActivateDelay(0);

*// Встать в очередь к своему экскаватору*

Wait(par.ExcavatorQueue[Index]);

**end**;

**end**;

Соответствующий массив счетчиков добавляется в класс имитации:

*// Количество самосвалов, движущихся на разгрузку*

ForwardTrip : **array** **of** Integer;

Для учета разгруженных самосвалов по типам добавим в класс имитации два счетчика:

*// Количество разгруженных самосвалов по типам*

HeavyCount : Integer;

LightCount : Integer;

В алгоритм процесс работы измельчителя после завершения разгрузки следует добавить следующие строки:

**if** trk.Tonnage > LightTonnage **then**

Inc(par.HeavyCount)

**else**

Inc(par.LightCount);

Подсчет общей массы доставленного груза можно производить непосредственно в процедуре визуализации одновременно с отображением данных по количеству завершивших разгрузку самосвалов каждого типа:

lbHeavyCount.Caption := IntToStr(qry.HeavyCount);

lbLightCount.Caption := IntToStr(qry.LightCount);

lbDelivered.Caption := FloatToStr(qry.LightCount \* LightTonnage +

qry.HeavyCount \* HeavyTonnage);

Полный текст программы находится в папке 3.6 Quarry Visual.

## 3.7. Модель сетевого планирования

Одним из популярных методов планирования и анализа проектов, состоящих из нескольких взаимосвазанных работ, является метод *PERT* (*Program Evaluation and Review Technique* — техника оценки и анализа программ). Важной частью этого методя является *сеть PERT*. Она представляет собой граф, дуги которого представляют отдельные работы. Когда все работы, входящие в некоторый узел, завершены, данный узел оказывается в состоянии *готовности*, что позволяет начать работы, исходящие из этого узла. Начальный узел сети изначально находится в состоянии готовности. Когда в этом состоянии оказывается конечный узел сети, работа над проектом считается завершенной. Таким образом, сеть PERT позволяет наглядно представить взаимную зависимость отдельных работ проекта. Существует много работ, посвященных исследованию этих сетей. Рассмотрим на примере, как можно с помощью средств имитационного моделирования смоделировать работу такой сети.

*Постановка задачи*. Пусть работа по замене и наладке оборудования состоит из ряда работ, перечисленных в таблице (намерацию сразу будем проводить с нуля, чтобы было удобнее переносить их в программу):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер работы | Описание | Предшествующие работы | Время | | |
| Минимум | Мода | Максимум |
| 0 | Разборка силового оборудования и оснащение инструментами | – | 1 | 3 | 5 |
| 1 | Освоение нового монтажа | – | 3 | 6 | 9 |
| 2 | Подготовка к проверке наладки | – | 10 | 13 | 19 |
| 3 | Чистка, проверка и ремонт силового оборудования | 0 | 3 | 9 | 12 |
| 4 | Калибровка инструмента | 0 | 1 | 3 | 8 |
| 5 | Проверка контактов | 1, 4 | 8 | 9 | 16 |
| 6 | Проверка правильности сборки | 1, 4 | 4 | 7 | 13 |
| 7 | Сборка и проверка силового оборудования | 3 | 3 | 6 | 9 |
| 8 | Проверка наладки | 2, 6 | 1 | 3 | 8 |

Из этой таблицы видно, что, к примеру, работы 0, 1 и 2 начинаются сразу же, в самом начале работы над проектом. После завершения работы 0 могут начинаться работы 3 и 4. После завершения работ 1 и 4 могут начинаться работы 5 и 6 и т. д. Время исполнения каждой работы распределено треугольно с параметрами, также указанными в таблице. Проект завершается, когда заканчиваются работы 5, 7 и 8, потому что остальные работы должны завершить раньше, так как они упомянуты в столбце «Предшествующие работы».

Изобразим взаимоотношения между отдельными работами в виде сети PERT:

0

1

2

3

4

7

5

6

6

8

8

8

|  |  |  |
| --- | --- | --- |
| Номер дуги | Исходный узел | Конечный узел |
| 0 | 0 | 1 |
| 1 | 0 | 2 |
| 2 | 0 | 3 |
| 3 | 1 | 4 |
| 4 | 1 | 2 |
| 5 | 2 | 5 |
| 6 | 2 | 3 |
| 7 | 4 | 5 |
| 8 | 3 | 5 |

Узлы сети для удобства пронумерованы. Таким образом, получается, что узел номер 0 является начальным, он символизирует начало проекта, а узел номер 5 — конечным, он обозначает завершение проекта. Номера работ подписаны рядом с соответствующими дугами. Соответствие номеров дуг (работ) и узлов (этапов выполнения проекта), которые они соединяют, приведено в таблице.

Собственно прогон модели, как нетрудно догадаться, состоит в исполнении девяти работ. Вся информация, которую можно при жтом получить — времена готовности пяти узлов (нулевой узел учитывать, естественно, нет смысла). При этом время готовности узла номер 5 — это время завершения работы над проектом в целом.

Естественно, практическая полезность данной информации сомнительна, поскольку время исполнения каждой из работ — величина случайная. Поэтому разумно будет выполнить достаточно много прогонов и собрать по их результатам статистики по временам готовности каждого из узлов.

*Решение*. Представить однократный прогон в виде одного процесса, естественно, нельзя, поскольку в нем предусматривается в ряде случаев параллельное исполнение нескольких работ, чего одиночный процесс сделать не позволяет. Можно создать несколько процессов, каждый из которых выполняет несколько работ, расположенных в сети друг за другом, и обеспечить их синхронизацию в точках разветвления, слияния и пересечения. Например, такие процессы могли бы выполнять следующие последовательности работ: 0-3-7, 1-5, 2-8 и 4-6. Однако такая схема построения модели не является гибкой: в случае изменения структуры сети потребуется заново находить такие цепочки процессов и обеспечивать их синхронизацию. Поэтому разработаем более универсальный метод построения модели, который позволит имитировать сети произвольной конфигурации только посредством изменения набора исходных данных. Такую программу несложно было бы доработать так, чтобы она прочитывала структуру сети из файла и проводила ее прогоны. Таким образом, программа стала бы более универсальным инструментом исследования.

Представим каждую дугу сети в виде отдельного процесса. Работа такого процесса заключается в следующем: начав свою работу по команде активации от узла, из которого он исходит, он исполняет предусмотренную работу (то есть, выполняет задержку на заданное время), а потом сообщает тому узлу, в который входит, о том, что его работа завершена.

Каждый узел также будет представлен в виде отдельного процесса. Его работа состоит в том, что он дожидается, пока все входящие в него дуги не закончат работу (естественно, начальный узел ничего не ждет), затем заносит в статистику время своей готовности (кроме начального узла), а затем активирует процессы всех исходящих из него дуг (конечный узел вместо этого завершает прогон имитации).

Взаимосвязь узлов и дуг обеспечивается соответствующей настройкой исходной конфигурации сети: каждая дуга содержит ссылку на узел, в котором она заканчивается, а каждый узел содержит список дуг, которые в нем начинаются. Естественно, каждая дуга также содержит время исполнения своей работы, а узел — ссылку на свой объект сбора статистики. Последние, кстати, являются глобальными, поскольку они относятся не к конкретному прогону, а собирают результаты по многим прогонам.

Прежде чем обсуждать структуры и алгоритмы работы процессов, рассмотрим способ задания конфигурации сети. К сожалению, язык Delphi (да и все массовые языки программирования) не позволяет непосредственно описать исходную конфигурацию структуры объектов. Поэтому воспользуемся следующим стандартным приемом: представим исходные данные ранее приведенных таблиц в виде массива записей, а потом разработаем процедуру, которая по этим данным автоматически строит соответствующую конфигурацию сети. Эти исходные данные содержат для каждой дуги следующую информацию: индексы исходного и конечного узлов и параметры времени исполнения работы. Соответствующие данные записываются в программе следующим образом:

*// Число узлов и дуг*

**const**

NodeCount = 6;

ArcCount = 9;

*// Структура информации по каждой дуге*

**type**

TArcData = **record**

FromNode, ToNode : Integer;

MinTime, ModaTime, MaxTime : Double;

**end**;

**var**

*// Исходные данные о конфигурации сети*

ArcData : **array** [0 .. ArcCount - 1] **of** TArcData = (

(FromNode : 0; ToNode : 1; MinTime : 1; ModaTime : 3; MaxTime : 5),

(FromNode : 0; ToNode : 2; MinTime : 3; ModaTime : 6; MaxTime : 9),

(FromNode : 0; ToNode : 3; MinTime : 10; ModaTime : 13; MaxTime : 19),

(FromNode : 1; ToNode : 4; MinTime : 3; ModaTime : 9; MaxTime : 12),

(FromNode : 1; ToNode : 2; MinTime : 1; ModaTime : 3; MaxTime : 8),

(FromNode : 2; ToNode : 5; MinTime : 8; ModaTime : 9; MaxTime : 16),

(FromNode : 2; ToNode : 3; MinTime : 4; ModaTime : 7; MaxTime : 13),

(FromNode : 4; ToNode : 5; MinTime : 3; ModaTime : 6; MaxTime : 9),

(FromNode : 3; ToNode : 5; MinTime : 1; ModaTime : 3; MaxTime : 8)

);

Алгоритм, создающий структуру сети по этим данным, естественно, исполняется перед началом исполнения каждого прогона, значит, он должен исполняться в методе Init имитации. Поэтому, в отличие от предыдущих примеров, рассмотрение этого начнем с класса имитации. Класс содержит массив узлов сети. При создании имитации сначала создается этот массив, а затем необходимое количество дуг, которые привязываются к соответствующим узлам. Собственно процесс имитации состоит в активации всех узлов и ожидании завершения:

TPert = **class**(TSimulation)

**public**

*// Массив узлов*

Nodes : **array** **of** TProcess;

**destructor** Destroy; **override**;

**protected**

**procedure** RunSimulation; **override**;

**procedure** Init; **override**;

**end**;

**procedure** TPert.Init;

**var**

i : Integer;

**begin**

**inherited**;

*// Создать узлы*

SetLength(Nodes, NodeCount);

**for** i := 0 **to** NodeCount - 1 **do**

Nodes[i] := TNode.Create(NodeStat[i]);

*// Пометить начальный и конечный*

(Nodes[0] **as** TNode).First := True;

(Nodes[NodeCount - 1] **as** TNode).Last := True;

*// Создать дуги*

**for** i := 0 **to** ArcCount - 1 **do**

**begin**

**with** ArcData[i] **do**

**begin**

*// Задается конечный узел,*

TArc.Create(Nodes[ToNode] **as** TNode,

*// время выполнения работы*

rndArc.Triangular(MinTime, ModaTime, MaxTime)).

*// Вставить в список исходящих дуг соответствующего узла*

Insert((Nodes[FromNode] **as** TNode).OutArcs);

*// Посчитать в соответствующем узле*

Inc((Nodes[ToNode] **as** TNode).LimitIncomeArcs);

**end**;

**end**;

**end**;

**procedure** TPert.RunSimulation;

**begin**

*// Запустить все узлы*

ActivateAllDelay(Nodes, 0);

*// Ждать завершения*

Passivate;

**end**;

По ранее упоминавшимся причинам (см. параграф 2.3) массив узлов определен как массив объектов класса TProcess. По этой причине при обращении к компонентам узлов требуется выполнять их приведение к классу TNode при помощи операции **as**. Массив объектов сбора статистики определен в глобальных переменных следующим образом:

*// Объекты сбора статистики*

NodeStat : **array** [0 .. NodeCount - 1] **of** TStatistics;

Поскольку узел номер 0 является начальным и находится в состоянии готовности сразу же после запуска имитации, его статистику собирать не требуется. Однако для корректной работы процедуры создания имитации требуется его наличие. Поэтому индекс 0 в массиве предусматривается, однако объект сбора статистики по этому индексу не создается, и данный элемент массива остается равен **nil** (в Delphi все глобальные переменные сразу после своего создания получают нулевые значения).

Теперь опишем структуру и алгоритмы процессов. Процесс, имитирующий работу узла, описывается следующим образом:

*// Узел сети*

TNode = **class**(TProcess)

**public**

*// Список исходящих дуг*

OutArcs : TList;

*// Счетчик завершенных входящих дуг*

IncomeArcs : Integer;

*// Количество входящих дуг, которые должны быть завершены*

*// для готовности узла*

LimitIncomeArcs : Integer;

*// Является ли узел начальным или конечным*

First, Last : Boolean;

*// Объект сбора статистики узла*

NodeStat : TStatistics;

**constructor** Create(Stat : TStatistics);

**destructor** Destroy; **override**;

**protected**

**procedure** RunProcess; **override**;

**end**;

**procedure** TNode.RunProcess;

**var**

par : TPert;

**begin**

par := Parent **as** TPert;

*// Ожидать завершения всех входящих дуг*

**while** IncomeArcs < LimitIncomeArcs **do**

Passivate;

*// Собрать статистику по времени готовности (кроме начального узла)*

**if** **not** First **then**

NodeStat.AddData(SimTime);

*// Последний узел завершает имитацию*

**if** Last **then**

par.ActivateDelay(0)

*// Все остальные - запускают исходящие дуги*

**else**

ActivateAllDelay(OutArcs, 0);

**end**;

Учет завершенных работ производится с помощью счетчика IncomeArcs, который сравнивается с количеством входящих дуг LimitIncomeArcs. Последнее значение устанавливается в рассмотренном ранее алгоритме построения сети в методе Init имитации.

Процесс, соответствующий дуге, предельно прост и не нуждается в дополнительных комментариях:

*// Дуга сети*

TArc = **class**(TProcess)

**public**

*// Конечный узел дуги*

ToNode : TNode;

*// Время выполнения работы*

ActionTime : Double;

**constructor** Create(AToNode : TNode; ActTime : Double);

**protected**

**procedure** RunProcess; **override**;

**end**;

**procedure** TArc.RunProcess;

**begin**

*// Выполнить работу*

Hold(ActionTime);

*// Сообщить узлу о завершении работы*

Inc(ToNode.IncomeArcs);

ToNode.ActivateDelay(0);

**end**;

Заметим, что можно было и не использовать процессы для имитации работы узлов. Достаточно иметь класс, содержащий те же самые поля, а соответствующие операторы предусмотреть в алгоритме работы дуги. Однако показанный вариант организации работы делает алгоритмы более простыми и понятными, а также позволяет более гибко производить их усложнение.

Работа главной программы также очень проста. Следует создать объекты сбора статистики, выполнить серию прогонов и отобразить статистику:

**for** i := 1 **to** NodeCount - 1 **do**

NodeStat[i] := TStatistics.Create;

**for** i := 1 **to** RunCount **do**

**begin**

prt := TPert.Create;

SwitchTo(prt);

prt.Free;

**end**;

**for** i := 1 **to** NodeCount - 1 **do**

**begin**

WriteStat('Node ' + IntToStr(i) + ' statistics:', NodeStat[i]);

WriteLn;

NodeStat[i].Free;

**end**;

Визуальное отображение хода имитации (как это делалось в пункте 3.2.2) не обязательно, поскольку имитация выполняется очень быстро даже на компьютерах со сравнительно слабым процессором.

При одной из серий в 400 прогонов были получены следующие результаты:

Node 1 statistics:

Average = 3.001 +- 0.854

Min = 1.083, max = 4.926

Total = 400 values

Node 2 statistics:

Average = 7.272 +- 1.400

Min = 3.887, max = 11.577

Total = 400 values

Node 3 statistics:

Average = 15.952 +- 2.032

Min = 11.249, max = 22.387

Total = 400 values

Node 4 statistics:

Average = 11.118 +- 2.026

Min = 5.935, max = 15.914

Total = 400 values

Node 5 statistics:

Average = 20.603 +- 1.994

Min = 14.922, max = 28.176

Total = 400 values

Интересно сравнить полученные результаты с теми, которые получаются при ручном вычислении, исходя из некоторых простых предположений. Если предположить, что все работы выполняются за минимально возможное время, то весь проект завершается через 11 единиц времени. Если же предположить, что каждая работа выполняется за максимально возможное время, весь проект выполняется за 34 единицы времени. Как видно из результатов проведенных 400 прогонов, реальный диапазон времени выполнения оказался ощутимо ***у***же. Если предположить выполнение каждой работы за время, равное моде, то проект завершится через 18 единиц времени, а если за математическое ожидание — то через 19 единиц. Эти результаты близки к полученному среднему значению, но все же ощутимо от него отличаются — примерно на величину стандартного отклонения.

Полный текст программы находится в папке 3.7 Pert.

Визуализация данной модели в том качестве, как это рассматривалось ранее, вряд ли имеет смысл. Поэтому рассмотрим оконное приложение, которое отображает с помощью элементов управления результаты прогонов, а также гистограммы по этим же данным. Собственно добавление гистограмм к программе не является проблемой. Для этого следует: поместить массив объектов-гистограмм в программу; обеспечить их создание при запуске программы и очистку перед началом каждой серии прогонов точно так же, как это делается для объектов-статистик; поместить в класс узла ссылку на гистограмму и добавить соответствующий параметр в конструктор; добавить в алгоритм узла запись значения времени в гистограмму. Все эти действия несложно самостоятельно изучить в приведенном примере.

Заслуживает внимания один прием, примененный при разработке. В приложении имеется 5 графических сеток для построения 5 гистограмм. Поскольку, как отмечалось в параграфе 1.6, гистограмма отображается с помощью процедур событий OnDrawCell и OnTopLeftChanged, следовало бы построить процедуры обоих этих событий для каждой гистограммы. Однако содержимое процедур события OnTopLeftChanged полностью идентично, а событий OnDrawCell отличается только индексом объекта-гистограммы в массиве.

Следовательно, можно создать общую процедуру события OnTopLeftChanged и сопоставить ее всем сеткам гистограмм:

**procedure** TfrPert.dgHist1TopLeftChanged(Sender: TObject);

**begin**

(Sender **as** TDrawGrid).Repaint;

**end**;

В этой процедуре происходит обращение к параметру Sender, который указывает на объект-источник события.

Ситуация с событием OnDrawCell несколько осложняется тем, что при построении каждой отдельной гистограммы следует обращаться к разным объектам, находящимися в массиве. Разрешить эту проблему можно с помощью свойства Tag, которое имеется у всех объектов управления. В это свойство можно поместить любое целочисленное значение, не влияющее на работу объекта по его основному назначению. При работе программы можно прочитать значение этого свойства и использовать его в любом качестве. В нашем случае поместим в это свойство каждого из объектов-сеток рисования индекс того объекта гистограммы, содержимое которого требуется отобразить от 1 до 5. В итоге получается следующая процедура отображения:

**procedure** TfrPert.dgHist1DrawCell(Sender: TObject; ACol, ARow: Integer;

Rect: TRect; State: TGridDrawState);

**begin**

DrawHistCell(Sender **as** TDrawGrid, ACol, ARow, Rect,

NodeHist[(Sender **as** TDrawGrid).Tag]);

**end**;

Полный текст программы находится в папке 3.7 Pert Visual.

# 4. Дополнительные средства процессно-ориентированного моделирования

В моделируемых системах бывают ситуации, когда несколько объектов не могут одновременно выполнять действие по причине нехватки необходимых возможностей. Например, обрабатывающие устройства заняты, не хватает деталей для выполнения сборки, места на стоянке заняты автомобилями и т. п. В таком случае объект, который в данный момент не может выполнить действие, вынужден ожидать, пока ему не предоставится такая возможность. Такие средства общего использования, которые могут ограничивать возможности процессов, называются *ресурсами*.

## 4.1. Понятие ресурса

Для добавления возможности работать с ресурсами в имитациях имеется класс, который называется TResource. Прежде че рассматривать его свойства и методы, обсудим общие подходы к организации работы с ним.

Каждый ресурс обладает некоторой *мощностью* — суммарным целочисленным количеством этого ресурса, доступным всем процессам. Некоторое количество единиц ресурса может быть *занято*, то есть передано в распоряжение некоторым процессам. Оставшееся количество ресурса является *доступным*.

Процесс, который желает получить в свое распоряжение некоторое количество единиц ресурса, обращается к нему посредством метода Get, в котором указывается потребное количество. Если в данный момент у ресурса имеется достаточное количество доступных единиц, то они предоставляются процессу. Соответственно, количество занятых единиц увеличивается на запрошенное количество, а количество доступных уменьшается на ту же величину. Если количество доступных единиц ресурса меньше запрашиваемого, то передачи их процессу не происходит. В случае успешного предоставления ресурса данный метод возвращает значение True, а в случае неудачи — False.

В общем случае, процесс, который не смог получить ресурс, переходит в состояние ожидания. Для этого в объекте ресурса предусмотрено несколько очередей. Наличие нескольких очередей позволяет управлять приоритетом разных процессов по возможности доступа к ресурсу. Дополнительным средством управления приоритетом являются функции сравнения очередей, которые определяют, в каком порядке процессы, поступающие в очередь, размещаются в ней.

После того как процесс закончил действие с использованием ресурса, он *освобождает* этот ресурс, то есть возвращает использованные единицы. При этом происходит обратное изменение количества занятых и доступных единиц. В большинстве случаев ресурс освобождается тем же самым процессом, который его получил. Хотя, вообще говоря, это не обязательно должно быть именно так: в ресурсе не ведется учет процессов, которые получали ресурс. После освобождения ресурса возобновляется работа одного или нескольких процессов, ожидающих в очередях этого ресурса.

Возобновление процессов происходит в порядке их размещения в очередях. При этом сначала возобновляются проессы, расположенные в очереди с нулевым индексом, потом — в очереди с первым индексом и т. д. Таким образом, наличие у ресурса нескольких очередей позволяет управлять их приоритетом с точки зрения порядка возобновления. Еще один способ управления приоритетом — функции сравнения, которые можно задавать для очередей.

Однако совершенно не обязательно, что при освобождении ресурса фактически будет возобновлен первый процесс из очереди с нулевым индексом: это зависит еще от количества единиц ресурса, запрашиваемого каждым из ожидающих процессов. Например, если первый процесс в очереди запросил две единицы ресурса, а второй — только одну, то при освобождении одной единицы ресурса первый процесс не сможет продолжить свою работу, потому что необходимого ему количества ресурса нет в наличии. Зато второй процесс сможет сделать это беспрепятственно.

Мощность ресурса может изменяться в процессе работы программы. Для этого в классах ресурса и процессов предусмотрены соответствующие методы. Если мощность ресурса увеличивается, то предпринимается попытка активировать все процессы, ожидающие ресурса. Если мощность уменьшается, возможны две ситуации. Когда она уменьшается на величину, не превышающую доступного количества ресурса, его мощность просто уменьшается. Если же величина уменьшения превышает доступное количество (например, когда доступно 0 елиниц ресурса), то новая мощность ресурса станет меньше количества занятых единиц. Однако при обращении с запросом к ресурсу о количестве доступных единиц результатом будет 0, а не отрицательное число, как можно было бы получить простым вычитанием. Это можно трактовать так: мощность ресурса уменьшается не на полное указанное число единиц, а на столько, сколько было доступно на момент обращения. Дальнейшее уменьшение произойдет, когда будет освобождено еще некоторое количество ресурса.

Например, пусть ресурс имеет мощность 5 единиц, из которых доступна 1 (то есть, 4 единицы заняты). Если выполнить уменьшение мощности на 2 единицы, то она станет равна 3, что меньше текущего занятого количества. Однако занятые 4 единицы по-прежнему остаются в распоряжении взявших их процессов. Но как только какой-либо из них освободит одну единицу ресурса, количество занятого ресурса сравняется с его мощностью, и на этом процесс уменьшения мощности можно считать завершенным.

Объекты ресурсов автоматически собирают интервальную статистику по количеству занятых и доступных единиц ресурса. Если мощность ресурса не меняется в процесс работы имитации, то сумма средних значений этих статистик равна мощности ресурса, а их стандартные отклоения равны друг другу.

Для работы с ресурсами предусмотрен уже упомянутый класс TResource и несколько методов класса TProcess. Рассмотрим сначала основные средства класса TResource (но не все; часть из них отложим до следующего параграфа):

TResource = **class**

**public**

**constructor** Create(InitCap, InitBusy, QueCnt : Integer;

StartTime : Double); **overload**;

**constructor** Create(InitCap, InitBusy, QueCnt : Integer); **overload**;

**constructor** Create(InitCap : Integer; StartTime : Double); **overload**;

**constructor** Create(InitCap : Integer); **overload**;

**constructor** Create; **overload**;

**destructor** Destroy; **override**;

**function** Available : Integer;

**function** Get(cnt : Integer; NewTime : Double) : Boolean; **overload**;

**function** Get(cnt : Integer) : Boolean; **overload**;

**function** Get : Boolean; **overload**;

**procedure** Release(cnt : Integer; NewTime : Double); **overload**;

**procedure** Release(cnt : Integer); **overload**;

**procedure** Release; **overload**;

**procedure** Add(cnt : Integer; NewTime : Double); **overload**;

**procedure** Add(cnt : Integer); **overload**;

**procedure** Add; **overload**;

**procedure** Sub(cnt : Integer; NewTime : Double); **overload**;

**procedure** Sub(cnt : Integer); **overload**;

**procedure** Sub; **overload**;

**procedure** ClearStat(NewTime : Double); **overload**;

**procedure** ClearStat; **overload**;

**procedure** StopStat(NewTime : Double); **overload**;

**procedure** StopStat; **overload**;

**property** Busy : Integer **read**;

**property** Capacity : Integer **read**;

**property** AvailStat : TIntervalStatistics **read**;

**property** BusyStat : TIntervalStatistics **read**;

**property** QueueCount : Integer **read**;

**property** Queue[Idx : Integer] : TList **read**;

**end**;

Сначала рассмотрим параметры конструктора Create. Это в порядке их расположения: исходная мощность, исходное количество занятого ресурса, количество очередей и время, начиная с которого производится сбор статистики. Для вариантов конструктора с меньшим или нулевым количеством параметров подразумеваются следующие их значения. Мощность ресурса полагается равной 1, изначально занято 0 единиц ресурса, ресурс имеет одну очередь ожидания, а начальное время совпадает с текущим имитационным временем в момент создания. Деструктор удаляет объекты статистики, а также все очереди ожидания (что, как мы помним, означает автоматическое завершение всех находящихся в них процессов).

Метод Available возвращает имеющееся на данный момент доступное количество ресурса. Метод Get направляет запрос на получение заданного количества ресурса. Вариант с двумя параметрами предусматривает указание запрашиваемого количества ресурса и момента времени для фиксации факта получения в статистиках. Вариант с одним параметром в качестве момента времени использует текущее имитационное время. Вариант без параметро, кроме того, подразумевает запрос на выделение одной единицы ресурса. Если доступного количества ресурса достаточно для удовлетворения запроса, ресурс выделяется, и метод возвращает результат True. Если доступного ресурса недостаточно, никаких действий не производится, и метод возвращает результат False. Эти методы не производят никаких действий с очередями, они только ведут учет количества единиц ресурса.

Методы Release освобождают заданное число единиц ресурса (или 1, если этот параметр не указан) в заданный момент времени (или в текущий момент). Методы Add и Sub увеличивают и уменьшают мощность ресурса соответственно. Как и в предыдущих случаях, если количество не указано, увеличение или уменьшение проихводится на 1 единицу.

Как правило, процессы не обращаются напрямую к указанным методам, потому что они не выполняются никаких действий с очередями ожидания. Для выполнения этих действий в классе TProcess предусмотрены специальные методы, рассматриваемые далее.

Методы ClearStat и StopStat, аналогично ранее рассмотренным одноименным методам других классов, производят очистку объектов статистики ресурса и их коррекцию к заданному имитационному времени.

Свойства Busy и Capacity дают количество занятых единиц и мощность ресурса соответственно. Свойства AvailStat и BusyStat позволяют обращаться к объектам статистики по доступным и занятым единицам ресурса. Свойство QueueCount дает количество очередей ожидания в объекте ресурса, а свойство-массив Queue[i] позволяет обращаться к каждой из этих очередей отдельно. Индекс очереди может быть в диапазоне от 0 до QueueCount – 1.

В классе TProcess для работы с ресурсами предусмотрены следующие методы:

**procedure** GetResource(Res : TResource; Count, Index : Integer); **overload**;

**procedure** GetResource(Res : TResource; Count : Integer); **overload**;

**procedure** GetResource(Res : TResource); **overload**;

**procedure** GetResource(Res : TResource; GetRes : TGetResourceFunc;

Index : Integer); **overload**;

**procedure** GetResource(Res : TResource; GetRes : TGetResourceFunc); **overload**;

**procedure** ReleaseResource(Res : TResource; Count : Integer); **overload**;

**procedure** ReleaseResource(Res : TResource); **overload**;

**procedure** ChangeResource(Res : TResource; Count : Integer);

Методы GetResource предназначены для получения заданного количества ресурса. Параметр Res указывает на запрашиваемый ресурс, Count — количество запрашиваемых единиц (1, если не указан), Index — номер очереди, в которую помещается процесс в случае ожидания (0, если не указана). Работа первых трех методов происходит одинаково: процесс помещается в указанную очередь ресурса и выполняет цикл ожидания до тех пор пока не удастся получить запрошенное количество ресурса. После успешного получения ресурса метод завершается, и процесс продолжает работу.

Еще два варианта метода GetResource предназначены для так называемого *совместного* получения ресурса, то есть одновременного получения некоторого количества сразу нескольких ресурсов. В этих вариантах метода указывается параметр GetRes, который является ссылкой на метод класса без параметров и результатом типа Boolean, определенный следующим образом:

TGetResourceFunc = **function** : Boolean **of** **object**;

Метод, передаваемый в качестве параметра, должен быть определен в классе процесса, выполняющего запрос на получение ресурса. Он должен проверять возможность выделения ресурсов; для этого он может обращаться к любым элементам процесса и имитации в целом. Если ресурса возможно, метод производит его и возвращает результат True. Если нет, то возвращает результат False. Попутно заметим, что даже если производится попытка выделения одновременно нескольких ресурсов, процесс помещается в очередь ожидания только одного из них — того, который указан параметром Res, этот ресурс можно назвать в данном контексте *основным*. Как и в предыдущем случае, если номер очереди не указан, подразумевается 0.

Методы ReleaseResource освобождают заданное количество единиц ресурса (или 1, если этот параметр не указан). Метод ChangeResource изменяет мощность ресурса. Естественно, если параметр Count положителен, то мощность увеличивается, а если отрицателен — уменьшается. Методы ReleaseResource и ChangeResource автоматически активируют процессы, ожидающие выделения ресурса, если есть такая возможность.

Одно из удобных применений ресурсов — имитация действий, которые могут одновременно выполняться ограниченным числом процессов. Ранее для организации подобных действий мы использовали обслуживающий процесс. Ограничение количества действий обеспечивалось автоматически, поскольку количество обслуживающих процессов также ограничено.

С помощью ресурсов те же действия можно организовать следующим образом. Перед началом действия процесс запрашивает одну единицу ресурса, затем выполняет действие и освобождает ресурс. Это записывается в следующем виде:

GetResource(par.ActionRes);

*// Выполнение действия*

ReleaseResource(par.ActionRes);

Ограничение количества одновременно выполняющихся действий обеспечивается объектом ресурса: если его не хватает, то «лишние» процессы будут приостановлены до освобождения занятых. При такой организации работы отпадает потребность в обслуживающих процессах. Также появляются более широкие возможности для формулировки алгоритма действия: теперь выполняемое действие может состоять из нескольких этапов и даже быть построено по сколь угодно сложному алгоритму. Кроме того, отпадает необходимость в явной постановке процесса в очередь ожидания, организации цикла ожидания, извлечения процесса из очереди. Следует отметить, что действия процессов, выполняемые в таком контексте, трактуются уже не как обслуживающие, а как самостоятельные.

Процесс клиента банка из модели, рассмотренной в параграфе 3.1, при использовании ресурсов может быть построен следующим образом (основная часть комментариев из текста опущена, поскольку все указанные действия уже обсуждались ранее):

**procedure** TClient.RunProcess;

**var**

par : TBankSimulation;

**begin**

par := Parent **as** TBankSimulation;

*// Дождаться освобождения кассира*

GetResource(par.Cashman);

*// Приступить к действию*

**if** StartingTime = SimTime **then**

Inc(par.NotWaited);

*// Сбор статистики по действию удален,*

*// так как он выполняется объектом ресурса*

Hold(rndCashman.Uniform(MinCashTime, MaxCashTime));

InTime := SimTime - StartingTime;

par.InBankTime.AddData(InTime);

par.InBankHist.AddData(InTime);

*// Количество завершенных процессов определяется с помощью статистики*

*// по времен пребывания в системе*

**if** par.InBankTime.Count = MaxClientCount **then**

par.ActivateDelay(0);

*// Действие завершено, освободить кассира*

ReleaseResource(par.Cahsman);

Finish;

**end**;

В измененном варианте модели следует удалить процесс кассира и заменить его на ресурс. Кроме того, удалить следует очередь ожидания: у объекта ресурса имеется собственная очередь. Объекта статистики также удаляется: вся статистика собирается непосредственно объектом ресурса. Соответственно меняются методы инициализации и деструктора имитации. Полный текст переработанной с использованием ресурсов имитации находится в папке 4.1 Bank Resources.

Отметим еще некоторые преимущества данного варианта построения модели. Во-первых, теперь изменить количество касс очень просто: достаточно указать его при создании ресурса. Во-вторых, количество касс может меняться непосредственно в процессе работы программы: например, если количество покупателей стало достаточно большим, его можно увеличить, а если их стало меньше, закрыть лишние кассы. Другой вариант — организовать изменение числа касс в зависимости от времени суток. Ответственность за это изменение можно возложить, например, на процесс генератора покупателей.

Указанные преимущества при организации действий посредством ресурсов, а не обслуживающих процессов, однако, не должны приводить к иллюзии, будто использование ресурсов всегда предпочтительнее. В ряде случаев имитировать действие с их помощью менее удобно или вообще невозможно.

Так, например, с помощью ресурсов нельзя имитировать работу нескольких аналогичных, но все же различных обрабатывающих устройств, отличающихся по некоторым параметрам. С помощью ресурсов также невозможно обеспечить индивидуальный учет статистики по каждому устройству в отдельности. Нельзя обеспечить наличие отдельной очереди к каждому из них. Последние два ограничения можно обойти, если вместо одного ресурса создать массив ресурсов единичной мощности. Однако в этом случае невозможно создать общую очередь ко всем устройствам. Кроме того, с помощью ресурсов нельзя имитировать работу обслуживающих устройств, которые, помимо собственно обслуживающего действия, выполняют что-то еще.

Еще один возможный взгляд на ресурс состоит в том, что ресурс — это «умный» счетчик, как и объект сбора статистики TActionStatistics, который автоматически контролирует нахождение своего значения в заданном диапазоне, собирает интервальную статистику о своем значении и управляет обращающимися к нему процессами. В отличие от упомянутого класса TActionStatistics, значение ресурса не может стать отрицательным. Процесс, который пытается уменьшить равное нулю значение, приостанавливается до тех пор, пока его значение не станет положительным.

Таким образом, ресурс — это лишь еще один инструмент построения имитации, удобный в одних ситуациях и неудобный в других. Выбор конкретного способа имитации следует осуществлять в результате анализа поставленной задачи.

## 4.2. Модель планирования запасов

Рассмотрим несколько примеров построения имитации с использованием ресурсов. Как и в примерах предыдущей главы, формулировки задач взяты из работы [2], но решены они с использованием процессно-ориентированного подхода.

*Постановка задачи*. Рассмотрим работу магазина по продаже определенного вида товара (пусть это будут, например, радиоприемники). Интервал прихода покупателей распределен экспоненциально с математическим ожиданием, равным 0,2 недели. Если в момент прибытия покупателя на складе магазина имеется хотя бы один радиоприемник, покупатель приобретает его и покидает систему. Если в данный момент радиоприемников данной модели нет, с вероятностью 0,2 покупатель делает предварительный заказ и приобретает приемник после его появления в магазине. Если покупатель не делает заказа, то это расценивается как несостоявшаяся покупка, он сразу покидает систему, и данные об интервалах времени между такими покупателями следует статистически обработать.

Максимально возможное количество радиоприемников на складе равно 72. Периодически, раз в 4 недели, производится проверка содержимого склада. Если в момент проверки выясняется, что количество имеющихся в наличии радиоприемников не превышает 18, производится заказ поставщику. Объем заказа рассчитывается исходя из того, чтобы пополнить общее количество приемников до 72 с учетом текущего наличия на складе либо имеющихся предварительных заказов. При этом возможные будущие продажи или предварительные заказы, которые могут поступить во время исполнения этого заказа, не принимаются во внимание. Заказанные приемники поступают на склад через 3 недели после того, как заказ был отправлен. Для удобства учета заказанных приемников вводится понятие *учетного количества*. Это количество приемников с учетом сделанных предварительных заказов и заказа поставщику. Каждая продажа и предварительный заказ уменьшает это количество на 1. При отсутствии товара на складе и наличии предварительных заказов это количество становится отрицательным. Если формируется заказ поставщику товара, учетное количество доводится до максимального значения (собственно говоря, получающаяся разница и составляет объем заказа поставщику).

Требуется провести имитацию в течение 6 лет (312 недель). При этом по истечении 1 года (52 недель) с начала имитации следует очистить все статистические данные. Поэтому фактически собирается информация за 5 лет работы (260 недель). Такой подход позволяет устранить погрешности, вносимые переходным процессом, и собирать данные уже в установившемся режиме работы.

Следует собрать следующие статистические данные:

* о состоянии запаса радиоприемников (то есть, об их фактическом количестве на складе по времени);
* об учетном количестве радиоприемников;
* об интервалах времени между несостоявшимися покупками;
* о фактическом количестве имеющихся на складе приемников на момент исполнения заказа поставщиком;
* о времени ожидания покупателей в очереди.

*Решение*. В этой задаче удобно учитывать количество имеющихся на складе радиоприемников с помощью ресурса, мощность которого равна 72, то есть максимально возможному количеству приемников на складе. При приобретении приемника покупатель получает 1 единицу этого ресурса. Если на складе приемников нет, то покупатель становится в очередь ожидания; это трактуется как предварительный заказ, сделанный покупателем. При выполнении заказа поставщиком этот ресурс освобождается в количестве, равном объему заказа, при этом возобновляют свою работу покупатели, ожидавшие появления приемников.

Очевидно, покупатель в данной модели является компонентом. Войдя в систему, он должен проверить наличие на складе приемников, а если их нет, принять решение о предварительном заказе. Если покупатель отказывается от выполнения покупки, он должен зафиксировать текущее время в соответствующей статистике. Если же принято решение о покупке (немедленной или путем предварительного заказа), следует уменьшить учетное количество примеников. В случае выполнения предварительного заказа это количество станет отрицательным. Затем следует получиь одну единицу ресурса приемников. В случае предварительного заказа процесс на этом будет приостановлен до выполнения заказа производителем. В итоге получается следующий алгоритм его работы:

**procedure** TCustomer.RunProcess;

**var**

par : TSavings;

**begin**

par := Parent **as** TSavings;

*// Покупатель приобретает приемник, если:*

*// 1. Приемники имеются в наличии*

*// 2. Приемников нет, тогда он с вероятностью OrderProb*

*// встает в очередь ожидания*

**if** (par.Radios.Available > 0) **or** rndCustomer.Draw(OrderProb) **then**

**begin**

*// Уменьшить учетное количество приемников (отрицательное число*

*// означает наличие заказов, ожидающих выполнения)*

Dec(par.CurrentAvailable);

par.InvPosStat.AddData(par.CurrentAvailable, SimTime);

*// Приобрести приемник или ждать, пока они не появятся в наличии*

GetResource(par.Radios);

**end**

**else**

*// Зафиксировать статистику по отказам*

par.LostSalesStat.AddData(SimTime);

*// Встать в очередь завершенных процессов*

Finish;

**end**;

Генератор покупателей построен по стандартной схеме и не нуждается в пояснениях:

**procedure** TCustomerGenerator.RunProcess;

**begin**

**while** True **do**

**begin**

*// Удалить завершенные процессы (если есть)*

ClearFinished;

*// Создать покупателя*

TCustomer.Create.ActivateDelay(0);

*// Покупатели прибывают с экспоненциально распределенным интервалом*

Hold(rndCustomer.Exponential(MeanCustTime));

**end**;

**end**;

За проверку наличия приемников на складе отвечает отдельный процесс, который с интервалом 4 недели проверяет текущее состояние склада. Если количество приемников оказывается меньше или равно предельно допустимого значения, производится заказ поставщику, который исполняется в течение 3 недель. После исполнения заказа требуется выждать остающуюся неделю. Упрощает постоение алгоритма то, что, во-первых, интервал между проверками и продолжительность исполнения заказа — величины постоянные, а во-вторых, продолжительность исполнения заказа меньше интервала между проверками. Если бы это было не так, пришлось бы определять отдельный процесс, выполняющий проверку (по аналогии с генератором процессов), и другой процесс, имитирующий исполнение заказа (по аналогии с компонентами). Алгоритм проверки выглядит следующим образом:

**procedure** TCheck.RunProcess;

**var**

par : TSavings;

OrderCount : Integer;

**begin**

par := Parent **as** TSavings;

**while** True **do**

**begin**

*// Если учетное количество приемников меньше предельно допустимого*

**if** par.CurrentAvailable < OrderLimit **then**

**begin**

*// Вычислить объем заказа производителю*

OrderCount := InvPosition - par.CurrentAvailable;

*// Восстановить учетное количество*

par.CurrentAvailable := InvPosition;

par.InvPosStat.AddData(par.CurrentAvailable, SimTime);

*// Дождаться выполнения заказа*

Hold(OrderTime);

*// Зафиксировать статистику по резервному запасу*

par.SafetyStockStat.AddData(par.Radios.Available);

*// Восстановить количество ресурса*

ReleaseResource(par.Radios, OrderCount);

*// Выждать оставшееся время до следующей проверки*

Hold(CheckInterval - OrderTime);

**end**

**else**

*// Ждать следующей проверки*

Hold(CheckInterval);

**end**;

**end**;

Заметим еще раз, что учетное количество восстанавливается немедленно при обнаружении факта необходимости заказа поставщику (фактически, при отправке этого заказа), а количество приемников на складе, то есть доступное количество ресура — после его исполнения.

Что касается класса имитации, то его построение также не содержит ничего нового. В классе находятся все необходимые объекты для его работы: ресурс, счетчик учетного количества, объекты статистики, ссылки на процессы. Что касается алгоритма, то его работа состоит из двух этапов. Первый этап, предварительный прогон, выполняется в течение 1 года для приведения системы в некоторое рабочее состояние. После этого все объекты статистики очищаются, и запускается второй, основной этап имитации, работающий в течение 5 лет. Определение класса и такст алгоритма приведены ниже:

TSavings = **class**(TSimulation)

**public**

*// Ресурс по имеющимся в наличии приемникам*

Radios : TResource;

*// Статистика по интервалам между отказами от покупки*

LostSalesStat : TTimeBetStatistics;

*// Статистика по запасам на момент исполнения заказа*

SafetyStockStat : TStatistics;

*// Статистика по наличию приемников на складе*

InvPosStat : TIntervalStatistics;

*// Генератор покупателей*

CustGen : TCustomerGenerator;

*// Процесс проверки состояния склада*

Check : TCheck;

*// Учетное количество приемников*

CurrentAvailable : Integer;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**procedure** ClearStat; **override**;

**protected**

**procedure** RunSimulation; **override**;

**procedure** Init; **override**;

**end**;

**procedure** TSavings.RunSimulation;

**begin**

CustGen.ActivateDelay(0);

Check.ActivateDelay(0);

*// Предварительная имитация*

Hold(PreSimulationTime);

*// Очистка статистики*

ClearStat;

*// Продолжение имитации*

Hold(MainSimulationTime);

*// Коррекция статистики*

StopStat;

**end**;

В этом примере впервые пришлось переопределить метод ClearStat, отвечающий за очистку всех объектов статистики. Его определение не представляет трудностей:

**procedure** TSavings.ClearStat;

**begin**

**inherited**;

InvPosStat.Clear(SimTime);

SafetyStockStat.Clear;

LostSalesStat.Clear;

Radios.ClearStat(SimTime);

**end**;

Полный текст программы находится в папке 4.2 Savings. Один из прогонов программы дал следующий результат:

Simulation stopped at 312.00

Statistics last cleared at 52.00

Lost sales statistics:

Average = 2.305 +- 6.340

Min = 0.003, max = 36.439

Total = 103 values

Safety stock statistics:

Average = 1.167 +- 2.407

Min = 0.000, max = 9.000

Total = 18 values

Inventory position statistics:

Average = 42.416 +- 20.256

Min = -1.000, max = 72.000

Total = 260.000 time units, current value = 41.00

Radio resource statistics:

Current capacity = 72

Utilization = 42.894 +- 20.369

Max utilization = 72, current = 31

Current available = 41, average = 29.106

Min available = 0, max = 71

Order waiting statistics:

Average length = 0.167 +- 0.756

Max = 7, current = 0

Average waiting time = 0.037

Calendar statistics:

Average length = 3.000 +- 0.000

Max = 3, current = 3

Average waiting time = 0.214

Отметим, что согласно данным по учетному количеству товара (inventory position) минимальное значение равно –1, что означает наличие не более одного предварительного заказа. В то же время статистика по очереди ожидания (order waiting), в которой находятся покупатели, сделавшие предварительные заказы, говорит, что максимальная длина очереди была равно 7. Нет ли здесь противоречия?

Нет, если вспомнить, что учетное количество товара увеличивается в момент отправки заказа поставщику, то есть перед этим моментом имеет локальный минимум. Однако предварительные заказы от покупателей могут добавляться в течение времени исполнения этого заказа. Например, могло быть так, что к моменту проверки имелся один предварительный заказ от покупателя, а за 3 недели, пока поставщик исполнял заказ, было получено еще 6 предварительных заказов.

Статистика по состоянию резервного запаса на момент исполнения заказа (safety stock) сообщает, что всего поставщиком было исполнено 18 заказов, при этом количество товара на складе к моменту исполнения заказа было от 0 до 9 единиц при среднем значении чуть больше 1.

Как и в предыдущих примерах, при выполнении визуализации добавим к постановке задачи некоторые дополнительные условия. Во-первых, будем учитывать количество совершенных продаж (учет несостоявшихся продаж труда не представляет: эта информация доступна из соответствующего объекта статистики). Во-вторых, добавим учет данных по времени ожидания тех покупателей, которые делали предварительные заказы. Дело в том, что, поскольку большинство покупателей становятся в очередь и сразу же ее покидают, величина среднего времени ожидания не позволяет сделать сколько-нибудь достоверных предположений по времени ожидания предварительных заказов. В-третьих, время исполнения заказа становится величиной случайной, имеющей нормальное распределение с математическим ожиданием 3 недели, стандартным отклонением 1 неделя и минимальным значением 2 недели.

Внесение первых двух дополнений не составляет труда. В класс имитации следует добавить целочисленный счетчик совершенных продаж и объект сбора статистики по времени ожидания, а в алгоритм работы покупателя добавить действия, увеличивающие счетчик и добавляющие значение времени ожидания, если оно было положительно:

TSavings = **class**(TSimulation)

**public**

…

*// Количество сделанных покупок*

BuysCount : Integer;

*// Статистика по покупателям, сделавшим предварительный заказ*

WaitStat : TStatistics;

…

**end**;

**procedure** TCustomer.RunProcess;

…

*// Покупка выполнена*

Inc(par.BuysCount);

*// Если покупатель ожидал заказа, учесть его*

**if** SimTime > StartingTime **then**

par.WaitStat.AddData(SimTime - StartingTime);

…

**end**;

Ситуация с третьим дополнением несколько сложнее. Дело не только в том, что время выполнения заказа различно, но еще и в том, что в некоторых случаях длительность исполнения заказа может превышать промежуток времени между последовательными проверками. Чтобы решить проблему, разделим обязанности между двумя процессами. Существующий процесс TCheck только лишь периодически проверяет, следует ли выполнять очередной заказ поставщику. Если следует, он активирует другой процесс, TOrder, который, собственно, и моделирует исполнение заказа. Собственно, действия, входящие в теперь уже два алгоритма, остаются теми же, что и в исходной задаче, только теперь они распределены между двумя процессами. В итоге, алгоритмы обоих процессов будут выглядеть следующим образом:

**procedure** TCheck.RunProcess;

**var**

par : TSavings;

**begin**

par := Parent **as** TSavings;

**while** True **do**

**begin**

*// Если учетное количество приемников меньше предельно допустимого*

**if** par.CurrentAvailable < OrderLimit **then**

*// Запустить процесс выполнения заказа поставщику*

par.Order.ActivateDelay(0);

Hold(CheckInterval);

**end**;

**end**;

**procedure** TOrder.RunProcess;

**var**

par : TSavings;

OrderCount : Integer;

**begin**

par := Parent **as** TSavings;

**while** True **do**

**begin**

*// Вычислить объем заказа поставщику*

OrderCount := InvPosition - par.CurrentAvailable;

*// Восстановить учетное количество*

par.CurrentAvailable := InvPosition;

par.InvPosStat.AddData(par.CurrentAvailable, SimTime);

*// Дождаться выполнения заказа*

Hold(Max(MinOrderTime,

rndOrder.Normal(MeanOrderTime, DeviationOrderTime)));

*// Зафиксировать статистику по резервному запасу*

par.SafetyStockStat.AddData(par.Radios.Available);

*// Восстановить количество ресурса*

ReleaseResource(par.Radios, OrderCount);

*// Ожидать следующего заказа*

Passivate;

**end**;

**end**;

Полный текст программы находится в папке 4.2 Savings Visual.

## 4.3. Модель работы порта

*Постановка задачи*. Смоделируеи работу порта, в котором танкеры нагружаются нефтью. В порту имеются три причала, которые могут одновременно загружать не более трех танкеров. Танкеры прибывают в порт с интервалом, равномерно распределенным в диапазоне от 4 до 18 ч. Танкеры могут быть трех типов, условно обозначенных от 0 до 2. относительная частота появления танкеров каждого типа и граничные времена равномерно распределенного времени их погрузки приведены в таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Относительная частота появления | Время погрузки | |
| Минимальное | Максимальное |
| 0 | 0,25 | 16 ч | 20 ч |
| 1 | 0,55 | 21 ч | 27 ч |
| 2 | 0,20 | 32 ч | 40 ч |

В порту имеется один буксир, который используется при пирчаливании и отчаливании танкеров. Причаливание и отчаливание занимает время в 1 ч, причем, если одновременно ест танкеры, ожидающие причаливания и отчаливания, приоритет отдается операции причаливания.

Судовладелец предлагает администрации порта принять на обслуживание новый тип танкера (номер 3). Время погрузки этого типа танкеров распределено равномерно в интервале от 18 до 24 ч. Всего имеется пять таких танкеров, которые, загрузившись, отправляются на разгрузку и обратно. Полное время обращения такого танкера распределено равномерно в интервале от 216 до 264 ч. Первый из таких танкеров прибывает в момент времени 0, а остальные — через 48 ч каждый.

Работу порта осложняют штормы. Каждый шторм имеет равномерно распределенную длительность от 2 до 6 ч. Промежуток времени между окончанием шторма распределен экспоненциально с математическим ожиданием 48 ч. На такое же время намечается начало первого шторма. Во время шторма буксир не работает, однако операция причаливания или отчаливания, выполняющаяся в момент начала шторма, доводится до завершения.

Администрация порта перед заключением контракта решила определить влияние, которое окажут на время пребывания судов в порту пять дополнительных танкеров. Выводы предлагается сделать на основе имитации работы порта в течение года (8760 ч). Следует оценить время пребывания в порту танкеров всех типов, а также загруженность буксира и причалов.

*Решение*. Сначала определим процессы и ресурсы, необходимые для решения задачи. Очевидно, роль компонентов в этой задаче играют танкеры. Однако их поведение несколько различается в зависимости от типа. А именно, танкеры типов 0—2 выполняют однократное действие по прибытию в порт, погрузке и отплытию, в то время как танкеры типа 3 выполняют переход для разгрузки и обратно и снова выполняют погрузку, то есть, их работа организована циклически.

Занятость причалов и танкера удобно контролировать при помощи ресурсов. Мощности их равны 3 и 1 соответственно. Поскольку в формулировке задачи явно указано на приоритеты при выборе действия для буксира (для причаливания или отчаливания), предусмотрим в соответствующем ресурсе две очереди. Более приоритетная очередь с индексом 0 будет использоваться для ожидания танкеров, ожидающих причаливания, а менее приоритетная очередь с индексом 1 — для отчаливания.

Влияние шторма можно моделировать, изменяя мощность ресурса буксира. А именно, при начале шторма его мощность уменьшается на 1 (то есть, до нуля), а при окончании — увеличивается на 1 (то есть, восстанавливается ее исходное значение). При таком подходе ни один из танкеров во время шторма не сможет получить в свое распоряжение ресурс буксира. Однако, если во время начала шторма ресурс буксира был занят, уменьшение его мощностьи фактически откладывается до момента его освобождения, что позволяет довести текущую операции причаливания или отчаливания до конца. Управлять мощностью ресурса буксира будет отдельный процесс, который через заданные интервалы времени будет увеличивать и уменьшать ее.

Естественно, для организации прибытия танкеров потребуется процесс-генератор, который, однако, может создавать процесс танкеров только типов 0—2. Для порождения пяти танкеров типа 3 создадим отдельный процесс-генератор.

В качестве результата будем собирать и выводить следующую статистику: по времени нахождения танкера в порту от момента прибытия до момента окончания отчаливания; по загруженности ресурсов причалов и буксира; по очередям ожидания свободного причала и буксира, для последнего — отдельно по операциям причаливания и отчаливания.

Сначала определим структуру классов танкера и имитации. В классе танкера следует указать номер его типа. В классе имитации создаются все составляющие процесса и объекты сбора статистики:

**type**

*// Процесс, имитирующий работу танкера*

TTanker = **class**(TProcess)

**public**

*// Тип танкера*

TankerType : Integer;

**constructor** Create(TType : Integer);

**protected**

**procedure** RunProcess; **override**;

**end**;

*// Имитация работы порта*

TPort = **class**(TSimulation)

**public**

*// Причалы*

Berth : TResource;

*// Буксир*

Tug : TResource;

*// Генераторы танкеров*

Gen : TTankerGenerator;

Gen3 : TTanker3Generator;

*// Шторм*

Storm : TStorm;

*// Статистика по времени пребывания танкеров по типам*

TimeStat : **array** **of** TStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** RunSimulation; **override**;

**procedure** Init; **override**;

**end**;

Определения не указанных здесь остальных классов процессов не содержат ничего нового; в них только переопределяется метод RunProcess, как было в предыдущих примерах.

Алгоритм процесса-танкера включает в себя следующие действия:

* запоминание времени прибытия;
* получение по 1 единице ресурсов причала и танкера, причем именно в указанном порядке. В противном случае возможна ситация, когда, получив в свое распоряжение буксир, танкер не сможет продолжить работу, так как все причалы заняты. В то же время, ни один танкер не сможет освободить причал, поскольку буксир также занят;
* выполнение причаливания и освобождение буксира;
* выполнение погрузки;
* получение 1 единицы ресурса буксира для отчаливания (на этот раз с постановкой в очередь номер 1);
* отчаливание и освобождение ресурсов причала и буксира;
* сбор статистики по времени пребывания в порту.

Все эти действия танкеры типов 0—2 выполняют однократно, а танкеры типа 3 — многократно в цикле (последние, кроме того, еще и выполняют переход для разгрузки и обратно). Чтобы по-разному организовать выполнение действий для танкеров разных типов, в алгоритме работы танкера предусмотрена проверка номера его типа и завершение работы после первого исполнения для танкера типов 0—2:

**procedure** TTanker.RunProcess;

**var**

par : TPort;

ArriveTime : Double;

**begin**

par := Parent **as** TPort;

**while** True **do**

**begin**

*// Запомнить время прибытия*

ArriveTime := SimTime;

*// Получить свободный причал*

GetResource(par.Berth);

*// Получить буксир для причаливания*

GetResource(par.Tug);

*// Выполнить причаливание*

Hold(1);

*// Освободить буксир*

ReleaseResource(par.Tug);

*// Выполнить погрузку. Время выполнения зависит от типа танкера*

Hold(rndTanker.Uniform(MinLoadingTime[TankerType],

MaxLoadingTime[TankerType]));

*// Получить буксир для отчаливания*

GetResource(par.Tug, 1, 1);

*// Выполнить отчаливание*

Hold(1);

*// Освободить причал*

ReleaseResource(par.Berth);

*// Освободить буксир*

ReleaseResource(par.Tug);

*// Зафиксировать статистику времени пребывания по типу танкера*

par.TimeStat[TankerType].AddData(SimTime - ArriveTime);

*// Если танкер типа 0-2, закончить работу*

**if** TankerType < TypeCount - 1 **then**

Break;

*// Выполнить переход на разгрузку и обратно*

Hold(rndTanker.Uniform(MinTripTime, MaxTripTime));

**end**;

Finish;

**end**;

Два процесса генераторов различаются только способом организации цикла и указания типа танкера:

**procedure** TTankerGenerator.RunProcess;

**var**

TType : Integer;

**begin**

**while** True **do**

**begin**

*// Очистить список завершенных процессов*

ClearFinished;

*// Случайно выбрать тип танкера*

TType := rndTanker.TableIndex(TypeProb);

*// Создать танкер*

TTanker.Create(TType).ActivateDelay(0);

*// Ожидать прибытия следующего танкера*

Hold(rndTanker.Uniform(MinInterval, MaxInterval));

**end**;

**end**;

**procedure** TTanker3Generator.RunProcess;

**var**

i : Integer;

**begin**

*// Всего создается 5 танкеров типа 3*

**for** i := 1 **to** Type3Count **do**

**begin**

*// Создать танкер типа 3*

TTanker.Create(3).ActivateDelay(0);

*// Ожидать прибытия следующего танкера*

Hold(Interval3);

**end**;

**end**;

Что касается процесса, имитирующего работу шторма, то в нем также нет ничего сложного. При начале каждого шторма этот процесс уменьшает мощность буксира на 1, а по окончании снова увеличивает:

**procedure** TStorm.RunProcess;

**var**

par : TPort;

**begin**

par := Parent **as** TPort;

**while** True **do**

**begin**

*// Время до следующего шторма*

Hold(rndStorm.Exponential(StormInterval));

*// Сделать буксир нерабочим на время шторма*

ChangeResource(par.Tug, -1);

*// Длительность шторма*

Hold(rndStorm.Uniform(MinStormTime, MaxStormTime));

*// Освободить буксир после шторма*

ChangeResource(par.Tug, 1);

**end**;

**end**;

Построение собственно имитации не вызывает трудностей:

**procedure** TPort.RunSimulation;

**begin**

*// Запустить процессы*

Gen.ActivateDelay(0);

Gen3.ActivateDelay(0);

Storm.ActivateDelay(0);

*// Время имитации – 1 год*

Hold(SimulationTime);

StopStat;

**end**;

Полный текст программы находится в папке 4.3 Port. Один из прогонов дал следующие результаты:

Type 0 in port time:

Average = 39.863 +- 15.067

Min = 18.014, max = 101.800

Total = 204 values

Type 1 in port time:

Average = 45.987 +- 16.275

Min = 23.062, max = 110.253

Total = 438 values

Type 2 in port time:

Average = 56.437 +- 14.451

Min = 34.182, max = 109.122

Total = 160 values

Type 3 in port time:

Average = 42.615 +- 14.784

Min = 20.079, max = 92.036

Total = 155 values

Berth resource:

Current capacity = 3

Utilization = 2.929 +- 0.282

Max utilization = 3, current = 3

Current available = 0, average = 0.071

Min available = 0, max = 2

Tug resource:

Current capacity = 1

Utilization = 0.219 +- 0.413

Max utilization = 1, current = 1

Current available = 0, average = 0.697

Min available = 0, max = 1

Berth queue:

Average length = 2.087 +- 1.753

Max = 9, current = 0

Average waiting time = 19.044

Tug for arrival queue:

Average length = 0.012 +- 0.108

Max = 1, current = 0

Average waiting time = 0.108

Tug for depart queue:

Average length = 0.043 +- 0.214

Max = 2, current = 0

Average waiting time = 0.392

Calendar:

Average length = 10.093 +- 0.784

Max = 12, current = 11

Average waiting time = 10.041

По этим результатам видно, что загруженность причалов равна 2,929, то есть, близка к предельной. Кроме того, танкеры проводят значительное время в ожидании освобождения причала — в среднем около 19 ч, а максимальная длина очереди достигает 9 танкеров. Все эти обстоятельства наводят на предположение о том, что предложенные дополнительные 5 танкеров ощутимо осложнят работу порта. Чтобы в этом убедиться, выполним прогон, в котором не участвуют танкеры типа 3. Для этого достаточно убрать из текста (или закрыть комментарием) строчку Gen3.ActivateDelay(0) в алгоритме процесса имитации. Строго говоря, для соответствующего изменения программы следовало бы полностью исключить все ее фрагменты, относящиеся к танкерам типа 3, а именно, класс генератора, элемент массива объектов статистики, элементы массивов, задающих границы времен погрузки, а также упростить алгоритм работы танкера, исключив из него цикл. Однако для пробного прогона вполне достаточно временного отключения генератора. Один из прогонов дает следующие результаты:

Type 0 in port time:

Average = 22.170 +- 3.805

Min = 18.043, max = 37.384

Total = 201 values

Type 1 in port time:

Average = 28.107 +- 3.944

Min = 23.032, max = 46.925

Total = 439 values

Type 2 in port time:

Average = 40.342 +- 4.236

Min = 34.053, max = 58.794

Total = 147 values

Type 3 in port time:

Average = 0.000 +- 0.000

Min = 0.000, max = 0.000

Total = 0 values

Berth resource:

Current capacity = 3

Utilization = 2.440 +- 0.641

Max utilization = 3, current = 2

Current available = 1, average = 0.56

Min available = 0, max = 2

Tug resource:

Current capacity = 0

Utilization = 0.180 +- 0.384

Max utilization = 1, current = 0

Current available = 0, average = 0.75

Min available = 0, max = 1

Berth queue:

Average length = 0.158 +- 0.416

Max = 3, current = 0

Average waiting time = 1.751

Tug for arrival queue:

Average length = 0.012 +- 0.109

Max = 1, current = 0

Average waiting time = 0.133

Tug for depart queue:

Average length = 0.021 +- 0.152

Max = 2, current = 0

Average waiting time = 0.233

Calendar:

Average length = 5.407 +- 0.647

Max = 6, current = 5

Average waiting time = 8.070

Сравнивая эти два результата, нетрудно убедиться, что добавление танкеров нового типа существенно увеличивает загруженность причалов, время пребывания танкеров в порту и длину очереди ожидания. Поэтому по результатам выполнения данной имитации наиболее разумное решение администрации порта — отказаться от заключения нового контракта до расширения мощностей порта.

Рассмотренный вариант построения модели обладает одним недостатком. Каждый танкер, прибыв в порт, сначала дожидается получения ресурса причала, а потом получает ресурс буксира. Вполне может оказаться так, что при получении причала буксир еще занят, в этом случае танкер ожидает его освобождения. Этот период ожидания отмечается в статистике как промежуток времени, в течение которого причал занят, хотя реально он еще некоторое время не используется. Фактически, как занятое отмечается время, когда ресурс причала не занят, а назначен тому или иному танкеру. Чтобы исправить это несоответствие, следует сделать так, чтобы ресурс причала не выделялся до тех пор, пока в наличии не будет свободного буксира. Другими словами, выделение этих двух ресурсов следует выполнять *совместно*. Как отмечалось в параграфе 4.1, это можно следать при помощи отдельного варианта метода GetResource и дополнительного метода, выполняющего выделение обоих ресурсов. Добавим в класс танкера метод. Его объявление и текст приведены ниже:

*// Объявление метода в классе TTanker*

**private**

*// Совместное выделение ресурсов причала и буксира*

function GetBerthAndTug : Boolean;

**function** TTanker.GetBerthAndTug: Boolean;

**var**

par : TPort;

**begin**

par := Parent **as** TPort;

*// Совместное получение ресурсов причала и буксира*

*// Если оба ресурса имеются в наличии*

**if** (par.Berth.Available > 0) **and** (par.Tug.Available > 0) **then**

**begin**

*// Получить по 1 единице ресурса*

par.Berth.Get;

par.Tug.Get;

*// Возвратить True*

Result := True;

**end**

**else**

*// Наличных ресурсов нет в наличии, возвратить False*

Result := False;

**end**;

Для совместного выделения внесем одно изменение в метод RunProcess танкера. А именно, заменим два расположенных подряд действия по выделению ресурсов одним:

*// Получить свободный причал и буксир*

GetResource(par.Tug, GetBerthAndTug);

Обратите внимание на следующее обстоятельство. Несмотря на то, что данным действием выделяется по одной единице каждого из двух ресурсов, процесс встает в очередь ожидания только к одному из них — в данном случае, к ресурсу буксира. Это вполне объяснимо, так как процесс, являясь ячейкой списка, не может находиться одновременно более чем в одном списке, и его включение в любой список сопровождается автоматическим исключением из того списка, в котором он находился раньше. Ресурс, в очередь которого включается процесс, будем называть в данном контексте *основным*.

Выбор основного ресурса при их совместном выделении следует производить, исходя из логики исполнения моделируемого процесса. Так, в данном случае в качестве основного ресурса выбран буксир, чтобы обеспечить автоматическое его предоставление тем или иным танкерам в зависимости от их приоритета (вспомним, что причаливающие танкеры бользуются б***о***льшим приоритетом в предоставлении буксира по сравнению с отчаливающими). В ряде же случаев выбор основного ресурса может быть произвольным.

Еще одно следствие неравноправия ресурсов при совместном выделении состоит в том, что процесс, ожидающий ресурса, активируется только при освобождении основного ресурса. Если же освобождается ресурс, не являющийся основным, ожидающий процесс не может об этом узнать, поскольку ожидает в очереди основного ресурса. В рассмотренном примере это не представляло проблемы, поскольку ресурсы причала и буксира всегда освобождаются одновременно. Однако в других моделях возможны и иные ситуации, поэтому рассмотрим один из возможных подходов к разрешению указаннйо проблемы.

Пусть, например, процесс-компонент одновременно выделяет два ресурса: resA (основной) и resB (неосновной). Тогда их выделение выполняется при помощи следующего вызова (метод GetAAndB, как и в рассмотренном примере модели порта, выполняет совместное выделение обоих ресурсов):

GetResource(resA, GetAAndB);

Теперь, если по завершении какого-то этапа работы требуется освободить единицу ресурса resB (неосновного), то следует уведомить об этом другие компоненты, ожидающие в очереди ресурса resA (основного). Это можно сделать, например, выполнив фиктивное освобождение ресурса resA объемом 0 единиц:

*// Действительное освобождение 1 единицы ресурса resB*

ReleaseResource(resB);

*// Фиктивное освобождение ресурса resA*

ReleaseResource(resA, 0);

Если в очереди к ресурсу resB нет никаких процессов, то порядок этих вызовов безразличен. Однако если в очереди к этому ресурсу также могут находить процессы (скорее всего, другого класса), от порядка вызова зависит их взаимный приоритет: процессы в очереди, указанной первой, возобновляются раньше, чем во второй, а, следовательно, имеют более высокий приоритет.

Еще один возможный вариант состоит в создании специального вспомогательного процесса, который становится в очередь к неосновному ресурсу и работа которого состоит только в активации связанного с ним процесса, находящегося в очереди к основному ресурсу. Этот процесс можно назвать *активатором*. Не будем рассматривать построение и использование такого процесса-активатора подробнее, поскольку в наших примерах необходимости в нем возникать не будет. Заметим лишь, что использование активаторов позволит обеспечить практически любую логику взаимодействия процессов и ресурсов ценой создания нового типа процессов, а, следовательно, усложнения структуры программы.

Вообще говоря, желательно избегать ситуации, когда разные процессы пользуются совместно выделяемыми ресурсами по-разному, поскольку это может привести к запутанной логике их взаимодействия и ощутимо затруднить построение работоспособной программы. И вообще, любой случай совместного выделения ресурсов следует продумывать очень тщательно и многократно проверять работоспособность программы при разных вариантах исполнения.

Полный текст модели с совместным выделением ресурсов причала и буксира находится в папке 4.3 Port 2.

При построении визуализированной версии программы желательно отображать текущее состояние буксира: выполняет ли он причаливание или отчаливание или свободен, — а также наличие шторма. Что касается состояния буксира, то прямо получить информацию о нем из имеющихся объектов имитации затруднительно (хотя и возможно). Факт наличия шторма обнаруживается достаточно просто: по текущей мощности ресурса буксира.

Для регистрации состояния буксира добавим в программу тип-перечисление, отражающий его возможные состояния, а в класс имитации — переменную, в которой фиксируется текущее состояние буксира:

**type**

*// Состояние буксира*

TTugState = (tsFree, tsArrive, tsDepart);

TPort = **class**(TSimulation)

**public**

…

TugState : TTugState;

…

**end**;

Алгоритм танкера должен менять эти состояния в процессе своей работы (указаны только действия по изменению состояния и имеющиеся строки программы, после которых они должны быть размещены):

…

GetResource(par.Tug, GetBerthAndTug);

par.TugState := tsArrive;

…

ReleaseResource(par.Tug);

par.TugState := tsFree;

…

GetResource(par.Tug, 1, 1);

par.TugState := tsDepart;

…

ReleaseResource(par.Berth);

ReleaseResource(par.Tug);

par.TugState := tsFree;

…

Еще одно удобное дополнение — при показе количества загружающихся танкеров (оно берется как текущее количество занятых единиц ресурса причала) отдельно указывать танкеры, которые уже загружены, но ожидают буксира для отчаливания (это количество можно узнать как длину очереди на отчаливание в ресурсе буксира). Поскольку занятость ресурса причала включает как еще не загруженные танкеры, так и те, которые уже загружены и ожидают отправки, количество загружаемых танкеров получается как разность указанных двух величин. Отображение всех этих значений производится в процедуре визуального отображения следующим образом:

*// Отобразить танкеры, находящиеся у причала,*

*// отдельно еще не загруженные и загруженные, но ожидающие отчаливания*

lbLoading.Caption := Chars(prt.Berth.Busy - prt.Tug.Queue[1].Size, '\*') +

Chars(prt.Tug.Queue[1].Size, 'o');

*// Отобразить состояние буксира*

**case** prt.TugState **of**

tsFree :

lbTug.Caption := '';

tsArrive :

lbTug.Caption := '===>';

tsDepart :

lbTug.Caption := '<===';

**end**;

*// Отобразить состояние шторма*

**if** prt.Tug.Capacity > 0 **then**

lbStorm.Caption := '\_\_\_\_\_\_'

**else**

lbStorm.Caption := '^^^^^';

Отображение остальных данных о текущем состоянии имитации не содержит ничего нового по сравнению с предыдущими примерами. Полный текст программы с визуализацией находится в папке 4.3 Port Visual.

## 4.4. Модель конвейерной линии

*Постановка задачи*. Вдоль конвейерной ленты расположены пять одинаковых обрабатывающих устройств. Обрабатываемые детали поступают равномерно со скоростью 4 детали в минуту непосредственно к первому из них. Если в момент поступления детали это устройство свободно, оно снимает деталь с конвейера и обрабатывает ее в течение времени, распределенного экспоненциально с математическим ожиданием, равным 1 минуте. Если устройство занято, деталь двигается дальше по конвейеру. Если, когда деталь приближается к свободному устройству, оно снимает ее с конвейера и обрабатывает ее, как описано выше. Время, в течение которого деталь перемещается от предыдущего устройства к следующиему, равна 1 минуте. Если все устройства до пятого включительно заняты, деталь делает полный круг по конвейеру в течение еще 5 минут и снова поступает к первому.

Требуется смоделировать работу конвейера в течение суток (1440 минут) и собрать следующую статистику: по загруженности каждого из обрабатывающих устройств, по времени нахождения деталей в системе, по общей занятости обрабатывающих устройств и по количеству деталей, находящихся на конвейере.

*Решение*. Будем моделировать обрабатывающие устройства посредством ресурсов, поэтому потребуются только два класса процессов: собственно деталь и генератор деталей. Поскольку при приближении к очередному обрабатывающему устройству требуется узнать занятость именно его, а не всех устройств в целом, создадим массив ресурсов, представляющих каждое устройство отдельно. Так как по условию задачи требуется собрать статистику не только по каждому устройству отдельно, но еще и по всей совокупности устройств, создадим дополнительный объект сбора статистики по действиям, в котором будет фиксироваться начало и конец обработки на любом устройстве. Для сбора статистики по количеству деталей на конвейере удобно также воспользоваться объектом сбора статистики по действиям. В этом случае началом действия считается постановка детали на конвейер (то есть, ее прибытие в систему), а концом — снятие детали с коныейера некоторым обслуживающим устройством. Для сбора данных по времени пребывания деталей в системе следует воспользоваться объектом точечной статистики. Данные в него помещаются при каждом завершении обслуживания детали.

Таким образом, структура класса имитации выглядит следующим образом:

TConveyor = **class**(TSimulation)

**public**

*// Генератор деталей*

Generator : TDetailGenerator;

*// Массив ресурсов, представляющих обрабатывающие устройства*

MachRes : **array** **of** TResource;

*// Статистика по занятости обрабатывающих устройств*

ActStat : TActionStatistics;

*// Статистика по времени пребывания деталей в системе*

TimeStat : TStatistics;

*// Статистика по занятости конвейера*

ConvStat : TActionStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

Что касается классов процессов генератора и детали, они не содержат ничего, кроме переопределенного метода RunProcess. Алгоритмы работы генератора и имитации не содержат ничего нового и в комментариях не нуждаются:

**procedure** TDetailGenerator.RunProcess;

**begin**

**while** True **do**

**begin**

ClearFinished;

*// Создат новую деталь*

TDetail.Create.ActivateDelay(0);

*// Ожидать прибытия следующей*

Hold(DetailInterval);

**end**;

**end**;

**procedure** TConveyor.RunSimulation;

**begin**

*// Начать поступление деталей*

Generator.ActivateDelay(0);

*// Ждать конца имитации*

Hold(SimulationTime);

StopStat;

**end**;

Что касается алгоритма процесса детали, главная сложность в нем состоит в процессе выбора свободного обслуживающего устройства. Поскольку, возможно, детали предстоит пройти мимо нескольких обслуживающих устройств, эта часть алгоритма должна представлять собой цикл. Процесс не должен непосредственно получать 1 единицу ресурса устройства, поскольку в случае его занятости это будет означать постановку в очередь ожидания. Вместо этого процесс должен запросить количество доступных единиц ресурса и в случае, если оно больше нуля, получить 1 единицу. Если же количество доступных единиц равно 0, процесс должен имитировать перемещение детали к следующему устройству: увеличить номер устройства на 1, а если он превысит максимально возможное значение, сделать его равным 0, и выдержать паузу равную 1 или 5 минутам в зависимости от того, на каком фрагменте конвейера находится деталь. Кроме того, в процессе детали нужно зафиксировать две статистики действия: по пребыванию детали на конвейере и по загруженности обрабатывающих устройств, — а также статистику по времени пребывания детали в системе. В результате этого обсуждения получается следующий алгоритм раобты процесса детали:

**procedure** TDetail.RunProcess;

**var**

par : TConveyor;

i : Integer;

**begin**

par := Parent **as** TConveyor;

*// Деталь встала на конвейер*

par.ConvStat.Start(SimTime);

i := 0;

*// Пока очередное устройство занято*

**while** par.MachRes[i].Available = 0 **do**

**begin**

*// Двигаться к следующему*

Inc(i);

*// Если все устройства пройдены, идти к первому*

**if** i = MachineCount **then**

i := 0;

*// Пройти участок конвейера*

**if** i > 0 **then**

Hold(StepTime)

**else**

Hold(BackStepTime);

**end**;

*// Деталь сошла с конвейера*

par.ConvStat.Finish(SimTime);

*// Занять устройство*

GetResource(par.MachRes[i]);

*// Выполнить действие*

par.ActStat.Start(SimTime);

Hold(rndDetail.Exponential(1));

par.ActStat.Finish(SimTime);

*// Освободить устройство*

ReleaseResource(par.MachRes[i]);

*// Зафиксировать статистику по времени*

par.TimeStat.AddData(SimTime - StartingTime);

Finish;

**end**;

При одном из прогонов модели получились такие результаты:

Machine 0 statistics:

Current capacity = 1

Utilization = 0.882 +- 0.322

Max utilization = 1, current = 1

Current available = 0, average = 0.118

Min available = 0, max = 1

Machine 1 statistics:

Current capacity = 1

Utilization = 0.856 +- 0.351

Max utilization = 1, current = 1

Current available = 0, average = 0.144

Min available = 0, max = 1

Machine 2 statistics:

Current capacity = 1

Utilization = 0.811 +- 0.392

Max utilization = 1, current = 1

Current available = 0, average = 0.189

Min available = 0, max = 1

Machine 3 statistics:

Current capacity = 1

Utilization = 0.753 +- 0.431

Max utilization = 1, current = 1

Current available = 0, average = 0.247

Min available = 0, max = 1

Machine 4 statistics:

Current capacity = 1

Utilization = 0.686 +- 0.464

Max utilization = 1, current = 1

Current available = 0, average = 0.314

Min available = 0, max = 1

Time in system:

Average = 8.939 +- 10.173

Min = 0.002, max = 73.471

Total = 5728 values

Activity statistics:

Average = 3.988 +- 0.929

Max = 5

Current running = 5, completed = 5728

Details on conveyor statistics:

Average = 31.709 +- 9.349

Max = 58

Current running = 27, completed = 5733

Calendar statistics:

Average length = 37.697 +- 9.504

Max = 65, current = 34

Average waiting time = 1.115

Эти результаты показывают неравномерную загруженность обслуживающих устройств: она уменьшается от первого устройства к последнему. Это вполне ожидаемый результат, так как детали поступают сначала к первому устройству, и вероятность того, что именно оно примет деталь на обслуживание, выше, чем для последующих. Средняя загруженность устройств примерно равна 4, что также вполне естественно. Действительно, за 1 минуту на конвейер поступают 4 детали, в то время как в среднем все устройства могут обслужить 5 деталей.

Результаты также показывают значительное количество одновременно присутствующих в системе процессов: максимальная длина календаря равна 65. Это означает значительные по сравнению с предыдущими примерами затраты оперативной памяти. Последний показатель, а также общее количество обработанных компонентов (свыше 5 тысяч), наводят на предположение о более значительном времени, которое требуется на исполнение программы. Правда, однократный прогон программы с заданными исходными данными по-прежнему выполняется очень быстро, практически мгновенно. Однако эксперименты, проведенные в экстремальной ситуации, когда интервал между моментами поступления деталей равен 0,2 минуты, то есть, когда обрабатывающие устройства уже не справляются с нагрузкой, показывают следующие результаты. Количество деталей на конвейере достигает примерно 700, что приводит к выделению памяти для программы порядка 12 Мбайт, а время выполнения на компьютере с процессором Athlon 64 X2 Dual Core 5200+ превысило 10 секунд.

Полный текст программы находится в папке 4.4 Conveyor.

При построении визуализации добавим отображение информации по количеству деталей на каждом участке конвейера, а также количество деталей, обработанное каждым устройством в отдельности. Это потребует, во-первых, размещения дополнительных полей в классе имитации. Во-вторых, алгоритм работы процесса детали должен будет изменять эти значения в соответствующие моменты:

*// Дополнения к классу имитации*

…

*// Количество деталей на каждом фрагменте конвейера*

ConvCount : **array** **of** Integer;

*// Количество деталей, обработанных каждым устройством*

Completed : **array** **of** Integer;

*// Дополнения к алгоритму процесса детали*

…

**while** par.MachRes[i].Available = 0 **do**

**begin**

…

*// Деталь на следующем участке конвейера*

Inc(par.ConvCount[i]);

…

*// Участок пройден*

Dec(par.ConvCount[i]);

**end**;

…

*// Обработка детали завершена*

Inc(par.Completed[i]);

Еще одно дополнение — программа сначала выполняет предварительную имитацию в течение 720 минут для приведения системы в некоторое рабочее положение, а уже потом, очистив статистику, запускает основную имитацию. Это дополнение здесь подробно обсуждать не будем, так как эта тема уже затрагивалась ранее. Полный текст программы с визуализацией находится в папке 4.4 Conveyor Visual.

## 4.5. Перехват ресурса

При работе с ресурсами возможна такая ситуация как *перехват* ресурса. Если реусрс занят каким-то процессом, то другой процесс может, не дожидаясь освобождения ресурса, забрать его у текущего обладателя ресурса для выполнения более срочных действий. При этом текущее действие с использованием ресурса может быть либо *прервано*, либо *приостановлено*. В первом случае процесс, ранее обладавший ресурсом, возобновляет свое исполнение, но действие считается незавершенным. Во втором случае процесс помещается в начало той же очереди, в которой он находился до получения ресурса. После того как процесс, перехвативший ресурс, освобождает его, процесс, ранее обладавший ресурсом, снова получает ресурс в свое распряжение и заканчивает исполнение прерванного действия, то есть, ожидает в течение времени, которого не хватило до завершения действия в момент перехвата. Процесс, действие которого было приостановлено, работает так, как будто никакого перехвата ресурса не было. Единственный для него способ определить, был ли произведен перехват — сравнить фактическое время окончания действия с ранее запланированным. Напротив, если при перехвате действие прерывается, процесс может это определить, причем он будет знать, какой промежуток времени, отведенный для действия, не был использован.

Процесс, получивший ресурс в результате перехвата, сам может лишиться его в результате перехвата еще более приоритетным процессом. При этом он гарантированно снова получит ресурс раньше, чем тот процесс, у которого он перехватил ресурс первоначально (разумеется, если в обоих случаях происходит приостановка текущего действия, а не прерывание).

Заметим, что перехват ресурса возможен не в любой ситуации. Точнее говоря, перехват возможен только при одновременном выполнении следующих условий:

1. Мощность ресурса должна быть единичной на момент попытки перехвата и последующего освобождения перехваченного ресурса. Лучше всего, если мощность ресурса будет единичной в течение всего времени его существования.
2. В момент попытки перехвата ресурса он должен быть выделен некоторому процессу, причем этот процесс в этот момент должен выполнять некоторое самостоятельное действие. Другими словами, требуется, чтобы в календаре событий имелась некоторое уведомление для процесса, обладающего ресурсом.
3. Процесс, пытающийся перехватить ресурс, должен обладать более высоким приоритетом. Приоритет определяется по двум критериям. Первый — индекс очереди, в который планируется постановка процесса при получении ресурса: процесс, встающий в очередь с меньшим индексом, обладает более высоким приоритетом. Второй критерий применяется, если текущий обладатель ресурса при получении ресурса находился в той же очереди, в котороую планируется постановка перехватывающего процесса. Это функция сравнения. Если она задана, то с ее помощью сравнивается приоритет двух процессов и, если перехватывающий процесс обладает более высоким приоритетом, перехват становится возможен. Если функция не задана, то сравнение не выполняется, и перехват невозможен.

Если любое из указанных условий не выполняется, то перехват ресурса невозможен, и перехватывающий процесс обычным образом встает в очередь ожидания ресурса. Это, кстати, не гарантирует, что этот процесс впоследствии получит ресурс в соответствии с критериями его приоритета по отношению к процессам, уже ожидающим ресурса.

Рассмотрим, например, ситуацию, когда ресурс занят некоторым процессом A, в очереди ожидания находится процесс B, и его пытается перехватить процесс C. Пусть функция критерия перехвата определена таким образом, что процесс C не является приоритетным по отношению к процессу A, зато является приоритетным по отношению к процессу B, и все процессы встают в одну очередь в порядке поступления. В этом случае процесс C не сможет перехватить ресурс у процесса A и встанет в очередь к ресурсу после процесса B, и, таким образом, получит ресурс после него. Если бы процесс C попытался выполнить перехват ресурса, когда им обладал процесс B, тогда перехват произошел бы успешно. Чтобы устранить такую нелогичность поведения, следует согласовать определения функции критерия перехвата ресурса и функции сравнения очереди таким образом, чтобы они давали однотипное поведение программы во всех случаях.

Для организации возможности перехвата в объекте ресурса имеются следующие поля и методы:

**public**

CurrentProc : TProcess;

CurrentIndex : Integer;

**constructor** Create(InitCap, InitBusy, QueCnt : Integer;

StartTime : Double; PriorFunc : TCompareFunc); **overload**;

**constructor** Create(InitCap, InitBusy, QueCnt : Integer;

PriorFunc : TCompareFunc); **overload**;

**constructor** Create(InitCap : Integer; StartTime : Double;

PriorFunc : TCompareFunc); **overload**;

**constructor** Create(InitCap : Integer;

PriorFunc : TCompareFunc); **overload**;

**constructor** Create(PriorFunc : TCompareFunc); **overload**;

**function** PreemptedProcs(Index : Integer) : Integer; **overload**;

**function** PreemptedProcs : Integer; **overload**;

**property** Priority : TCompareFunc **read**;

Первое, что здесь указано — поля CurrentProc и CurrentIndex. В них записаны ссылка на последний процесс, получивший ресурс, и индекс очереди, в которой он его ожидал. Поскольку перехват возможен только для ресурса единичной мощности, эти поля содержат сведения о единственном процессе, обладающим данным ресурсом.

Следом указаны еще пять вариантов конструктора, в которых задается параметр-функция PriorFunc, определяющая критерий сравнения приоритета процессов при перехвате. Эта функция строится по тому же шаблону, что и ранее определенная функция сравнения для очереди, то есть, она имеет два параметра-процесса и возвращает логический результат. Первым параметром в эту функцию передается ссылка на процесс, пытающийся перехватить ресурс. Второй параметр — процесс, обладающий ресурсом в данный момент. Результат функции должен быть равен True, если первый параметр является более приоритетным, чем второй (то есть, перехват возможен), и False в противном случае. Напомним еще раз, что сравнение с помощью этой функции производится только тогда, когда перехватывающий процесс становится в ту же очередь, что и текущий. Свойство Priority озволяет получить ссылку на функцию сравнения приоритета.

Функция PreepmtedProcs возвращает количество приостановленных процессов, ожидающих в очереди с указанным индексом (или нулевым индексом, если параметр не указан). Заметим, что подсчитываются только приостановленные процессы, находящиеся в начале очереди. Как только при просмотре встречается процесс, еще не получивший ресурса, просмотр завершается. Это нормально, поскольку приостановленные процессы всегда помещаются в начало очереди. Если для очереди задана функция сравнения, она должна быть определена таким образом, чтобы новые процессы не могли встать в очередь раньше приостановленных, а то работа программы будет происходить не так, как ожидалось. Как отличить приостановленный при перехвате процесс от процесса, еще не получившего ресурса, будет сказано несколько ниже.

Теперь рассмотрим средства класса процесса, используемые при перехвате ресурсов:

**public**

TimeLeft : Double;

**procedure** PreemptResource(Res : TResource; Index : Integer); **overload**;

**procedure** PreemptResource(Res : TResource); **overload**;

**procedure** PreemptResourceNoWait(Res : TResource;

Index : Integer); **overload**;

**procedure** PreemptResourceNoWait(Res : TResource); **overload**;

Поле TimeLeft содержит промежуток времени, оставшийся до завершения текущего действия. При выполнении перехвата в это поле записывается значение, оставшееся до ранее запланированного момента окончания действия. Если в результате перехвата текущий процесс приостанавливается, это значение используется для нового планирования события окончания действия, а в это поле заносится нулевое значение. Если же при перехвате действие прерывается, то в этом поле будет находиться ненулевое значение. Именно по значению этого поля процесс после завершения действия может определить, было ли действие выпонено полностью. Если в нем находится ненулевое значение, ресурс был перехвачен у процесса, и освобождать его не следует.

Методы PreemptResource, собственно, и выполняют попытку перехвата ресурса с приостановкой текущего действия. Если индекс очереди не указан, он предполагается равным нулю. Количество получаемого ресурса не указывается, поскольку в случае перехвата оно может быть равно только 1. Методы PreemptResourceNoWait выполняют попытку перехвата ресурса с прерыванием текущего действия. Обе этих пары методов в случае невозможности перехвата ресурса обычным образом ставят текущий процесс в очередь ожидания ресурса, то есть, фактически, вызывают методы GetResource. Перехваченный ресурс освобождается обычным образом, при помощи метода ReleaseResource, который автоматически учитывает наличие приостановленных действий.

Если при выполнении перехватываемого действия собирается статистика, то может возникнуть потребность в том, чтобы учитывать промежутки времени, когда действие приостановлено. Для этого в классе TActionStatistics предусмотрены следующие методы:

**procedure** Preempt(NewTime : Double); **overload**;

**procedure** Preempt; **overload**;

**procedure** Resume(NewTime : Double); **overload**;

**procedure** Resume; **overload**;

Методы Preempt учитывают прерывание действия, а методы Resume — его возобновление. По существу, первые из них уменьшают на 1 количество выполняющихся действий, а вторые — увеличивают на 1. Таким образом, их действие очень похоже на методы Finish и Start соответственно. Различие между Finish и Preempt в том, Finish дополнительно учитывает количество завершенных действий, и в моменты вызова считает, что действие завершено. Методы Preempt такого подсчета не ведут, то есть, действие не считается завершенным. Разницы между методами Resume и Start нет никакой. Фактически, это просто синонимы одной и той же пары методов, введенные только для более ясного указания цели вызова.

В классе TServiceStatistics аналогичных средств не предусмотрено, поскольку он предназначен для сбора статистики по обслуживающим действиям, при выполнении которых перехват ресурса невозможен.

## 4.6. Модель работы станка с поломками

*Постановка задачи*. Задания поступают на станок в среднем один раз в час, причем интервалы между их поступлениями распределены экспоненциально. Задания выполняются в порядке их поступления. Перед выполнением каждого задания производится наладка станка, которая занимает время, равномерно распределенное в интервале от 0,2 до 0,5 часа. Само задание выполняется за время, распределенное нормально со средним значением 0,5 часа и среднеквадратичным отклонением 0,1 часа. После выполнения задания покидают систему.

Периодически станок подвергается поломкам. Когда происходит поломка, текущее задание (независимо от того, производится ли в данный момент наладка или непосредственно выполнение) снимается со станка и помещается в начало очереди заданий. Устранение неисправности состоит из трех этапов, каждый из которых требует времени, распределенного экспоненциально с математическим ожиданием 0,75 часа (таким образом, общее время устранения неисправности распределено в соответствии с законом Эрланга с указанными параметрами). После устранения неисправности прерванное задание возобновляется с того места, на котором оно было прервано. Интервал времени от устранения неисправности до появления следующей распределен нормально с математическим ожиданием 20 часов и среднеквадратичным отклонением 2 часа. Первая неисправность происходит в момент времени, распределенный по этому же закону.

Требуется проанализровать работу станка в течение 500 часов для получения информации по загрузке станка и времени выполнения задания (отдельно по этапам наладки и непосредственно исполнения), времени нахождения заданий в системе и загруженности ремонтной бригады.

### 4.6.1. Приостановка действия

*Решение*. Формулировка задачи явно соответствует описанному выше действию перехвата ресурса, причем при перехвате текущий процесс, обладающий ресурсом, приостанавливает свое действие. Следовательно, в модели необходимо предусмотреть наличие ресурса, соответствующего моделируемому станку. Чтобы обеспечить возможность перехвата, предусмотрим в ресурсе две очереди ожидания. При этом более приоритетная очередь с индексом 0 будет использоваться для перехвата ресурса при появлении поломки, а очередь с индексом 1 — для получения ресурса при обработке деталей.

Поломки будем имитировать при помощи отдельного процесса. Этот процесс, дождавшись очередной поломки, перехватывает ресурс станка, выполняет ремонт, а затем освобождает ресурс, давая возможность приостановленному процессу продолжить действие. Поскольку по действиям, выполняемым на станке, ведется учет статистики, при перехвате и освобождении ресурса следует предусмотреть учет этих обстоятельств в объекте сбора статистики. Здесь, правда, возникает одна сложность: в момент поломки станок может находиться в процессе наладки, исполнения основного действия или быть свободным. В первых двух случаях следует указать приостановку и возобновление работы в одном из двух соответствующих объектов сбора статистики, а в третьем случае этого не требуется. Чтобы сделать возможным проверку состояния станка и выбор одного из вариантов действий, предусмотрим хранение информации о состоянии станка с помощью переменной типа-перечисления:

**type**

*// Перечисление TMachineState обозначает состояния станка*

TMachineState = (msFree, msPreparing, msWorking, msRepaired);

В классе имитации предусмотрим следующие компоненты. Кроме традиционных компонентов-процессов, представляющих генератор компонентов (заданий) и управление поломками, содадим ресурс, представляющий моделируемый станок, и поле, отражающее текущее состояние станка. Также создадим ряд объектов сбора статистики: точечную статистику по времени пребывания в системе, а также статистику действия и точечную статистику по времени выполнения для каждой из операций наладки, исполнения задания и ремонта:

*// Класс TTools имитирует работу станка с поломками*

TTools = **class**(TSimulation)

**public**

*// Генератор заданий*

Generator : TDetailGenerator;

*// Управление поломками*

Faults : TFaults;

*// Ресурс станка*

Tool : TResource;

*// Состояние станка*

State : TMachineState;

*// Статистика по времени пребывания в системе*

TimeStat : TStatistics;

*// Статистика по действию наладки*

PrepStat : TActionStatistics;

PrepTimeStat : TStatistics;

*// Статистика по основному действию*

OperStat : TActionStatistics;

OperTimeStat : TStatistics;

*// Статистика по ремонту станка*

RepairStat : TActionStatistics;

RepairTimeStat : TStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

В классах процессов, участвующих в имитации, никаких дополнительных компонентов, помимо переопределения метода RunProcess, не потребуется. Поэтому займемся рассмотрением этих методов. Как обычно, они достаточно просты, и главная сложность состоит в том, чтобы не забыть собрать всю необходимую статистику. Алгоритм работы компонента-задания включает получение ресурса станка, выполнение наладки, затем основного действия, и, наконец, освобождение ресурса, чтобы следующее задание могло начать обработку. Дополнив соответствующие операторы действиями по сбору статистики и переключению состояния станка, получаем следующий алгоритм:

**procedure** TDetail.RunProcess;

**var**

par : TTools;

ActionStartTime : Double;

**begin**

par := Parent **as** TTools;

*// Получить доступ к станку*

GetResource(par.Tool, 1, DetailQueueIndex);

*// Состояние станка - наладка*

par.State := msPreparing;

*// Выполнить наладку*

par.PrepStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(rndDetail.Uniform(MinPrepTime, MaxPrepTime));

par.PrepStat.Finish(SimTime);

par.PrepTimeStat.AddData(SimTime - ActionStartTime);

*// Состояние станка - основное действие*

par.State := msWorking;

*// Выполнить основное действие*

par.OperStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(rndDetail.Normal(MeanMainTime, DeviationMainTime));

par.OperStat.Finish(SimTime);

par.OperTimeStat.AddData(SimTime - ActionStartTime);

*// Состояние станка - свободен*

par.State := msFree;

*// Освободить станок*

ReleaseResource(par.Tool);

*// Собрать статистику по времени пребывания в системе*

par.TimeStat.AddData(SimTime - StartingTime);

Finish;

**end**;

Процесс, управляющий поломками, во многом похож на процесс, имитирующий штормы в задаче из параграфа 4.3. Отличие состоит в том, что он не меняет мощность ресурса, а вместо этого перехватывает его. Кроме того, здесь следует предусмотреть операторы, выполняющие приостановку и возобновление при сборе статистики о действиях:

**procedure** TFaults.RunProcess;

**var**

par : TTools;

RepairStartTime : Double;

LastState : TMachineState;

**begin**

par := Parent **as** TTools;

**while** True **do**

**begin**

*// Ожидать следующей поломки*

Hold(rndFaults.Normal(MeanFaultInterval, DeviationFaultInterval));

*// Прервать текущее действие*

PreemptResource(par.Tool, FaultQueueIndex);

*// Приостановить сбор статистики действия*

**case** par.State **of**

msPreparing :

par.PrepStat.Preempt(SimTime);

msWorking :

par.OperStat.Preempt(SimTime);

**end**;

*// Запомнить предыдущее состояние станка и задать состояние ремонта*

LastState := par.State;

par.State := msRepaired;

*// Начать ремонт*

RepairStartTime := SimTime;

par.RepairStat.Start(SimTime);

Hold(rndFaults.Erlang(MeanRepairIntervalTime, RepairIntervalCount));

par.RepairStat.Finish(SimTime);

par.RepairTimeStat.AddData(SimTime - RepairStartTime);

*// Восстановить предыдущее состояние*

par.State := LastState;

*// Возобновить сбор статистики*

**case** par.State **of**

msPreparing :

par.PrepStat.Resume(SimTime);

msWorking :

par.OperStat.Resume(SimTime);

**end**;

*// Закончить ремонт*

ReleaseResource(par.Tool);

**end**;

**end**;

Алгоритмы работы генератора и процесс имитации не требуют дополнительных пояснений:

**procedure** TDetailGenerator.RunProcess;

**begin**

**while** True **do**

**begin**

ClearFinished;

*// Создать задание и запустить процесс*

TDetail.Create.ActivateDelay(0);

*// Подождать перед созданием следующего*

Hold(rndDetail.Exponential(MeanDetailInterval));

**end**;

**end**;

**procedure** TTools.RunSimulation;

**begin**

Generator.ActivateDelay(0);

Faults.ActivateDelay(0);

*// Ожидать окончания имитации*

Hold(SimulationTime);

StopStat;

**end**;

Полный текст программы находится в папке 4.6.1 Tools. Один из ее прогонов дал следующие результаты:

Time in system statistics:

Average = 5.205 +- 3.216

Min = 0.549, max = 15.037

Total = 500 values

Preparation time statistics:

Average = 0.389 +- 0.368

Min = 0.200, max = 4.557

Total = 501 values

Operation time statistics:

Average = 0.543 +- 0.357

Min = 0.196, max = 4.627

Total = 500 values

Repair time statistics:

Average = 2.267 +- 1.382

Min = 0.290, max = 5.110

Total = 22 values

Preparation statitics:

Average = 0.349 +- 0.477

Max = 1

Current running = 0, completed = 501

Operation statitics:

Average = 0.500 +- 0.500

Max = 1

Current running = 0, completed = 500

Repair statitics:

Average = 0.102 +- 0.302

Max = 1

Current running = 1, completed = 22

Tool resource statitics:

Current capacity = 1

Utilization = 0.950 +- 0.217

Max utilization = 1, current = 1

Current available = 0, average = 0.050

Min available = 0, max = 1

Repair queue statistics:

Average length = 0.000 +- 0.000

Max = 0, current = 0

Average waiting time = 0.000

Details queue statistics:

Average length = 4.364 +- 3.404

Max = 16, current = 2

Average waiting time = 4.184

Calendar statistics:

Average length = 3.849 +- 0.358

Max = 4, current = 3

Average waiting time = 0.313

Как видно из этих результатов, за время имитации было выполнено 500 заданий (последнее, 501-е, в момент завершения имитации находилось в состоянии приостановленного исполнения). За это время произошло 22 поломки, причем в момент завершения имитации происходил 23-й ремонт станка. Среднее время выполнения наладки и основного действия составило 0,389 и 0,543 часа соответственно, что несколько больше тех значений, которые должны быть согласно исходным данным (0,35 и 0,5 соответственно). Очевидно, это превышение связано с тем, что часть действий прерывалась поломкой станка. Это хорошо видно также по величине максимального времени выполнения этих действий: она на порядок превышает среднее значение в обоих случаях.

Результаты занятости станка показывают, что наладка выполнялась в течение 34,9 % времени его работы, основное действие — в течение 50 %, а ремонт — в течение 10,2 %. Таким образом, получается, что в течение 100 % – 34,9 % – 50 % – 10,2 % = 4,9 % времени станок свободен и не занят никаким действием. Этот же результат можно получить из статистики по ресурсу станка. Разница в 0,1 % объясняется погрешностями округления.

В результатах также содержатся данные по статистикам двух очередей к ресурсу станка: очередь на ремонт (Repair queue) и очередь на обработку (Details queue). Первая очередь показывает полностью нулевые результаты. Так, собственно, и должно быть: в эту очередь встает единственный процесс, который, являясь приоритетным, сразу же получает ресурс — либо посредством перехвата, который всегда успешен, либо когда ресурс свободен. Поэтому данная статистика не содержит никакой полезной информации и приведена здесь только для иллюстрации указанного обстоятельства.

При разработке визуализированной версии программы в постановку задачи не будем вносить никаких изменений. Однако здесь есть два момента, требующие пояснения. Первый — это отображение очереди ожидания с обозначением того, есть ли в ней задание, обработка которого была приостановлена в результате поломки станка. Таких заданий может быть не больше одного. Будем обозначать его другим символом, например, буквой **о**. Тогда очередь отображается строкой звездочек, длина которой на 1 меньше длины очереди, с добавлением к ней этой буквы. Вторая особенность касается отображения состояния станка. В зависимости от значения, записанного в поле State объекта имитации, будем отображать его с помощью различных строк. Описанные действия обеспечиваются следующими строками кода:

*// Отображение очереди с обозначением приостановленного задания*

**if** tls.Tool.PreemptedProcs(DetailQueueIndex) > 0 **then**

lbQueue.Caption := Chars(tls.Tool.Queue[DetailQueueIndex].Size - 1, '\*') + 'o'

**else**

lbQueue.Caption := Chars(tls.Tool.Queue[DetailQueueIndex].Size, '\*');

*// Отображение состояния станка*

**case** tls.State **of**

msFree :

lbState.Caption := '';

msPreparing:

lbState.Caption := '(.)';

msWorking :

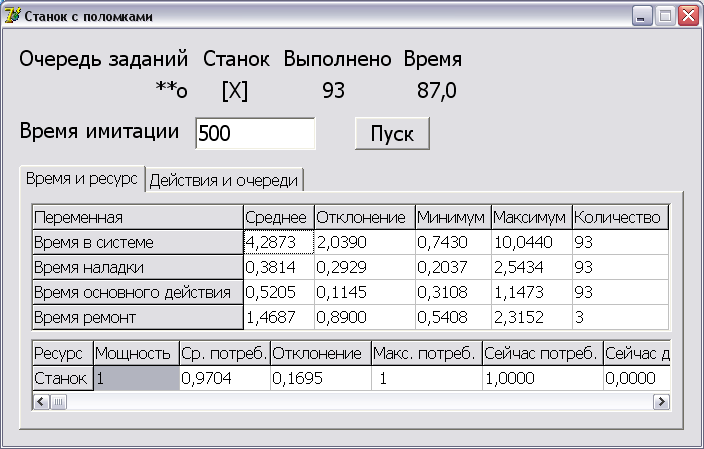
lbState.Caption := '(\*)';

msRepaired:

lbState.Caption := '[X]';

**end**;

Процесс исполнения модели приведен ниже:



Полный текст программы находится в папке 4.6.1 Tools Visual.

### 4.6.2. Прерывание действия

Чтобы проиллюстрировать перехват ресурса с прерыванием действия, внесем изменения в постановку задачи. Предположим, в случае поломки станка текущее задание переносится на дополнительный станок старой модели, все операции на котором занимают вдвое б***о***льшее время. Если поломка происходит во время выполнения наладки, задание снимается с основного станка и переносится на дополнительный станок, на котором заново выполняется наладка и основное действие с параметрами времени, вдвое превышающими параметры основного станка. Если поломка происходит во время выполнения оснвной обработки, задание переносится на дополнительный станок, на котором заново выполняется наладка на тех же условиях, а затем остаток основной обработки за время, вдвое б***о***льшее времени, не хватившего на обработку на основном станке. Задания, которые не были прерваны, ожидают окончания ремонта обычным образом. Естественно, дополнительно к имеющимся статистическим данным требуется собрать данные по обработке на дополнительном станке.

К имеющимся компонентам класса имитации следует добавить данные, относящиеся к дополнительному станку: ресурс дополнительного станка, его состояние (для консольной задачи эта информация не нужна, однако будет полезна для визуализации), объекты сбора статистики по загруженности и времени выполнения наладки и основного действия для дополнительного станка. Соответствующие элементы класса приведены ниже:

ExtraTool : TResource;

ExtraState : TMachineState;

ExtraPrepStat : TActionStatistics;

ExtraPrepTimeStat : TStatistics;

ExtraOperStat : TActionStatistics;

ExtraOperTimeStat : TStatistics;

Алгоритм процесса, управляющего поломками, отличается тем, что, во-первых, перехватывает ресурс при помощи метода PreemptResourceNoWait, а во-вторых, не восстанавливает предыдущее состояние станка, поскольку незавершенная обработка переносится на другой станок (приведена только часть текста метода):

*// Прервать текущее действие*

PreemptResourceNoWait(par.Tool, FaultQueueIndex);

*// Остановить сбор статистики действия*

**case** par.State **of**

msPreparing :

par.PrepStat.Preempt(SimTime);

msWorking :

par.OperStat.Preempt(SimTime);

**end**;

*// Задать состояние ремонта*

par.State := msRepaired;

*// Начать ремонт*

RepairStartTime := SimTime;

par.RepairStat.Start(SimTime);

Hold(rndFaults.Erlang(MeanRepairIntervalTime, RepairIntervalCount));

par.RepairStat.Finish(SimTime);

par.RepairTimeStat.AddData(SimTime - RepairStartTime);

*// Станок свободен*

par.State := msFree;

*// Закончить ремонт*

ReleaseResource(par.Tool);

Наиболее сложные действия выпадают на алгоритм, имитирующий процесс выполнения задания. Поскольку любое из двух действий может быть прервано, после окончания каждого из них следует проверить, завершено ли оно. Эт выполняется посредством проверки значения свойства TimeLeft. Как указывалось в предыдущем параграфе, нулевое значение этого свойства говорит о том, что оно было завершено. Если же его значение больше 0, то действие не было завершено, а значение свойства указывает, сколько времени осталось до его завершения.

Порядок действий в случае прерывания каждого из двух действий почти один и тот же: следует получить ресурс дополнительного станка, выполнить наладку, затем основное действие, и, наконец, освободить ресурс. Все эти действия необходимо сопровождать сбором статистики — точно так же, как это происходит для основного станка, но используя объекты сбора статистики, относящиеся к дополнительному станку. Разница между продолжением прерванной наладки и основного действия состоит в том, что в первом случае продолжительность основного действия вычисляется обычным образом (но умножается на 2), а во втором — равна удвоенной продолжительности незаконченного действия. В итоге получается довольно большой по объему, но простой по структуре текст алгоритма (приведена только часть алгоритма, которая отличается от предыдущего примера):

*// Если наладка прервана, перейти на дополнительный станок*

**if** TimeLeft > 0 **then**

**begin**

*// Работа начинается заново*

TimeLeft := 0;

*// Получить ресурс дополниельного станка*

GetResource(par.ExtraTool);

*// Начать наладку на дополнительном станке*

par.ExtraState := msPreparing;

par.ExtraPrepStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(rndDetail.Uniform(MinPrepTime, MaxPrepTime) \* 2);

*// Закончить наладку*

par.ExtraPrepStat.Finish(SimTime);

par.ExtraPrepTimeStat.AddData(SimTime - ActionStartTime);

*// Начать основное действие*

par.ExtraState := msWorking;

par.ExtraOperStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(rndDetail.Normal(MeanMainTime, DeviationMainTime) \* 2);

*// Закончить действие*

par.ExtraOperStat.Finish(SimTime);

par.ExtraOperTimeStat.AddData(SimTime - ActionStartTime);

par.ExtraState := msFree;

*// Освободить дополнительный станок*

ReleaseResource(par.ExtraTool);

**end**

**else**

**begin**

*// Закончить наладку на основном станке*

par.PrepStat.Finish(SimTime);

par.PrepTimeStat.AddData(SimTime - ActionStartTime);

*// Состояние станка - основное действие*

par.State := msWorking;

*// Выполнить основное действие*

par.OperStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(rndDetail.Normal(MeanMainTime, DeviationMainTime));

*// Если основное действие было прервано*

**if** TimeLeft > 0 **then**

**begin**

*// Запомнить оставшееся время выполнения основного действия*

OperTime := TimeLeft;

TimeLeft := 0;

*// Получить ресурс дополнтельного станка*

GetResource(par.ExtraTool);

*// Начать наладку*

par.ExtraState := msPreparing;

par.ExtraPrepStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(rndDetail.Uniform(MinPrepTime, MaxPrepTime) \* 2);

*// Закончить наладку*

par.ExtraPrepStat.Finish(SimTime);

par.ExtraPrepTimeStat.AddData(SimTime - ActionStartTime);

*// Начать основное действие*

par.ExtraState := msWorking;

par.ExtraOperStat.Start(SimTime);

ActionStartTime := SimTime;

Hold(OperTime \* 2);

*// Закончить основное действие*

par.ExtraOperStat.Finish(SimTime);

par.ExtraOperTimeStat.AddData(SimTime - ActionStartTime);

par.ExtraState := msFree;

*// Освободить ресурс*

ReleaseResource(par.ExtraTool);

**end**

**else**

**begin**

*// Закончить действие на основном станке*

par.OperStat.Finish(SimTime);

par.OperTimeStat.AddData(SimTime - ActionStartTime);

*// Состояние станка - свободен*

par.State := msFree;

*// Освободить станок*

ReleaseResource(par.Tool);

**end**;

**end**;

При одном из прогонов были получены следующие результаты:

Time in system statistics:

Average = 3.463 +- 2.311

Min = 0.516, max = 11.048

Total = 482 values

Preparation time statistics:

Average = 0.351 +- 0.086

Min = 0.200, max = 0.500

Total = 475 values

Operation time statistics:

Average = 0.499 +- 0.100

Min = 0.232, max = 0.803

Total = 464 values

Repair time statistics:

Average = 2.602 +- 1.803

Min = 0.448, max = 7.232

Total = 21 values

Extra preparation time statistics:

Average = 0.741 +- 0.179

Min = 0.473, max = 0.999

Total = 18 values

Extra operation time statistics:

Average = 0.782 +- 0.349

Min = 0.091, max = 1.420

Total = 18 values

Preparation statitics:

Average = 0.337 +- 0.473

Max = 1

Current running = 1, completed = 475

Operation statitics:

Average = 0.468 +- 0.499

Max = 1

Current running = 0, completed = 464

Extra preparation statitics:

Average = 0.027 +- 0.161

Max = 1

Current running = 0, completed = 18

Extra operation statitics:

Average = 0.028 +- 0.165

Max = 1

Current running = 0, completed = 18

Repair statitics:

Average = 0.109 +- 0.312

Max = 1

Current running = 0, completed = 21

Tool resource statitics:

Current capacity = 1

Utilization = 0.914 +- 0.280

Max utilization = 1, current = 1

Current available = 0, average = 0.086

Min available = 0, max = 1

Extra tool resource statitics:

Current capacity = 1

Utilization = 0.055 +- 0.228

Max utilization = 1, current = 0

Current available = 1, average = 0.945

Min available = 0, max = 1

Repair queue statistics:

Average length = 0.000 +- 0.000

Max = 0, current = 0

Average waiting time = 0.000

Details queue statistics:

Average length = 2.484 +- 2.423

Max = 10, current = 2

Average waiting time = 2.568

Extra tool queue statistics:

Average length = 0.000 +- 0.000

Max = 0, current = 0

Average waiting time = 0.000

Calendar statistics:

Average length = 3.860 +- 0.367

Max = 5, current = 4

Average waiting time = 0.401

Как видно из этих результатов, за время прогона было обработано 482 задания, из них 464 были обработаны на основном станке, а 18 — на дополнительном. Из последних 18 заданий 9 были прерваны на этапе наладки, и еще 9 — на этапе основного действия. Это видно из того обстоятельства, что основное действие на основном станке было завершено 464 раза, а действие наладки — 475 раз. Ремонт был произведен 21 раз, это значит, что 3 поломки произошли, когда станок простаивал.

Среднее время пребывания заданий в системе несколько сократилось, однако этот результат следует уточнить посредством многократных прогонов данной и предыдущей моделей. В то же время, как видно, дополнительный станок занят лишь в течение 5,5 % времени имитации. Это значит, что эффективность поддержания работоспособности дополнительного станка в целях ускорения исполнения заданий должна быть серьезно взвешена на основании данных, не указанных в постановке задачи.

Полный текст программы находится в папке 4.6.2 Tools Extra. Визуализированное приложение находится в папке 4.6.2 Tools Extra Visual. Поскольку в последнем не содержится каких-либо особенностей по сравнению с ранее разработанными приложениями, рассматривать его подробно не будем.

## 4.7. Затворы

*Затвор* — это объект, позволяющий управлять исполнением процессов. Затвор может находиться в двух состояниях: *открытом* и *закрытом*. Обращение процесса к затвору называется его *прохождением*. Если процесс пытается пройти через открытый затвор, это обращение немедленно успешно завершается, и процесс продолжает свою работу. Если же процесс пытается пройти через закрытый затвор, он встает в очередь ожидания и находится в ней до тех пор, пока какой-либо другой процесс не откроет затвор.

Работу затвора можно смоделировать при помощи ресурса единичной мощности. Чтобы закрыть его, следует получить единицу ресурса, а чтобы открыть — освободить ее. При прохождении затвора процесс запрашивает одну единицу ресурса, а потом сразу же освобождает ее. Если затвор закрыт, наличное количество ресурса равно нулю, и процесс будет приостановлен до тех пор, пока оно не увеличится до 1. Поскольку после получения ресурса процесс сразу же освобождает его, другие процессы, ожидаюшие в очереди, также смогут успешно пройти через затвор.

Однако некоторые средства ресурса при работе с затворами не имеют смысла. Так, перехват затвора не имеет смысла. Кроме того, поскольку при открытии затвора все ожидающие его открытия процессы возобновляются в один и тот же момент имитационного времени, говорить об их приоритете также нет смысла. Это значит, что в затворе достаточно иметь одну очередь ожидания. В некоторых тонких случаях, когда важен порядок возобновления ожидающих процессов, можно обеспечить его при помощи функции сравнения очереди.

По вышеперечисленным причинам в библиотеке предусмотрен более простой класс для реализации затворов, который называется TGate. Рассмотрим его структуру:

TGate = **class**

**public**

**constructor** Create(InitState : Boolean; StartTime : Double); **overload**;

**constructor** Create(InitState : Boolean); **overload**;

**constructor** Create; **overload**;

**destructor** Destroy; **override**;

**procedure** Open(NewTime : Double); **overload**;

**procedure** Open; **overload**;

**procedure** Close(NewTime : Double); **overload**;

**procedure** Close; **overload**;

**procedure** StopStat(NewTime : Double); **overload**;

**procedure** StopStat; **overload**;

**procedure** ClearStat(NewTime : Double); **overload**;

**procedure** ClearStat; **overload**;

**property** State : Boolean **read**;

**property** Stat : TIntervalStatistics **read**;

**property** Queue : TList **read**;

**end**;

Свойство State позволяет узнать состояние затвора: она равно False, если затвор закрыт, и True, если он открыт. Управлять состоянием затвора можно с помощью методов Open и Close, в которых либо указывается момент имитационного времени, либо используется текущее время в момент вызова метода. Методы ClearStat и StopStat, как обычно, позволяют очистить собранную статистику и скорректировать ее к текущему имитационному времени. Свойство Stat дает ссылку на объект сбора статистики по состоянию затвора. Открытое состояние в этой статистике фиксируется как числовое значение 1, закрытое — как 0. Таким образом, среднее значение этой статистики соответствует доле времени, в течение которого затвор был открыт. Свойство Queue дает ссылку на очередь ожидания затвора.

Конструкторы Create позволяют указать исходное состояние затвора (в соответствии со смыслом свойства State) и время его создания. Как обычно, если не указано время создания, используется текущее имитационное время. Если не указано исходное состояние затвора, оно равно False (то есть, затвор закрыт).

Для работы с затворами в классе TProcess предусмотрены следующие методы:

**procedure** WaitGate(Gate : TGate);

**procedure** CloseGate(Gate : TGate);

**procedure** OpenGate(Gate : TGate);

Метод WaitGate выполняет попытку прохождения затвора. Если она неудачна, процесс встает в очередь ожидания затвора. Методы CloseGate и OpenGate меняют состояние затвора, закрывая и открывая его соответственно.

Для отображения статистики затвора используются следующие две процедуры, предназначенные для работы в консольных и оконных приложения соответственно:

**procedure** WriteStat(Header : **string**; Gate : TGate); **overload**;

**procedure** ShowStat(Grid : TStringGrid; **const** Headers : **array** **of** **string**;

**const** Gate : **array** **of** TGate); **overload**;

Использование этих процедур абсолютно аналогично одноименным процедурам для других объектов сбора статистики. Они отображают для затвора только два параметра: среднюю загруженность затвора (то есть, долю времени, когда он находился в открытом состоянии) и текущее его состояние. Последнее обозначается словами OPEN или CLOSED в консольном приложении, и ОТКРЫТ или ЗАКРЫТ в оконном.

Практически работу с затворами рассмотрим на примере в следующем параграфе.

## 4.8. Модель участка с ограниченным движением

*Постановка задачи*. На участке двухполосной дороги с двусторонним движением на одной из полос производится ремонт, в результате чего движение производится только по одной полосе. Чтобы обеспечить возможность проезда, с двух сторон от участка установлены светофоры, которые попеременно открывают движение в одну и другую стороны (для определенности будем считать, что это направления слева направо и справа налево). Промежуток времени между закрытием одного светофора и открытием другого составляет 55 секунд, чтобы автомобили, въехавшие на этот участок, успели с него выехать.

Автомобили прибывают к участку с двух сторон с интервалом времени между ними, распределенным экспоненциально. Математическое ожидание интервала для автомобилей, прибывающих слева, равно 9 секундам, справа — 12 секундам. Длительность горения разрешающего сигнала светофора для автомобилей, прибывающих слева, равна 60 секундам, справа — 45 секундам. Минимальный промежуток времени между автомобилями, проезжающими мимо открытого светофора, равен 2 секундам.

Требуется провести имитацию в течение 1 часа (3600 секунд) и собрать статистику по времени ожидания автомобилей в очередях перед светофорами.

*Решение*. Сначала отметим очевидные обстоятельства. Во-первых, автомобили, проезжающие по изучаемому участку, будут представлены процессами-компонентами. Во-вторых, для их порождения потребуются два процесса-генератора, каждый для своего направления. Различаются эти генераторы средним временем задержки между моментами создания процессов-компонентов. Следовательно, представить их можно двумя объектами одного и того же класса, в котором содержится поле, задающее это время.

Моделировать светофоры, управляющие прохождением автомобилей, будем при помощи затворов, по одному для каждого направления движения. Открытое состояние затвора соответствует зеленому сигналу светофора, закрытое — красному. Таким образом, процесс-автомобиль должен предпринимать попытку прохождения затвора-светофора. Если затвор открыт, автомобиль сразу же проедет светофор и закончит работу. Если затвор открыт, автомобиль вынужден будет дождаться зеленого сигнала. Управлять состоянием затворов-светофоров будет отдельный процесс, который с заданной периодичностью открывает и закрывает их.

Однако затворов еще недостаточно, поскольку в условии указано, что интервал проследования автомобилей мимо светофора должен быть не менее 2 секунд. Обеспечить выполнение этого требования можно при помощи ресурсов (по одному для каждого направления движения). Автомобиль, проезжающий мимо светофора, получает единицу ресурса, удерживает ее в течение 2 секунд и освобождает, тем самым разрешая следующему автомобилю проследование светофора. Таким образом, перед прохождением светофора каждый автомобиль обращается к затвору и ресурсу своего направления. То есть, автомобили, ожидающие прохождения мимо светофора, будут находиться в двух очередях для каждого направления.

Последовательность обращений к затвору и ресурсу имеет существенное значение. Если сначала процесс будет проходить затвор, а потом получать ресурс, то все автомобили, ожидавшие в очереди к затвору, немедленно встанут в очередь к ресурсу и постепенно проследуют мимо светофора независимо от того, сколько автомобилей реально успевает проехать за то время, пока горит зеленый сигнал светофора. Следовательно, такой порядок обращения делает работу модели некорректной.

Если же процесс автомобиля сначала получает единицу ресурса, а только потом обращается к затвору, то при закрытом светофоре складывается следующая ситуация. Первый процесс успешно получает ресурс и встает в очередь ожидания к затвору. Последующие прибывающие автомобили не могут получить единицу ресурса и встают в очередь к нему. Как только светофор открывается, первый автомобиль проезжает мимо него, и через 2 секунды освобождет единицу ресурса. Это позволяет начать дивжение следующему автомобилю, который получается единицу ресурса, проходит через открытый затвор и через 2 секунды освобождает ресурс, позволяя двигаться следующему за ним автомобилю. Так происходит до тех пор, пока очередь к ресурсу не исчерпается либо затвор не закроется. Если затвор закроется раньше, чем все ожидавшие автомобили проследуют мимо светофора. Автомобиль, получивший ресурс, будет задержан в очереди затвора и будет ожидать его открытия.

При таком варианте организации алгоритма в очереди к затвору оказывается не более одного автомобиля, а все остальные ожидают открытия светофора в очереди к ресурсу. Это никоим образом не противоречит постановке задачи, но потребует соответствующей интерпретации результатов имитации. В частности, средняя длина очереди ожидания автомобилей равна сумме средних длин очередей ресурса и затвора. Однако статистику по времени ожидания перед светофорами следует собирать отдельно с помощью специально предназначенных для этого объектов.

Поскольку объекты сбора статистики, ресурсы и затворы, а также процессы-генераторы должны быть определены парами, следует предумотреть величину, позволяющую различать два направления движения, а также использовать ее в качестве индекса при обращении к соответствующим массивам объектов. Удобно в этом качестве использовать тип-перечисление:

**type**

*// Направление движения*

TDirection = (dirLeftRight, dirRightLeft);

Процесс-автомобиль должен знать направление своего движения, чтобы обращаться к надлежащим объектам ресурса, затвора и сбора статистики. Процесс-генератор также должен иметь информацию о направлении движения, чтобы при создании автомобилей задавать им направление. Кроме того, каждый из двух генераторов порождает автомобили с разным средним интервалом. Для реализации этих возможностей в классах процессов следует предусмотреть соответствующие поля и конструктор, задающий их начальные значения:

*// Класс TGenerator – генератор автомобилей*

TGenerator = **class**(TProcess)

**public**

**constructor** Create(Dir : TDirection; Mean : Double);

**protected**

*// Направление движения автомобилей*

Direction : TDirection;

*// Средний интервал движения*

MeanInterval : Double;

**procedure** RunProcess; **override**;

**end**;

*// Класс TCar - автомобиль*

TCar = **class**(TProcess)

**public**

**constructor** Create(Dir : TDirection);

**protected**

*// Направление движения*

Direction : TDirection;

**procedure** RunProcess; **override**;

**end**;

Теперь структура класса проекта определяется достаточно просто:

*// Класс TTraffic – имитация одностороннего участка дороги*

TTraffic = **class**(TSimulation)

**public**

*// Генераторы автомобилей*

Gen : **array** [TDirection] **of** TGenerator;

*// Процесс, управляющий светофорами*

Lights : TLights;

*// Ресурсы, управляющие прохождением автомобилей*

LightRes : **array** [TDirection] **of** TResource;

*// Затворы, обозначающие светофоры*

LightGate : **array** [TDirection] **of** TGate;

*// Статистика по времени ожидания перед светофорами*

WaitStat : **array** [TDirection] **of** TStatistics;

**destructor** Destroy; **override**;

**procedure** StopStat; **override**;

**protected**

**procedure** Init; **override**;

**procedure** RunSimulation; **override**;

**end**;

Алгоритмы описываемых процессов также устроены чрезвычайно просто. Поскольку работа алгоритма автомобиля была описана выше, а остальные алгоритмы построены по типовым правилам, приводим их здесь без дополнительных пояснений:

**procedure** TGenerator.RunProcess;

**begin**

**while** True **do**

**begin**

ClearFinished;

*// Создать автомобиль*

TCar.Create(Direction).ActivateDelay(0);

*// Дождаться создания следующего*

Hold(rndCar.Exponential(MeanInterval));

**end**;

**end**;

**procedure** TCar.RunProcess;

**var**

par : TTraffic;

**begin**

par := Parent **as** TTraffic;

*// Ожидать возможности проезда*

GetResource(par.LightRes[Direction]);

*// Ожидать разрешающего сигнала светофора*

WaitGate(par.LightGate[Direction]);

*// Зафиксировать статистику по времени ожидания*

par.WaitStat[Direction].AddData(SimTime - StartingTime);

*// Проехать мимо светофора*

Hold(CarPauseTime);

*// Разрешить проезд следующего автомобиля*

ReleaseResource(par.LightRes[Direction]);

Finish;

**end**;

**procedure** TLights.RunProcess;

**var**

par : TTraffic;

**begin**

par := Parent **as** TTraffic;

**while** True **do**

**begin**

*// Ожидать завершения проезда автомобилей*

Hold(RedLightTime);

*// Открыть светофор для движения слева направо*

OpenGate(par.LightGate[dirLeftRight]);

*// Выждать время*

Hold(OpenTime[dirLeftRight]);

*// Закрыть светофор*

CloseGate(par.LightGate[dirLeftRight]);

*// То же для направления справа налево*

Hold(RedLightTime);

OpenGate(par.LightGate[dirRightLeft]);

Hold(OpenTime[dirRightLeft]);

CloseGate(par.LightGate[dirRightLeft]);

**end**;

**end**;

**procedure** TTraffic.RunSimulation;

**begin**

Gen[dirLeftRight].ActivateDelay(0);

Gen[dirRightLeft].ActivateDelay(0);

Lights.ActivateDelay(0);

*// Дождаться окончания имитации*

Hold(SimulationTime);

StopStat;

**end**;

При одном из прогонов программы получился следующий результат:

Wait time left to right statistics:

Average = 76.499 +- 49.297

Min = 0.000, max = 199.213

Total = 391 values

Wait time right to left statistics:

Average = 76.790 +- 53.838

Min = 0.000, max = 170.000

Total = 262 values

Gate left to right statistics:

Current state = CLOSED, open percentage = 0.283

Gate right to left statistics:

Current state = CLOSED, open percentage = 0.200

Resource left to right statistics:

Current capacity = 1

Utilization = 0.903 +- 0.296

Max utilization = 1, current = 1

Current available = 0, average = 0.097

Min available = 0, max = 1

Resource right to left statistics:

Current capacity = 1

Utilization = 0.890 +- 0.312

Max utilization = 1, current = 1

Current available = 0, average = 0.110

Min available = 0, max = 1

Gate left to right queue statistics:

Average length = 0.686 +- 0.464

Max = 1, current = 1

Average waiting time = 6.204

Gate right to left queue statistics:

Average length = 0.745 +- 0.436

Max = 1, current = 1

Average waiting time = 9.658

Resource left to right queue statistics:

Average length = 7.691 +- 5.864

Max = 25, current = 8

Average waiting time = 70.120

Resource right to left queue statistics:

Average length = 5.171 +- 4.019

Max = 17, current = 14

Average waiting time = 66.877

Calendar statistics:

Average length = 4.363 +- 0.481

Max = 5, current = 4

Average waiting time = 1.730

Полный текст программы находится в папке 4.8 Traffic.

На основании построенной модели можно провести анализ того, какие интервалы открытого состояния обоих светофоров оптимальны с той или иной точки зрения. Пусть, к примеру, требуется выяснить, при каких значениях этих интервалов средние времена ожидания автомобилей имеют наименьшие значения. Поскольку результаты зависят от значений двух параметров, их следует перебирать в двух вложенных циклах.

Рассмотрим, например, значения параметров в диапазоне, границы которого отстоят от ранее указанных значений на 3 секунды, причем перебирать их будем с шагом 2 секунды. Тогда для каждого изз параметров получается 4 возможных значения: *t* – 3, *t* – 1, *t* + 1, *t* + 3, — где *t* — исходный интервал открытого состояния светофора. Всего получается 16 возможных сочетаний интервалов.

Для каждого сочетания интервалов открытия левого и правого светофоров чледует провести серию прогонов и собрать статистику по среднему времени ожидания автомобилей, движущихся с обеих сторон (например, по 400 прогонов для каждого сочетания). Поскольку стандартная процедура отображения статистики занимает достаточно много места на экране (4 строки), а нам требуется только среднее значение и его стандартное отклонение, не будем использовать имеющуюся процедуру отображения, а непосредственно выведем на экран необходимые результаты. В остальном при построении программы будем использовать подход, рассмотренный в пункте 3.2.2. В итоге получается следующий текст главного модуля программы (приведены только существенные с точки зрения поставленной задачи фрагменты алгоритма):

**var**

*// Статистика по среднему времени ожидания*

GlWaitStat : **array** [TDirection] **of** TStatistics;

…

*// Перебор параметров имитации*

**for** iLeft := 0 **to** LeftStepCount - 1 **do**

**for** iRight := 0 **to** RightStepCount - 1 **do**

**begin**

*// Задать времена открытия светофоров*

OpenTime[dirLeftRight] := MinLeftTime + iLeft \* LeftTimeStep;

OpenTime[dirRightLeft] := MinRightTime + iRight \* RightTimeStep;

*// Очистить статистику*

GlWaitStat[dirLeftRight].Clear;

GlWaitStat[dirRightLeft].Clear;

*// Выполнить серию прогонов*

**for** i := 1 **to** RunCount **do**

**begin**

*// Создать имитацию и запустить ее*

trf := TTraffic.Create;

SwitchTo(trf);

*// Собрать статистику*

GlWaitStat[dirLeftRight].AddData(trf.WaitStat[dirLeftRight].Mean);

GlWaitStat[dirRightLeft].AddData(trf.WaitStat[dirRightLeft].Mean);

*// Удалить имитацию*

trf.Free;

*// Отобразить ход выполнения*

**if** i **mod** 10 = 0 **then**

Write('.');

**end**;

*// Отобразить результаты серии прогонов*

WriteLn;

WriteLn('Left open time = ', OpenTime[dirLeftRight] : 4 : 0,

', right open time = ', OpenTime[dirRightLeft] : 4 : 0);

WriteLn('Left wait time = ', GlWaitStat[dirLeftRight].Mean : 8 : 4,

' +- ', GlWaitStat[dirLeftRight].Deviation : 6 : 4);

WriteLn('Right wait time = ', GlWaitStat[dirRightLeft].Mean : 8 : 4,

' +- ', GlWaitStat[dirRightLeft].Deviation : 6 : 4);

WriteLn;

**end**;

Заметим, что в этой программе выполняется большое число прогонов (для указанных параметров оно равно 4 \* 4 \* 400 = 6400), на которые затрачивается значительное время. Поэтому в процессе выполнения серии прогонов требуется визуализировать ее ход посредством отображения точек. К примеру, общее время выполнения программы на компьютере, оснащенном процессором AMD Athlon 64 X2 Dual Core 5200+, превысило 1,5 минуты. При одной из серии прогонов были получены следующие результаты:

........................................

Left open time = 57, right open time = 42

Left wait time = 73.0613 +- 5.6275

Right wait time = 90.7184 +- 11.2943

........................................

Left open time = 57, right open time = 44

Left wait time = 74.5417 +- 5.7729

Right wait time = 84.9568 +- 7.1904

........................................

Left open time = 57, right open time = 46

Left wait time = 76.3619 +- 5.7401

Right wait time = 83.6406 +- 6.0939

........................................

Left open time = 57, right open time = 48

Left wait time = 78.0614 +- 6.8024

Right wait time = 81.1025 +- 5.3060

........................................

Left open time = 59, right open time = 42

Left wait time = 71.2930 +- 4.2290

Right wait time = 92.1801 +- 11.5558

........................................

Left open time = 59, right open time = 44

Left wait time = 72.6017 +- 4.7677

Right wait time = 88.2595 +- 8.1777

........................................

Left open time = 59, right open time = 46

Left wait time = 74.3353 +- 5.4534

Right wait time = 84.6616 +- 6.6517

........................................

Left open time = 59, right open time = 48

Left wait time = 76.3017 +- 5.8210

Right wait time = 81.6708 +- 5.0406

........................................

Left open time = 61, right open time = 42

Left wait time = 70.0079 +- 4.0807

Right wait time = 95.2318 +- 13.3555

........................................

Left open time = 61, right open time = 44

Left wait time = 71.7748 +- 4.6604

Right wait time = 89.6501 +- 10.8366

........................................

Left open time = 61, right open time = 46

Left wait time = 72.7172 +- 4.6655

Right wait time = 85.7507 +- 7.8581

........................................

Left open time = 61, right open time = 48

Left wait time = 74.6240 +- 4.6991

Right wait time = 84.2009 +- 6.4219

........................................

Left open time = 63, right open time = 42

Left wait time = 68.7669 +- 3.5898

Right wait time = 96.9002 +- 14.4614

........................................

Left open time = 63, right open time = 44

Left wait time = 70.0132 +- 4.3029

Right wait time = 91.0528 +- 10.0081

........................................

Left open time = 63, right open time = 46

Left wait time = 72.3948 +- 4.2474

Right wait time = 88.3255 +- 7.3576

........................................

Left open time = 63, right open time = 48

Left wait time = 73.1849 +- 4.5948

Right wait time = 84.8234 +- 5.5536

Done.

Как видно, изменения параметров нельзя оценить однозначно. Так, увеличение длительности открытого состояния светофора уменьшает время ожидания для «своего» направления, но увеличивает время ожидания для противоположного. Если судить по среднему арифметическому двух времен ожидания, то оптимальным вариантом оказывает сочетание 59 секунд для левого светофора и 48 — для правого. По-видимому, есть смысл продолжить поиск оптимального значения с меньшим шагом вблизи этих значений.

Впрочем, правило выбора оптимального варианта может быть и иным. Так, к примеру, в исходной постановке задачи автомобили слева направо следуют чаще, чем справа налево. Следовательно, есть смысл учитывать среднее время ожидания для направления слева направо с б***о***льшим весом, чем для направления справа налево. Кроме того, в реальной анализируемой ситуации одно из двух направлений по какой-либо причине может иметь приоритет, и тогда нужно стремиться в первую очередь уменьшить это время ожидания.

Полный текст программы находится в папке 4.8 Traffic Multi.

## 4.9. Событийное проектирование. Модель банка для автомобилистов

# 5. Задания для самостоятельной работы

1. Внесите в любую из рассмотренных визуальных имитаций (выполняющих однократный прогон) следующие дополнения:

* Добавьте кнопку «Пауза», которая позволяет приостанавливать и возобновлять исполнение текущего исполняющегося прогона. Если прогон был полностью остановлен, эта кнопка не должна выполнять никаких действий.
* Добавьте возможность настройки всех параметров, управляющих выполнением имитации. Поскольку добавление соответствующих элементов на главное окно программы потребует много места, предусмотрите для этого дополнительное диалоговое окно, открываемое с помощью отдельной кнопки или команды меню.
* Сформулируйте общие подходы к выполнению указанных дополнений.

2. Внесите следующие изменения в модель магазина самообслуживания:

* К открытию магазина скапливается несколько покупателей, количество которых распределено в соответствии с законом Пуассона с математическим ожиданием, равным 3. Эти покупатели сразу же после открытия оказываются в торговом зале.
* В магазине работают три кассы, очередь к которым общая. Статистику следует собирать по всем кассирам одновременно.
* Максимальная длина очереди составляет 10 человек. Если покупатель после выбора покупок обнаруживает, что длина очереди максимальна, с вероятностью 0,2 он покидает магазин, не выполняя покупок. Следует учесть количество таких покупателей и общее число их покупок (упущенная выгода). Если покупатель не отказывается от совершения покупки, он продолжает выбор в течение времени, равномерно распределенного на интервале от 2 до 6 минут. За это дополнительное время количество его покупок возрастает на величину, равномерно распределенную от 0 до 2 включительно. После выбора дополнительных покупок покупатель вновь пытается встать в очередь на тех же условиях. Количество дополнительных попыток не ограничено.
* С вероятностью 0,3 покупатель может сделать одну дополнительную покупку непосредственно на кассе. На времени выбора это не сказывается. Время обслуживания дополнительной покупки распределено экспоненциально с математическим ожиданием 0,5 минуты.
* По истечении заданного времени имитации прекращается вход покупателей в торговый зал, однако имитация продолжается до завершения обслуживания последнего покупателя.
* Среднее время обслуживания одной покупки составляет 0,4 минуты, а средний интервал прибытия покупателей — 1,3 минуты.

3. Внесите следующие изменения в модель магазина самообслуживания:

* Покупатель, обойдя торговый зал, с вероятностью 0,05 отказывается от выполнения покупки по причине отсутствия необходимого ассортимента.
* В зале работают две кассы, к каждой из которых имеется своя очередь. Статистику по занятости кассиров следует собирать по каждой кассе отдельно.
* Покупатель, выбравший покупки, становится в более короткую очередь. Если очереди имеют равную длину, выбор делается с равной вероятностью в пользу любой из них.
* После обслуживания кассиром покупателю требуется экспоненциально распределенное время со средним значением 0,3 минуты для того чтобы забрать покупки. Если кассир заканчивает расчет первой покупки следующего покупателя до того как предыдущий забрал свои покупки, его работа блокируется до тех пор пока предыдущий покупатель не заберет покупки. После ухода предыдущего покупателя работа кассира возобновляется, и остальные покупки он рассчитывает в обычном режиме.
* Введите сбор статистики стоимости, которую каждый покупатель заплатил за свои покупки. Стоимость каждой отдельной покупки распределена нормально со средним значением 60 р., и стандартным отклонением 20 р., но не меньше 10 р.
* Соберите статистику по занятым инвентарным корзинам: покупатель, входя в торговый зал, берет корзину, и возвращает ее, забрав покупки либо покидая торговый зал без покупок.

4\*. В производственном цехе изготавливаются комплектующие трех наименований, а затем производится сборка из них готового изделия. Изготовлением каждого наименования комплектующих, а также сборкой, занимаются по одному рабочему. Время изготовления каждого вида изделия распределено нормально со средним значением 0,5, 0,7 и 0,4 часа соответственно и стандартным отклонением 0,1 часа во всех случаях. При этом время изготовления ограничено снизу значением 0,2 часа во всех случаях. Для работы сборщика необходимо наличие хотя бы по одной детали каждого типа, которые он берет и создает из них готовое изделие. На сборку требуется время, распределенное треугольно в интервале от 0,2 до 0,8 часа с модой 0,4 часа. Если текущее количество изделий какого-либо типа превышает 10, работа по изготовлению этого типа изделий блокируется до тех пор, пока их количество не уменьшится до 9. Собранные изделия поступают на склад, имеющий максимальный объем 50 изделий. Если склад полностью заполнен, работа сборщика блокируется. Оптовые покупатели прибывают за изделиями с интервалом времени, распределенным нормально со средним значением 24 часа и стандартным отклонением 4 часа, но не меньше 12 часов. Прибывшие покупатели забирают все имеющиеся на складе изделия.

Проведите моделирование работы данного цеха в предположении его круглосуточной его работы в течение недели (168 часов). Соберите статистику по занятости каждого рабочего, а также подсчитайте общее количество изделий, приобретенных покупателями. Кроме того, в результате многократных прогонов соберите статистику по количеству изделий, забираемых покупателями в течение недели.

*Дополнение*. Измените модель с учетом того, что изготовление комплектующих первого типа занимаются одновременно двое рабочих, при этом среднее время изготовления составляет 1 час со стандартным отклонением 0,2 часа, но не менее 0,5 часа. Если после изготовления какой-либо из этих рабочих оказывается заблокирован по причине переполнения очереди, его блокировка продолжается до тех пор пока количество деталей этого типа в очереди не уменьшится до 5. При этом другой рабочий не обязательно тоже блокируется.

5. Для модели поточной производственной линии проанализируйте влияние на результаты имитации распределения 6 свободных мест для размещения деталей между первой и второй очередями. Для этого сначала выберите статистические показатели, по которым имеет смысл проводить сравнение. Постройте программу (консольную или оконную), которая собирает статистику по выбранным показателям в результате 400 прогонов для каждого возможного распределения и отображает результаты в удобном для сравнения виде. Прокомментируйте полученные результаты. Какой вариант, с вашей точки зрения, является оптимальным?

# Литература

1. Андрианов А. Н., Бычков С. П., Хорошилов А. И. Программирование на языке симула-67. — М.: Наука. Глав. ред. физ.-мат. лит., 1985. — 288 с.
2. Прицкер А. Введение в имитационное моделирование и язык СЛАМ II: Пер. с англ. — М.: Мир, 1987. — 646 с., ил.

Оглавление

[Введение 1](#_Toc229502983)

[1. Базовые классы 2](#_Toc229502984)

[1.1. Списки 2](#_Toc229502985)

[1.2. Сопрограммы 6](#_Toc229502986)

[1.3. Примеры использования сопрограмм 8](#_Toc229502987)

[1.4. Случайные величины 15](#_Toc229502988)

[1.5. Классы сбора статистики и гистограмм 18](#_Toc229502989)

[1.6. Отображение статистики и гистограмм 22](#_Toc229502990)

[2. Процессы 26](#_Toc229502991)

[2.1. Классы процессов 26](#_Toc229502992)

[2.2. Имитационное время и календарь процессов 27](#_Toc229502993)

[2.3. Планирование работы процессов 29](#_Toc229502994)

[2.4. Создание процессов 32](#_Toc229502995)

[2.5. Завершение процессов 33](#_Toc229502996)

[2.6. Типовые роли процессов в имитации 35](#_Toc229502997)

[2.7. Процесс имитации 38](#_Toc229502998)

[2.8. Визуализация имитации 41](#_Toc229502999)

[2.9. Сбор и отображение статистики для действий 43](#_Toc229503000)

[3. Примеры моделей 47](#_Toc229503001)

[3.1. Модель банка 47](#_Toc229503002)

[3.1.1. Исходная задача 48](#_Toc229503003)

[3.1.2. Визуализация 55](#_Toc229503004)

[3.1.3. Использование нескольких обслуживающих устройств 61](#_Toc229503005)

[3.1.4. Имитация без использования процессов 67](#_Toc229503006)

[3.2. Модель магазина 68](#_Toc229503007)

[3.2.1. Исходная задача 68](#_Toc229503008)

[3.2.2. Многократные прогоны 72](#_Toc229503009)

[3.3. Модель поточной линии 75](#_Toc229503010)

[3.3.1. Блокировка 76](#_Toc229503011)

[3.3.2. Визуализация 80](#_Toc229503012)

[3.4. Модель линии контроля качества 82](#_Toc229503013)

[3.5. Модель работы по перевозке сыпучих грузов 86](#_Toc229503014)

[3.6. Модель работы карьера 93](#_Toc229503015)

[3.7. Модель сетевого планирования 101](#_Toc229503016)

[4. Дополнительные средства процессно-ориентированного моделирования 109](#_Toc229503017)

[4.1. Понятие ресурса 109](#_Toc229503018)

[4.2. Модель планирования запасов 115](#_Toc229503019)

[4.3. Модель работы порта 122](#_Toc229503020)

[4.4. Модель конвейерной линии 132](#_Toc229503021)

[4.5. Перехват ресурса 137](#_Toc229503022)

[4.6. Модель работы станка с поломками 140](#_Toc229503023)

[4.7. Затворы 140](#_Toc229503024)

[4.8. Модель участка с ограниченным движением 140](#_Toc229503025)

[4.9. Событийное проектирование. Модель банка для автомобилистов 140](#_Toc229503026)

[5. Задания для самостоятельной работы 140](#_Toc229503027)

[Литература 142](#_Toc229503028)