23
Модели, управляемые событиями

Основные понятия компьютерного моделирования

Объекты модели

Модели

Пример “по умолчанию”: класс NothingAtAll

Реализация классов моделирования

Класс SimulationObject

Класс DelayedEvent

Класс Simulation

Трассировка примера NothingAtAll

Моделирование — это представление системы объектов, существующих в реальном или воображаемом мире. Цель создания компьютерной модели — обеспечить рабочую среду для понимания моделируемой ситуации, например, поведения очереди, рабочей нагрузки клерка или своевременности обслуживания клиентов. Некоторые модели относятся к так называемым “моделям массового обслуживания”. Они описывают ситуации, когда существует место, за которым работает обслуживающий персонал (клерки). Заказчик (клиент) приходит сюда для того, чтобы клерк его обслужил. Если клерк не доступен, клиент становится в очередь. Первый клиент из очереди обслуживается первым освободившимся клерком. Часто модель такого типа имеет несколько мест обслуживания и несколько видов клиентов, каждый со своим перечнем мест обслуживания. Есть очень много примеров подобных моделей: банк, автомобильная мойка, парикмахерская, больница, кафетерий, аэропорт, почта, парк развлечений, фабрика. Компьютерные модели предоставляют возможность собирать статистику о таких ситуациях и проверять новые идеи об их организации.

Объекты, участвующие в моделях массового обслуживания, действуют более или менее независимо друга от друга. Поэтому приходится решать задачу координирования или синхронизации действий различных моделируемых объектов. Объекты обычно координируют свои действия через механизм обмена сообщениями. Одни объекты должны синхронизировать свои действия в определенные критические моменты, другие объекты не могут продолжать действия, если не получают доступа к некоторым ресурсам, которые могут оказаться недоступными в данный момент. В системе Smalltalk-80 синхронизирующие средства обеспечивают классы Process, Semaphore и SharedQueue. Чтобы поддерживать общее описание моделей массового обслуживания необходимы механизмы координации следующих действий:

- использования ресурсов фиксированного размера;
- использования изменяющихся ресурсов;
- одновременности действий двух объектов.

Ресурсы фиксированного размера могут быть расходуемыми или не расходуемыми. Для примера, джем — расходуемый фиксированный ресурс кондитерской; книги — не расходуемый ресурс библиотеки. Изменяющиеся ресурсы обычно подразделяются на возобновляемые и синхронизированные между производителем и потребителем. Кондитерская может моделировать свое обеспечение джемом как изменяющимся ресурсом, потому что джем может быть снова завезен на склад. Можно также представить себе ресурсы одновременно и возобновляемые, и нерасходуемые. Например, при построении модели пункта аренды автомобилей, автомобиль — возобновляемый ресурс, так как производятся новые автомобили и добавляются к имеющимся в аренде автомобилям; в то же время, автомобили — не расходуемый ресурс, поскольку арендованный автомобиль возвращается для использования его другими клиентами. В действительности, большинство не расходуемых ресурсов становятся расходуемыми, например, библиотечные книги в конечном счете становятся слишком изношенными для дальнейшего чтения; автомобили со временем ломаются. “Нерасходуемость ресурса” понимается как нерасходуемость его в течение времени функционирования модели.

Когда действия двух объектов модели надо синхронизировать для того, чтобы создать видимость совместного выполнения ими некоторой задачи, говорят, что два объекта находятся в отношении клиент-сервер. Например, врачу необходимо содействие пациента, чтобы провести обследование. Сервер — координируемый ресурс; это объект модели, чьи задачи могут быть выполнены только тогда, когда затребует ресурс один или более клиентов.

Важный аспект моделирования состоит в том, что создаются модели, изменяющиеся во времени; клиенты приходят в банк и покидают его; на мойку приезжают автомобили, их моют, сушат и они покидают мойку; самолет приземляется в аэропорту, разгружается, загружается и покидает аэропорт. Часты случаи, когда активность объектов соотнесена со временем: события происходят в определенное время или в определенном интервале времени. Следовательно, действия должны быть синхронизированы с некоторым понятием времени. Очень часто само понятие времени также моделируется.

Есть несколько способов представления действий моделируемых объектов в реальном или моделируемом времени. При одном подходе, часы идут обычным образом и в каждый такт часов все объекты модели получают возможность выполнить необходимые действия. Часы здесь выступают как синхронизирующее устройство модели, предоставляя удобный механизм обеспечения параллельных действий, поскольку часы “ожидают”, пока завершатся все действия, которые должны произойти в данное время. Часто бывает так, что в заданный такт часов ничего не происходит.

При альтернативном подходе, часы продвигаются вперед сразу ко времени, когда должно начинаться следующее действие. В этом случае система управляется следующим дискретным действием или происходящим по расписанию событием. Такая реализация модели зависит от поддержки очереди событий, упорядоченных в соответствии с временем модели. Каждый раз, когда завершается обработка некоторого события, из очереди выбирается следующее событие и часы переводятся вперед на соответствующее время.

Модели, описываемые в данной главе, основаны на таком, управляемом событиями подходе. Это модели, в которых существует множество независимых объектов, каждый со своим набором определенных к исполнению задач (получить услуги или ресурсы), и каждый нуждается в координировании своих действий во времени с другими объектами модели.

Данная глава описывает среду, в которой могут разрабатываться такие модели. Класс SimulationObject (ОбъектМодели) описывает общий вид объекта, который может появиться в модели, а именно, объекта с набором необходимых к выполнению задач. Протокол сообщений класса обеспечивает программную среду, в которой выполняются предусмотренные задачи. Экземпляр класса Simulation (Модель) поддерживает модель часов и очередь событий. Этим же классом координируются поступления новых объектов в систему (таких, как клиент) и определение ресурсов (таких, как клерки).

Следующая глава 24 посвящена способам сбора данных, порожденных действующей моделью. Статистику можно собирать с помощью общих механизмов, определенных в подклассах класса Simulation и/или класса SimulationObject. Любая модель может предоставлять и свой собственный механизм сбора информации о своем поведении.

В главе 25 приведены примеры моделей, которые используют два вида синхронизации: совместное использование фиксированных ресурсов и совместное использование изменяющихся ресурсов. Глава 26 вводит дополнительную поддержку координации между двумя объектами модели, один из которых ожидает обслуживания, а второй такое обслуживание обеспечивает.

Основные понятия компьютерного моделирования

Этот раздел содержит описание основного протокола классов SimulationObject и Simulation. Классы представлены дважды. Сначала приводится описание протокола каждого класса и объясняется, как создавать пример по умолчанию; затем дается описание реализации.

Объекты модели

Рассмотрим модель автомобильной мойки. Главные компоненты мойки: место для мытья, место для сушки, место для оплаты, мойщики, сушильщики, кассиры и автомобили различного вида, такие как грузовики или легковые автомобили. Мы можем классифицировать эти компоненты согласно их поведения. Основная классификация такая: места, где размещаются рабочие и выполняется работа; рабочие — мойщики, сушильщики и кассиры; автомобили — клиенты рабочих мест. Эта классификация может быть переведена в три класса объектов системы: Place (Место), Worker (Рабочий) и Customer (Клиент). Все эти классы похожи тем, что каждый из них описывает объекты, которые имеют задачи для выполнения: клиент требует обслуживания, рабочий обеспечивает такое обслуживание, а место предоставляет ресурсы для проведения работы. В частности, последнее

предоставляет очередь ожидания обслуживания, когда клиентов больше, чем могут обслужить рабочие. Общее поведение моделируется в суперклассе SimulationObject, который описывает объекты, появляющиеся в моделируемой ситуации; Экземпляр этого класса — объект модели — любой объект, которому можно указать последовательность задач для выполнения. Каждый такой объект определяет главную последовательность действий, которая инициализируется в момент, когда объект входит в модель. Например, действия автомобиля на мойке могут быть такими: запрос мойщика, ожидание в процессе мойки, запрос сушильщика, ожидание в процессе сушки, оплата за обслуживание и отъезд.

Класс SimulationObject определяет общую управляющую последовательность сообщений, в соответствии с которыми объект входит в модель, выполняет свои задачи и покидает модель. Эта последовательность состоит из посылки объекту сообщений startUp (войти), tasks (выполнить задачи) и finishUp (выйти). Инициализация описанных протоколом переменных происходит в ответ на сообщение initialize. Это сообщение вызывается методом, связанным с сообщением startUp. Методы для сообщений tasks и initialize реализуются подклассами класса SimulationObject.

SimulationObject instance protocol

initialization	
initialize	Инициализирует переменные экземпляра, если они есть.
simulation control	
startUp	Инициализирует переменные экземпляра. Информирует модель о том, что получатель входит в нее, после чего инициализирует задачи получателя.
tasks	Определяет последовательность действий, которые должен выполнить получатель.
finishUp	Информирует модель о том, что задачи получателя выполнены.

Есть несколько сообщений, которые любой объект модели может использовать для описания своих задач. Одно из них holdFor: aTimeDelay, где аргумент aTimeDelay — отрезок времени модели, на величину которого объект задерживает выполнение следующего действия. Смысл такой задержки — создать период времени, в течении которого объект предположительно доведет до конца некоторое действие.

Мы называем категорию таких сообщений языком задач модели (task language), тем самым подчеркивая, что эти сообщения посылаются объекту модели как часть реализации сообщения tasks.

Модель может содержать простые (или статические) ресурсы, например, банки с джемом, которые может приобретать объект модели. Кроме того, модель может состоять из координируемых ресурсов, то есть объектов модели, чьи задачи должны синхронизироваться с задачами других объектов модели. Язык задач включает сообщения доступа к каждому типу ресурсов, как для его получения, так и для выдачи.

Существуют четыре сообщения для статических ресурсов. Два из них — сообщения для получения ресурса с именем resourceName в количестве amount:

```
acquire: amount ofResource: resourceName
acquire: amount ofResource: resourceName withPriority: priorityInteger
```

Одно сообщение — для выдачи количества amount ресурса с именем resourceName:

```
produce: amount ofResource: resourceName
```

Еще одно — для отказа от полученного статического ресурса:

```
release: aStaticResource
```

Существуют также три сообщения для координируемых ресурсов. Прежде всего, это сообщение для получения ресурса с именем resourceName (здесь ресурс - объект модели, представляющий клиента, а запрос идет от сервера, такого как клерк):

```
acquireResource: resourceName
```

Произвести ресурс с именем resourceName можно сообщением (запрос идет от клиента):

produceResource: resourceName

Отказаться от полученного ресурса (объекта модели, чьи задачи могут теперь быть продолжены) можно сообщением:

resume: anEvent

Когда объект модели запрашивает статический ресурс, он может установить уровень важности запроса (его приоритет). Число 0 представляет наименьший уровень важности и, чем больше число, тем выше приоритет запроса. Сообщение acquire:ofResource: предполагает, что приоритет равен 0; сообщение acquire:ofResource:withPriority: задает уровень приоритета своим третьим аргументом.

Следующие два запроса проверяют наличие в модели статического ресурса и какое количество ресурса доступно. Сообщение resourceAvailable: resourceName определяет, есть или нет в модели ресурс, обозначаемый строкой resourceName. Сообщение inquireFor: amount ofResource: resourceName определяет, есть ли в модели указанный ресурс в количестве не менее amount.

Когда объект модели синхронизирует свои задачи с задачами другого объекта модели, бывает полезно выяснить, доступен ли такой объект. Два дополнительных запроса-сообщения

numberOfProvidersOfResource: resourceName
numberOfRequestersOfResource: resourceName

позволяют выяснить, доступен ли, соответственно, поставщик или потребитель координируемой задачи.

Кроме того, объекту модели можно послать сообщение stopSimulation, которое требует чтобы модель, экземпляр класса Simulation, в которой объект находиться, прекратила функционирование.

SimulationObject instance protocol

task language

holdFor: aTimeDelay
Задержать выполнение следующей задачи получателя, пока не истечет количество aTimeDelay времени модели.

acquire: amount ofResource: resourceName
Запросить модель об обеспечении получателя простым ресурсом, обозначаемым строкой resourceName. Если ресурс существует, выделить получателю количество amount этого ресурса. Если ресурса нет — известить пользователя модели (программиста) о том, что произошла ошибка.

acquire: amount ofResource: resourceName withPriority: priorityNumber
Аналогично предыдущему, за исключением того, что приоритет запроса на приобретение ресурса устанавливается равным числу priorityNumber.

produce: amount ofResource: resourceName
Запросить модель об обеспечении простым ресурсом, обозначаемым строкой resourceName. Если ресурс есть — добавить его в количестве amount к существующему ресурсу, если ресурса нет — создать его.

release: aStaticResource
Получатель использовал ресурс с именем aStaticResource. Ресурс больше не нужен и он возвращается в модель для повторного использования другими объектами.

inquireFor: amount ofResource: resourceName
Отвечает на вопрос о том, есть ли в модели по крайней мере количество amount ресурса, обозначаемого строкой resourceName.

resourceAvailable: resourceName
: Отвечает на вопрос о том, есть ли в модели ресурс, обозначаемый строкой resourceName.

acquireResource: resourceName
: Запросить модель об обеспечении ресурсом, обозначаемым строкой resourceName. Если он существует, предоставить получателю его услуги. Если ресурса нет, известить пользователя модели (программиста) о том, что произошла ошибка.

produceResource: resourceName
: Заставить получатель действовать как ресурс, обозначаемый строкой resourceName. Ждать другой объект модели, который пожелает получить этот ресурс.

resume: anEvent
: Получателю было предоставлено обслуживание ресурсом, обозначаемым аргументом anEvent. Обслуживание завершено, поэтому ресурс, объект модели, может продолжить свои задачи.

numberOfProvidersOfResource: resourceName
: Возвращает число объектов модели, ожидающих координирования своих задач, действуя как ресурс, обозначаемым строкой resourceName.

numberOfRequestersOfResource: resourceName
: Возвращает число объектов модели, ожидающих координирования своих задач, приобретая ресурс, обозначаемый строкой resourceName.

stopSimulation
: Заставляет модель, в которой действует получатель, остановиться. Все запланированные события удаляются и в модели ничего больше не происходит.

Примеры, которые мы представим в последующих главах, проиллюстрируют каждое сообщение языка управления задачами.

Модели

Цель класса Simulation (Модель) — управлять топологией объектов модели и планировать действия, которые происходят согласно времени модели. Экземпляры класса Simulation имеют ссылку на набор экземпляров класса SimulationObject, на текущее время модели и на очередь событий, ожидающих обработки.

Единица времени, соответствующая модели, хранится в переменной экземпляра и представляет собой число с плавающей точкой. Единица может быть миллисекундой, минутой, днем и т.д. Модель продвигает время, проверяя очередь для того, чтобы определить, когда запланировано следующее событие, и устанавливая свою переменную экземпляра на время, связанное с этим событием. Если очередь событий пуста, работа модели завершается.

Объекты модели входят в модель в ответ на одно из нескольких сообщений планирования, таких как

```
scheduleArrivalOf: aSimulationObjectClass
    accordingTo: aProbabilityDistribution
```

или

```
scheduleArrivalOf: aSimulationObject at: aTimeInteger.
```

Эти сообщения посылаются модели или во время, когда модель первоначально инициализируется в ответ на сообщение defineArrivalSchedule, или как часть последовательности задач, которые выполняет объект модели. Второй аргумент первого сообщения (aProbabilityDistribution) — экземпляр одного из классов распределения вероятностей, которые были определены в главе 22. В настоящей главе мы предполагаем, что все эти определения доступны. Распределение вероятностей определяет временной интервал, в течение которого будет создаваться очередной экземпляр первого аргумента — класса aSimulationObjectClass, и этому экземпляру будет посылаться сообщение startUp.

Кроме того, класс Simulation поддерживает сообщения, имеющие дело с планированием заданной последовательности действий; это сообщения

schedule: actionBlock at: timeInteger[1]

и

schedule: actionBlock after: amountOfTime.

Чтобы определять ресурсы модели, можно посылать модели одно или более из следующих двух сообщений. В первом сообщении

self produce: amount of: resourceName

второй аргумент — строка resourceName — обозначает доступный в модели простой количественно определенный ресурс; первый аргумент amount определяет количество этого ресурса, которое нужно сделать доступным. Во втором сообщении

self coordinate: resourceName

аргумент — строка resourceName — обозначает ресурс, который должен обеспечиваться некоторыми объектами модели, а запрашиваться другими объектами. Например, ресурс — мытье автомобиля, его поставщик — объект-мойка, а запрашивающий объект — автомобиль.

Simulation instance protocol

initialization

initialize — Инициализирует переменные экземпляра получателя.

modeler's initialization language

defineArrivalSchedule — Планирует объекты модели, которые будут вводиться в модель в определенные интервалы времени, как правило, на основе функций распределения вероятностей. Этот метод реализуется подклассами и включает последовательность сообщений к получателю (то есть к self), которые имеют вид
schedule:at:,
scheduleArrivalOf:at:,
scheduleArrivalOf:accordingTo:,
scheduleArrivalOf:accordingTo:startingAt:
(см. следующую категорию сообщений с их описанием).

defineResources — Определяет ресурсы, которые первоначально вводятся в модель. Обычно они выступают как ресурсы, которые можно приобретать. Этот метод реализуется подклассами и включает последовательность сообщений к получателю (то есть к self) вида produce: amount of: resourceName.

modeler's task language

produce: amount of: resourceName — Дополнительное количество amount ресурса, обозначаемого строкой resourceName, становится частью получателя; если ресурса еще нет в получателе, он создается; если он уже есть, его количество увеличивается.

coordinate: resourceName — Использование ресурса, обозначаемого строкой resourceName, должно координироваться получателем.

schedule: actionBlock after: timeDelayInteger — Запланировать блок actionBlock, который будет выполняться после того, как истечет количество timeDelayInteger времени модели.

schedule: actionBlock at: timeInteger — Запланировать блок actionBlock, который будет выполняться в заданное время timeInteger.

[1] во всех последующих примерах время — вещественное число. Название аргумента timeInteger (времяЦелое) осталось, видимо, по историческим причинам. — примеч. перев.

scheduleArrivalOf: aSimulationObject at: timeInteger

Запланировать объект модели, экземпляр класса SimulationObject, который будет введен в модель в заданное время timeInteger.

scheduleArrivalOf: aSimulationObjectClass
accordingTo: aProbabilityDistribution

Запланировать объекты модели, экземпляры класса aSimulationObjectClass, которые будут вводиться в модель через интервалы времени, определяемые на основе распределения вероятностей aProbabilityDistribution. Первый такой объект ввести в модель сразу же.

scheduleArrivalOf: aSimulationObjectClass
accordingTo: aProbabilityDistribution
startingAt: timeInteger

Аналогично предыдущему, но первый объект нужно ввести в модель во время timeInteger.

Обратите внимание, что в вышеупомянутых сообщениях планирования первый аргумент сообщения scheduleArrivalOf:at: — экземпляр класса SimulationObject, в то время как в сообщении scheduleArrivalOf:accordingTo: — класс SimulationObject class. Сообщения используются по-разному: первое — объектом модели, чтобы перепланировать самого себя, а второе — чтобы инициализировать поступление объектов модели в систему.

Протокол класса Simulaton включает несколько сообщений доступа. Одно из них, includesResourceFor: resourceName, может быть послано объектом модели, чтобы определить, существует ли в модели ресурс с заданным именем.

Simulation instance protocol

accessing

includesResourceFor: resourceName

Отвечает на вопрос, имеет ли получатель ресурс, обозначаемый строкой resourceName. Если такого ресурса нет — сообщает об ошибке.

provideResourceFor: resourceName

Возвращает ресурс, обозначаемый строкой resourceName.

time

Возвращает текущее время получателя.

Система управления моделью аналогична той, которая существует в классе SimulationObject. Инициализация происходит при создании модели и посылке ей сообщения startUp. Объекты модели и планирование новых объектов создают события, которые помещаются в очередь событий. Будучи инициализированной, модель запускается посредством посылки ей сообщения proceed и работает до тех пор, пока не будет исчерпана очередь событий.

В ходе работы модели объекты будут входить в модель и выходить из нее. Как часть протокола планирования, объект информирует свою модель о том, что он входит в нее или выходит. Соответствующие сообщения — enter: anObject и exit: anObject. В ответ на эти сообщения можно собирать статистику об объектах модели, входящих и покидающих модель. У подклассов класса Simulation есть возможность запретить вход объекта в модель, а вместо того, чтобы позволить объекту выйти из модели, можно перепланировать объект.

Simulation instance protocol

simulation control

startUp

Определить первоначальные объекты модели и запланировать поступление новых объектов.

proceed

Это сигнал на обработку одного события. Первое событие в очереди, если оно там есть, удаляется из нее, время модели устанавливается на время события, и событие инициализируется.

finishUp

Удалить ссылки на все оставшиеся объекты модели.

enter: anObject	Аргумент anObject сообщает получателю, что он входит.
exit: anObject	Аргумент anObject сообщает получателю, что он выходит.

В классе Simulation вышеупомянутые сообщения по умолчанию, как правило, ничего не делают. В частности, в ответ на сообщения enter: и exit: ничего не происходит, так же впрочем, как и в ответ на сообщения defineArrivalSchedule и defineResources. В результате, сообщение startUp ничего не совершает. Все эти сообщения обеспечивают только программную среду, которую должны использовать подклассы. Последние и создаются для того, чтобы переопределить эти сообщения и тем самым добавить модели ее особое поведение.

Пример "по умолчанию": класс NothingAtAll

В отличие от большинства классов системы, рассмотренных в предыдущих главах, суперклассы Simulation и SimulationObject обычно не реализуют свои основные сообщения как self subclassResponsibility. Тем самым, оба этих класса могут с успехом создавать свои экземпляры. Затем эти экземпляры могут использоваться как часть базовой модели или модели "по умолчанию", которая служит основой для нашего примера. В такой модели все объекты планируются так, что они ничего не делают и не содержат никаких событий. Разработка более содержательных моделей происходит как постепенное усовершенствование "моделей по умолчанию". Программист постепенно изменяет и проверяет модель, заменяя неинтересные экземпляры суперклассов на содержательные экземпляры соответствующих подклассов. Пример модели NothingAtAll (СовсемНичего) иллюстрирует идею "модели по умолчанию".

Предположим, что мы буквально ничего не делаем кроме того, что объявляем класс NothingAtAll как подкласс класса Simulation. Экземпляр класса NothingAtAll не имеет начальных ресурсов, поскольку ничего не делает в ответ на сообщение defineResources. Он также не имеет объектов, входящих в модель в различные временные интервалы, потому что ничего не происходит в ответ на сообщение defineArrivalSchedule. Теперь мы выполняем следующее выражение:

```
NothingAtAll new startUp proceed
```

В результате создается экземпляр класса NothingAtAll и ему посылается сообщение startUp. Образуется модель без ресурсов и без спланированных объектов, так что очередь событий пуста. В ответ на сообщение proceed модель определяет, что ее очередь событий пуста, и ничего не делает.

В качестве модификации описания класса NothingAtAll переопределим ответ на сообщение defineArrivalSchedule. В нем спланируем вход в модель экземпляров класса DoNothing. Класс DoNothing создадим просто как подкласс класса SimulationObject. Экземпляр класса DoNothing не имеет задач для выполнения, поэтому как только он входит в модель, он тут же и выходит из нее.

class name	DoNothing
superclass	SimulationObject

instance methods

no new methods

class name	NothingAtAll
superclass	Simulation

instance methods

initialization

```
defineArrivalSchedule
    self scheduleArrivalOf: DoNothing
            accordingTo: (Uniform from: 1 to: 5)
```

Эта версия класса NothingAtAll могла бы представлять ряд визитеров, входящих по очереди в пустую комнату, осматривающих ее без затраты времени, и тут же выходящих. Класс распределения вероятностей (в нашем примере Uniform) предполагается таким, как он был определен в главе 22. Согласно сделанным определениям, новый экземпляр класса DoNothing должен прибывать в модель каждые 1–5 единиц времени модели, начиная с 0. Следующие ниже выражения при их выполнении создают модель, посылают ей сообщение startUp, а затем циклически посылают ей сообщение proceed.

```
aSimulation ← NothingAtAll new startUp.
20 timesRepeat: [aSimulation proceed].
```

Сообщение startUp вызывает сообщение defineArrivalSchedule, которое составляет расписание для экземпляров класса DoNothing. Всякий раз, когда модели посылается сообщение proceed, экземпляр класса DoNothing входит и сразу же выходит из нее. Выполнение могло бы привести к следующей последовательности событий. В таблице время каждого события показывается слева, а описание события дается справа.

0.0	экземпляр DoNothing входит
0.0	экземпляр DoNothing выходит
3.21	экземпляр DoNothing входит
3.21	экземпляр DoNothing выходит
7.76	экземпляр DoNothing входит
7.76	экземпляр DoNothing выходит

и так далее.

Мы можем теперь сделать модель более интересной, планируя поступление в модель объектов модели разных видов, которые получают задачи что-либо сделать. Для этого определим класс Visitor как подкласс класса SimulationObject, экземпляры которого будут иметь задачу войти в пустую комнату и осмотреть ее, затрачивая на осмотр от 4 до 10 единиц времени модели. Время осмотра определяется как случайная величина, возникающая при вычислении выражения (Uniform from: 4 to: 10) next.

```
class name                  Visitor
superclass                  SimulationObject

instance methods

simulation control
    tasks
        self holdFor: (Uniform from: 4.0 to: 10.0) next
```

Класс NothingAtAll теперь определяется следующим образом.

```
class name                  NothingAtAll
superclass                  Simulation

instance methods

initialization
    defineArrivalSchedule
        self scheduleArrivalOf: DoNothing
                accordingTo: (Uniform from: 1 to: 5).
        self scheduleArrivalOf: Visitor
                accordingTo: (Uniform from: 4 to: 8)
                startingAt: 3
```

В модель входят два вида объектов, один не имеет времени на осмотр комнаты (экземпляр класса DoNothing), другой имеет на это короткий отрезок времени (экземпляр класса Visitor). Выполнение выражений

```
aSimulation ← NothingAtAll new startUp.
20 timesRepeat: [aSimulation proceed].
```

могло бы привести к следующей последовательности событий:

```
0.0     экземпляр DoNothing входит
0.0     экземпляр DoNothing выходит
3.0     экземпляр Visitor входит
3.21    экземпляр DoNothing входит
3.21    экземпляр DoNothing выходит
7.76    экземпляр DoNothing входит
7.76    экземпляр DoNothing выходит
8.23    первый экземпляр Visitor выходит через 5.23 секунды
```
и так далее.

Реализация классов моделирования

Для того, чтобы проследить путь, по которому происходило выполнение последовательности событий в предыдущих примерах, необходимо показать реализацию двух основных классов моделирования. Эти реализации также иллюстрируют управление несколькими независимыми процессами, ранее рассмотренное в главе 15.

Класс SimulationObject

Каждому объекту модели, созданному в системе, необходим доступ к тому экземпляру класса Simulation, в котором он функционирует. Такой доступ необходим, например, чтобы посылать сообщения, которые информируют модель о том, что объект входит в нее или выходит. Чтобы поддерживать такой доступ, класс Simulation отвечает на сообщение active. Задачи любого подкласса класса SimulationObject посылают сообщения объекту Simulation active для того, чтобы запросить информацию о модели, получить или произвести ресурсы.

```
class name                  SimulationObject
superclass                  Object

class methods

instance creation
    new
        ↑super new initialize
```

Система управления моделью, иногда называемая “жизненным циклом” объекта модели, включает последовательность сообщений startUp — tasks — finishUp. Когда объект первоначально появляется в модели, ему посылается сообщение startUp.

```
instance methods

initialization
    initialize
        "Ничего не делать. Инициализировать переменные экземпляра
        будут подклассы"
        ↑self

simulation control
    startUp
        Simulation active enter: self.
        "Сначала сообщить модели, что получатель начинает
        выполнять свои задачи"
        self tasks.
        self finishUp.
```

```
tasks
    "Ничего не делать. Необходимые действия будут
    спланированы в подклассах"
    ↑self

finishUp
    "Сообщить модели, что получатель выполнил свои задачи."
    Simulation active exit: self.
```

Категория task language состоит из сообщений, которые разработчик модели может использовать для определения последовательности действий объекта модели. Объект может задержать приращение времени модели (holdFor:), пытаться получить доступ к другому объекту модели, играющему роль ресурса (acquier:ofResource:), или определить, доступен ли необходимый ресурс (resourceAvailable:).

```
task language
    holdFor: aTimeDelay
        Simulation active delayFor: aTimeDelay

    acquire: amount ofResource: resourceName
        "Получить ресурс и затем сообщить ему, что он требуется в
        количестве amount. Возвратить экземпляр класса StaticResource"
        ↑(Simulation active provideResourceFor: resourceName)
            acquire: amount
            withPriority: 0

    acquire: amount
      ofResource: resourceName
      withPriority: priority
        ↑(Simulation active provideResourceFor: resourceName)
            acquire: amount
            withPriority: priority

    produce: amount ofResource: resourceName
        Simulation active produce: amount of: resourceName

    release: aWaitingSimulationObject
        ↑aWaitingSimulationObject

    inquireFor: amount ofResource: resourceName
        ↑(Simulation active provideResourceFor: resourceName)
            amountAvailable >= amount

    resourceAvailable: resourceName
        "В активной модели есть ресурс с этим именем?"
        ↑Simulation active includesResourceFor: resourceName

    acquireResource: resourceName
        ↑(Simulation active provideResourceFor: resourceName)
            acquire

    produceResource: resourceName
        ↑(Simulation active provideResourceFor: resourceName)
            producedBy: self

    resume: anEvent
        ↑anEvent resume
```

```
numberOfProvidersOfResource: resourceName
    | resource |
    resource ← Simulation active provideResourceFor: resourceName.
    resource serversWaiting
      ifTrue: [↑resource queueLength]
      ifFalse: [↑0]

numberOfRequestersOfResovrce: resourceName
    | resource |
    resource ← Simulation active provideResourceFor: resourceName.
    resource customersWaiting
      ifTrue: [↑resource queueLength]
      ifFalse: [↑0]

stopSimulation
    Simulation active finishUp
```

Модель хранит ресурсы в множестве, экземпляре класса Set. В случае статических ресурсов хранятся экземпляры класса ResourceProvider; в случае ресурсов, состоящих из задач, координируемых среди двух или более объектов модели, хранятся экземпляры класса ResourceCoordinator.

Когда объект модели запрашивает статический ресурс (acquire:ofResource:) и запрос удовлетворяется, тогда этому объекту передают экземпляр класса Resource. Экземпляр класса Resource ссылается на представляющий его ресурс и на его количество. Приведенные выше методы класса SimulationObject показывают, что ресурс в ответ на сообщение amountAvailable возвращает доступное в настоящее время количество ресурса, которое объект модели мог бы приобрести. Это сообщение посылается в методе inquireFor:ofResource:.

В методах из класса SimulationObject, связанных с сообщениями acquire:ofResource: и acquire:ofResource:withPriority:, возникает экземпляр класса ResourceProvider, которому посылается сообщение acquire: amount withPriority: priorityNumber. Результат этого сообщения — экземпляр класса WaitingSimulationObject. Однако, если количество amount ресурса не доступно, процесс, в котором сделан этот запрос, будет приостановлен до тех пор, пока необходимое количество ресурса не станет доступным. Экземпляру класса WaitingSimulationObject посылают сообщение release, чтобы отказаться от приобретенного ресурса.

Когда объект модели запрашивает координируемый ресурс (acquireResource:) и этот запрос удовлетворяется, тогда объект вводит в модель другой объект, действующий как ресурс (объект нуждающийся в обслуживании) до тех пор, пока не завершатся некоторые задачи (услуги). Если такой ресурс не доступен, то процесс, в котором сделан запрос, приостанавливается, пока необходимые ресурсы не станут доступными. Экземпляр класса ResourceCoordinator понимает сообщение как acquire, чтобы сделать запрос на координирование задач обслуживания, так и сообщение producedBy: aSimulationObject, чтобы указать на то, что аргумент сообщения должен быть введен в модель другим объектом для синхронизации действий. Из реализации класса SimulationObject следует, что экземпляр класса ResourceCoordinator может отвечать на запросы типа customersWaiting или serversWaiting, определяя, есть ли ресурсы (клиенты) или поставщики услуг (серверы), ожидающие координации своих действий, а также на сообщение queueLength, определяя число ожидающих.

Объяснение реализации классов ResourceProvider и ResourceCoordinator приводится в главах 25 и 26.

Класс DelayedEvent

Реализация механизма планирования для класса Simulation широко использует классы управления процессами Smalltalk-80, представленные в главе 15. Есть несколько проблем, которые необходимо решить при проектировании класса Simulation. Во-первых, как сохранять событие, которое должно быть отложено на некоторый промежуток времени

модели? Во-вторых, каким образом гарантировать, что все процессы, инициализированные в заданное время, завершатся до изменения показания часов? И, в-третьих, при условии решения первых двух проблем, как реализовать запрос на повторяющееся планирование последовательности действий, в частности, порождение и инициализацию некоторого вида объектов модели?

Чтобы решить первую проблему, класс Simulation поддерживает очередь всех планируемых событий. Эта очередь — экземпляр класса SortedCollection, чьи элементы — события, отсортированные согласно времени, в которое они должны вызываться. Каждое событие помещается в очередь внутри пакета — экземпляра класса DelayedEvent (ОтложенноеСобытие). Во время создания пакета соответствующее событие — активный процесс системы. Как таковой, он может сохраняться с необходимым для его выполнения контекстом путем создания семафора. Когда событие помещается в очередь, отложенному событию посылают сообщение pause, которое посылает своему семафору сообщение wait; когда событие забирается из очереди, его обработка возобновляется посылкой ему сообщения resume. Метод, связанный с сообщением resume, посылает семафору соответствующего отложенного события сообщение signal.

Протокол экземпляра класса DelayedEvent состоит из пяти сообщений.

DelayedEvent instance protocol

accessing	
condition	озвращает условие, согласно которому событие должно быть упорядочено.
condition: anObject	станавливает аргумент anObject как условие, согласно которому событие должно быть упорядоченно.
scheduling	
pause	риостановить текущий активный процесс, то есть текущее событие, которое обрабатывается.
resume	родолжить приостановленный процесс.
comparing	
<= aDelayedEvent	пределяет, следует ли аргумент aDelayedEvent за получателем.

Отложенное событие создается посылкой классу DelayedEvent сообщение new или onCondition: anObject. Реализация класса DelayedEvent приведена ниже.

class name	DelayedEvent
superclass	Object
instance variable names	resumptionSemaphore resumptionCondition

class methods

instance creation

```
new
    ↑super new initialize

onCondition: anObject
    ↑super new setCondition: anObject
```

instance methods

accessing

```
condition
    ↑resumptionCondition

condition: anObject
    resumptionCondition ← anObject
```

```
comparing
    <= aDelayedEvent
        resumptionCondition isNil
            ifTrue: [↑true]
            ifFalse: [↑resumptionCondition <= aDelayedEvent condition]

scheduling
    pause
        Simulation active stopProcess.
        resumptionSemaphore wait

    resume
        Simulation active startProcess.
        resumptionSemaphore signal.
        ↑resumptionCondition

private
    initialize
        resumptionSemaphore ← Semaphore new

    setCondition: anObject
        self initialize.
        resumptionCondition ← anObject
```

Согласно приведенному выше определению, любой объект, используемый в качестве условия возобновления resumptionCondition, должен отвечать на сообщение <=.

Класс Simulation

Экземпляры класса Simulation (Модель) имеют четыре переменные: множество объектов, которые действуют как ресурсы модели (resources); число, представляющее текущее время (currentTime); отсортированный набор, представляющий очередь задержанных событий (eventQueue); и целое число, обозначающее количество активных процессов в текущий момент времени (processCount). Переменная класса, называемая RunningSimulation, содержит ссылку на текущую активную модель, позволяя существовать нескольким моделям и контролируя в каждый текущий момент времени только одну из них.

Инициализация модели устанавливает переменные экземпляра в первоначальные значения. Когда модели посылают сообщение планирования startUp, она посылает себе сообщение activate, которое сообщает другим заинтересованным классам, какая модель сейчас активна.

```
class name                  Simulation
superclass                  Object
instance variable names     resources currentTime
                            eventQueue processCount
class variable names        RunningSimulation

class methods
instance creation
    new
        ↑super new initialize

accessing
    active
        ↑RunningSimulation

instance methods
```

initialization

initialize

```
resources ← Set new.
currentTime ← 0.0.
processCount ← 0.
eventQueue ← SortedCollection new
```

activate

```
"Этот экземпляр становится теперь активной моделью"
Running Simulation ← self.
```

Сообщения инициализации нужны и в подклассах. Эти сообщения позволяют разработчику модели задавать расписание поступления объектов и подходящее множество ресурсов. Они представляют интерфейс к сообщениям планирования процессов.

defineArrivalSchedule

```
"Подкласс определяет план, по которому объекты
динамически входят в модель."
↑self
```

defineResources

```
"Подкласс определяет объекты,
которые первоначально вводятся в модель."
↑self
```

task language

produce: amount of: resourceName

```
(self includesResourceFor: resourceName)
   ifTrue: [(self provideResourceFor: resourceName) produce: amount]
   ifFalse: [resources add:
            (ResourceProvider named: resourceName with: amount)]
```

coordinate: resourceName

```
(self includesResourceFor: resourceName)
   ifFalse: [resources add:
            (ResourceCoordinator named: resourceName)]
```

schedule: actionBlock after: timeDelayInteger

```
self schedule: actionBlock at: currentTime + timeDelayInteger
```

schedule: aBlock at: timeInteger

```
"Это механизм планирования единственного действия."
self newProcessFor:
        [self delayUntil: timeInteger.
         aBlock value]
```

scheduleArrivalOf: aSimulationObject at: timeInteger

```
self schedule: [aSimulationObject startUp] at: timeInteger
```

scheduleArrivalOf: aSimulationObjectClass
accordingTo: aProbabilityDistribution

```
"Это означает немедленный запуск модели."
self scheduleArrivalOf: aSimulationObjectClass
   accordingTo: aProbabilityDistribution
   startingAt: currentTime
```

```
scheduleArrivalOf: aSimulationObjectClass
  accordingTo: aProbabilityDistribution
  startingAt: timeInteger
    "Обратите внимание на то, что aSimulationObjectClass – класс
    SimulationObject или один из его подклассов. Реальная работа
    выполняется частным сообщением schedule:startingAt:andThenEvery:"
    self schedule: [aSimulationObjectClass new startUp]
      startingAt: timeInteger
      andThenEvery: aProbabilityDistribution
```

Заметим, что для определения сообщений с именами produce:of: и coordinate: сначала нужно определить, соответственно, классы ResourceProvider и ResourceCoordinator (в книге они определены в последующих главах).

Сообщения планирования категории task language реализуют подсчет ссылок для отслеживания запущенных процессов. Эта техника используется для решения второй упомянутой выше проблемы: как гарантировать, что все процессы, инициализированные в данное время, будут выполнены планировщиком единственного процессора Smalltalk-80 прежде, чем другой процесс получит возможность изменить время на часах модели. Используя счетчик ссылок, мы гарантируем, что время модели не изменится, пока счетчик ссылок не станет равен нулю.

Ключевые методы связаны с сообщениями

```
schedule: aBlock at: timeInteger
```

и

```
schedule: aBlock
    startingAt: timeInteger
    andThenEvery: aProbabilityDistribution.
```

Второе сообщение частное и вызывается методом

```
scheduleArrivalOf: aSimulationObjectClass
    accordingTo: aProbabilityDistribution
    startingAt: timeInteger.
```

Он обеспечивает общий механизм планирования повторяющихся действий, следовательно, дает решение и для третьей сформулированной ранее проблемы: как реализовать запрос на повторяющееся планирование последовательности действий.

Основная идея сообщения schedule: aBlock at: timeInteger — создать процесс, в котором выполнение последовательности действий (aBlock) откладывается до тех пор, пока модель не достигнет соответствующего времени timeInteger. Задержка выполняется сообщением delayUntil: delayedTime. Связанный с ним метод создает отложенное событие, которое добавляется в очередь событий модели. Процесс, связанный с этим отложенным событием, приостанавливается (посылкой событию сообщения pause). Когда это отложенное событие оказывается в очереди первым, оно удаляется и время устанавливается на ранее сохраненное время. Затем этому отложенному событию будет послано сообщение resume, которое вызовет выполнение блока; этот блок должен спланировать некоторое действие самой модели.

Процесс, который был активен, приостанавливается, когда соответствующее отложенное событие получает сообщение wait. Следовательно, число активных процессов должно быть уменьшено (stopProcess). Когда отложенное событие возобновляется, процесс продолжает выполнение с последнего выражения в методе delayUntil:; следовательно, число активных процессов должно быть увеличено (startProcess).

```
scheduling
  delayUntil: aTime
    | delayEvent |
    delayEvent ← DelayedEvent onCondition: aTime.
    eventQueue add: delayEvent.
    delayEvent pause.
```

```
delayFor: timeDelay
    self delayUntil: currentTime + timeDelay

startProcess
    processCount ← processCount + 1

stopProcess
    processCount ← processCount – 1
```

Подсчет ссылок на процессы также делается в методе, связанном с сообщением newProcessFor: aBlock (из категории планирования класса Simulation), который реализован следующим образом:

```
newProcessFor: aBlock
    self startProcess.
    [aBlock value.
     self stopProcess] fork
```

Первое выражение увеличивает счетчик процессов. Второе выражение — блок, для которого порождается новый процесс. Когда планировщик выполняет этот блок, выполняется последовательность действий модели, содержащаяся в аргументе aBlock. По завершении выполнения, блок дает сигнал, требующий уменьшить счетчик активных процессов. Таким образом, единая последовательность действий планируется в очереди событий экземпляра класса Simulation и задерживается до требуемого времени модели. Итак, счетчик ссылок на процессы увеличивается всякий раз, когда инициализируется новая последовательность действий, и уменьшается всякий раз, когда последовательность действий завершается; он уменьшается, когда происходит задержка отложенного события, и увеличивается, когда отложенное событие возобновляется.

Метод для частного сообщения

```
schedule: aBlock
    startingAt: timeInteger
    andThenEvery: aProbabilityDistribution
```

порождает процесс, который итеративно планирует действия. Алгоритм состоит в итерации двух сообщений:

```
self delayUntil: timeInteger.
self newProcessFor: aBlock
```

Повторение сообщений delayUntil: и newProcessFor: зависит от функции распределения вероятностей. Число повторений равняется числу элементов в выборке, порожденной данным распределением. Число, попавшее в выборку, определяет задержку до следующего времени вызова последовательности действий aBlock. Элементы распределения перечисляются посредством посылки распределению сообщения do:.

```
private
    schedule: aBlock
      startingAt: timeInteger
      andThenEvery: aProbabilityDistribution
        self newProcessFor:
          ["Это время для первого действия."
           self delayUntil: timeInteger.
           "Выполнить действие."
           self newProcessFor: aBlock copy.
           aProbabilityDistribution
               do: [:nextTimeDelay |
                    "Для каждой выборки из распределения, задержать
                    модель на выбранное количество времени."
                    self delayFor: nextTimeDelay.
                    "Теперь выполнить действие."
                    self newProcessFor: aBlock copy]]
```

У класса Simulation среда управления подобна той, которая есть у класса SimulationObject. Ответ на сообщение startUp — сделать модель активной, а затем определить объекты модели и расписание их поступления в модель. Внутренний цикл планируемых действий определяется как ответ на сообщение proceed. Всякий раз, когда экземпляр класса Simulation получает сообщение proceed, он проверяет счетчик ссылок на процессы, посылая сообщение readyToContinue. Если число процессов не равно нулю, то в модели еще есть активные процессы. Поэтому системный процессор (Processor) перепланирует процессы так, чтобы разрешить им выполниться. Если же число процессов — нуль, то проверяется очередь событий. Если она не пуста, очередное событие удаляется из очереди, время изменяется и задержанный процесс возобновляется.

```
simulation control
    startUp
        self activate.
        self defineResources.
        self defineArrivalSchedule

    proceed
        | eventProcess |
        [self readyToContinue] whileFalse: [Processor yield].
        eventQueue isEmpty
          ifTrue: [↑self finishUp]
          ifFalse: [eventProcess ← eventQueue removeFirst.
                    currentTime ← eventProcess condition.
                    eventProcess resume]

    finishUp
        "Сделать пустой очередь событий"
        eventQueue ← SortedCollection new.
        ↑nil

    enter: anObject
        ↑self

    exit: anObject
        ↑self

    atEnd
        "Завершить в модели все активные процессы и затем спросить ее,
        готова ли она к продолжению."
        [self readyToContinue]
          whileFalse: [Processor yield].
        ↑eventQueue is empty.

private
    readyToContinue
        ↑processCount = 0
```

В дополнение к сообщениям из категорий языка модели и управления моделью, в протоколе класса Simulation есть несколько сообщений доступа.

```
accessing
    includesResourceFor: resourceName
        | test |
        test ← resources
          detect: [:each | each name = resourceName]
          ifNone: [nil].
        ↑test notNil
```

```
provideResourceFor: resourceName
    ↑resources detect: [:each | each name = resourceName]

time
    ↑currentTime
```

Реализации классов Simulation и SimulationObject иллюстрируют посылки сообщения fork — экземпляру класса BlockContext, сообщения yield — экземпляру класса ProcessorScheduler, сообщений signal и wait — семафору.

Трассировка примера NothingAtAll

Теперь мы можем проследить исполнение первого примера модели NothingAtAll, в котором планировались только экземпляры класса DoNothing. После посылки сообщения

```
NothingAtAll new
```

переменные экземпляра новой модели состоят из

```
resources = Set()
currentTime = 0.0
processCount = 0
eventQueue = SortedCollection()
```

Затем мы посылаем модели сообщение startUp, которое вызывает сообщение scheduleArrivalOf: DoNothing accordingTo: (Uniform from: 1 to: 5). Это равносильно посылке модели сообщения

```
schedule: [DoNothing new startUp]
startingAt: 0.0
andThenEvery: (Uniform from: 1 to: 5)
```

Ответ состоит в вызове сообщения с именем newProcessFor:.

Шаг 1. newProcessFor: увеличивает processCount (self startProcess) и затем создает новый процесс, выполняющий следующий блок, на который далее будем ссылаться как на блок A:

```
[self delayUntil: timeInteger.
 self newProcessFor: block copy.
 aProbabilityDistribution do:
      [:nextTimeDelay |
       self delayFor: nextTimeDelay.
       self newProcessFor: block copy]
```

где переменная block — блок [DoNothing new startUp], на который далее будем ссылаться как на блок B.

Шаг 2. Когда процесс возвратится ко второму выражению метода newProcessFor:, выполнение продолжится с блока A. Его первое выражение уменьшит счетчик процессов и приостановит активность до времени 0.0 (то есть, создаст отложенное событие для планировщика активной модели, чтобы послать экземпляру класса DoNothing сообщение startUp во время 0.0, и поместит это событие в очередь событий).

```
resources = Set()
currentTime = 0.0
processCount = 0
eventQueue = SortedCollection(aDelayedEvent 0.0)
```

Теперь мы посылаем модели сообщение proceed. Счетчик процессов равен 0, так что сообщение readyToContinue возвращает true; очередь событий не пуста, поэтому первое отложенное событие удаляется из нее, время устанавливается в 0.0, и задержанный процесс возобновляется (это был блок A). Следующее действие увеличивает счетчик процессов и выполняет блок B. Это позволяет экземпляру класса DoNothing войти в модель и выполнить свои задачи, которых нет, и немедленно покинуть модель. Новое сообщение

newProcessFor: уменьшит счетчик процессов до 0, получит число из распределения вероятностей, сделает задержку на это число единиц времени модели, и спланирует новый процесс для некоторого более позднего времени. Эта последовательность событий продолжается до тех пор, пока модели вновь не будет послано сообщение proceed.

В зависимости от метода, связанного с сообщением tasks в подклассе класса SimulationObject, в очереди событий будут размещаться и другие задачи. Таким образом, если в модели NothingAtAll планируется экземпляр класса Visitor, то выражение self holdFor: someTime будет помещать событие в очередь, перемежая его с событиями, которые планируют новые поступления экземпляров классов Visitors и DoNothings.