Министерство образования и науки Российской Федерации

Филиал федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный экономический университет» в г. Чебоксары

В. И. ГУРЬЯНОВ

ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ НА UML SP

Монография

Чебоксары
2014

УДК 004.94
ББК 30в6
Г 95

Гурьянов В. И. **Имитационное моделирование на UML SP**: монография / В.И. Гурьянов. – Чебоксары : Филиал СПбГЭУ в г. Чебоксары, 2014. – 135 с.

ISBN 978-5-4246-0279-5

Печатается по решению редакционно-издательского совета
Филиал федерального государственного бюджетного образовательного учреждения
высшего профессионального образования
«Санкт-Петербургский государственный экономический
университет» в г. Чебоксары

РЕЦЕНЗЕНТЫ:
доктор физ.-мат. наук, профессор В. Н. Орлов
кандидат техн. наук, доцент М. В. Богданов
кандидат физ.-мат. наук, доцент В. П. Филиппов

В монографии излагаются основы имитационного моделирования с точки зрения объектно-ориентированного подхода. Имитационные модели представлены на разработанном автором специальном языке имитационного моделирования – UML Scientific Profile. Дается определение основных стереотипов профиля. Подробно рассмотрена методология Modeling SP – методология разработки объектных моделей. Приводятся многочисленные примеры. Реализация имитационных моделей выполнена на С++. Монография рассчитана на научных работников, инженеров и аспирантов, интересующихся проблемами имитационного моделирования.

© Гурьянов В.И., 2014
ISBN 978-5-4246-0279-5
© Филиал СПбГЭУ в г. Чебоксары, 2014

ОГЛАВЛЕНИЕ

Предисловие 4
Глава I. Основы имитационного моделирования 6
1.1. Унифицированный процесс 6
1.2. Пример имитационной модели 8
1.3. О сущности имитационного моделирования 13
Глава II. Разработка имитационных моделей 15
2.1. Визуальный язык имитационного моделирования UML SP 15
2.1.1. Концепция UML SP. 15
2.1.2. MSP и основные стереотипы UML SP 17
Примеры и пояснения 37
2.2. Особенности моделирования социально-экономических систем 64
2.2.1. Субпрофиль Scientific Profile for Active Systems 64
2.2.2. Моделирование сложного поведения систем 73
Примеры и пояснения 77
2.3. Моделирование организационных систем 80
Примеры и пояснения 84
2.4. Моделирование бизнес-процессов 85
Примеры и пояснения 87
2.5. Адаптация, развитие и эволюция 87
2.5.1. Адаптация 87
2.5.2. Развитие систем 88
2.5.3. Эволюция 92
Примеры и пояснения 97
Глава III. Изучение программных симуляций 101
3.1. Аналитические методы 101
Примеры и пояснения 108
3.2. Имитационный эксперимент 111
Примеры и пояснения 115
Заключение 118
Библиографический список 119
Приложение I 122
Приложение II 128
Приложение III 132

Предисловие

Возникновение имитационного моделирования обычно связывают с появлением метода Монте-Карло. Согласно легенде, имитационное моделирование «изобрел» Станислав Улам в 40-х годах прошлого века, когда понял, что задача о вероятности выкладывания пасьянса может быть решена не только математически, но и экспериментально. Успешное решение задачи о движении нейтрона в изотропной среде, выполненное методом Монте-Карло Дж. фон Нейманом и С. Уламом, по-видимому, и может считаться отправной точкой в развитии имитационного моделирования.

В 50-е и 60-е годы для создания имитационных моделей использовались универсальные языки программирования, такие как FORTRAN и ALGOL, что получило название *алгоритмического подхода*. В 70-е годы Т. Нейлор попытался применить имитационные модели для изучения реальных экономических процессов. Однако на протяжении как 70-х, так и 80-х годов эти попытки были по большей части безуспешными. С середины 70-х годов появились инструментальные средства, имеющие собственные языковые возможности. Благодаря этому многие проблемы создания имитационных моделей в 80-х годах удалось преодолеть, снизив уровень сложности разработки модели до приемлемого уровня. Это породило то, что сейчас называют *инженерным подходом*. В последующие два десятилетия имитационные модели стали использоваться в контурах управления экономических субъектов и заняли вполне заслуженное место в практике управления.

Столь успешное развитие технологий специализированных пакетов имитационного моделирования неизбежно оттенило алгоритмический подход на второй план. В настоящее время создано множество замечательных учебных пособий по имитационному моделированию. За редким исключением (например, Труб И.И. «Объектно-ориентированное моделирование на С++» 2006 года издания), все они за основу берут тот или иной инструментарий (GPSS, Simulink, AnyLogic и др.). В тоже время применение имитационного моделирования в научных исследованиях требует использования всего арсенала информационных технологий, что предполагает построение имитационных моделей на универсальных языках программирования с учетом последних достижений в этой сфере.

В монографии выражена личная точка зрения автора на имитационное моделирование, а материал книги представляет собой некую компиляцию более ранних работ. Большая часть материалов была представлена на конференциях и семинарах, а также в различных публикациях 2010 – 2013 гг. Автор отдает себе отчет в том, что ряд гипотез и утверждений, высказанных в этой книге, носит дискуссионный характер, и ни в коей мере не настаивает на их абсолютной истинности. Большая часть подобных высказываний вынесена в разделы «*Примеры и пояснения*». Этот материал предполагает существование собственного мнения у читателя.

В монографии мы придерживаемся «коммуникационной» парадигмы имитационного моделирования, т.е. некий объект, является имитационной моделью другого объекта, если их компоненты имеют одинаковые протоколы обмена сообщениями. Для описания объектных моделей используется специальный профиль UML (*Scientific Profile* или *UML SP*), основная идея которого состоит в том, чтобы каждой программной сущности назначать предметную семантику. Для этого используются стандартные механизмы расширения UML. Язык UML SP подобен языку SysML/UML (Systems Modeling Language), однако в отличие от последнего наделен двойственной семантикой, что и делает его инструментом имитационного моделирования. Кроме того, дано описание адаптированного *унифицированного процесса*, названного *Modeling SP*, что позволяет говорить о методологии разработки имитационных моделей.

В первой главе рассмотрены основы имитационного моделирования, в частности, обсуждается «коммуникационная» парадигма и понятие «язык моделирования». В разделе 1.2 приведен пример модели на UML SP. Желающие могут ограничиться прочтением только этого раздела, чтобы получить полное представление о нашем подходе. Во второй главе вводятся основные концепты и стереотипы UML SP; указаны элементы адаптации *унифицированного процесса*; приводятся примеры объектных моделей. Рассматриваются особенности моделирования экономических процессов, объектные модели организационных систем и бизнес-процессов. Кратко обсуждаются вопросы моделирования развивающихся систем и некоторые аспекты моделирования эволюции. В третьей главе представлены отдельные методы и подходы к изучению программных симуляций. В книге почти не рассматриваются традиционные вопросы имитационного моделирования, такие как метод Монте-Карло, статистическая обработка, системы массового обслуживания. Касаясь подобных вопросов, мы обычно рекомендуем ту или иную литературу.

Два слова о практической значимости предлагаемого подхода. Использование визуального языка как языка имитационного моделирования открывает новые возможности чисто инженерного приложения имитационных моделей. Рассматриваемый в этой книге профиль UML можно использовать как *язык проектирования* систем различной природы, в том числе экономических, политических и общественных. Причем проект системы будет не менее строгим, чем математическая модель. Мы полагаем, что имитационные модели можно построить даже в таких областях, в которых математическое моделирование пока недостаточно эффективно.

Имитационное моделирование широко применяется в самых разнообразных областях науки и техники. Нельзя ожидать, что все специалисты в конкретной области являются профессиональными программистами. Поэтому монография написана в форме учебного пособия по языку UML SP. Материал представлен так, чтобы не заставлять читателя обращаться к специальной литературе по программной инженерии. Единственным исключением является необходимость владения навыками объектно-ориентированного программирования на языке C++. Для разработки приложений использовались системы программирования Borland C++ Builder и Cincom Smalltalk (VisualWorks 7.6).

Книга рассчитана на специалистов различных областей знаний, использующих имитационное моделирование в своих научных исследованиях. Кроме того, мы надеемся, что книга будет интересна также профессиональным программистам, поскольку «компьютерные игры», созданные природой, отличаются нетривиальностью и некоторой характерной эстетикой.

Автор выражает благодарность руководителю семинара «Математическое моделирование и прикладные задачи» доктору физ.-мат. наук В.Н. Орлову, а также всем участникам семинара, за плодотворное обсуждение некоторых тем, затронутых в этой книге.

Глава I. Основы имитационного моделирования

1.1. Унифицированный процесс

Программная симуляция создается с помощью компьютерной программы. Поэтому для создания имитационной модели вполне применим процесс разработки программного обеспечения (Software Development Process). В монографии мы ограничимся унифицированным процессом разработки программного обеспечения (Unified Software Development Process, USDP).

Унифицированный процесс есть конечный результат исследований, проводимых в Ericsson (метод Ericsson, 1967), Rational (Rational Objectory Process, 1996–1997) и других ведущих компаниях. Особая роль принадлежит корпорации Rational Software, которая выпустила на рынок структурированную базу знаний под названием *Rational Unified Process* (RUP). RUP представляет собой набор рекомендаций по созданию практически любых программных продуктов. Обобщая опыт лучших разработок, RUP детально определяет *когда*, *кто* и *что* должен делать в проекте, чтобы в результате получить программную систему в установленные сроки, с определенной функциональностью и в рамках отведенного бюджета. Далее мы будем говорить об *унифицированном процессе* (UP) и будем придерживаться терминологии (и переводу терминов) книги [4].

Напомним основные термины и принципы *унифицированного процесса*. Унифицированный процесс наглядно может быть представлен в виде матрицы *Рабочие потоки* × *Фазы* (см. рис.1) [4].

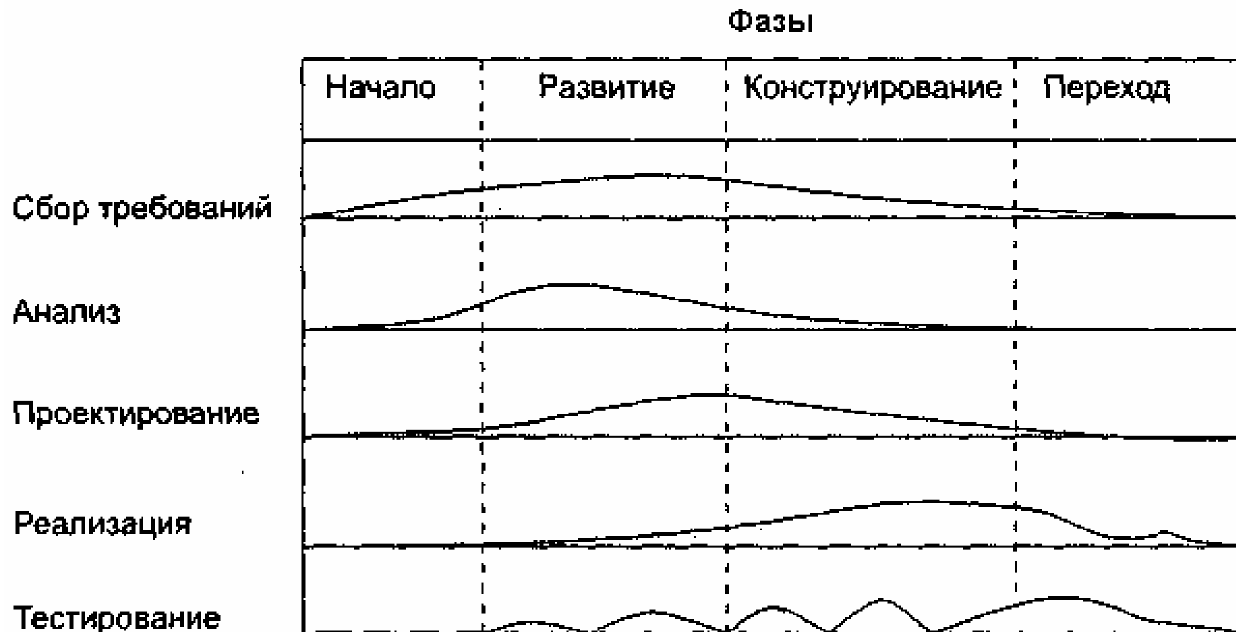


Рис.1. Матрица унифицированного процесса

Как видно на рис. 1, *унифицированный процесс* состоит из последовательности четырех фаз, каждая из которых завершается контрольной точкой:

- Inception (Начало) – цели и видение продукта;
- Elaboration (Развитие, Уточнение) – архитектура;
- Construction (Конструирование, Построение) – базовая функциональность;
- Transition (Переход, Внедрение) – выпуск продукта.

По мере прохождения этих фаз UP объем работ, выполняемый в каждом из пяти рабочих потоков, меняется.

К пяти основным рабочим потокам относятся:

• определение требований (построение Use Case Model) – сбор данных о том, что должна делать система, основные артефакты – диаграммы прецедентов и спецификация ПО;
• анализ (создание Analysis Model) – анализ требований и создание эскиза системы, основные артефакты – архитектура и диаграммы классов анализа;
• проектирование (разработка Design Model) – детальная проработка проекта системы, учет особенностей платформы;
• реализация – построение программного обеспечения;
• тестирование – проверяется, отвечает ли реализация предъявляемым требованиям.

Наиболее важными артефактами проекта являются *модели*: Use Case Model, Analysis Model, Design Model и д.р. Модель – целостный комплекс артефактов, который предоставляет самодостаточный взгляд на разрабатываемую систему. Самодостаточность моделей означает, что разработчик может из конкретной модели почерпнуть всю необходимую ему информацию, не обращаясь к другим источникам. Модели позволяют увидеть ее глазами будущих пользователей снаружи (Use Case Model) и глазами разработчиков изнутри (Analysis Model и Design Model) задолго до первой строки исходного кода. Большинство моделей представляются наборами UML-диаграмм.

В *унифицированном процессе* обычно используют следующие диаграммы: диаграмма классов (class diagram), диаграмма состояний (statechart diagram), диаграмма деятельности (activity diagram), диаграмма последовательности (sequence diagram), диаграмма кооперации (collaboration diagram), диаграмма компонентов (component diagram), диаграмма развертывания (deployment diagram). Подробное описание диаграмм смотрите в книге [9].

Унифицированный процесс предполагает итерационную модель жизненного цикла. Для средних и крупных проектов содержание этих итераций подробно рассмотрено в [35]. Разработка небольших моделей, в т.ч. тех, что рассматриваются в данной книге, может быть выполнена за три итерации. Для таких моделей удобно начинать разработку непосредственно с программы. Приведем следующую последовательность работ. В скобках приведены термины рабочих потоков.

Начало. На этой фазе производится выбор некоторого прототипа, который выражает концепцию проекта, причем этот выбор носит исключительно мыслимый характер. Поэтому мы это действие как итерацию выделять не будем.

Развитие (Уточнение). Итерация 1. Изучается предметная область. Если существует математическая модель моделируемого объекта, то ее следует включить в описание предметной области. Затем формулируются цели исследования, это позволяет определить, какие показатели должны измеряться в имитационных экспериментах (*определение требований*). Затем создается заготовка имитирующей программы (*модель анализа*). Можно воспользоваться шаблоном, приведенным в ПРИЛОЖЕНИИ I. Шаблон определяет *архитектуру* приложения. Затем необходимо внести изменения в шаблон, определив необходимые классы и задав наиболее важные связи между объектами (*классы анализа*). На этой итерации процедуры методов (*модель дизайна*) детализировать не стоит. Конечный артефакт – заготовка программы. Конечно же, программа должна работать.

Развитие (Уточнение). *Итерация 2.* Методом обратного инжиниринга (*reverse engineering*) следует построить диаграммы UML, определяющие имитационную модель. Некоторые CASE-системы поддерживают этот режим. Если такой возможности нет, то это можно сделать «вручную». Следует уточнить требования и построить диаграмму прецедентов; для каждого прецедента составить спецификацию (*определение требований*). В *модели анализа* нужны две диаграммы: диаграмма основных классов и архитектура (т.е. разложить классы по пакетам). Обе диаграммы должны показывать, как

приложение реализует каждый прецедент. *Модель дизайна* на этой итерации обычно не уточняется. Конечный артефакт – UML-модель.

Конструирование (Построение). Итерация 3. Цель этой итерации – согласование кода программы и UML-модели. UML-модель, построенная на второй итерации как правило отличается от программы, построенной на первой итерации. Поэтому в программу следует внести изменения. Кроме того, дописываются коды методов, а в UML-модели создается *модель дизайна*. Модель дизайна обычно содержит диаграммы деятельности, кооперации, последовательности действий. Эти диаграммы определяют алгоритмы процедур приложения. Конечный артефакт – согласованные UML-модель и программа.

Затем на тестах производится *верификация* модели. Цель верификации состоит в том, чтобы убедиться в правильности модели и программы. Для этого в качестве тестов используются некоторые частные случаи, точные решения для которых известны.

После третьей итерации собственно и начинается исследование имитационной модели.

Как уже было сказано выше, методология *унифицированного процесса* – это методология разработки любого программного обеспечения. В следующем разделе рассмотрен вариант унифицированного процесса, адаптированного под задачи имитационного моделирования (далее – *Modeling SP*).

1.2. Пример имитационной модели

Продемонстрируем методологию Modeling SP на простом примере. Этот пример мы будем рассматривать как шаблон для моделирования на UML SP, а данный раздел – как концентрированное изложение языка UML SP. Все последующее изложение можно рассматривать как обсуждение различных вопросов, возникающих на практике.

Итак, допустим, что необходимо построить модель процесса покупки товара.

Описание предметной области. Покупатель встречается с продавцом с целью покупки товара. Продавец предлагает цену, покупатель обдумывает предложение. Если цена покупателя устраивает, то он покупает товар. Мы осознанно утрируем ситуацию, чтобы не отвлекать внимание читателя на детали модели сделки.

Код программы. Как было сказано в предыдущем разделе, для разработки относительно простых имитационных моделей удобно начинать процесс разработки имитационной модели непосредственно с кода. Опишем моделируемый процесс на языке программирования C++. Создадим главную форму (см. рис. 2; далее будем говорить о *сцене*), посредством которой *Исследователь (Researcher)* будет управлять имитационным экспериментом и визуализировать результаты моделирования. Сначала систему надо создать и приготовить начальное состояние. Это будет делать метод Button1Click (кнопка с надписью *Приготовить*). Пусть метод Button2Click (кнопка с надписью *Продвинуть время*) моделирует процесс сделки; результат эксперимента отображается в полях формы (см. рис. 2).

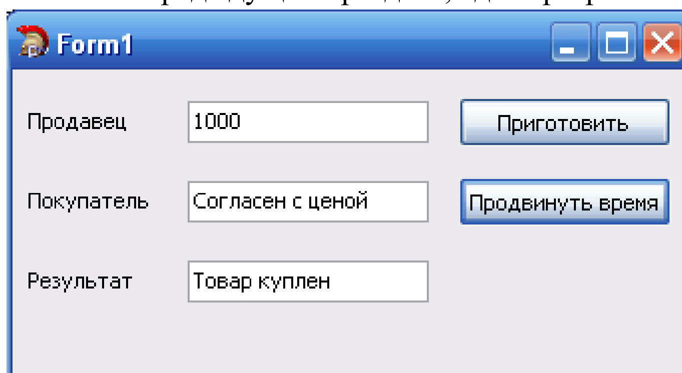


Рис. 2. Интерфейс модели

Главная форма с элементами управления – это модель экспериментальной установки. Для того чтобы снимать показания, разместим также некоторый код в самой

модели изучаемой системы. Совокупность этих операторов, включая сцену и некоторые другие объекты, назовем *исследовательской установкой* (*Research Instruments*). Для подобных программных сущностей (*Epistemology Entity*) далее будем использовать прилагательное «*гносеологический*». В имитационную модель процесса покупки гносеологические сущности не входят; мы их сознательно отделяем от самой модели (модуль Unit1, см. ПРИЛОЖЕНИИ I. Листинг 1).

Собственно имитационную модель представим классами Market., Buying, Agent и Fabric. Классы, подобные перечисленным, характерны для большинства имитационных моделей. *Изучаемая система (Ontology System)* моделируется экземплярами класса Buying (модуль Unit3, см. ПРИЛОЖЕНИИ I. Листинг 3). В дальнейшем для описания системы всегда будет создаваться специальный класс, который имеет, по меньшей мере, один метод, помеченный как exist (определяет единицу дискретно – событийного времени или, можно сказать, квант существования системы). Класс позволит инкапсулировать процесс покупки товара в объекте.

Кроме класса, моделирующего систему, необходим также класс, моделирующий *окружение системы* (*Ontology Environment*). Этот класс необходим для того, чтобы строго определить системные показатели, характеристики системы как целого, в частности, начальные и граничные условия. В нашем случае покупка осуществляется в контексте рынка (класс Market, модуль Unit2, см. ПРИЛОЖЕНИИ I. Листинг 2). Поэтому класс Buying должен иметь свойство deal, которое принимает значение true, если покупка совершилась и false – в противном случае. Обратите внимание – именно эту информацию мы и выводим на главную форму, т.е. *Исследователь* наблюдает за системой как бы со стороны, из контекста Market. Класс Market также обладает методом exist (сообщение посылает *Исследователь*), который в свою очередь посылает сообщение exist к объекту класса Buying. Экземпляр класса Agent – это *Атомарный объект (Ontology Atom);* нижний уровень объектной (точнее – объектно-темпоральной) декомпозиции системы. Код класса Agent приведен в Листинге 4, и в особом комментарии не нуждается. Единственно стоит обратить внимание на схематичность агента. Это не случайно. Единственное требование состоит в том, чтобы атомарный объект правильно описывал взаимодействие с системой. Внутреннее строение агента «физического» смысла не имеет. Прилагательное «*онтологический*» используется для того, чтобы отличать атомарные объекты модели от других атомарных объектов (например, атомов языка программирования).

В классе главной формы определим одно поле pWorld (*мир модели*) и метод Button1Click запишем следующим образом:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ // Приготовить начальное состояние
pWorld = new Market;
int r =  StrToInt(Edit1->Text);
pWorld->Prepare(r);
}  ,
```

а метод Button2Click:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ // Эксперимент
pWorld->Exist();  // квант существования мира модели
bool d = pWorld->observe; // <- наблюдение
if (d) { Edit2->Text = "Согласен с ценой"; // <- анализ данных
      Edit3->Text = "Товар куплен"; // <- анализ данных }
else {Edit2->Text = "Не согласен с ценой"; // <- анализ данных
      Edit3->Text = "Товар не куплен";  // <- анализ данных}
}  .
```

Экземпляры классов Market, Buying и Agent должны иметь некий общий интерфейс, который позволит им обмениваться сообщениями. Для этого вводится специальный абстрактный класс Fabric (общий предок классов модели; название для этого

класса подсказано известной книгой «The Fabric of Reality» [25]), который мы будем называть субстанциональным (*Substance*). Этот класс не может иметь экземпляров, но присутствует в каждом элементе модели и определяет всеобщие свойства (атрибуты) мира модели. С точки зрения программирования, этот класс определяет пользовательский тип TFabric, который и задает общий интерфейс для всех элементов модели. Код этого класса приведен в Листинге 5.

Полные листинги классов (см. ПРИЛОЖЕНИЕ I) будем рассматривать как шаблон для разработки других моделей. Модуль Unit1 задает *Сцену*, Unit2 – контекст модели, Unit3 – саму изучаемую систему, Unit4 – классы атомарных объектов и Unit5 – абстрактные классы модели.

Код программы, приведенный выше, можно рассматривать как описание предметной области на некотором языке. Если мы воспользуемся другим языком программирования, то получим другое описание. Интуитивно понятно, что оба способа описания имеют некоторую инвариантную составляющую, которая скрыта за синтаксисом конкретных языков. Мы явно можем определить этот инвариант, если обратимся к языку UML.

Имитационная модель на UML SP. Как сказано в первом разделе этой главы, на второй итерации создается имитационная модель на языке UML SP. В этой модели концепты верхнего уровня (*Researcher*, *Ontology System, Ontology Environment* и т.п.) обозначаются элементами UML, помеченными соответствующими стереотипами. В определении стереотипов вводится помеченное значение с меткой Concept, что позволяет задать регулярный механизм назначения предметной семантики элементам модели. Концепты верхнего уровня и концепты задачи находятся в отношении *инстанция* (InstOF). Концепт верхнего уровня – все множество смыслов, инстанция – элемент этого множества (например, «Антоновка» – инстанция «Фрукт»; причем важно *инстанцию* не путать с подмножеством «Яблоко») [68], [38]. Предметная область должна быть задана явной концептуальной моделью, например тезаурусом или онтологией. В нашем примере мы используем концепты: Рынок, Сделка, Продавец, Покупатель, Товар, Цена, Предлагает, Слушает, Принимает решение, Покупает и д.р. В случае необходимости Concept показывается на диаграмме.

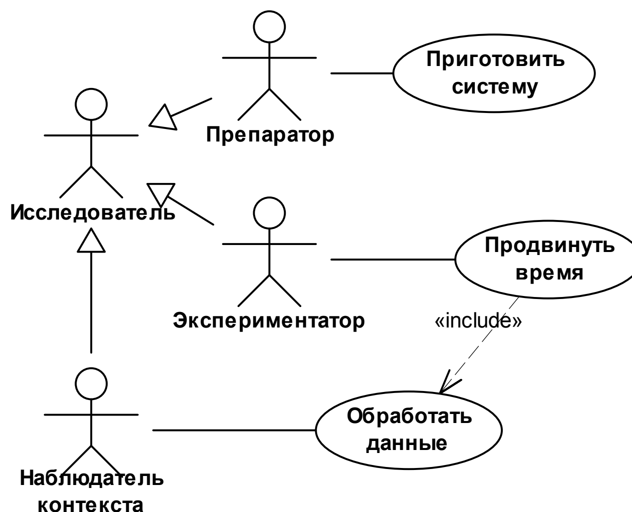


Рис. 3. Диаграмма прецедентов

По аналогии с UP в методологии *Modeling SP* предполагается построение трех моделей, которые определяются стереотипами *Research Use Case Model*, *Research Analysis Model* и *Research Design Model*.

Стереотип ***Research Use Case Model*** определяет спецификацию имитационной модели, т.е. задает модель функциональных требований имитационной модели. В нашем случае модель прецедентов показана на рис. 3.

Стереотип ***Research Analysis Model*** в рамках профиля задает семантику (предметную и вычислительную) имитационной модели и является «центром тяжести» процесса разработки. Основными артефактами модели анализа требований являются архитектура, диаграммы классов и зависимости между архитектурными пакетами.

Архитектура – это эскиз системы; основные компоненты системы изображаются как пакеты. На рис. 4 приведена типичная архитектура простой имитационной модели. На этом рисунке показаны названия пакетов, а не концепты. Пакет «*World*» содержит саму изучаемую имитационную модель. Пакет «*Research Instruments*» – это модель экспериментальной установки; использует некоторые общие алгоритмы обработки данных из пакета «*Epistemology Entity*». Пакет со стереотипом *Ontology Entity* группирует повторно используемые компоненты имитационной модели. Конкретное содержание этих пакетов определяется по диаграмме классов. Классы раскладываются по пакетам, а отношения между классами определяют зависимости архитектурных пакетов.

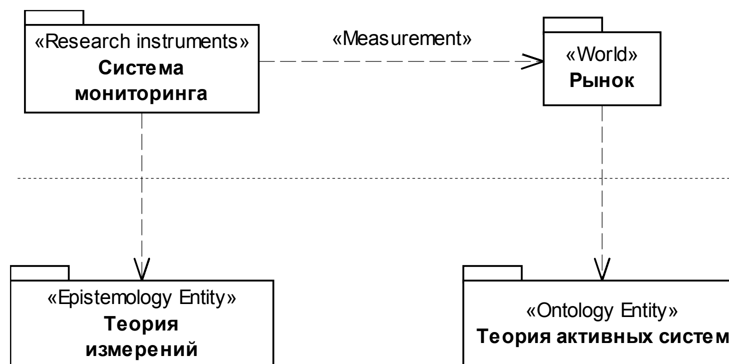


Рис. 4 Архитектура имитационной модели

Диаграмма классов анализа показана на рис. 5. Абстрактный класс Fabric – это *субстанциональный класс* (стереотип *Substance*), т.е. класс, который задает всеобщие свойства классов модели. В нашем случае такими качествами могли бы быть качества активных систем. С точки зрения вычислительной семантики этот класс определяет общий интерфейс для всех элементов системы. Класс Fabric мы должны поместить в пакет со стереотипом *Ontology Entity*.

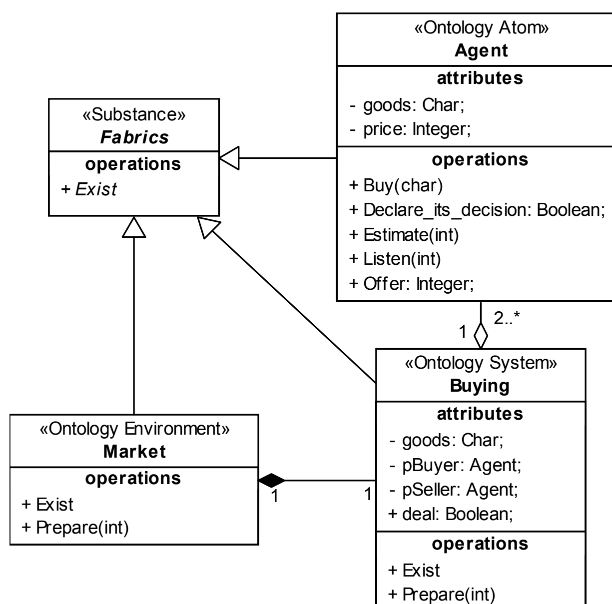


Рис. 5. Диаграмма классов имитационной модели

Классы Market, Buying и Agent поместим в пакет со стереотипом *World*. Пакет «*Epistemology Entity*» будет содержать базовые классы TForm, TButtom, TEdit, а пакет «*Research Instruments*» – класс TForm1. Фрагмент кода метода Market::Exist(), содержащий

один оператор observe = pBuying->deal и поле observe класса Market, также войдет в «*Research Instruments*» – это и есть отношение «*Measurement*».

Ключом к пониманию диаграммы классов является диаграмма кооперации (см. рис.6). Исследователь инициирует моделируемый процесс, посылая сообщение «*Exist*» объекту класса Market, который, в свою очередь, активизирует процедуру метода Buying::Exist(). Код этой процедуры имеет вид

```
int m = pSeller->Offer();  // продавец предлагает цену
pBuyer->Listen(m);  // покупатель слушает предложение
bool  d  =  pBuyer->Declare_its_decision();  //  покупатель  дает
ответ
if (d) { pBuyer->Buy(goods); // покупатель покупает товар
deal = true; }
else { deal = false; }
```

На диаграмме видно, как агенты обмениваются сообщениями. Последовательность обмена сообщениями составляет коммуникационный протокол в этой системе, а в более сложной – модель системы документооборота. Каналом передачи сообщений будет среда (объекты m и d) класса Buying.

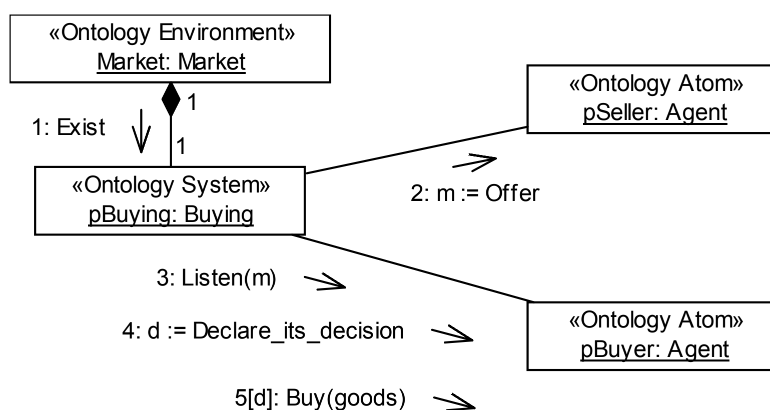


Рис. 6. Диаграмма кооперации

Диаграмма классов и диаграмма кооперации тесно связаны. Каждый объект – это экземпляр класса, каждая связь – экземпляр ассоциации. Поэтому на диаграмме классов должны быть показаны те ассоциации, для которых есть связи на диаграммах кооперации. Проводя параллель с математикой, можно сравнить диаграмму классов с уравнением, а диаграмму кооперации – с одним из решений этого уравнения (диаграмма последовательности действий – аналог графика функции). Тем самым, диаграммы классов задают модель процесса коммуникации.

«*Research Design Model*» имеет технический характер и позволяет перейти от модели анализа к коду программы на конкретном языке программирования. Такой переход выражается в определении конкретных алгоритмов методов и обычно является результатом перехода от композиции параллельных процессов к последовательным. В нашем случае такой переход не вызывает трудностей, поскольку процессы (будучи параллельными) выполняются последовательно.

В рассмотренном примере код программы и модель на UML уже являются согласованными. В процессе разработки такая согласованность достигается ценой многочисленных итераций модель-код, причем обычно код программы корректируется по UML-модели.

1.3. О сущности имитационного моделирования

Сначала определим наиболее общие понятия, такие как модель и моделирование. Под *моделью* будем понимать объект *m*, который в некотором определенном смысле может замещать объект *o*. Таким образом, модель – это некий заменитель, т.е. суррогат объекта. Соответственно, *моделирование* – это процесс подбора объекта *m*, такого, чтобы он мог использоваться вместо объекта *o*, относительно некоторого действия. На наш взгляд, «некоторое действие», как правило, связано с познавательной ситуацией. Термин *модель* имеет два значения – то, которое мы определили выше, и более древнее значение, как объект для подражания. Термин *модельная задача*, который далее мы часто будем использовать, надо понимать именно в этом, последнем значении.

Есть разные точки зрения на имитационное моделирование. Можно, например, говорить об агентной парадигме или парадигме системной динамики. В данной книге мы будем придерживаться «коммуникационной» парадигмы, которая имеет много общего с дискретно-событийной парадигмой.

Между математической моделью и объектом моделирования общими являются отношения между элементами. Мы предположим, что между имитационной моделью и объектом исследования общим будет протокол обмена сообщениями между компонентами. Т.е. *имитационная модель – всякий объект, который воспроизводит процесс коммуникаций исследуемого объекта*. Конечно, сообщение – это тоже некоторое отношение между элементами. Поэтому имитационная модель – это математическая модель. Но если математика рассматривает отношения «вне времени», то коммуникативный акт характеризуется дискретностью и событийностью и предполагает дискретно-событийное время (так как сообщение всегда направлено от передатчика к приемнику и, тем самым, делает их не равноценными во времени). В имитационных моделях есть и другие отношения. Однако они вторичны – выбираются так, чтобы обеспечить определенный протокол обмена сообщениями.

Коммуникационная парадигма позволяет довольно четко определить объект исследования в задачах имитационного моделирования. Объектом исследования является коммуникационный процесс, или протоколы обмена сообщениями. Алгоритмический подход [10] мы получим, если выполним редукцию протоколов на процессную модель. Отсюда следует, что имитационная модель и объект исследования имеют одинаковые алгоритмы поведения. Далее мы покажем, как системная динамика и многоагентные системы связаны с объектно-ориентированным имитационным моделированием.

Обсудим подробнее, что значит *язык имитационного моделирования*. Мы будем исходить из того, что код программы имитационной модели – это текст, описывающий некоторый кусочек действительности. Нам бы хотелось подчеркнуть это отдельно, потому что большинство авторов рассматривают объектно-ориентированное моделирование как способ написания «хороших» программ для имитационного моделирования. Различие состоит в том, что в нашем понимании код программы должен адекватно отражать описываемый феномен. Критерий адекватности мы уже указали – эквивалентность процессов коммуникации. Например, если мы моделируем лифт, то цикл на массиве – это не адекватное описание, а вот итератор – адекватное.

Текст всегда пишется на некотором языке. Напомним выразительное высказывание Галилея: «Философия написана в величественной книге (я имею в виду Вселенную), которая постоянно открыта нашему взору, но понять ее может лишь тот, который сначала научился постигать ее язык и толковать знаки, которыми она написана. Написана же она на языке математики». Точка зрения на математику как на мир идеальных объектов интересна и полезна (см. [22]), но, нам ближе понимание математики как языка. Мы будем также понимать имитационное моделирование как языковое описание действительности, точно так же как описанием действительности является математическое уравнение. Такая

точка зрения конструктивна. Далее мы покажем, как может быть строго построен такой *язык моделирования*.

Идея *языка моделирования* появилась в начале 60-х годов. Джеффи Гордон (Geoffrey Gordon) в 1961 г. разработал язык моделирования GPSS (General Purpose Simulating System – моделирующая система общего назначения). Язык GPSS ориентирован на описание систем массового обслуживания и в нем реализован блочно-ориентированный подход. Язык GPSS стал языком, который и на сегодняшний день определяет технологические решения в дискретном имитационном моделировании. Почти в то же время Кристен Нюгор (Kristen Nygaard) предложил язык моделирования СИМУЛА, концепции которого в дальнейшем легли в основу объектно-ориентированного программирования. Концепции языка СИМУЛА мы рассмотрим подробнее.

В 1962 г. К. Нюгорт (Норвежский компьютерный центр) приступил к реализации проекта Simulation Language (отсюда и SIMUlation LAnguage), предназначенного для программного моделирования методом Монте-Карло. Он привлек к сотрудничеству Уле Джохана Дала (Ole-Johan Dahl). За основу был выбран язык программирования АЛГОЛ-60. Идея объединить данные с процедурами, их обрабатывающими, появилась в 1965 г. Несколько позже появились и другие понятия – класс, объект, наследование, имитация параллельных вычислений. Новый язык вызвал интерес в Дании, Германии и СССР (в СССР в конце 60-х появилась реализация Симулы для УРАЛа-14 и БЭСМ-6). Окончательная версия языка была закончена в январе 1967-го года и получила название СИМУЛА-67. В настоящее время разработаны и другие языки моделирования, среди которых можно назвать Modelica (пакеты MathModelica и Dymola), SLAM II (развитие FORTRANа), ObjectMath (пакет Mathematica).

Язык СИМУЛА предназначен для моделирования систем с дискретно-событийным временем, т.е. систем, представляющих последовательность сменяемых друг друга мгновенных событий. У. Дал определяет моделирование как «процесс представления динамической системы моделью для получения информации об этой системе путем проведения экспериментов над моделью». Цель разработчиков языка была определена так: «предоставить в распоряжение исследователя, строящего модель системы, концептуальную основу для ясного и четкого мышления; предоставить средства для описания динамических моделей; облегчить процесс программирования».

Появление многочисленных понятий ООАП продиктовано желанием разработчиков языка использовать СИМУЛУ-67 в качестве основы для построения специализированных языков, ориентированных на различные предметные области. У. Дал пишет: «Язык Симула-67 выходит за традиционные рамки языков программирования и может служить основой, на которой строятся различные математические и естественнонаучные теории от геометрии и алгебры до химической технологии и сельского хозяйства, даже в тех случаях, когда речь идет не об имитации или программировании, а лишь о получении количественной информации». Нам эта мысль кажется весьма привлекательной.

Эти идеи, несколько подзабытые в последние годы, нашли свое отражение также и в языке UML SP. С одной стороны UML SP предоставляет разработчику имитационной модели «концептуальную основу для ясного и четкого мышления», с другой – позволяет использовать методологию UP для разработки компьютерных моделей. От существующих языков моделирования наш подход отличается тем, что отделяет саму модель от способа реализации модели (язык программирования, методы аппроксимации, организация вычислений и т.п.), делая имитационную модель объективной и существенно более ясной.

Глава II. Разработка имитационных моделей

2.1. Визуальный язык имитационного моделирования UML SP

Визуальный язык UML был разработан для проектирования программного обеспечения. Унифицированный язык моделирования (Unified Modeling Language) появился в конце 80-х в начале 90-х годов в основном благодаря усилиям Гради Буча, Джима Рамбо и Ивара Якобсона [9]. В настоящее время консорциум OMG принял этот язык как стандартный язык моделирования ПО. Мы будем в основном опираться на версию 1.5. UML имеет механизмы расширения, которые позволяют специализировать область применения языка. Такие расширения называются профилями UML. В данном разделе рассматривается профиль UML, далее называемый как *научный профиль* (*Scientific Profile*), предназначенный для описания объектных имитационных моделей [15].

2.1.1. Концепция UML SP. Мы определим три положения: (а) стереотипы обозначают концепты, (б) стереотипы имеют двойную семантику и (в) предполагается коммуникационная парадигма.

Профиль UML – это набор стереотипов, помеченных значений и ограничений. Предполагается, что модель предметной области представлена в *эксплицитной* форме, например, как тезаурус, семантическая сеть или онтология. Эксплицитная форма – это такая форма представления знаний, которая допускает машинную обработку. Стереотип может определять новые метки и ограничения, которые не являлись частью исходного элемента. В профиль вводится *двойственная семантика* посредством механизма помеченных значений для стереотипов в виде {Concept = *имя концепта*}. Для разграничения семантик, будем говорить о *вычислительной семантике* (см., например, [34]) и о *предметной семантике* стереотипа. Поскольку профиль – расширение UML, то можно воспользоваться разнообразными методологиями проектирования ПО, такими как *Unified Process* [4], *порождающее программирование* и др. Идея *профиля* состоит в том, чтобы распространить эти методологии на разработку имитирующих программ. Вследствие двойственной семантики можно будет говорить о методологиях построения имитационных моделей. Как будет показано далее, *Scientific Profile* предоставляет возможность создания повторяемого, предсказуемого и контролируемого процесса разработки имитационных моделей.

Приведем некоторые пояснения.

В языке UML предусмотрены *механизмы расширения,* которые применяется для уточнения синтаксиса и семантики. Механизм расширения включает в себя стереотипы, помеченные значения и ограничения.

Стереотипы позволяют создавать новые элементы языка на основе существующих элементов. Для применения стереотипа перед именем элемента необходимо добавить имя стереотипа во французских кавычках («…»). Стереотип обычно определяет ряд помеченных значений и ограничений, которые применяются к элементу, помеченному этим стереотипом.

Помеченные значения используются тогда, когда к элементу необходимо добавить некоторую специальную информацию. Синтаксис помеченных значений следующий: {метка_1 = значение_1, … , метка_N = значение_N}. Мы используем помеченное значение Concept.

Ограничения задают условия или правила для элемента модели, которые должны быть истинными. Ограничения отображаются, как строка текста, заключенная в фигурные скобки ({}), расположенная рядом с элементом. Ограничения можно задавать на специальном языке OCL, который входит в стандарт UML. Этот язык подобен языку SQL и отличается от процедурного языка.

То, что каждой программной сущности следует назначать предметную семантику, вовсе не означает, что на диаграммах всегда следует показывать концепты. Напротив, это

следует делать только тогда, когда это повышает наглядность диаграмм. Тем не менее, помеченные значения всегда присутствуют на так называемом *семантическом заднем плане* (*semantic backplane*) элемента UML, расширяя тем самым его спецификацию [4].

Мы будем придерживаться понимания *онтологии* в смысле системы Protégé. Protégé – это одна из наиболее популярных систем работы с онтологиями, первая версия которой создана в Стэндфордском университете (США) в 1987 г. По мнению разработчиков системы Protégé все понятия предметной области делятся на классы, подклассы, экземпляры [68].

Онтологии, если классифицировать их по принципу «общее-частное», подразделяются на следующие виды:

− верхнего уровня – система наиболее общих концептов, не зависящих от конкретной предметной области (например, CYC – база знаний понятий окружающего мира);

− онтологии предметной области (или онтология домена) – содержат понятия конкретной предметной области;

− онтологии задач – описывают понятия, которые определяются конкретными задачами. Назначение такой онтологии в том, чтобы описать концептуальную модель конкретной задачи или изучаемой системы.

Таким образом, каждый элемент модели может иметь, хотя это и не обязательно, несколько концептов. Концепты онтологии верхнего уровня фиксируются в стереотипах. Концепты онтологии задачи – в специальных именованных значениях {Concept = *имя концепта*} и ограничениях.

Онтологии предметной области образуют субпрофили (например, *Scientific Profile for Active Systems*); это позволяет специализировать профиль для конкретных научных областей. Можно подумать, что стереотипы субпрофиля строятся так же, как и стереотипы профиля, но только теперь в качестве метаклассов выступают стереотипы профиля. Это неверно. Стереотипы субпрофиля находятся в отношении обобщения со стереотипами профиля. Стереотипы субпрофиля замещают стереотипы профиля в конкретных моделях.

Продолжая параллель с математикой, заметим, что совершенно правомерно говорить об онтологии математических моделей, например об онтологиях уравнений математической физики. В научных текстах концепты определяются после ключевого слова «где».

Двойственная семантика UML SP предполагает две методологии, которые имеет смысл рассматривать раздельно. Методология, оперирующая с предметной семантикой – это методология моделирования систем. Методология отличается от других системных методологий. По своей форме и используемым методам данная методология подобна унифицированному процессу, но предназначена для моделирования систем произвольной природы. Методология, оперирующая с вычислительной семантикой – это методология ООАП. Методология подобна унифицированному процессу (хотя явно тяготеет к методу *Domain-Driven Design*), но формально не совпадает с ним, поскольку унифицированный процесс использует свой набор стереотипов и потому пользуется другим UML-языком. Важно подчеркнуть, что каждая из двух методологий может применяться совершенно автономно: для разработки моделей систем в системном анализе и для проектирования ПО соответственно. Что касается *Domain-Driven Design*, то применительно к имитационному моделированию интересна работа [28]; см. сайт [40].

Однако если обратится к компьютерному имитационному моделированию, эти методологии следует рассматривать только как единое целое. Это выражается в том, что процесс разработки имитационной модели предполагает итерационную разработку, при которой используется либо один, либо другой аспект UML SP. В этом случае возможно возникновение противоречий между моделью системы и моделью программы. Чтобы гарантированно исключить возникновение противоречий, потребуем, что бы обе модели

описывали некий инвариант, хотя и разными понятиями. В качестве такого инварианта мы будем рассматривать коммуникационную парадигму, т.е. и модель системы, и модель ПО должны описывать один и тот же коммуникационный процесс. В данной монографии единую методологию мы будем называть MSP (Modeling SP – моделирование на научном профиле) с тем, чтобы подчеркнуть то, что данная методология – это методология разработки имитационных моделей, а не методология имитационного моделирования. Сделаем еще одно примечание. Разработка ПО имитационного моделирования выполняется согласно унифицированному процессу и не сводится к разработке имитационной модели. Методология MSP как методология ООАП специально максимально приближена к UP с тем, чтобы можно было *встраивать* артефакты MSP в UP с минимальными усилиями.

Далее обе методологии рассматриваются параллельно, причем в большинстве случаев, сначала будем рассматривать вычислительную семантику, а затем – предметную. Тем не менее, не следует забывать, что каждая из методологий может быть определена независимо одна от другой. В первом случае предметом моделирования является реальная, существующая в физическом мире система. Во-втором – программная система.

2.1.2. MSP и основные стереотипы UML SP. Стереотипы UML SP имеют много общего со стереотипами SysML, однако эта тема заслуживает отдельного анализа, и здесь мы этой темы касаться не будем. Строгое определение стереотипов UML SP см. в ПРИЛОЖЕНИИ III. Сейчас мы дадим определение основных стереотипов в контексте их использования в MSP. Как сказано выше, методология разработки имитационных моделей строится по аналогии с UP. Напомним, что UP предполагает следующие рабочие потоки: *Требования* (метод прецедентов), *Анализ*, *Проектирование* и д.р. Рассмотрим рабочие потоки в этом порядке.

В качестве основы моделирования систем будем использовать концепцию «трех миров» К.Р. Поппера [46]. Структура реальности, согласно К. Попперу, имеет три компонента: мир ментальных состояний и процессов, физический мир и мир продуктов сознания. Именно такой смысл мы будем вкладывать в концепты стереотипов *Research Use Case Model*, *Research Analysis Model*, *Research Design Model*. С точки зрения вычислительной семантики для интерпретации стереотипов полезна точка зрения семиотики. Мир продуктов сознания – это код имитирующей программы, который, в свою очередь, рассматривается как текст, описывающий предметную область. UML-профиль будет играть роль метаязыка. Прагматика задается стереотипом *Research Use Case Model*, семантика – *Research Analysis Model*, а синтактика – *Research Design Model*.

Стереотип ***Research Use Case Model*** определяет цели и намерения *Исследователя* (стереотип *Researcher* "metaClass" Actor), последние обслуживаются прецедентами (случаями использования). Модель прецедентов есть метод *определения требований*, и предназначена для спецификации имитирующей программы.

В этом рабочем потоке производится сбор информации об объекте моделирования, определяются цели и задачи исследования, строится модель требований, диаграммы прецедентов, составляется спецификация прецедентов. Последнее и есть собственно спецификация имитирующей программы.

Типичные прецеденты: *Приготовить начальное состояние*, *Вычислить новое состояние*, *Обработать результаты наблюдения*, *Визуализировать результат*. Среди ролей *Исследователя* отметим роли *Наблюдатель контекста* и *Наблюдатель системы*, и будем различать их тем, где расположены датчики измерений (прежде всего детекторы и хронометры). Для примера п. 1.2. диаграмма прецедентов будет иметь вид, показанный на рис. 3.

Спецификация прецедента в простейшем случае должна содержать: *предусловие*, *этапы сценария*, *постусловие*. Более подробно о спецификации прецедентов см. [4]. Спецификации всех прецедентов образуют спецификацию программы (SRS).

«*Research Analysis Model*» создается в рабочем потоке *Анализ* и имеет два основных артефакта – архитектуру имитирующей программы и диаграммы классов (см. рис. 4 и рис. 5). Данная модель фокусируется на том, *что* должна делать система, чтобы прецеденты были выполнены. Детали того, *как* система будет это делать, предоставляются потоку *Проектирование* [4]. В рамках профиля аналитическая модель задает семантику (предметную и вычислительную) имитационной модели, и мы рассмотрим эту модель наиболее подробно.

Модель анализа требований представлена пакетом со стереотипом *Универсум* («*Universe*»). Типичная архитектура «*Research Analysis Model*» определяется четырьмя пакетами, распределенными по двум уровням абстракции (специфическому и общему) и двум разделам (*гносеологическому* и *онтологическому*); см. пример на рис. 4 . Заметим, что архитектура не входит в определение UML SP и является артефактом рабочего потока анализа. Знание архитектуры значительно облегчает понимание ограничений, накладываемых на стереотипы. Вычислительная семантика архитектуры имитационной модели может быть задана архитектурным паттерном *MVC*, который сам состоит из трех паттернов – *Observer*, *Strategy* и *Composite* [14]. Паттерны *Observer* и *Strategy* моделируют наблюдение за системой и экспериментальное воздействие на систему, а паттерн *Composite* определяет иерархию классов для *Observer* и *Strategy*. Возможны и другие архитектурные паттерны.

На специфическом уровне заданы пакеты со стереотипами *Исследовательская установка* («*Research Instruments*») и *Мир модели* («*World*»). Стереотип *Измерение* («*Measurement*») применяется к отношению зависимости, связывающему данные пакеты. Мы определяем это отношение как контролируемое нарушение инкапсуляции классов пакета «*World*». Стереотип *Мир модели* отражает исследуемую систему и ее окружение и содержит описание имитационной модели в традиционном понимании. Пакеты со стереотипами *Epistemology Entity* и *Ontology Entity* общего уровня задают методологии проведения измерений и экспериментов и законы функционирования системы в стереотипах UML. Их вычислительная семантика определяется как повторно используемые компоненты линейки ПО имитационной модели. В пакетах онтологического раздела запрещено использовать какие-либо классы, кроме помеченных меткой Concept (есть два исключения, см. далее). В корневой пакет «*Universe*» заносится класс Magnitude и все или некоторые из его потомков. Это обеспечивает доступ всех пакетов к данному классу. Предметная семантика этих классов определяется как агент взаимодействия гносеологических и онтологических объектов и предназначена для определения общего типа интерфейса. Выбор этих классов определяет набор доступных измерительных шкал (классификационные, сравнительные и количественные шкалы).

Для определения концептов архитектурных стереотипов будем опираться на концепцию «четырех миров», предложенную К.К. Колиным (доклад на 10-м заседании семинара «Методологические проблемы наук об информации» под названием «Философия информации: Структура реальности и феномен информации»**;** Москва, ИНИОН РАН, 7 февраля 2013 г., см. [31]). Обсуждая вопрос о сущности информации, К.К. Колин предложил двухуровневую модель реальности. На верхнем уровне два компонента – физическая и идеальная реальность. На втором уровне идеальная реальность представлена тремя компонентами: *объективная идеальная реальность первого рода* (ИР-1), *субъективная идеальная реальность* (ИР-2) и *объективная идеальная реальность второго рода* (ИР-3).

В контексте методологии MSP мы будем понимать данную концепцию следующим образом. Концепт стереотипа *Research Instruments* определим как ИР-2 («феномен сознания *человека,* а также продукты деятельности сознания, *существующие внутри него*») с той поправкой, что «феномен сознания человека» выражен в инструментах научного познания. Концепт стереотипа *Epistemology Entity* определим как ИР-3 («совокупность нематериальных продуктов деятельности сознания, *находящихся вне*

его»). Под ИР-1 понимается идеальная реальность, которая «возникает в результате взаимодействия отдельных компонентов физической реальности и их *взаимного отражения*». Примем это как определение концепта «*Ontology Entity*». Пакет «*Ontology Entity*» содержит абстрактные классы, которые не могут иметь экземпляров. Приняв данное определение, мы можем сказать, что сущности данного пакета моделируют элементы объективной идеальной реальности первого рода. Понятие «физическая реальность» назначим концепту «*World*». Зависимость пакета «*World*» от «*Ontology Entity*» будем понимать как проявление феномена так называемой «физической информации». Классы Magnitude, доступные всем пакетам *Универсума*, по-видимому, тесно связаны с обобщенным определением информации, предложенным К.К. Колиным (это определение применимо как к материальным, так и к идеальным компонентам реальности). К.К. Колин придерживается атрибутивного подхода к определению понятия информация: «Совокупность … различий и есть информация» (в смысле А.Д. Урсула).

Итак, мы определяем четыре стереотипа для архитектурных пакетов. Два из них («*Research Instruments*» *и* «*Epistemology Entity*») мы рассмотрим в третьей главе монографии, а сейчас обратимся к подробному рассмотрению пакетов, помечаемых стереотипами *World и Ontology Entity.*

Архитектурный пакет «*World*». Этот пакет моделирует изучаемую систему. Пакет содержит конкретные классы и диаграммы деятельности, отражающие алгоритмы поведения системы. Его содержимое есть результат декомпозиции системы на подсистемы.

Структурная декомпозиция занимает центральное место в системном анализе и известна достаточно хорошо. Этого нельзя сказать о декомпозиции поведения. По нашему мнению, для обоснования подбора стереотипов можно воспользоваться одной из логик изменения. Мы предлагаем следующий подход.

Алгоритмическая логика – это формальная система, которая допускает различные интерпретации. С точки зрения вычислительной семантики данную формальную систему мы будем интерпретировать как *логику программирования*, а с точки зрения предметной семантики – как некую *логику изменения*. Логику мы будем понимать в ее первоначальном смысле – как «науку о правильном мышлении», «искусство рассуждения». Наша цель исключительно утилитарная – использовать логику изменения как набор правил декомпозиции *активности*.

Формулы логики – это тройки $\{P\}$ S $\{Q\}$ (тройки Хоара), где P и Q – предикаты, а S – оператор или список операторов. Интерпретация логики отображает каждую формулу в ложь или истину.

Интерпретация тройки с точки зрения вычислительной семантики. Тройка $\{P\}$ S $\{Q\}$ истинна при условии, что если выполнение S начинается в состоянии, для которого высказывание P истинно, и завершается, то для постусловия высказывание Q истинно.

Интерпретация тройки с точки зрения предметной семантики. Тройка $\{P\}$ S $\{Q\}$ истинна при условии, что ситуация, для которой истинно высказывание P, исчезает, а ситуация, для которой истинно высказывание Q, возникает. Процесс S исчезновения и возникновения будем называть *активностью*. Пример: перемещение пешехода из пункта A в пункт B.

Подобная интерпретация алгоритмической логики приводит к временной (темпоральной) логике. Действительно, тройка Хоара очень похожа на формулу фон Вригта $A\mathbf{T}B$, которая читается так: сейчас A, а в следующий момент B. Поэтому можно поступить иначе – взять за основу некоторую временную логику, например логику Пнуели, и определить для этой логики вычислительную семантику [23], [30]. Заметим, что логика времени является составной частью логики изменения. Мы решили взять за основу именно алгоритмическую логику потому, что такой подход ближе к стилю изложения данной книги. Для декомпозиции поведения воспользуемся алгоритмической логикой [62] и приведем ее краткое описание (подробнее см. [23]). Наиболее важными правилами

вывода являются: аксиома присвоения, правило последовательной композиции, правило оператора **if**, правило оператора **while**, правило следования и правила, связанные с параллелизмом. Их вычислительная семантика соответствует семантике языков программирования. Предметная семантика будет следующая.

Оператор присвоения. Аксиома присвоения: $\{P_{x\leftarrow e}\}\ x = e\ \{P\}$, где $P_{x\leftarrow e}$ есть замена переменных x на выражение e (текстуальная подстановка). Смысл этой аксиомы поясним на частном случае $\{P_{a\leftarrow b}\}\ a = b\ \{P\}$, где a, b – некоторые объекты. Пусть утверждение P для объекта b истинно, тогда после присвоения $a = b$ утверждение P будет истинно применительно к объекту a. Тем самым, эта аксиома постулирует такую разновидность активности, когда один объект подменяется на другой. Относительно предиката P объект a неотличим от b. Можно также сказать, что эта разновидность активности есть копирование.

Оператор композиции (;).

Правило композиции. $$\frac{\{P\}\,S_1\{Q\},\{Q\}S_2\{R\}}{\{P\}S_1;S\ \{R\}}$$

С точки зрения предметной семантики оператор композиции постулирует такой вид активности, которая соответствует понятию *последовательность*. Пример из теории динамических систем – свойство фазовой траектории, которое заключается в том, что фазовую траекторию можно разбить на два участка.

*Оператор **skip***. Для этого оператора имеет место аксиома: $\{P\}$ **skip** $\{P\}$.

С точки зрения предметной семантики оператор **skip** постулирует существование такого вида активности, который может быть определен термином «статика». Для этого оператора предикат P является инвариантом. Пример из теории динамических систем – устойчивый или неустойчивый узел. Этот вид активности тесно связан с понятиями *активное ожидание*. В системной динамике Дж. Форрестера этот вид активности нашел свое отражение в концепции *запаздывания.*

Приведем следующий код

```
int x = 0; // высказывание «x == 0» истинно
while (true) {
 x = SetValue(x);
 if (x>0) x = x - 1; // эти два оператора задают оператор skip
 if (x<0) x = x + 1; // относительно предиката x == 0
};   .
```

Функция SetValue(x) возвращает входное значение x. Если функция SetValue вернет в какой-то момент значение x отличное от нуля, то через некоторое конечное число итераций x станет равным 0. Если в данном коде изменить условия на противоположные, то получим неустойчивый узел (репеллер).

*Оператор **if***.

Правило оператора if. $$\frac{\{P\wedge B\}S\{Q\},\{P\wedge\neg B\}\Rightarrow Q}{\{P\}if(B)S;\{Q\}}$$

С точки зрения предметной семантики оператор **if** постулирует такой вид активности, который может быть определен термином «управление». Тогда предикат B – параметр управления. В системной динамике Дж. Форрестера этот вид активности отражен в концепции *функций решения*.

*Оператор **while***.

Правило оператора while. $$\frac{\{I\wedge B\}S\{I\}}{\{I\}while(B)S;\{I\wedge\neg B\}}$$

С точки зрения предметной семантики оператор **while** постулирует такой вид активности, которая может быть определена термином «стационарность». Пример из теории динамических систем – предельный цикл (устойчивый и неустойчивый). В системной динамике данный вид активности называется *петлей обратной связи*. Можно говорить о петле обратной связи с отрицательной (аттрактор) и положительной (репеллер) связью. Данное правило предполагает существование некоторого инварианта *I*, разного для разных *S*. Если предикат *I* истинный перед выполнением тела цикла, то выполнение *S* должно сделать *I* опять истинным.

Приведем еще одно правило вывода – **правило следования**.

Правило следования. $$\frac{P' \Rightarrow P, \{P\}S\{Q\}, Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

Это правило можно интерпретировать как правило «сопряжения» диаграмм деятельности. Например, если пешеход из пункта *A* перемещается в некоторую точку *y* пункта *B*, то пешеход из точки *x* пункта *A* (усиленное предусловие) перемещается в пункт *B* (ослабленное постусловие). Это правило мы будем использовать для обеспечения целостности дерева декомпозиции.

Еще два важнейших правила связаны с параллельностью. К сожалению, параллелизм как свойство реальности изучено очень и очень мало. В логиках изменения (точнее в логиках направленности) рассматривают четыре типа существования: бытие, небытие, возникновение, исчезновение. В связи с параллелизмом мы вынуждены изменить трактовку термина «небытие» и отличать его от отрицания «не существует». Это ясно из следующего примера: поток, связанный с объектом, может быть создан, но приостановлен. Правильно говорить, что объект существует, но существует в небытии.

*Оператор **await***. Синхронизация определяется посредством оператора **await**: $\langle$ await(*P*) *S*;$\rangle$, где *P* – булево выражение, задающее условие задержки; угловые скобки указывают, что *S* выполняется как *неделимое действие*. Сокращенная форма $\langle S;\rangle$ определяет только взаимное исключение, а сокращенная форма $\langle$ await(*P*); $\rangle$ – только условную синхронизацию.

Правило оператора await. $$\frac{\{P \wedge B\}S\{Q\}}{\{P\}\langle await(B)S;\rangle\{Q\}}$$

В основу интерпретации этого вида активности с точки зрения предметной семантики положим следующие простые рассуждения. Пусть заданы потоки *a*, *b* и *c*. Пусть в некоторый момент *t* поток *b* входит в критическую секцию и в некоторый другой момент *t′* выходит из критической секции. В интервале [*t*, *t′*] потоки *a* и *b* находятся в приостановленном состоянии. С точки зрения наблюдателей, связанных с потоками *a* и *c*, интервал времени [*t*, *t′*] будет событием. С точки зрения наблюдателя потока *b* – активностью. С переходом к квазипараллельному описанию эта ситуация сохраняется, правда в более завуалированной форме.

Определение предметной семантики темпоральной декомпозиции возможно на основе концепции темпомиров С.П. Курдюмова. В различных областях науки отмечены факты, когда структуры и системы имеют разные темпы развития («горящих с разной мощностью»). Для обозначения этого феномена иногда используют понятие «темпомир» (см., например, [33], [2]). Предельную, абсолютную форму этого понятия можно назвать *абсолютным темпомиром*. С точки зрения *наблюдателя контекста*, абсолютный темпомир представляет собой событие, не имеющее продолжительности. С точки зрения *наблюдателя темпомира*, контекст абсолютного темпомира представляет собой статическую картину, а изменение доступных переменных контекста запрещено. С точки зрения наблюдателя темпомира, абсолютный темпомир определяется как сеть событий и причинно-следственных связей и описывается ориентированным графом (по Ф.А.

Новикову – нагруженным мульти-псевдо-гипер-орграфом), представлением которого в UML является диаграмма деятельности (Activity diagram) или диаграмма состояний (State diagram). *Абсолютный темпомир* не является объектом, хотя и может быть инкапсулирован в объекте. С переходом к квазипараллельному описанию интерпретация темпомира становится еще более наглядной. В языках программирования (вычислительная семантика) *абсолютный темпомир* моделируется процедурой; создание *абсолютного темпомира* – это вызов процедуры из текущей процедуры. Абстракция темпомира позволяет представить природный процесс как совокупность вложенных, параллельных, последовательных, синхронизированных, асинхронных, открытых, замкнутых темпомиров и тем самым снимает многие логические проблемы непротиворечивого описания. Абстракция абсолютного темпомира – это такая же абстракция как, например, материальная точка или абсолютно черное тело.

Оператор ***co***. Оператор **co** указывает, что несколько операторов могут выполняться параллельно:

```
co оператор1;
// …
// операторN;
oc
```

- начать параллельное выполнение операторов, помеченных символом параллелизма //, и ждать их завершения.

Правило оператора co.
$$\frac{\{P_i\} S_i \{Q_i\}\ \textit{свободны от взаимного вмешательства}}{\begin{array}{l}\{P_1 \wedge \ldots \wedge P_n\} \\ co\ S_1; // \ldots // S_n; oc \\ \{Q_1 \wedge \ldots \wedge Q_n\}\end{array}}$$

С точки зрения предметной семантики это правило можно рассматривать как определение такой разновидности активности, которая выражается термином «параллелизм». Ограничение «потоки свободны от взаимного вмешательства» позволяет определить взаимодействия в системе, как по горизонтальным, так и по вертикальным связям. Формальное определение взаимного вмешательства см. в [62].

Определение стереотипов. Рассмотренный выше аналог логики изменения не входит в метамодель профиля. Вместо данной логики можно использовать другие логики изменения. Тем самым нельзя утверждать, что определенные выше разновидности активности и правила декомпозиции отражают истинные законы природы. Тем не менее, мы воспользуемся этой логикой для определения концептов стереотипов, и будем рассматривать эту логику как часть методологии MSP.

Стереотип Ontology Activity. Концепт стереотипа *Ontology Activity* определим, как уже было сказано ранее, формулой $\{P\}\ S\ \{Q\}$ рассмотренной выше алгоритмической логики. Договоримся о том, что в рамках «*Research Analysis Model*» вместо термина *деятельность* в онтологическом разделе будем использовать термин *активность*, а термин *деятельность* оставим за гносеологическими сущностями, т.е. данный стереотип применяется по умолчанию ко всякой деятельности онтологического раздела. Данный стереотип имеет смысл указывать на диаграммах деятельности и диаграммах состояний только в том случае, если на диаграммах указаны также измерительные процедуры.

Параллелизм. То, что объекты моделируемой системы существуют параллельно, фиксируется в утверждение, что подсистемы должны быть потомками класса, определяющего поток (т.е. потомками не TObject, а TThread). Все основные стереотипы пакета «*World*» есть следствие этого принципа.

Стереотип Exist. Этот стереотип используется для обозначения тех методов классов, которые определяют единицу дискретно-событийного времени объектов.

Процедуры методов определяют *квант существования систем*. С точки зрения вычислительной семантики – это процедуры, определяющие вычисления потока. В Borland C++ Builder поток определяется классом TThread, который имеет выделенный метод Execute (см. ПРИЛОЖЕНИЕ II). В языке Java класс Thread имеет выделенный метод Run. Таким образом, классы, моделирующие систему, должны быть потомками некоторого абстрактного класса (этот класс помечается стереотипом *Substance*), который имеет метод «*Exist*».

Квант существования системы – особый (выделенный) вид активности. Все, что было сказано относительно интерпретации оператора **await**, применимо и здесь. Под единичным *квантом существования* системы будем понимать атомарную (неразложимую) активность, наблюдаемую *наблюдателем контекста*, т.е. действие (action). *Время* – отображение конечного подмножества целых чисел на множество вершин графа. Это отображение выполняет специальный прибор (гносеологическая сущность) – счетчик времени, который следует отличать от хронометра. Тем самым, в *Scientific Profile* время не является онтологической сущностью, а представляет собой данные измерений.

В простейшем случае активность «*Exist*» определяется бесконечным циклом вида

```
do {
 // ... некие действия ...
 tick++; // счетчик тактов, генерирует внутренние события
 if (Terminated) { // действия завершения потока};
} while (!Terminated);  .
```

Если используется квазипараллельное описание, то тело цикла выполняется только один раз. *Исследователь* инициирует изучаемый процесс всякий раз, когда посылает в *мир модели* сообщение со стереотипом *Exist*.

Стереотипы категории «структура» («*Ontology Environment*», «*Ontology System*», «*Ontology Atom*»). Другое важное следствие параллелизма – существование объектов двух родов: *вещественых* и *информационных* объектов. Действительно, операция копирования не применима к потокам и, тем самым, к объектам, связанным с потоками. Например, для объектов можно написать следующий код:

```
Thing *pa = new Thing;
pa->characteristics = 0;
Thing b; b = *pa;   // используется копирующий конструктор.
```

Однако если класс Thing является потомком класса TThread, подобный код вызывает ошибку на стадии компиляции. В квазипараллельном моделировании подобные объекты можно моделировать классом, для которого копирующий конструктор определен явно, но объявлен как private.

В приведенном выше коде, по существу, имеют место два объекта: исходный *pa и его копия b. Такие объекты мы будем называть *информационными*, поскольку они допускают тиражирование. Напротив, объекты, которые не могут быть копированы, будем называть *вещественными*. Для этих объектов выполняется закон сохранения их общего количества, если только объекты не создаются и не уничтожаются. Для вещественных объектов допустимо копирование только указателей. Мы будем полагать, что абстрактный класс «*Substance*» фиксирует это свойство. Напомним, что аксиома присвоения – это одна из центральных аксиом алгоритмической логики. Несмотря на то, что эти два рода объектов имеют весьма разные свойства, модель коммуникативного акта применима в обоих случаях. Идея разделения потоков на вещественные и информационные наиболее четко была высказана Дж. Форрестором в его системной динамике. Интересно, что термин «коммуникация» в ряде словарей трактуется именно в

этих двух значениях: (а) как средства связи и (б) как средства сообщения (трубопроводы, пути, дороги).

Определим процедуры измерения для вещественных объектов. Мы предлагаем следующее наиболее общее решение. Рассмотрим класс, который не является потоковым классом, но полностью повторяет интерфейс исходного класса. В приведенном выше примере это будет класс IThing

```
class IThing {
public:
 int characteristics;
 const IThing& operator = (const IThing &m) {// операция '=' по умолчанию
       this->caracteristic = m.caracteristic;
       return *this;
 }
 const IThing& operator = (Thing &m) {
       m.Suspend();
       this->characteristics = m.characteristics;
       m.Resume();
       return *this;
 }
};   .
```

В этом классе перегружена операция присвоения и тем самым становится возможным получение мгновенных снимков потока Thing

```
Thing *a = new Thing(false);
IThing *b = new IThing; *b = *a;
IThing c; c = *b;   .
```

Объект *b класса IThing является информационным объектом, и его можно тиражировать (см. последнею строку кода). Можно не перегружать операцию присвоения, определенную по умолчанию (и лучше этого не делать); это сделано для большей наглядности. Для класса IThing необходимо также перегрузить операции равенства, отношения и арифметические операции, что позволит определить все необходимые измерительные процедуры. Мы для таких классов используем префикс I, подразумевая слово image, однако совпадение с общепринятым обозначением интерфейса также не случайно. Рассмотренное решение заставляет предположить, что необходимо определить отдельный стереотип, для определения таких классов. Однако мы не будем этого делать, поскольку предметная семантика этих классов не ясна.

По умолчанию все классы будем считать информационными объектами, если они не помечены стереотипами категории «структура». В эту категорию включим потоковые классы (потомки TThread) тройки объектной декомпозиции «*Ontology Environment*», «*Ontology System*» и «*Ontology Atom*».

Предметную семантику стереотипа *Ontology Environment* определим как «весь *мир*, кроме изучаемого объекта», и этот класс определяется на самом первом шаге декомпозиции. С точки зрения системного анализа объекты с этим стереотипом определяют функцию системы относительно окружающей среды и позволяют задать ее системные характеристики. Окружающая среда задается крайне схематично, однако ее связи с системой определяются с необходимой точностью. Как правило, анализируются горизонтальные и вертикальные связи. В простых случаях достаточно определить только начальные и граничные условия. Внутренним переменным класса «*Ontology Environment*» нельзя назначать концепты, а программный код составляется исходя из соображений удобства.

Стереотипом *Ontology System* будем помечать классы самой изучаемой системы и классы подсистем всех ярусов иерархии, кроме нижниго. Внутренним переменным класса,

локальным переменным и методам должны назначаться концепты (по крайней мере, в принципе), а программный код составляется исходя из соображений адекватности.

Предметная семантика стереотипа *Ontology Atom* определяется как конечный уровень декомпозиции изучаемой системы на подсистемы. Это могут быть акторы, интеллектуальные и рефлексирующие агенты. Композиция подобных объектов моделирует подсистему также весьма схематично, но их внешние связи определяются с необходимой точностью. Внутренним переменным класса «*Ontology Atom*» нельзя назначать концепты, а программный код составляется исходя из соображений удобства.

В настоящее время существует ряд математических теорий, опираясь на которые можно определить декомпозицию системы как формальную процедуру. Например, аппарат родов структур Н. Бурбаки. Очень интересный подход, основанный на этом математическом аппарате, развит Ю.И. Бродским [6]; см. сайт [40]. На наш взгляд, этот подход весьма тесно пересекается с проблемой декомпозиции систем, хотя сам автор придерживается несколько иной точки зрения.

Применение стереотипов. Сформулируем *методику* декомпозиции системы, которая в значительной степени опирается на рассмотренную выше логику изменения. Правила вывода мы будем рассматривать в обратном порядке – от следствия к условию. Тогда вместо правил вывода можно говорить о правилах декомпозиции активности.

Метод декомпозиции систем нашел свое отражение в таких методологиях, как IDEF0, DFD, а также в ООАП [8]. В имитационном моделировании особое значение имеет поведение систем, что требует обобщение этого метода на поведенческие аспекты декомпозиции. Для построения объектных моделей будем опираться на принцип, который можно назвать *принципом объектно-темпоральной декомпозиции.* Прежде всего, напомним, что система в UML SP – это система коммуникаций. Обычно, говоря о декомпозиции системы, подразумевают структурную декомпозицию. Мы под декомпозицией системы будем подразумевать выполнение декомпозиции как структуры, так и поведения системы. Результат декомпозиции системы представляет собой сеть коммуникаций, связывающую процессные центры, обрабатывающие сообщения, передаваемые по этой сети. Всякий раз, когда выполняется объектная декомпозиция системы, необходимо для этих объектов определять кванты их существования (т.е. единицы дискретно-событийного времени).

Прежде всего, необходимо определиться со способом описания параллельных процессов. Существуют два способа и, судя по всему, они эквиваленты. Пусть есть несколько процессных центров и несколько документов. Если отслеживать историю документа, то описание сводится к последовательности процессных центров. Если отслеживать историю процессного центра, то получим последовательность обработанных документов. В теории параллельного программирования первый случай называют *декомпозицией по информационным потокам*, второй – *декомпозицией по заданиям*. Декомпозиция по данным – это одновременная обработка двух или более документов одного типа; предполагается, что и обрабатывающих центров также больше. В бизнес-моделировании первый способ описания называется процессным подходом, второй – моделированием на основе организационно-штатной структуры. В механике сплошной среды первый способ называется методом Лагранжа, второй – методом Эйлера.

Процедура декомпозиции предполагает выполнение трех этапов: объектную декомпозицию, определение горизонтальных взаимодействий и определение квантов существования подсистем.

1. *Объектная декомпозиция* [8]. Декомпозиция системы должна быть проведена так, чтобы множества переменных стали непересекающимися [62]. Наиболее общее решение – это инкапсуляция переменных в объектах. Мы допускаем, что архитектурный паттерн, определяющий структурную декомпозицию – это паттерн *Composite*. Тем самым, модель изучаемой системы будет определяться стереотипами *Окружающая среда* («*Ontology Environment*»), *Система* («*Ontology System*»), *Атом* («*Ontology Atom*»), которыми будут

помечаться соответствующие классы. Это предположение соответствует основным положениям системного анализа и закреплено в определении стереотипов.

2. *Горизонтальные взаимодействия и горизонтальная синхронизация*. Поскольку компоненты системы взаимодействуют друг с другом посредством разделяемых переменных, то одной объектной декомпозиции недостаточно для обеспечения взаимного невмешательства. Объектная декомпозиция определяет модель системы, в которой взаимодействия отсутствуют. На данном этапе определяются связи между объектами, выделенными на первом этапе. Еще раз напомним, что система в UML SP – это система коммуникаций. Процесс коммуникаций описывается диаграммами кооперации. Каждая диаграмма кооперации должна быть экземпляром диаграммы классов, так что каждый объект есть экземпляр некоторого класса, а каждая связь – экземпляр некоторой ассоциации. Для строгого анализа взаимодействий можно использовать метод глобальных инвариантов [62].

3. *Вертикальная синхронизация*. После того, как определены горизонтальные взаимодействия, можно определить, какова должна быть структура атомарной операции ⟨await(P) S;⟩ выделенных объектов; т.е. определить, как должен выглядеть для основного потока квант существования подсистем. Нижележащий уровень декомпозиции будем рассматривать как абсолютный темпомир.

Эти атомарные операции нижележащего уровня должны быть синхронизированы с потоком текущего уровня. Уточним, что потоки не могут быть вложенными, но они могут быть синхронизированными. Мы будем исходить из *гипотезы темпоральной синхронизации иерархии*. Горизонтальные связи в системе образуют как асинхронные взаимодействия, так и синхронные взаимодействия. Напротив, вертикальные связи в системе образуют только синхронные взаимодействия. Действительно, в противном случае иерархическая система как целое существовать не может. Этот вопрос обсуждается, например, в книге И.В. Прангишвили [47] как «закономерность расхождения темпов жизненных функций элементов системы». Для проверки отсутствия взаимного вмешательства можно использовать *ослабленные утверждения* (правило следования) [62].

Декомпозиция выполняется рекурсивно. Процесс декомпозиции заканчивается тогда, когда дальнейшая декомпозиция *с точки зрения текущей задачи* моделирования становится излишней.

Рассмотрим пример декомпозиции системы, используя изложенную выше методику.

Прогулки Иммануила Канта. Пусть необходимо построить имитационную модель, которая описывает следующую ситуацию. Некий философ просыпается всегда в пять часов утра. В строго определенное время он выходит из дома для прогулки и ровно через час возвращается домой. В остальное время, свободное от сна и обеда, философ занимается размышлениями. В десять часов вечера философ ложится спать. Эта последовательность действий повторяется каждый день.

Первый шаг декомпозиции – контекстная диаграмма (рис. 7). В результате *объектной декомпозиции* получим два объекта – саму изучаемую систему (класс Koenigsberg, помечается стереотипом *Ontology System*) и «все остальное» (класс Setting, помечается стереотипом *Ontology Environment*). Объект класса Setting определяет начальные и граничные условия для объекта класса Koenigsberg. Основной цикл потока Setting имеет период 365 дней.

Взаимодействие по горизонтальным связям сводится к годовому изменению степени освещенности, которое задает изменение граничных условий. Поток Setting каждые сутки устанавливает свойство season класса Koenigsberg. Это взаимодействие описывается специальным вариантом паттерна *Producer*/*Consumer*, когда не только *Consumer* ждет записи в буфер season, но и *Producer* должен ждать считывания записи. Данный паттерн - одна из самых распространенных программных конструкций параллельного программирования. Если не вдаваться в подробности, то паттерн

Producer/*Consumer* представляет собой ни что иное, как обычную очередь, и используется для передачи данных посредством разделяемых переменных.

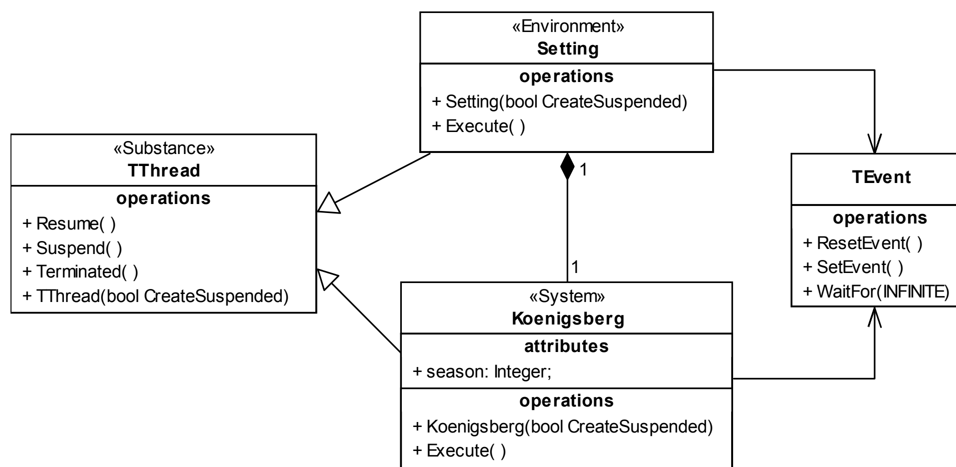


Рис. 7. Контекстная диаграмма

Для выполнения *синхронизации по вертикальным связям* необходимо обеспечить рандеву потоков Setting и Koenigsberg. Квант существования (атомарное действие) системы Koenigsberg определим как активность от одной точки рандеву до другой. Активность «*Exist*» (метод Execute) класса Koenigsberg моделирует суточное вращение и в нашей модели сводится к последовательному изменению значения булевой переменной day. Будем предполагать, что синхронизация происходит в полночь (day == false). Поскольку поток Setting не может вмешаться в вычисления потока Koenigsberg, с точки зрения наблюдателя Setting, активность «*Exist*» класса Koenigsberg представляет собой событие; выполнение некоторого оператора в основном цикле. По той же причине, с точки зрения наблюдателя Koenigsberg, поток Setting выглядит приостановленным.

Для обеспечения рандеву используем стандартную схему с двумя разделяемыми объектами синхронизации TEvent (см. ПРИЛОЖЕНИЕ II). Сначала событие e1 сброшено и поток Setting ждет, когда поток Koenigsberg придет в точку рандеву и установит его. Поток Koenigsberg приостанавливается. После чего поток Setting выполняет запись в свойство season класса Koenigsberg. Затем поток Setting устанавливает событие e2, что разрешает потоку Koenigsberg продолжить вычисления.

Второй шаг декомпозиции – декомпозиция контекстной диаграммы (рис. 8). В результате *объектной декомпозиции* получим объекты классов TowerClock и Townsman.

Жителей города моделируют экземпляры класса Townsman, а башенные часы – экземпляр класса TowerClock. Взаимодействие между жителями города и часами осуществляется через колокольню собора BellTower. Все вещественные объекты (Koenigsberg, TowerClock, Townsman) являются потомками класса TThread.

Горизонтальные связи. Взаимодействие между жителями города и башенными часами будем моделировать паттерном *Producer*/*Consumer* в его общей формулировке. Каждый житель города в роли *Consumer* ожидает наступления очередного часа, после чего синхронизирует свои часы. Если по какой-то причине он не синхронизировал часы, то соответствующее сообщение ставится в очередь. В роли *Producer* выступает объект класса BellTower. Поскольку жителей города много, а башенные часы ничего не знают о них, естественно воспользоваться паттерном *Observer* [14] (см. рис. 8). Класс Subject посредством метода Attach позволяет подписываться всем желающим жителям города (аргумент Observer*) на оповещение и посредством метода Notify оповещает их. Одной из конкретных реализаций класса Subject будет класс BellTower, который моделирует колокольню с колоколом (другая реализация – класс, моделирующий циферблат). Зная взаимодействие, необходимо определить способ *синхронизации*. Для объектов класса

Townsman мы будем использовать *активное ожидание*, т.е. активность каждого из объектов не приостанавливается, а сводится к циклической проверке ожидаемого события.

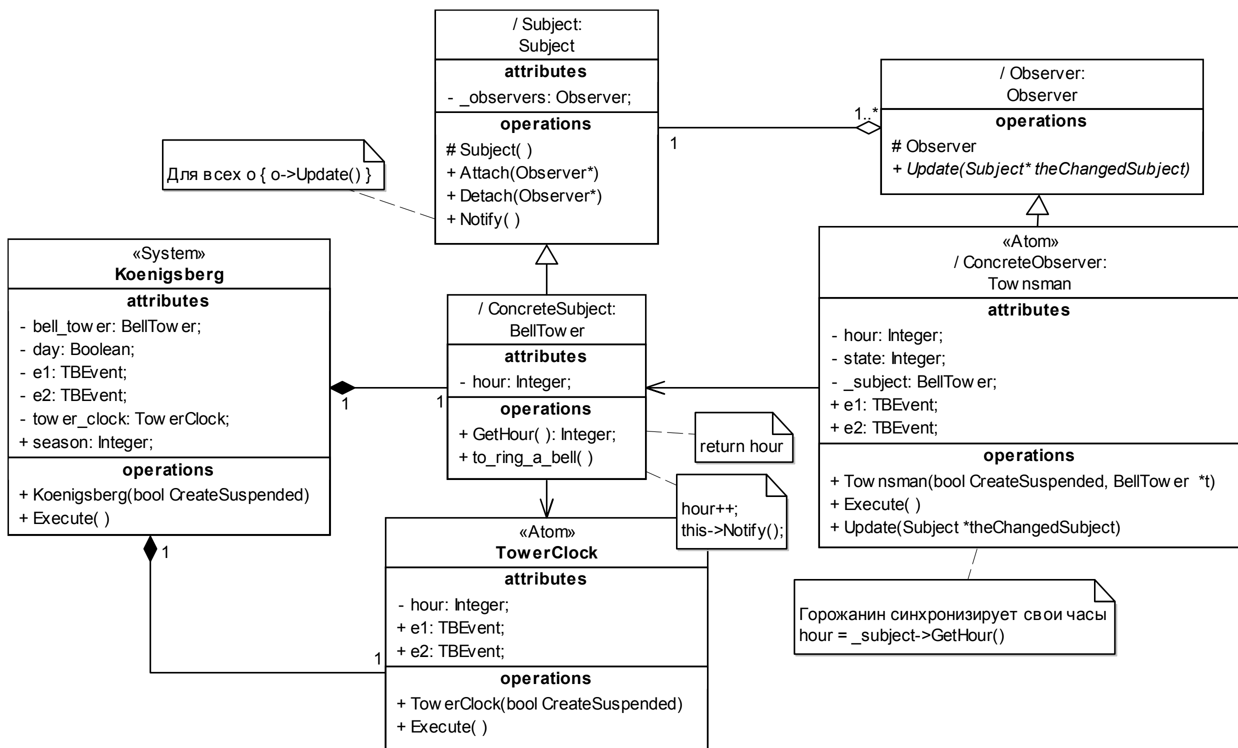


Рис. 8. Декомпозиция контекстной диаграммы

Вертикальные связи. Взаимодействие объекта TowerClock с системой Koenigsberg выражается в том, что сторож колокольни каждый день в астрономический полдень (day == true) устанавливает часы на 12 часов. Взаимодействие объектов класса Townsman с объектом Koenigsberg заключается в том, что физиологический процесс бодрствование/сон подстраивается под смену дня и ночи. Объекты имеют 24-х часовой цикл и всякий раз на 12-ой итерации приходят к точке рандеву. Точка рандеву у всех объектов одна и та же, поскольку рандеву моделирует взаимодействие с объектом Koenigsberg. Однако каждый объект может эту точку использовать по-разному, например, возможно взаимодействие с запаздыванием.

Для синхронизации потока TowerClock и потоков Townsman с потоком Koenigsberg будем использовать такую конструкцию, как *барьер* (barrier). Реализуем барьер как мультиобъектный вариант TEvent (класс TBEvent; этот класс можно собрать, используя TEvent). От класса TEvent этот класс отличается тем, что имеет внутренний счетчик. Объект Koenigsberg вызывает метод ResetBarrierEvent и устанавливает счетчик в значении, равном числу объектов синхронизации, а событие барьера – в сброшенное состояние. Всякий раз, когда синхронизируемые объекты вызывают метод SetEvent, счетчик понижается на единицу и поток приостанавливает себя. Когда счетчик становится равным нулю, устанавливается событие барьера. Объект Koenigsberg ждет, пока не произойдет событие барьера объекта TBEvent.

Как и на предыдущем шаге, активность «*Exist*» (метод Execute) классов TowerClock и Townsman рассматривается потоком Koenigsberg как атомарное действие, т.е. представляет собой событие. С точки зрения потоков TowerClock и Townsman время суток не меняется.

Третий шаг декомпозиции – определение атомарных объектов. В рамках данной задачи последующая декомпозиция является излишней. Поэтому все подсистемы, выделенные на предыдущем шаге декомпозиции, определим как атомарные объекты и

пометим стереотипом *Ontology Atom*. Методы, помеченные стереотипом *Exist*, уточняем любым удобным способом, например, кодируем так, как определено на предыдущем шаге.

Существенное теоретическое и практическое значение в построении имитационных моделей имеет решение вопроса о том, что есть физическое время. В научной литературе можно найти множество концепций времени. Мы будем придерживаться следующего определения: « … физическое время – это количественная мера упорядоченной эволюции материального объекта как целого от его возникновения до гибели» [49]. Из этого определения следует, что время – гносеологическая сущность, т.е. продукт процедуры измерения. Это в свою очередь означает, что время непосредственно не должно определять ход событий в модельном мире. В противном случае мы должны будем признать онтологический характер времени. Определение также указывает на то, что должно измеряться – эволюция материального объекта. Причем не просто какое-то локальное изменение, а изменение объекта как целого. Сразу оговоримся, что в сложных системах, таких как общественные системы, есть все основания для введения онтологического времени (например, модель хронотопа).

Мы предлагаем следующую измерительную процедуру. Необходимо отслеживать изменение *вещественного* объекта (потока). Однако невозможно сделать его копию и сравнить затем копию с оригиналом. Поэтому необходимо создавать информационные объекты-копии и сравнивать эти объекты друг с другом. Воспользуемся методом, рассмотренным выше. Построим класс, который не является потоковым классом, но полностью повторяет интерфейс исходного класса. Для нашего примера это будет класс IKoenigsberg

```
class IKoenigsberg {
public:
 int season;
 const IKoenigsberg& operator = (Koenigsberg &m) {
       m.Suspend();
       this->season = m.season;
       m.Resume();
       return *this;
 }
 bool operator == (const IKoenigsberg &m) {
       if (this->season == m.season) {return true;
       } else return false;
    }
};   .
```

В этом классе перегружена операция присвоения, и тем самым становится возможным получение мгновенных снимков потока Koenigsberg. Определенная по умолчанию операция присвоения не совпадает по сигнатуре с перегруженной операцией присвоения, и потому объекты IKoenigsberg могут быть копированы. Перегруженная операция равенства позволит сравнивать экземпляры класса IKoenigsberg друг с другом.

Теперь можно определить процедуру измерения контекстного или «космического» времени. В методе Execute класса Setting зададим две локальные переменные типа IKoenigsberg, и на каждой итерации цикла потока будем записывать туда текущее и предшествующее состояние объекта Koenigsberg. Если сравнение покажет, что оба объекта класса IKoenigsberg не равны друг другу, будем увеличивать переменную iTime на единицу. На множестве целых чисел задано отношение порядка, поэтому iTime может отражать «упорядоченную эволюцию материального объекта», что, и объясняет выбор этой переменной в качестве «количественной меры».

Время системы Koenigsberg должно измеряться так же, как происходит синхронизация системы с атомарными объектами. Наблюдаемая переменная – это переменная day. Для этого случая также может быть использован класс IKoenigsberg, который объявлен в классе Koenigsberg как дружественный. Тем самым появляется

возможность копировать и внутренние поля объектов класса Koenigsberg. Время атомарных объектов имеет смысл только как результат коммуникаций. Поэтому оно совпадает с показаниями часов жителей города. Во всех трех случаях применяются разные измерительные процедуры, что позволяет говорить о разновидностях времени. По-видимому, можно также говорить о мировом времени (время мира модели), понимая под этим синхронизированную иерархию всех трех разновидностей времени (год – сутки – час), или, точнее, как о результате применения комбинированной процедуры измерения. Заметим, что единое время мира модели не то же самое, что абсолютное время Ньютона.

Вернемся к нашему примеру. Дать краткое объяснение теории пространства и времени Канта затруднительно, поскольку многие аспекты теории неясны. По Канту время – априорная форма восприятия Мира, данная человеку с момента рождения (см. «Критика чистого разума»). Кант исходил из того, что наши ощущения имеют причины, которые он называет «вещами в себе». Восприятие состоит из двух частей: то, что обусловлено объектом, эту часть он называет ощущением, и то, что обусловлено нашим врожденным субъективным аппаратом. Эту последнюю часть он называет формой явления. Эта часть априорна в том смысле, что не зависит от опыта. Существуют две такие формы: пространство (для упорядочивания внешних ощущений) и время (для упорядочивания внутренних ощущений). Для моделирования философа создадим класс Philosopher, потомок класса Townsman. В этом классе метод Update замещен. Метод отличается тем, что содержит алгоритм (форма восприятия, данная человеку с момента рождения) последовательной записи событий обновления. Получается, что этот алгоритм и есть время по Канту. Точно так же ментальная карта местности, по которой совершает прогулки философ, есть пространство по Канту. В принципе, такое понимание времени согласуется с данным выше определением времени.

Итак, выше предложена методика объектно-темпоральной декомпозиции систем, в основе которой лежит одна из разновидностей логики изменения. Закончим на этом рассматрение архитектурного пакета «*World*» и обратимся к еще одному архитектурному пакету.

Архитектурный пакет «*Ontology Entity*». Определение классов этого пакета, возможно, наиболее трудоемкая часть рабочего потока *Анализ*. С точки зрения вычислительной семантики этот пакет определяет классы имитационной модели, пригодные для повторного использования в линейке имитационных моделей. Предметная семантика этого пакета значительно сложнее. Этот пакет описывает *теорию* изучаемого объекта и содержит классификацию, в которую входит моделируемая система; представлен сущностями, помечаемыми стереотипами «*Substance*», «*Taxonomy*», «*Meronomy*», «*Categorization*», «*Ontology Category*» и «*Ontology Space*». В простых случаях достаточно определить только два класса («*Substance*» и «*Ontology Space*»), причем переменные состояния системы («*Ontology Space*») часто допустимо моделировать базовыми типами используемого языка программирования (int, bool, char и т.п.).

Для обоснования предметной семантики обратимся к теории классификации, основанной на принципе двойственности таксономии и мерономии. Этот подход развит в работах С.В. Мейена и Ю.А. Шрейдера и в наиболее законченном виде изложен в книге [59]. Морфизм – это некоторый способ сравнения объектов. Таксономия – группировка объектов по сходству. Таксон – это множество видов (но не объектов классификации). На множестве таксонов заданы два отношения – отношение включения $\subset$ и отношение пересечения $\cap$. Таксономическая структура может быть иерархической и фасетной или же комбинированной. Мерономия – это сравнение объектов классификации по их строению. Мерон – это часть архетипа. Архетип – это план строения всех объектов таксона; некоторая конструкция из меронов. Архетип – это не структура. В физических системах архетип – это фазовое пространство системы. Математическую основу теории Мейена–Шрейдера составляет теория категорий. Множество таксонов является категорией C, где объекты категории – таксоны T, а морфизмы – отображения вложения таксонов.

Архетипы также образуют категорию *R*, где в качестве морфизмов выступают гомоморфизмы. Классификация – это функтор $\Im:C\rightarrow R$ из категории таксонов в категорию архетипов (функтор каждому таксону сопоставляет определенный архетип).

Определение стереотипов. Опираясь на эту теорию, введем стереотипы «*Taxonomy*», «*Meronomy*», «*Categorization*» (функтор), «*Ontology Category*» (таксон) и «*Ontology Space*» (мерон) (см. ПРИЛОЖЕНИЕ III). При таком выборе стереотипов отношение обобщения между классами «*Ontology Category*» можно рассматривать как метафору отношения включения между таксонами, а множественное наследование – как отношение пересечения. Таксономические структуры и архетипы не входит в метамодель UML SP, они создаются в процессе разработки модели. Пользовательские типы (классы «*Ontology Space*») определяют поля классов «*Ontology Category*»; либо непосредственно, либо в составе структур данных. Отношение зависимости «*Categorization*» пакета «*Taxonomy*» от пакета «*Meronomy*» отражает то, что внутренние переменные классов «*Ontology Category*» имеют типы «*Ontology Space*».

Под *пространством* будем понимать объект со стереотипом *Ontology Space*; предметная семантика определяется как объект для размещения других объектов, образующих структуру системы. Определяющий признак *пространства* – два и более объекта не могут быть в одной и той же ячейке пространства. Поэтому *пространство процесса* – вся совокупность переменных, доступных процессу. Вычислительная семантика «*Ontology Space*» близка к понятию «структура данных».

Применение стереотипов. Теория классификации Мейена–Шрейдера позволяет не только определить стереотипы категории «классификация», но и предоставляет некоторые методы для построения «*Research Analysis Model*».

Рассмотрим сначала *гносеологический раздел* модели анализа. В имитационных моделях операциональные процедуры (сравнение, счет, измерение и т.д.) рассматриваются как процессы коммуникации между исследовательской установкой и объектом исследования. Из теории Мейена-Шрейдера следует, что необходимо определять процедуры определения признаков, процедуру принадлежности к таксону, процедуру определения архетипа, процедуру сравнения архетипов, процедуру функтора (по заданному таксону выводит архетип), процедуру принадлежности к классификации. Для определения операциональных процедур хорошо подходят паттерны *Iterator*, *Observer* и *Template Method*.

Вернемся к *онтологическому разделу* модели анализа. Один из методов выделения классов анализа в UP – использование паттернов. Для выделения классов категории «классификация» предлагается использовать паттерн *Bridge* (см. [9]). Программная конструкция использует два класса – класс-манипулятор (*Abstraction*) и класс-тело (*Implementor*), причем класс-манипулятор имеет поле, в котором хранится указатель на класс-тело. Интерфейс (класс-манипулятор) определяет внешние признаки классифицируемых объектов, а реализация – их строение. По определению, паттерн *Bridge* отделяет абстракцию от реализации и минимизирует связность пакетов «*Taxonomy*» и «*Meronomy*». В ассоциации классов, прежде всего, необходимо найти класс, который может играть роль *Abstraction*. Этот класс должен быть абстрактным и объявлять интерфейс. Кроме того, класс должен иметь поле, которое содержит указатель на абстрактный класс, играющий роль *Implementor*. После чего классы раскладываются по пакетам «*Taxonomy*» и «*Meronomy*». Природа, конечно, не обязана быть устроена так, как этого требует паттерн *Bridge*. Поэтому данный метод следует рассматривать как один из возможных подходов к выделению классов.

Проиллюстрируем сказанное на следующем примере.

Автобусный рейс. Наиболее распространенная система классификации – иерархическая. Пусть необходимо построить имитационную модель автобусного рейса с автовокзала некоторого города. Расписание движения автобусов составляет классификационную систему для данной задачи. Мы допустим, что с данного автовокзала

выполняются рейсы по одному междугороднему маршруту и двум пригородным маршрутам. Диаграмма классов для имитационной модели приведена на рис. 9.

Конкретный рейс моделируется экземплярами класса STripA1, который выполняется по маршруту SuburbanRouteA. Этому классу назначен концепт «Рейс A1». Метод Exist задает единицу дискретно-событийного времени; моделирует посадку/высадку пассажиров в данном населенном пункте (операция Update) и перемещение объекта класса Bus в следующий населенный пункт. Маршрут моделируется динамическим списком из экземпляров подклассов класса Place. Классы BusStopStation и BusStation различаются операцией UpdateImp – в первом случае обновляется только часть пассажиров, во-втором – все.

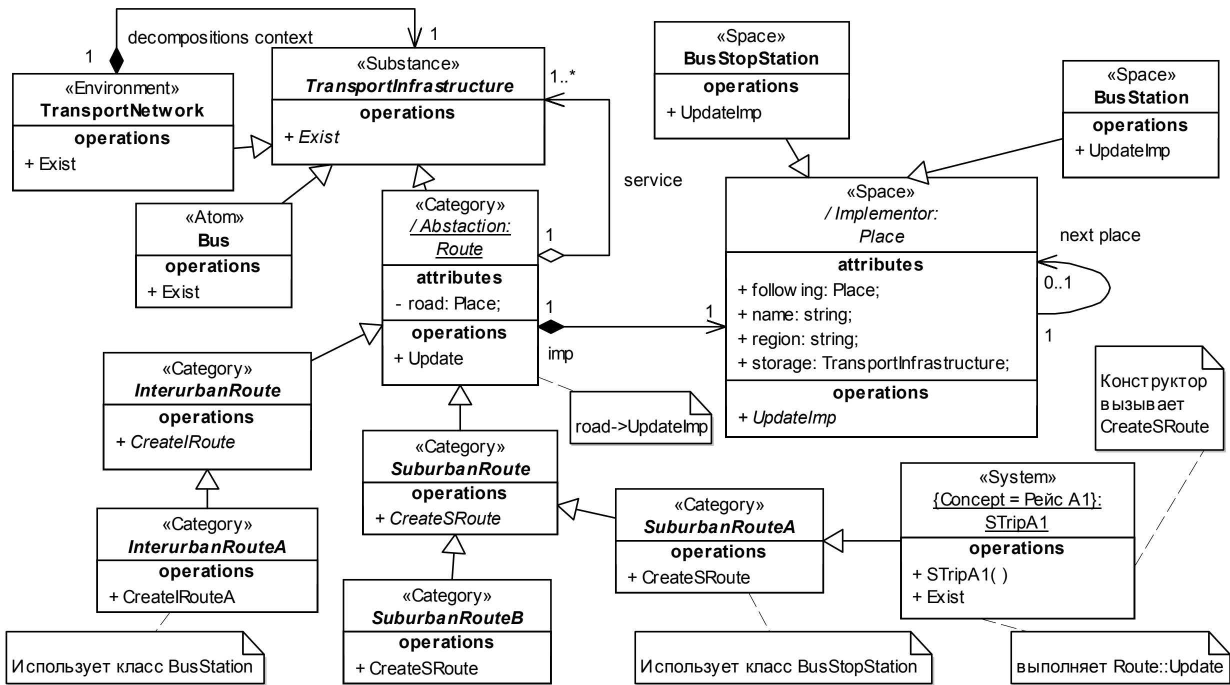


Рис. 9. Имитационная модель «Автобусный рейс»

Допустим, что диаграмма классов не размечена стереотипами. Воспользуемся описанным выше методом определения классов анализа. На роль *Abstraction* подходит класс Route, тогда класс Place будет играть роль *Implementor* (см. рис. 9). Предполагая типовую двухслойную архитектуру, разложим классы по основным пакетам модели анализа. Классы TransportNetwork, Bus и STripA1 поместим в пакет «*World*», остальные классы разместим в пакете «*Ontology Entity*» так как показано на рис. 10. Заметим, что на рис. 10 нет необходимости указывать стереотипы классов, поскольку пакеты как раз и группируют классы по категориям. Мы приводим стереотипы для большей наглядности.

После того, как выполнено разделение классов по пакетам, становится возможным определение их стереотипов (см. рис. 9). Из рис. 10 видно, что классификация зависит от выбора классов пакета «*Meronomy*», т.е. от способа определения функтора. Например, в нашем случае в качестве альтернативной мерономии можно было бы в качестве меронов выбрать дороги, через которые проходит маршрут. Таксономия не изменилась бы, но классификация была бы уже другая.

Для моделирования пересечения таксонов используется множественное наследование. Классический пример фасетной системы классификации – периодическая система химических элементов Д.И. Менделеева. Этот пример рассмотрен в работе [21]; см. сайт [40], где приведена объектная модель атомов химических элементов. В работе рассмотрена «чисто» фасетная система, т.е. рассмотрен упрощенный вариант модели. В общем случае надо рассмотреть все периоды, поскольку периодическая система Д.И.

Менделеева является комбинированной системой классификации. Заметим, что приведенная модель учитывает квантовые эффекты.

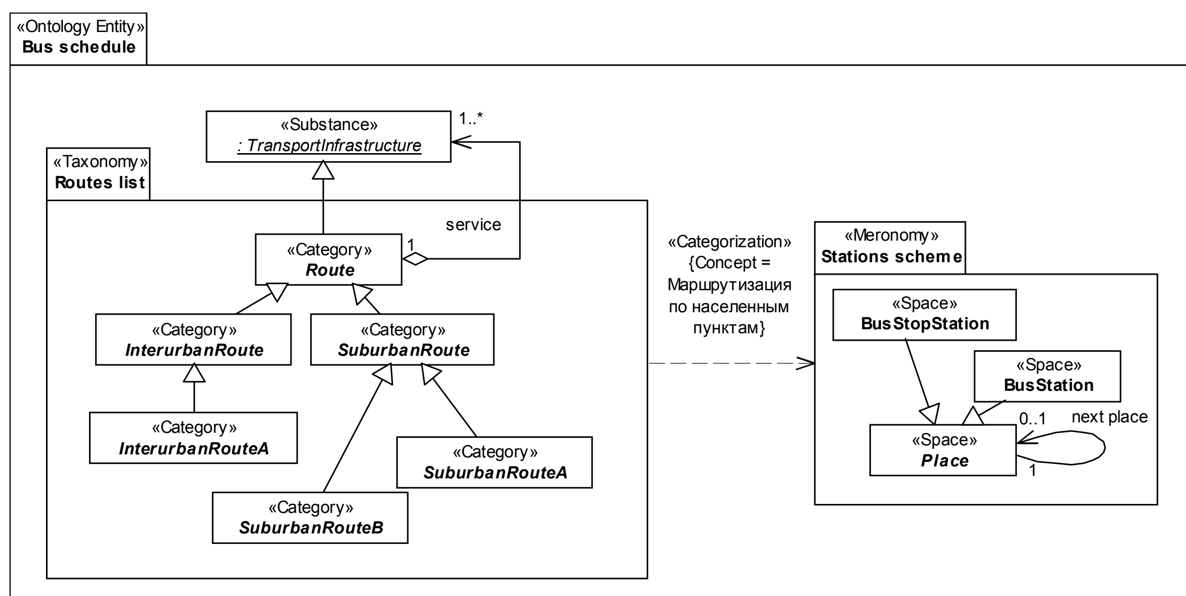


Рис. 10. Пример модели иерархической системы классификации

Как видно из рис. 10 классификация не исчерпывает всех сущностей пакета «*Ontology Entity*», хотя и составляет большую часть его наполнения. Это в первую очередь относится к классам, помеченным стереотипом *Substance*. С точки зрения вычислительной семантики, подобные классы – это абстрактные классы архитектурного паттерна. С точки зрения предметной семантики, концепт этого стереотипа может быть определен посредством такого математического понятия, как *каркас* [59], а также близкого понятия «системы, нарисованные на системах» В.А. Лефевра [37]. Каркас $K = \langle\langle M, \alpha\rangle, \Omega_2, A\rangle$ – это кортеж, содержащий модель $\langle M, \alpha\rangle$, сигнатуру Ω_2 и аксиоматику A. Применяется для определения моделей с отношениями из сигнатуры Ω_2 удовлетворяющих аксиомам A. Каркас можно понимать как модель, которая определена только частично, или как каркас модели. Авторы [59], говоря о моделях, представленных посредством каркаса, используют метафоры «ткань» и «рисунок» для модели $\langle M, \alpha\rangle$ и для отношений из сигнатуры Ω_2 соответственно. В вырожденном случае класс «*Substance*» должен объявлять, по крайней мере, один метод «*Exist*», так как бессмысленно рассматривать объекты, которые не могут существовать. Классы «*Substance*» могут образовывать иерархию наследования, что моделируется вложенными пакетами.

Закончим на этом рассмотрения «Research Analysis Model» и обратимся к последней модели UML SP.

«Research Design Model» определяет правила (способ) описания модели на конкретном языке программирования и позволяет отделить элементы имитационной модели от ограничений, налагаемых синтаксисом этих языков. В данной работе всюду предполагается использование C++ (за редким исключением, когда приходится обращаться к Smalltalk), что, конечно же, не исключает и другие языки программирования, включая специализированные языки GPSS World, AnyLogic и др. Особенностью потока работ по созданию модели дизайна в *Scientific Profile* является представление параллельных процессов, определенных в модели анализа, в виде последовательных операций, пригодных для описания на выбранном языке программирования. Затем вычислительный алгоритм должен быть исследован на точность и устойчивость. Модель дизайна создается в рабочем потоке *Проектирование*. Подчеркнем, сущности, вводимые в «*Research Design Model*», никакого «физического»

смысла не имеют, и им нельзя назначать концепты; в *Domain-Driven Design* такие классы называются *синтетическими*.

Остановимся подробнее на вопросе перехода к квазипараллельному процессу. Ясно, что между обоими процессами должна иметь место некая эквивалентность. Открытым остается вопрос: что следует понимать под эквивалентностью в данном случае?

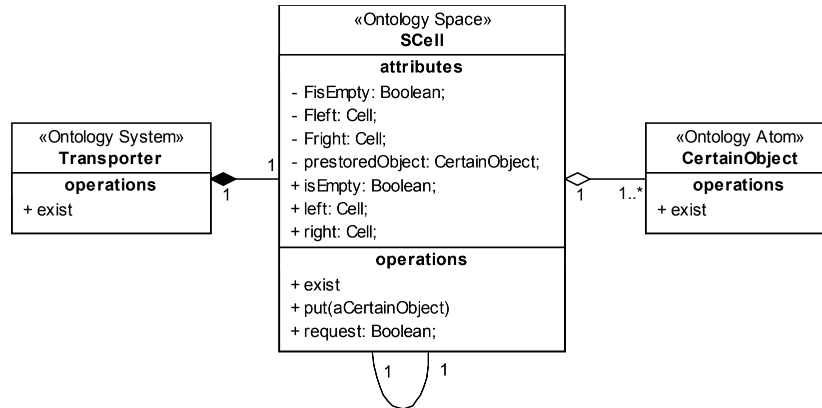


Рис. 11. Диаграмма классов для модели перемещения объектов

В качестве модельной задачи рассмотрим симуляцию перемещения объекта по ячейкам пространства. Ячейки являются экземплярами класса SCell, который имеет поле prestoredObject (указатель на хранимый объект), два свойства left и right и метод exist. Допустим, что объект может перемещаться только в одну сторону, например, вправо. Метод put с параметром aCertainObject передает указатель на хранимый объект одной (правой, в нашем случае) из смежных ячеек. Пространство будем моделировать динамическим списком, который образует кольцо. Диаграмма классов для этой модели показана на рис. 11.

В начальном состоянии объект aCertainObject находится в первой ячейке пространства. Всякий раз, когда *Исследователь* посылает сообщение exist, объект aCertainObject перемещается в правую ячейку.

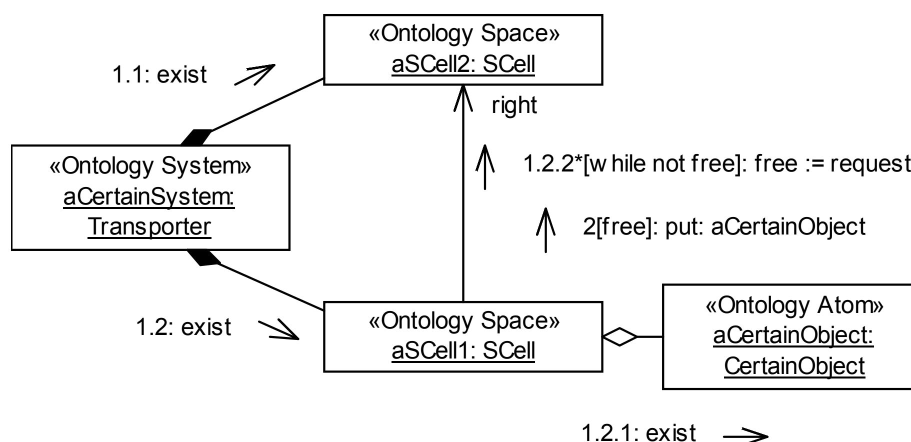


Рис. 12. Диаграмма кооперации для модели перемещения объектов

На каждом шаге дискретно-событийного времени (процессы метода exist) объекты обмениваются сообщениями так, как это показано на рис. 12. Заметим, что это существенно параллельная модель. Все процессы, инкапсулированные в объектах классов Transporter, SCell и CertainObject, считаются параллельными (см. ПРИЛОЖЕНИЕ II).

Основные модели параллельного программирования: *Процесс/канал* (*Process/Channel*), *Обмен сообщениями* (*Message Passing*), *Параллелизм данных* (*Data Parallel*), *Общая память* (*Shared Memory*). В нашем примере синхронизация может выполняться любым из этих способов. Например, *Процесс/канал* реализуется тем, что для метода put выделяется специальное поле, которое исполняет роль буфера ввода, и на следующем такте дискретно-событийного времени переписывается в поле prestoredObject. Механизм общей памяти может быть реализован посредством семафоров. Чаще других в имитационном моделировании используется механизм обмена сообщениями. Рассмотрим его подробнее.

Диаграмма деятельности показана на рис. 13. Процесс processing (обработать) выполняется тогда, когда ячейка находится в состояние full, а skip – в состояние empty (prestoredObject = NULL). Если ячейка full, то процесс начинается, определяет свое состояние как full, обрабатывает объект aCertainObject, передает указатель в смежную ячейку (сообщение с селектором put) и очищает поле prestoredObject:= NULL Если ячейка в состояние empty, то выполняется некоторое действие skip, выполняется прием сообщения из смежной ячейки, после чего полю prestoredObject будет присвоено значение сообщения (это может быть NULL).

Обратная задача распараллеливания, может быть решена одним из двух способов: опираясь на концепцию событийно-ориентированного или процессно-ориентированного моделирования. Введем фиктивное свойство isEnabled для класса CertainObject, которое может принимать значение true или false. В методе exist контекста последовательно проверим все ячейки по свойству isEmpty; если ячейка не пустая, установим свойство isEnabled = true. Затем повторно обойдем все ячейки. Если ячейка не пустая и isEnabled = true, то обработаем объект aCertainObject и установим isEnabled = false, затем передадим объект в смежную ячейку. Иначе – пропускаем ячейку. В противном случае aCertainObject будет перемещаться по ячейкам пространства мгновенно.

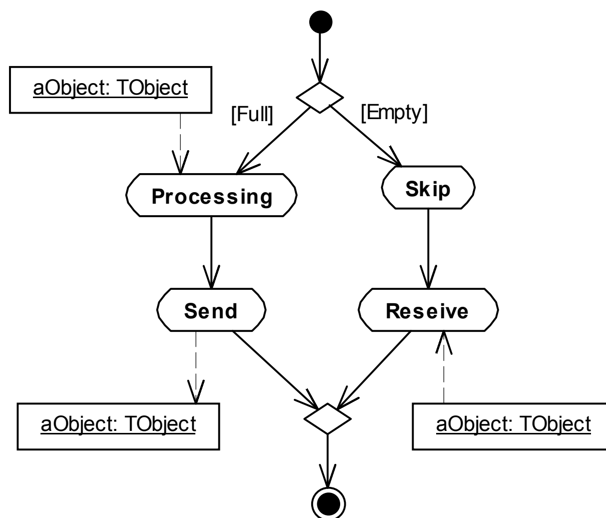


Рис.13. Диаграмма деятельности для активности пространственной ячейки

Можно допустить, что между композицией параллельных процессов «*Research Analysis Model*» и квазипараллельным процессом «*Research Design Model*» имеет место отношение *наблюдаемой конгруэнции* относительно операций процессной алгебры *CCS* (см. [17]). Конгруэнция означает то, что всякое допустимое преобразование одного процесса может быть выполнено и для другого; верно и обратное [67], [39]. Тем самым становится возможным изучение процесса «*Research Analysis Model*», работая с процессом «*Research Design Model*».

Итак, для обоснования подбора стереотипов UML SP мы использовали следующие научные концепции и теории: концепцию «трех миров» К. Поппера, концепцию «четырех миров» К. Колина, концепцию целостности и метод декомпозиции системного анализа, логику изменения, теорию классификации Мейена–Шрейдера. Эти теории и концепции не входят в метамодель UML SP, но составляют основу методологии MSP.

О языках программирования. Все современные объектно-ориентированные языки пригодны для создания имитационных моделей, в т.ч. такие популярные, как C++, Object Pascal (Delphi) и Java. В данной монографии практически всюду используется объектно-ориентированный язык C++. Этот язык склонен к сильной типизации, однако позволяет обойти порождаемые типизацией ограничения путем работы с указателями (как, впрочем, и в других языках). Не типизируемые языки, на наш взгляд, более выразительны с точки зрения имитационного моделирования. С другой стороны, типизация полезна и в имитационном моделировании, поскольку позволяет явно проследить таксономию пакета «*Ontology Entity*».

Именно в языке динамического программирования Smalltalk коммуникационная парадигма имитационного моделирования выражена лучше всего. Поэтому мы приведем краткое описание этого языка, тем более что в дальнейшем изложении будут встречаться некоторые понятия и термины, характерные для него. Некоторые темы рассматриваются в предположение, что реализация выполнена на Smalltalk.

Smalltalk был создан группой исследователей возглавляемой Аланом Кэйем в исследовательском центре Xerox PARC в 1970-х годах. В 1983 году была выпущена общедоступная реализация, известная как Smalltalk-80 Version 2 в виде образа (независимый от платформы файл, содержащий объекты) и спецификации виртуальной машины. Основная заслуга в реализации проекта принадлежит Ингалсу. Сейчас существует две реализации Smalltalk, являющиеся прямыми потомками Smalltalk-80 – Squeak и VisualWorks. Образ Smalltalk-80 version 2 запущен на Hobbes, виртуальной машине ST-80, реализованной на VisualWorks. Заметим, что VisualWorks – это программный продукт со свободной лицензией; его можно установить с сайта http://www.cincomsmalltalk.com/main/.

В языке есть всего три конструкции (см. http://ru.wikipedia.org/wiki/Smalltalk):

- посылка сообщения объекту,
- присваивание объекта переменной,
- возвращение объекта из метода.

и несколько синтаксических конструкций для определения объектов-литералов и временных переменных.

Основными идеями Smalltalk являются:

- Всё – объекты. Строки, целые числа, логические значения, классы, блоки кода, память – всё представляется в виде объектов. Можно сделать программу, которая работает с классами как с объектами, например, генерирует код класса, а затем загружает его. Выполнение программы состоит из посылок сообщений между объектами. Любое сообщение может быть послано любому объекту; объект-получатель определяет, является ли это сообщение правильным, и что надо сделать, чтобы его обработать.
- Динамическая типизация – это означает, что вы не указываете типы переменных в программе, а присваиваете переменным указатели на объекты. Тем самым одна и та же переменная может изменять свой тип в процессе выполнения программы. В языке C++ работа с указателями – альтернатива; в Smalltalk – это основная идея. Если сообщение не может быть обработано, то объект возвращает стандартное сообщение “Do not understand” и выполнение программы продолжается.

Мы часто ссылаемся на паттерны проектирования, изложенные в книге [14]. Для Smalltalk не все паттерны есть в указанном источнике, более полное изложение можно найти в [64].

Как показал опыт построения объектных моделей в ряде случаев одну и ту же модель имеет смысл построить на двух языках программирования. Желательно, чтобы один язык был типизируемым, а другой – не типизируемым. Это позволит избежать многих ловушек неадекватного моделирования, обусловленных особенностями конкретных языков.

Примеры и пояснения

1. ***Коммуникативный акт***

Как уже было сказано ранее, в монографии мы придерживаемся коммуникационной парадигмы имитационного моделирования. Поэтому нелишне напомнить основные модели коммуникативного акта [51], [29].

Модель К. Шеннона и У. Уивера. Одна из первых моделей коммуникации, предложена американскими учеными Клодом Шенноном и Уорреном Уивером (Warren Weaver) в конце 40-х годов.

Модель включает пять элементов: *источник информации, передатчик, канал передачи, приемник и цель* (место назначения), расположенные в линейной последовательности (линейная модель). В дальнейшем стало ясно, что в иных областях этой модели недостаточно. Другая модель включает шесть компонентов: *источник, кодирующее устройство, сообщение, канал, декодирующее устройство и приемник* («телефонная» модель). Помимо этого, Шеннон ввел такие понятия как *количество информации*, *шум и избыточность.*

В имитационном моделировании обеих моделей может оказаться недостаточно. В ряде случаев приходится прибегать к более сложным моделям, таким, как модель Якобсона.

Функциональная модель Р.О. Якобсона. В модели коммуникации или речевого события, по Якобсону, участвуют *адресант* и *адресат,* от первого ко второму направляется *сообщение*, которое написано с помощью *кода. Контекст* в модели Якобсона связан с содержанием сообщения, с информацией, им передаваемой, понятие *контакта* связано с регулятивным аспектом коммуникации. Оппозиция адресанта и адресата приводит к пониманию различения «лингвистики говорящего» и «лингвистики слушающего». Языковая реальность отправителя сообщения во многом не похожа на языковую реальность получателя сообщения.

Мы предлагаем следующую *минимальную* модель (рис. 14). Поскольку коммуникативный акт – базовое понятие UML SP, будем рассматривать модель, явно моделируя параллельные потоки.

Стандарт C++ ANSI непосредственно не поддерживает параллельное программирование, но в Borland C++ Builder существует возможность создания многопоточных приложений на основе класса TThreade (см. ПРИЛОЖЕНИЕ II). Пусть мир модели определен как экземпляр класса Setting. Систему будем моделировать экземпляром класса CommunicativeAct, а участников коммуникативного акта – экземплярами класса ActiveObject (см. рис. 14). Экземпляры классов Setting и ActiveObject имеют собственные потоки. Коммуникативный акт будем моделировать как рандеву (синхронное взаимодействие) потоков, связанных с экземплярами класса ActiveObject.

Рандеву организовано следующим образом. Метод потока Execute() класса ActiveObject имеет вид:

```
void __fastcall ActiveObject::Execute() { FreeOnTerminate = true;
do {
      if (tick == 5) this->Suspend();
      tick++;
```

```
} while (!Terminated);
}
```

и при условии tick == 5 порождает событие «точка рандеву достигнута», после чего поток приостанавливает себя (tick == 5 – это пример внутреннего события).

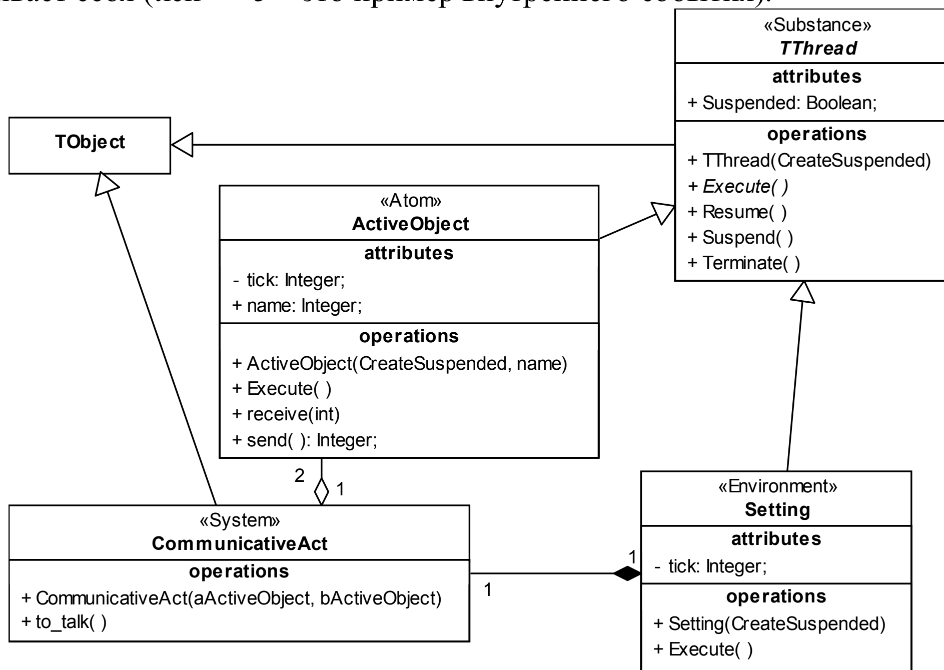


Рис. 14. Модель коммуникативного акта

В свою очередь, метод to_talk() класса CommunicativeAct имеет вид

```
void to_talk() {
if (a1->Suspended && a2->Suspended) {
      int m = a1->send();
      // … процесс передачи сообщения m …
      a2->receive(m);
      a1->Resume(); a2->Resume();
};
}   .
```

То есть, как только потоки обоих объектов достигнут точки рандеву, происходит пересылка сообщения. Поскольку оба потока приостановлены, ситуации гонок не возникает. После пересылки сообщений оба потока перезапускаются и продолжают свое выполнение. Выполнение метода to_talk() происходит из метода потока Execute() класса Setting.

В данной модели основное внимание концентрируется на организации коммуникативного акта, а не на процессах кодирования и декодирования. С точки зрения активных объектов пересылка сообщения происходит мгновенно. Модель является вырожденной: класс, помеченный «*Ontology System*» не имеет собственного потока, что указывает на фундаментальный характер модели. Модель позволяет учесть внешнее воздействие, в частности, шум. В рассмотренном примере сообщение имеет тип int; как правило, структура сообщения существенно сложнее – для моделирования сообщений определяются специальные классы сообщений.

После перехода к квазипараллельному описанию модель значительно упрощается и сводится к следующему коду:

```
a1-> Execute(); // метод «Exist»
int m = a1->send();
```

```
int n = m; //пересылка сообщения
a2->receive(n); a2-> Execute();.
```

Формула Лассуэлла. Процесс коммуникации может быть разделен на отдельные фрагменты, единицы коммуникации – *коммуникативные акты*. Г. Лассуэлл предложил формулу для анализа коммуникативного акта. В первоисточнике *(Harold Dwight Lasswell)* формула выглядит следующим образом: *Who says what to whom in which channel with what effect?* Данную формулу можно рассматривать как модель коммуникативного акта. Применим эту формулу к рассмотренному выше коду (см. табл. 2).

Таблица 2

Таблица анализа коммуникативного акта [29]

источник	сообщение и форма	адресат и тип	канал	цели и функции
кто?	что сообщает?	кому?	каким способом?	зачем?
1	2	3	4	5

1. Кто является отправителем/источником сообщения? Объект-отправитель a1.
2. Что является содержанием сообщения и в какой форме осуществляется коммуникация? Конкретное значение переменной m, тип int.
3. Кому направлено сообщение, кто является адресатом и каков тип коммуникации? Объект-получатель a2, типы коммуникации: между атомарными объектами, системой и атомарным объектом, между подсистемами.
4. По какому каналу передается и принимается сообщение? В нашем случае канал n.
5. Каковы цели и функции коммуникации в данном случае? Вызываемый метод. Чьи и какие потребности коммуникация обслуживает?

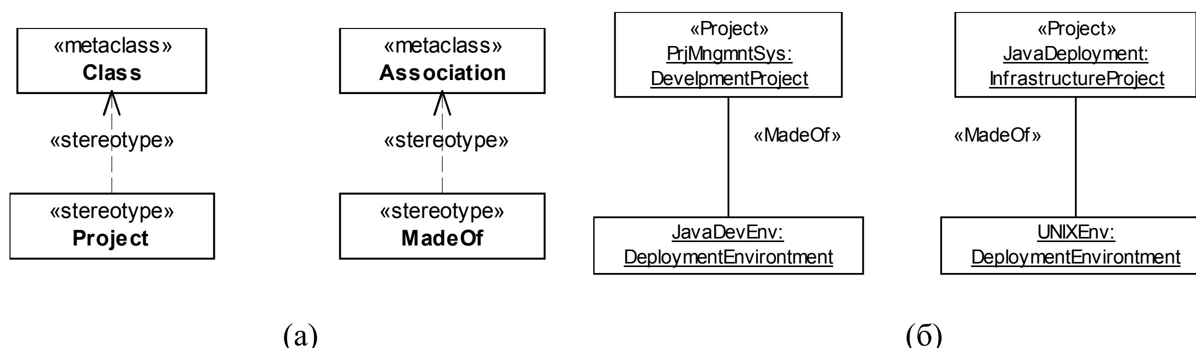
Самая простая коммуникационная система – это диалог, в котором один активный объект посылает сообщение другому активному объекту и дожидается ответа. Модель строится на основе модели коммуникативного акта, но, кроме того, используется паттерн *Strategy*, так как коммуникативный акт для вопроса и коммуникативный акт для ответа могут быть реализованы по разным алгоритмам и по разным каналам. Как, например, если один человек здоровается, используя речь, а другой отвечает ему кивком головы.

2. *Стереотип и метафора*

Если помеченные значения и ограничения не вызывают особых трудностей в понимании, то стереотипы требуют некоторых пояснений.

Напомним определение стереотипа. Стереотипы позволяют создавать новые элементы модели на основании *существующих*. Для этого используется элемент *класс* со специальным стереотипом «stereotype» и элемент *зависимость* с тем же стереотипом «stereotype», см. рис. 15. Зависимость указывает, на основе какого элемента метамодели UML создается новый элемент. Эти конструкции образуют метамодель системы стереотипов. В данном случае следует говорить именно о метамодели, потому, что она является моделью элементов модели. Метамодель нельзя объединять с обычными моделями. В *Scientific Profile* предполагается, что существует отдельная модель, предназначенная исключительно для стереотипов (см. ПРИЛОЖЕНИЕ III).

Пример на рис. 15 взят из книги Sinan Si Alhir, Learning UML, – Publisher: O'Reilly, 2003. – P. 252. Если необходимо, на рис. 15а можно указать помеченные значения (например, Name: String) и ограничения (например, {DataStart: String, DataEnd: String}).



(а) (б)

Рис.15. Стереотипы: (а) определение, (б) использование

Стереотипизация – это такое отношение между элементами, которое существенно отличается от других отношений UML и по своей природе имеет много общего с метафорой. *Метафора* (др.-греч. μεταφορά «перенос; переносное значение») – слово или выражение, использующая название объекта одного класса для описания объекта другого класса. Мышление в значительной мере опирается на метафору. Метафора рассматривается как понимание одного объекта через другой. Следующее предложение определяет метафору через метафору: «существует гипотеза, что греческое слово «метафора» происходит от слова «амфора» – сосуд для перенесения чего-то».

3. О вычислительной семантике

В профиле мы рассматриваем двойственную семантику элементов модели. Уточним понятие *вычислительной семантики*.

По-видимому, можно говорить всего о двух подходах к определению вычислительных семантик, а именно: о семантиках, ориентированными на компиляцию, и семантиках, ориентированными на интерпретацию (см., например, [34]).

Рассмотрим семантики, ориентированные на интерпретацию; их существуют три основных вида.

Во-первых, необходимо упомянуть об *операционных семантиках*. Смысл конструкций языка в таких семантиках выражается в терминах переходов той или иной абстрактной машины из одного состояния в другое. В качестве примера абстрактных машин обычно приводят так называемую SECD-машину П.Лендина, или категориальную абстрактную машину. В этой книге мы обычно подразумеваем именно эту семантику.

Во-вторых, следует сказать об *пропозиционных семантиках*. В этих семантиках значение конструкций языка выражается в терминах совокупности формул. Примерами являются, в частности, аксиоматический метод Хоара и метод индуктивных утверждений Флойда.

В-третьих, скажем об *денотационных семантиках*. Значение конструкций языка в этой семантике представляется в терминах абстракции функций, оперирующих состояниями программы. В частности, данный подход иллюстрирует теория вычислений Д. Скотта, основанная на семантических доменах.

С точки зрения имитационного моделирования понимание того, что есть «вычисление» (или «компьютинг»), конечно же, важно. Например, в рамках коммуникационной парадигмы, мысленный эксперимент (ментальная симуляция) тоже имитационный эксперимент, и что здесь понимать под вычислением далеко не ясно. Чтобы не углубляться в эту проблему, будем под вычислительной семантикой понимать

имитационную модель, рассматриваемую просто как компьютерную программу, а потому неявно будем допускать одну из приведенных выше семантик.

4. Механическое движение, изотахия и апории Зенона

Весьма непривычную физику мы получим, если обратимся к типичным физическим задачам.

Ванна Архимеда. Рассмотрим в качестве примера задачу о равновесии тела, погруженного в жидкость. Пусть система описывается классом Bath{Concept = Ванна}, окружение – классом Bathroom{Concept = Ванная}, а субстанциональный класс есть Material{Concept = Вещество}. Есть два атомарных объекта body и liquid; их классы – Body{Concept = Физическое тело} и Liquid{Concept = Жидкость}. Квант существования системы включает следующие процессы. Объект liquid обращается к свойству depth {Concept = Глубина погружения} объекта body и получает значение глубины погружения тела относительно поверхности жидкости как аргумент метода shift {Concept = Сдвинуться}. Объект liquid высчитывает выталкивающую силу F_a и сообщает объекту body как аргумент метода displace {Concept = Вытеснить}; объект bath в качестве второго аргумента передает величину силы тяжести F_g. Объект body сравнивает величину F_a с величиной силы тяжести F_g и изменяет значение свойства depth. Атомарные объекты будут обмениваться сообщениями до тех пор, пока вес вытесненной жидкости не сравняется с весом тела. Редукция коммуникации объектов на процессную модель дает не что иное, как численный алгоритм решения нелинейного уравнения $F_a(depth) - F_g = 0$. Описанный выше процесс – это модель статики. В динамической модели надо исходить из уравнения движения и явно учитывать вязкое трение, и если еще детализировать модель, то и волновые процессы в жидкости. Отметим различие между объектной моделью и математической моделью. В математической модели заданы начальные и граничные условия. В нашем случае явно описано окружение системы – класс Bathroom. В методе exist этого класса замеряется некая системная характеристика системы, например, высота жидкости в ванне.

Движение материальной точки. Для симуляции механического движения рассмотрим перемещение частицы по ячейкам дискретного пространства [13]. Напомним, что в этой книге мы не строим физических теорий, а только ищем адекватное описание физических процессов, опираясь на существующие модели. Модель механического движения, которую мы будем рассматривать далее, в основном основана на идеях Бэка, которые были высказаны еще в 1929 г. Далее предполагается реализация на Smalltalk.

Модель механического движения будем строить на стандартной двухуровневой архитектуре, включающей пакеты SI, Registrar, Mechanical World и Classical Mechanics. Первые два пакета описывают процедуры измерения механических величин и измерительные приборы. Модель представлена классами Fabric, GeometricSystem, MechanicalSystem, Particle и четырьмя активностями со стереотипом «*Ontology Activity*»: *Displacement* {Concept = Перемещение}, *Interaction* {Concept = Взаимодействие}*, Collision* ({Concept = Столкновение}) и *Disintegration* {Concept = Распад}. Пакет Mechanical World содержит реализации прецедентов «Приготовить начальное состояние», «Вычислить следующее состояние» и классы конкретной механической модели. Объект класса Cosmos моделирует контекст механической системы и определяет начальные и граничные условия. Объект класса Particle моделирует материальную точку и не является моделью элементарной частицы. Объекты класса Particle являются атомарными объектами, и говорить об их внутренней структуре не имеет смысла.

Концепт *Физическоое Пространство «Ontology Space»* предполагает определение ряда топологических и метрических свойств. Физическое пространство мы будем моделировать динамическим списком (объект space) из экземпляров класс Cell. Мы будем

рассматривать одномерное неограниченное пространство, имеющее правую и левую связанность. *Инфинитезимальные ячейки* пространства – это ячейки класса Cell, моделирующие часть пространства «от данной точки и до бесконечности». В такой ячейке поле right или поле left имеет значение nil. Мы будем подобные объекты понимать в смысле *нестандартного анализа*.

Рассмотрим алгоритм активности *displacement* {Concept = Перемещение} на примере свободного движения материальных точек. На рис.16 представлена диаграмма кооперации объектов при свободном движении материальной точки из ячейки cell1 в ячейку cell2, и описывается, *что* происходит, когда имеет место механическое движение.

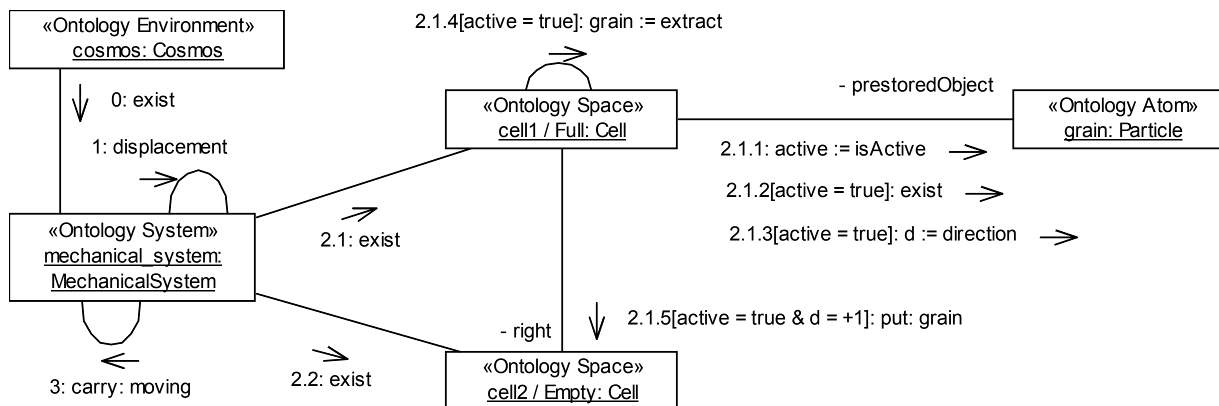


Рис.16. Диаграмма кооперации для свободного механического движения материальной точки

После получения сообщения exist объектом mechanical_system, активизируются два параллельных процесса – displacement и carry. Метод carry моделирует свободное движение механической системы. В процессе displacement всем ячейкам пространства одновременно посылаются сообщения exist и ячейки одновременно его получают. Если ячейка содержит частицу, то происходит следующее. Запрашивается состояние частицы; если частица активна, то частице посылается сообщение exist и затем запрашивается свойство direction. Затем частица изымается из ячейки и передается в смежную ячейку.

Остановимся только на двух моментах. Первый из них связан с тем, что необходимо признать эквивалентность механического движения и движения пространства. Процедура метода carry класса GeometricSystem моделирует движение пространства и представляет собой алгоритм смещения динамического списка space. Метод displacement класса MechanicalSystem перемещает частицу из одной ячейки в другую. Допустив эквивалентность обоих процессов, мы можем интерпретировать процесс carry как свободное движение механической системы. В частности, выбирая подходящее значение параметра moving, можно выбрать такую механическую систему, в которой частица будет покоиться.

Второй момент связан с пониманием таких свойств дискретного пространства-времени, как свойства изотахии, кекинемы и реновации (см. [13]). В рамках приведенной выше модели всем этим свойствам можно дать физическую интерпретацию. Например, свойство изотахии выражается в том, что все частицы могут иметь только одну скорость. Для того чтобы объяснить разные скорости у разных частиц, мы допускаем, что за квант существования системы частицы совершают не одно, а несколько перемещений. Причем медленные частицы совершают меньшее количество перемещений, чем быстрые. Обозначим также то, что рассмотренная модель позволяет разрешить апории Зенона, относящиеся к движению. Подробнее см. [19].

Отметим еще один важный момент. В физике большое значение имеют различные симметрии и потому необходимы строгие методы доказательства инвариантности имитационной модели. Например, инвариантность относительно пространственного сдвига доказывается следующим образом. Создадим новую механическую систему путем

сдвига пространства. Сравним объекты, задав процедуру измерения длины. Таких процедур будет две. Опираясь на аппарат процессных алгебр, можно показать, что оба процесса измерения находятся в отношении бисимуляции и тем самым между процессами имеет место сильная эквивалентность.

Шестеренки Максвелла. Максвелл, выводя уравнения электродинамики, пользовался наглядной моделью из шестеренок. Возможно, это один из первых удачных опытов применения имитационных моделей в физике.

Рассмотрим плоскую монохроматическую волну, природу которой пока не будем конкретизировать, и сопоставим ей класс Wave. Волна существует в контексте Cosmos. Введем два атомарных класса E-Field и M-Field. Пусть каждый атомарный объект может существовать в двух состояниях {0, 1}, имеет поле state и свойство state. Кроме того, атомарные объекты имеют метод vary{Concept = Измениться}, изменяющий поле state. Пусть объект m-Field посылает сообщение state объекту e-Field и получает значение (допустим 0) свойства в качестве аргумента метода vary. После чего объект m-Field изменяет свое состояние с 0 на 1. Затем все повторяется, только теперь сообщение state посылает объект e-Field. Получив значение 1, объект меняет свое состояние с 0 на 1. Таким образом, получаем осциллятор. Пусть класс Wave имеет поле space, представляющее собой динамический список из экземпляров класса Call. Разместим в каждой ячейке по экземпляру E-Field и M-Field. В результате в каждой ячейке получим осциллятор. Зададим связь между ячейками следующим образом: пусть объект m-Field, одновременно с запросом поля space у объекта e-Field, посылает сообщение vary объекту класса M-Field в соседней ячейке. В этом случае во всем пространстве возникает когерентная структура.

Звуковая волна моделируется проще. Для этого достаточно рассмотреть пространство, заполненное сцепленными осцилляторами, причем в каждой ячейке находится только один осциллятор. Самой простой моделью волновых процессов являются *булевы струны* [20].

Геометрическая оптика оперирует со своими объектами. Пусть строится изображение некоторого предмета в оптической системе с одной линзой. *Изображение* – это объект, который создается объектом *линза*, использующего объект *предмет* в качестве аргумента.

Воображаемая физика клеточных автоматов. После появления «Игры жизни» Конвэя стало окончательно ясно, что весьма простые системы, составленные из клеточных автоматов, могут демонстрировать весьма сложное поведение. Естественно, что должна была возникнуть идея объяснить физические явления с этих позиций (или более общее – с позиций самоорганизации). Наиболее грандиозная попытка построения такой физики, по-видимому, принадлежит С.Я. Берковичу [7]. Автор рассматривает дискретные уравнения как фундаментальные, а переход к дифференциальным уравнениям – как метод исследования. Этот прием мы часто используем в этой книге.

Алгоритмическая физика. Подход, созвучный нашему, обсуждается в монографии Ю.И. Ожигова [44]. Основная идея этой книги состоит в том, что вместо используемого в физике математического аппарата надо перейти к аппарату конструктивной математики, что, в свою очередь, приводит к понятию алгоритма. Ю.И. Ожигов такой подход называет *конструктивной физикой*. Приведем также такую цитату: «В описании алгоритмов содержатся фактические указания на механизмы физических законов» [38, с.44].

Концептуальная схема следующая. Вводится понятие сценариев – последовательности состояний исследуемой системы. Модель складывается из последовательности сценариев. Результат выполнения каждого сценария хранится в памяти компьютера и может использоваться в качестве начальных условий для другого сценария. Модель состоит из двух частей: административной и пользовательской. К административной части относятся алгоритмы, создающие визуальный образ. Пользовательская – сам этот образ. Рассматривается машина Тьюринга и вычисления с

оракулом. Автор оперирует с таким понятием, как эвристики, и понимает под этим термином некие идеологии построения алгоритмов. В монографии предлагаются алгоритмы описания явлений квантовой механики и квантовой электродинамики.

5. Экзотические пространства

Имитационные модели должны отражать то, что есть на самом деле. Валидность модели – то качество имитационной модели, которое представляет наибольшую ценность. Для моделей, которые не отражают действительности, даже существует специальный термин – симулякры (по Ж. Бодрийяру симулякры – копии, не имеющие оригинала, например, Адам и Ева). В то же время можно построить интересные конструкции, которые демонстрируют возможности объектного моделирования. В этом пункте мы рассмотрим некоторые из них.

Как было сказано в предыдущем пункте, основные концепты классической механики требуют глубокого обоснования. Анализ следует начинать с таких фундаментальных понятий, как пространство и время. Согласно определению концепта *Ontology Space*, пространство – это контейнер, т.е. объект, в который можно положить другой объект. В метрическом пространстве это уже не всегда можно сделать. Последнее как раз и позволяет определить измерительную процедуру «Размер» и затем «Расстояние».

Хорошо известные динамические списки в имитационном моделировании приобретают новый смысл. Их можно интерпретировать как объектные модели пространств. Несмотря на то, что это довольно простая объектная конструкция, с ее помощью можно построить весьма необычные пространства.

Пусть задан класс SCell, моделирующий ячейку пространства

```
class SCell {
public:
      void exist(){}
      SCell *space;
      SCell *left;  // левый сосед
      SCell *right; // правый сосед
      CertainObject *prestoredObject; // хранимый объект
};
```

Фрагмент кода

```
SCell *cell1,*cell2,*cell3;
cell1 = new SCell; cell2 = new SCell;  cell3 = new SCell;
cell1->left = NULL; cell1->right = cell2;
cell2->left = cell1; cell2->right = cell3;
cell3->left = cell2; cell3->right = NULL;
```

задает ограниченное пространство из трех ячеек с некоторой топологией. Линейный список уже обладает, кроме топологии, также упорядоченностью и метрикой - в пространство можно поместить счетное количество предметов, а потом их пересчитать.

Пример вложенного пространства (рекурсия), состоящего из трех ячеек:

```
SCell *head;
head->space = new SCell;
head->space->space = new SCell;
head->space->space->space = new SCell;
```

Отметим следующий тонкий момент. В данном примере не видна разница между списком и агрегацией. В случае списка поля left и right – это свойства (Public). Напротив, поле space должно быть Private и доступ к нему возможен только посредством специальных методов.

Комбинируя связанность и агрегацию, можно придумать множество всевозможных пространственных конструкций. Для исследования подобных пространств следует создавать прибор-анализатор (гносеологическую сущность) в виде программного агента, способного самостоятельно передвигаться.

Можно создавать пространства, структура которых меняется во времени. Например, пространство сворачивается в точку, а ячейки записываются рекурсивно в одну. Затем обратно разворачивается в линейный список.

А вот нечто, похожее на космологическую модель, некое пульсирующее пространство. Допустим, был создан список из некоторого количества ячеек (например, одной) так, как было показано выше. Рассмотрим код

```
/ ---- расширение пространства
SCell *curr; curr = cell1;
while (curr != NULL) {
      SCell *newCell = new SCell;
      newCell->right = curr->right;
      curr->right = newCell;
curr = newCell->right;}
// ---- сжатие пространства
SCell *p, *r; curr = cell1;
while (curr != NULL) {
      p = curr->right; // уничтожаемая ячейка
      r = p->right ;    // сохраняемая ячейка
      r->left = p->left; curr->right = r;
      delete p;
curr = r->right;}
```

На первой фазе каждая ячейка удваивается, а затем, на второй фазе, каждая четная ячейка уничтожается, и система возвращается в исходное состояние.

Загадка размерности. Казалось бы, трехмерное пространство (и вообще n-мерное) можно смоделировать, поместив указатели на один объект в три списка. Однако более глубокий анализ покажет, что это приводит к противоречию с определением концепта *Ontology Space*.

Тупики. Еще одна разновидность экзотических объектов, которые возможно очень распространены в природе. *Тупик*, *клинч* или *дедлок* (Deadlock) – это ситуация, когда один поток ждет другой поток, а последний – первого. Известный пример – сразу четыре автомобиля выезжают на перекресток, где пересекаются две равнозначные дороги. Каждый из них обязан ждать проезда автомобиля, который находится справа. Другой не менее известный пример – транспортная пробка в городе, возникающая в системе четырех дорог, образующих прямоугольник.

«Ноль» и «Омега». Конструктор можно интерпретировать как «Большой взрыв», который создает мир. Соответственно, деструктор интерпретируется как конечная точка существования мира.

Хаотический узор. Если внимательно посмотреть на абстрактный хаотический узор, подобный тому, какими бывают обои, то можно обнаружить в нем те или иные изображения («высший пилотаж» – увидеть анимированные картинки). Обычно это фигурки людей и животных. Если узор квазифрактальный, то эти картины будут меняться в зависимости от дистанции. Подобным образом устроен физический мир. В зависимости от того, с чем взаимодействует исследуемый объект, фрагмент реальности может становиться той или иной системой. Классы как раз и выступают в качестве средства описания этого процесса структурирования реальности.

6. Мир модели и декомпозиция. Концепт *World* (*Мир модели*) является прямым обобщением понятия *умвельта*, введенным Я.Д. Икскюльем в начале прошлого века. По мнению Икскюлья, каждое животное живет в уникальном, своем собственном

окружающем мире – Umwelt. Умвельт определяется типом организации данной живой особи и в первую очередь особенностями или спецификой рецепторов и аффекторов. В своей книге Икскюль пытается воссоздать жизненные миры некоторых видов животных, а также людей, и наглядно, в рисованных картинках, показывает, насколько по-разному тем или иным существам видится мир.

Понятия *контекста* и *декомпозиции* впервые достаточно четко было сформулировано в методологиях SADT и DFD. Первоначально была создана методология структурного анализа и проектирования (Structured Analysis/Structured Design – SA/SD), которая затем трансформировалась в метод SADT (Structured Analysis and Design Technique) и несколько позже вошла в стандарт IDEF как IDEF0. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Построение диаграмм начинается с контекстной диаграммы, которая затем подвергается декомпозиции. Для понимания этих диаграмм полезно функции представлять в смысле математических функций, а декомпозицию – как раскрытие условных обозначений в математическом выражении. Заметим, что диаграммы UML не покрывают функциональные модели полностью, а потому в некоторых случаях диаграммы SADT могут быть полезны.

Диаграммы потоков данных (DFD – Data Flow Diagrams) показывают, как происходит взаимодействие операций в бизнес-процессе. Диаграммы потоков данных известны давно. Приведем цитату из книги Г.Н. Калянова (Г.Н. Калянова, Консалтинг при автоматизации предприятий: подходы, методы, средства.). Рассказывают, что DFD были использованы для реорганизации переполненного клерками офиса еще в 20-х годах прошлого века. Осуществлявший реорганизацию консультант обозначил кружком каждого клерка, а стрелкой - каждый документ, передаваемый между ними. Используя такую диаграмму, он предложил схему реорганизации, в соответствии с которой двое клерков, обменивающиеся множеством документов, были посажены рядом, а клерки с малым взаимодействием были посажены на большом расстоянии. Так родилась первая модель, представляющая собой потоковую диаграмму – предвестника DFD

Контекстная диаграмма в DFD называется схемой внешней среды процесса. Процесс представляется в виде прямоугольника со скругленными краями (в системе BP Win). На схеме указываются основные контрагенты процесса как прямоугольники с тенью, а также входы и выходы процесса, изображаемые с помощью стрелок. Декомпозиция продолжается до уровня, когда будет достигнута необходимая степень детализации.

Понятие декомпозиции напрямую связано с появлением нового способа мышления, которое оформилось в так называемый *системный подход*. До двадцатого века наука знала только одну мысленную операцию – анализ, и обратную ей – синтез. Декомпозиция и агрегация – это другие мысленные операции, возможные только в том случае, когда есть понимание такого понятия, как *системность*.

В профиле вводятся концепты *Окружающая среда* («*Ontology Environment*» "metaClass" Class), *Система* («*Ontology System*» "metaClass" Class) и *Атом* («*Ontology Atom*» "metaClass" Class). В отличие от DFD и так же, как в SADT, контекст в объектных моделях не детализируется и, как правило, описывается в коде программы любым удобным образом. Однако протокол обмена сообщениями системы и внешней среды должен моделироваться предельно аккуратно. Как уже было сказано, это определяет показатели системы как целого, а значит и саму систему. В этом плане подход DFD вполне может оказаться полезным. Отметим, что в UML концепция декомпозиции выражена не так явно, как в DFD и SADT.

«*Ontology System*» является инвариантом относительно выбора *Исследователя*; «*Ontology Environment*» *и* «*Ontology Atom*», напротив, определяются положением *Исследователя*. Концепт «*Ontology Atom*» имеет много общего с концептом «*Ontology Environment*» и может рассматриваться как интерфейс с миром, расположенным в атоме.

Можно создать объектную модель, так сказать, с точки зрения атома. В этом случае мир атома станет окружающей средой для модели, а то, что было внешней средой – станет одним из атомов. Тоже можно проделать с любой подсистемой изучаемой системы. Коды программ во всех этих случаях будут разные, хотя изучаемая система одна и та же. Точно так же, как выбор системы отсчета меняет наблюдаемую скорость и положение материальной точки.

В связи с этим отметим одно из направлений в современной философии, называемое *эндофизикой*. «Эндофизика – это подход к изучению реальности, не взятой строго самой по себе, в чем ранее почти всегда состоял идеал естествознания, а с неустранимой насечкой – находящимся в ней наблюдателем» (*Alyushin A* Observing Reality on Different Time Scales // Endophysics, Time, Quantum and the Subjective / Ed by R Bucchen, A C Elitzur, M Saniga Proceedings of the ZiF Interdisciplinary Research Workshop, Bielefeld, Germany 17-22 January 2005 With CD-ROM World Scientific New Jersey, London, Singapore etc 2005 P 441-462). Одна из центральных идей эндофизики – мысленные эксперименты с виртуальным наблюдателем [2]. Наблюдатель выступает как базовый элемент мысленного эксперимента, в котором строятся воображаемые реальности. С наблюдателя снимаются все ограничения, которые накладывает человеческое восприятие. По мнению автора работы «инстанция наблюдателя в принципе не устранима из такой схемы».

В профиле также допустимо рассматривать множество сущностей «*Исследователь*». В определение этого стереотипа нет каких-либо ограничений. В частности, классы гносеологического раздела не обязаны быть потомками класса «*Substance*». Однако когда мы рассматриваем модели квантовой механики (или мегамира), приходится сделать следующее дополнительное ограничение. Эксперимент, измерение, наблюдение – это модель того, как одна часть физического мира взаимодействует с другой. Тем самым модели экспериментальной установки должны иметь такие конструкции, которые не противоречат законам физики.

7. Переход от математических моделей к имитационным моделям

Дискретность присуща предметам и явлениям природы. Неважно, придете вы за 15 или за 10 минут до отправления автобуса, точно так же, как неважно, пришли вы на 5 или 15 минут позже. В принципе любые отношения могут порождать дискретность. Например, если деталь не подходит к некоторой конструкции, то тот факт, что деталь на 10 сантиметров длиннее или 5 сантиметров короче, не имеет никакого значения.

Первым шагом в разработке имитационной модели является сбор информации о предметной области. Во многих случаях, по меньшей мере в науке, известна математическая модель системы или процесса. Как использовать эту информацию для создания имитационной модели? Типичным подходом является использование лингвистических переменных и формальных грамматик, которые позволяют достаточно строго выполнить *процедуру дискретизации*.

Воспользуемся понятием лингвистической переменной (см. Заде Л. Понятие лингвистической переменной и его применение к принятию приближенных решений. – М.: Мир, 1976.). Лингвистической переменной называется набор $\langle \aleph, T(\aleph), U, G, M \rangle$, где

- $\aleph$ – имя лингвистической переменной;
- *T* – множество его значений (терм-множество), представляющие имена нечетких переменных, областью определения, которых является множество *U*. Множество *U* называется базовым терм-множеством лингвистической переменной;
- *G* – синтаксическая процедура, позволяющая оперировать элементами терм-множества Т, в частности, генерировать новые термы (значения) с помощью связок "и", "или" и модификаторов типа "очень", "не", "слегка" и др. Например, "маленькая или средняя толщина", "очень маленькая толщина" и др.; Множество $T \cup G(T)$, где $G(T)$ - множество сгенерированных термов, называется расширенным терм-множеством лингвистической переменной;

- *M* – семантическая процедура, позволяющая преобразовать новое значение лингвистической переменной, образованной процедурой *G*, в нечеткую переменную, то есть сформировать соответствующее нечеткое множество.

Пример. Рассмотрим модель суточного вращения Земли (рис. 17, этот рисунок позаимствован из книги Я.И. Перельмана под названием «Занимательная физика»). Расположим систему координат в центре Земли и проведем радиус-вектор к центру Солнца. Посмотрим на систему с северного полюса (лучше – с Полярной звезды); Земля будет вращаться против часовой стрелки. Проведем сферу произвольного радиуса (например, в одну астрономическую единицу), которую будем отождествлять с нашим понятием небесной сферы. Зафиксируем точку на поверхности Земли и будем считать ее точкой наблюдения. Соединим эту точку с центром Солнца, что задаст вектор наблюдения. Проведем плоскость, касательную к поверхности Земли в этой точке – горизонт. Точка пересечения вектора с небесной сферой будет двигаться по дуге окружности с Востока на Запад через Зенит. Противоположная к Зениту точка – Надир. Таким образом, для неподвижного наблюдателя на поверхности Земли, Солнце восходит на Востоке и уходит за горизонт на Западе.

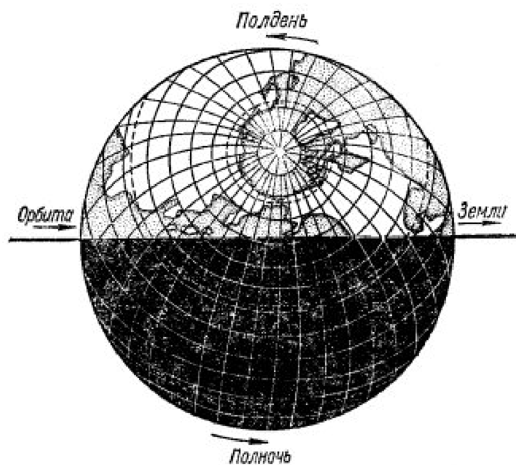


Рис. 17. Вращение Земли

Вращение вектора наблюдения описывается уравнением вращения $d\varphi/dt = \omega(\varphi)$, где следующее значение, задаваемое формулой $\varphi_{i+1} = \varphi_i + \omega(\varphi_i)\Delta t$, может зависеть от угла поворота, но не от t. Разобьем окружность на четыре равных сектора и присвоим им имена east, zenith, west, nadir. Это и есть основа дискретной модели.

Объектная модель показана на рис. 18. Смена дня и ночи моделируется объектом класса Day. Определим атомарный объект класса Sun, который имеет свойство state, которое может иметь четыре значения: west, nadir, east, zenith.

```
void Sun::exist()
{
switch (state[1]) {
case 'e': state = "zenith"; break;
case 'z': state = "west"; break;
case 'w': state = "nadir"; break;
case 'n': state = "east"; break;
}}
```

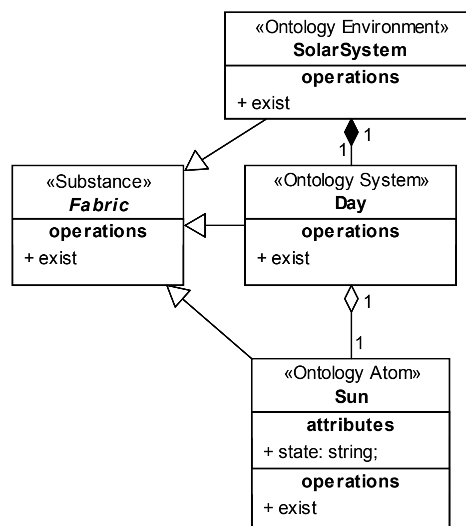


Рис. 18. Диаграмма классов Day

Описание, в котором следующее состояние зависит от текущего, вполне адекватно описывает процесс вращения Земли. Стоит обратить внимание на то, что в этой модели время как онтологическая сущность (т.е. переменная) не присутствует и тем самым в этом мире времени как такового не существует.

Формализация этого перехода может быть проведена с помощью лингвистической переменной <ℵ, *T*, *U*, *G*, *M*>, где

- ℵ – высота подъема Солнца над горизонтом;
- *T* – {"east ", "zenith ", "west ", "nadir" };

- U – [-45, 315];
- G – процедура образования новых термов, которая в нашем случае будет сводиться к модификатору «прошло четверть суток». Например, применение модификатора к терму "east" («прошло четверть суток с восхода») будет задавать терм "zenith" («Солнце стояло в Зените»).
- M – процедура задания на U= [-30, 330] нечетких подмножеств $A1$="east"[-45, 45], $A2$ = "zenith "[45, 135], $A3$="west"[135, 225], $A4$="nadir"[225, 315], а также нечетких множеств для термов из $G(T)$ соответственно правилам трансляции модификатора.

½ встречи. Дискретность и непрерывность в природе – явления довольно сложные. Приведем такой пример. Допустим, два человека договорились встретиться и обговорить некоторый вопрос. Встреча дискретна – она либо состоялась, либо нет. Допустим, что один из них видел другого издалека, но встреча не состоялась. Затем они созвонились и решили вопрос. Можно ли сказать, что встреча состоялась наполовину?

Перекладывания. Во многих случаях полезно использовать прием моделирования. Например, процесс перекладывания объектов из двух контейнеров класса Set в третий хорошо моделируется процессом вычисления $a + b = c$, где a,b и c – целые числа. Прием моделирования следует применять тогда, когда вопрос адекватности какого-либо процесса нас не интересует. Типичный пример – поведение атомарного объекта. Возможно, однако, потребуется доказательство эквивалентности процессов.

Принцип наименьшего действия. Обсудим методологические вопросы дискретного моделирования с более общей точки зрения. В системном анализе рассмотрение объекта как целого, как обладающего эмерджентными свойствами, стало общепринятой концепцией. По мнению И.З. Цехмистро [60] понятие системности должно рассматриваться в более широком смысле – как физически неделимого и неразложимого на какие-либо множества целого. С этой точки зрения не только структуру системы следует рассматривать как систему, но и изменения системы также обладают целостностью.

Поясним это на известном примере скатывания камня с вершины холма. Традиционный взгляд на этот процесс представляет собой представление о движении камня как последовательности причинно-следственных событий и описывается уравнениями динамики. В первом приближение движение камня описывается как движение материальной точки по наклонной поверхности под действием силы тяжести и силы трения. Однако в той же классической механике возможна и другая картина движения – основанная на понятии действия. В этом случае мы рассматриваем виртуальные траектории и, значит, обходимся без необходимости отслеживать причинно-следственные связи. Принцип наименьшего действия и есть взгляд на изменение системы как на некое, неразложимое на элементы, целостное понятие, аналогичное системным представлениям. Только теперь в качестве элементов анализа выступают не структурные элементы, а действия. Можно ожидать, что квант существования системы будет иметь такой набор темпоральных свойств, который не может быть выражен через элементарные причинно-следственые связи и носит эмерджентный характер. Подчеркнем, то, что принцип наименьшего действия есть твердо установленный научный факт, означает, что и квант существования системы – это объективная истина, а не результат философских спекуляций.

Именно такой смысл мы вкладываем в концепт «Квант существования системы». Концепт определяет стереотип «*Exist*», который мы будем использовать для методов, задающих единицу дискретно-событийного времени существования онтологических компонентов мира модели, включая сам мир.

Таким образом, последовательный с методической точки зрения, переход к дискретной модели будет предполагать следующие действия (на примере физических моделей).

1. Зададим контекст системы и определим начальные и граничные условия. Зададим для системы квант существования системы. Принцип наименьшего действия будем рассматривать как ограничения. Поведение системы мы можем анализировать только в терминах виртуальных траекторий. На этом этапе, как структура системы, так и структура времени неопределенны (а может и не существуют вообще). Для экономических систем, возможно, подобное описание дается системной динамикой Дж. Форестора.

2. Если структура системы существует объективно, то возможна декомпозиция системы на подсистемы. Квант существования системы, скорее всего, также можно декомпозировать и представить его в виде множества причинно-следственных связей. Это означает, что мы можем расположить элементы исследовательской установки внутри системы, а наблюдатель получает возможность наблюдать систему «изнутри». Для наблюдателя контекста квант существования системы остается событием, а с точки зрения наблюдателя системы, квант существования распадается на последовательность некоторых особых событий. Таким образом, траектория движения может быть разбита на ряд конечных интервалов (а значит, имеет место дискретность). В самом интервале мы по-прежнему будем придерживаться описания движения с точки зрения действия.

3. Данный процесс будем продолжать до тех пор, пока процесс декомпозиции имеет физический смысл в рамках данной задачи. Конечный результат декомпозиции – атомарные объекты и кванты их существования. В соответствии с концепцией целостности нет смысла пытаться моделировать структуру атома и временную структуру существования атома, даже если структура нам известна. Достаточно потребовать сохранение свойств атома (без выделения структуры) и выполнения принципа наименьшего действия (без выделения причинно-следственных связей). На практике это означает, что допускается искусственное и упрощенное описание атома. Можно, конечно, составить и точную модель, однако она будет уже избыточной. Пример – агент, это не модель человека, это схематическое описание поведения.

В примере, рассмотренном в начале этого пункта, квант существования системы в четыре раза меньше времени эксперимента. В примере с камнем квант существования совпадает со временем эксперимента, а атомарные объекты – камень и холм. Движение камня рассматривается как последовательность причин и следствий, описывается циклом, который есть не что иное, как численное решение уравнения движения.

Предлагаемый подход отличается от традиционного, и его можно назвать «моделированием сверху вниз». В традиционном подходе имеет место обратный процесс – от элементарных актов к кванту существования, от атомов к системе. По-видимому, такой подход также имеет право на существование, если только использовать понятие *агрегирование* вместо понятия *синтез*. Однако с практической точки зрения он представляется более сложным. Этот подход реализован в агентной парадигме имитационного моделирования и коррелирует с эндофизическими представлениями, которые уже обсуждались выше.

8. Пакеты миров. Выше мы определили стереотип *World* как пакет, в котором находятся классы конкретной имитационной модели. Однако чаще приходится иметь дело не с одной, а с несколькими моделями одновременно. Поэтому более правомерно говорить о *пакете миров(«Worlds»)*. Эти структуры можно рассматривать как одну из интерпретаций логики С. Крипке [23].

В логике Крипке вводится понятие *модельной структуры* $<G, K, R>$, в которой K является множеством возможных миров, G является *актуальным* (действительным) *миром* – $G \in K$, а R является отношением между мирами. Крипке рассматривал модальности «возможное» и «необходимое» и дал следующее определение:

- □А читается “необходимо А” и естественно интерпретируется как истинность А во всех возможных мирах, достижимых из актуального,

- ♦A читается “возможно A” и естественно интерпретируется как истинность A в некотором возможном мире, достижимом из актуального.

Актуальный мир будем определять как мир, в котором находится *Исследователь*.

Трансмировые связи опишем стереотипом *Отношение Достижимости* («*Accessibility Relation*», "metaClass" Dependency.

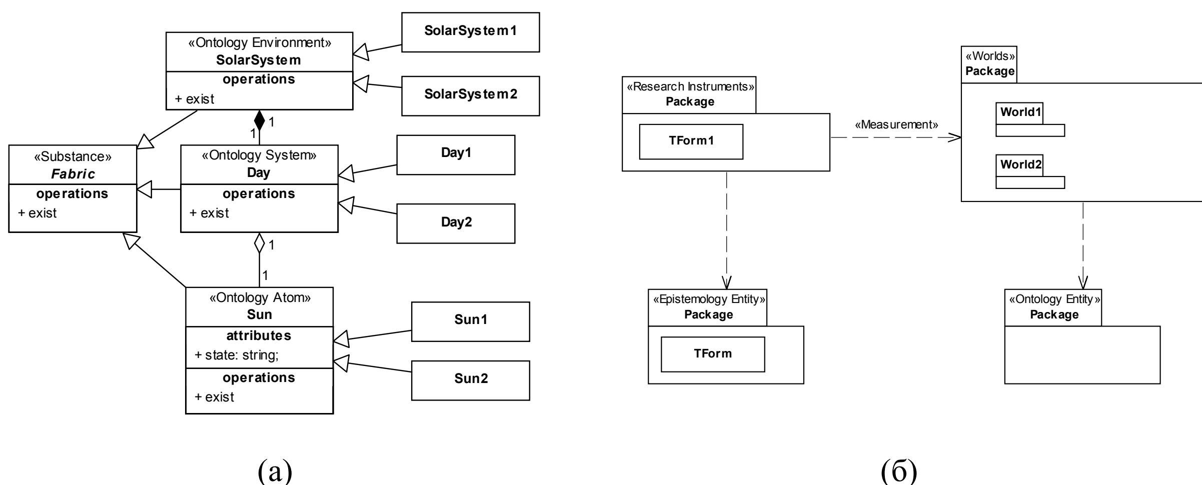
Использование *пакета миров* вместо одного *мира* практически ничего не меняет в архитектуре модели анализа. Актуальный мир, представляющий собой все сущности пакета Analysis System, кроме *пакета миров*, также может рассматриваться в описанном выше смысле как один из *миров*.

Пакет миров может иметь разную структуру. Рассмотрим три типа отношений между мирами. Для конкретности будем рассматривать модель из предыдущего пункта. Кроме того, далее мы будем полагать, что реализация моделей выполнена на Smalltalk.

1. *Параллельные миры*. Миры, связанные генетически. Все три компоненты имеют общего предка, см. рис. 19. Потомки одних и тех же классов образуют вариабельность (возможные миры). Процессы миров можно рассматривать как параллельные процессы. Достижимость – *Исследователь* имеет доступ ко всем мирам параллельно. Обычно такая структура пакета возникает тогда, когда исследуются модификации одной и той же имитационной модели. В этом случае нет необходимости обмениваться сообщениями, хотя такая возможность и существует. Необходимо – то, что определено в классе-предке; возможно – все допустимые модификации классов-потомков.

Веерные структуры. Пример – модель миров художественных произведений. Классы-потомки образуются в момент воспроизведения (устного или письменного) читателем текста произведения. Рассмотрим следующее художественное произведение с названием «Рассвет»: «Солнце взошло над горизонтом». Несмотря на краткость, это литературное произведение определяет вполне полноценный мир – некую вселенную (задает квант существования), систему (назовем ее dawning) и атомарный объект sun, который имеет два состояния (за горизонтом, над горизонтом) и метод stand up. Читатель создает потомков этих классов, например, так: «Солнце взошло над горизонтом, стало теплее».

Легенды. Легенды – это пример древовидных структур. Общий класс-предок – первое сообщение о некотором историческом событии. Пока легенда передается устным путем, она каким-то образом эволюционирует. Когда появляется письменность легенды-миры «вмораживаются» в носитель и перестают изменяться. Это позволяет говорить о некоторой космологии возможных миров.

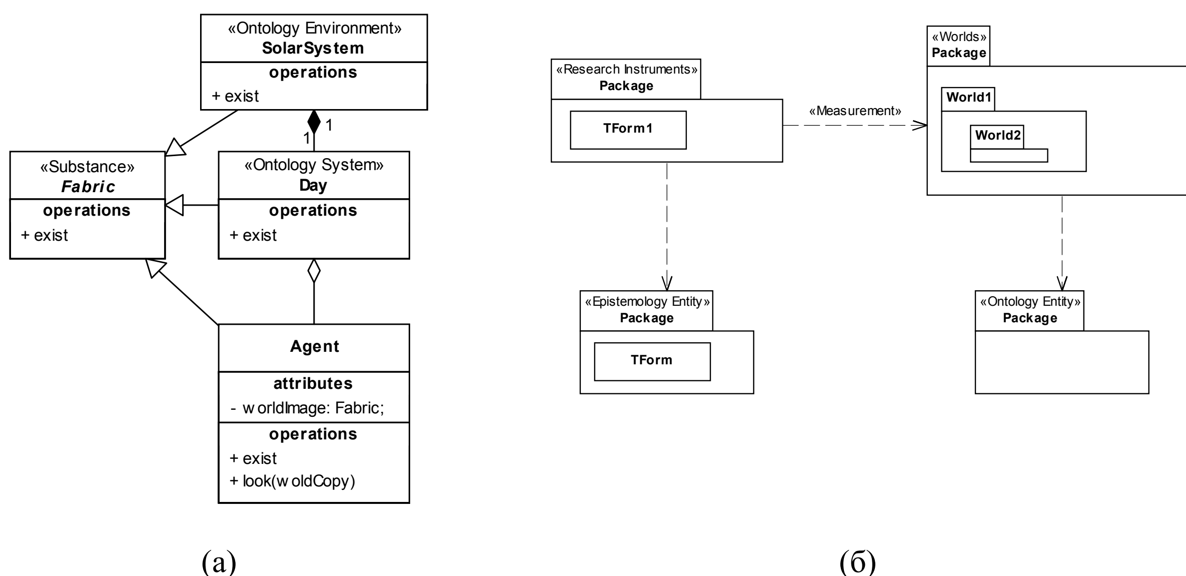


(а) (б)

Рис. 19. Диаграмма классов (а) и архитектура (б) для двух параллельных моделей Sun

2. Фрактальные миры

Построим нашу модель суточного вращения следующим образом. Сначала создадим модель компьютера, а затем уже в этом компьютере создадим имитационную модель. В такой конструкции доступ к модели будет возможен только через виртуальную машину. В более сложном случае можно создать разветвленную структуру со многими уровнями. Подобные структуры имеют много общего с фракталами. Нечто похожее имел в виду Лефевр, когда говорил о «системах, нарисованных на системах» [37].



(а) (б)

Рис. 20. Диаграмма классов (а) и архитектура (б) для вложенных моделей Sun

Одним из интересных частных случаев фрактальных миров являются модели рефлексии. В данном случае термин «рефлексия» мы понимаем в философском смысле – как способность индивидуума познавать свой мир. Как это ни удивительно, но рефлексия моделируется довольно просто. Для этого достаточно в классе Agent задать поле worldImage (см. рис. 20а) и определить операцию agent look: (self deepCopy) в методе exist класса Day. Оператор self deepCopy создает копию с независимыми переменными текущего экземпляра класса Day (в C++ псевдопеременной self соответствует псевдопеременная this). Метод look класса Agent присваивает полю worldImage указатель на эту копию. Операция agent look: (self deepCopy) моделирует познавательную деятельность рефлектирующего агента, которая протекает вне агента.

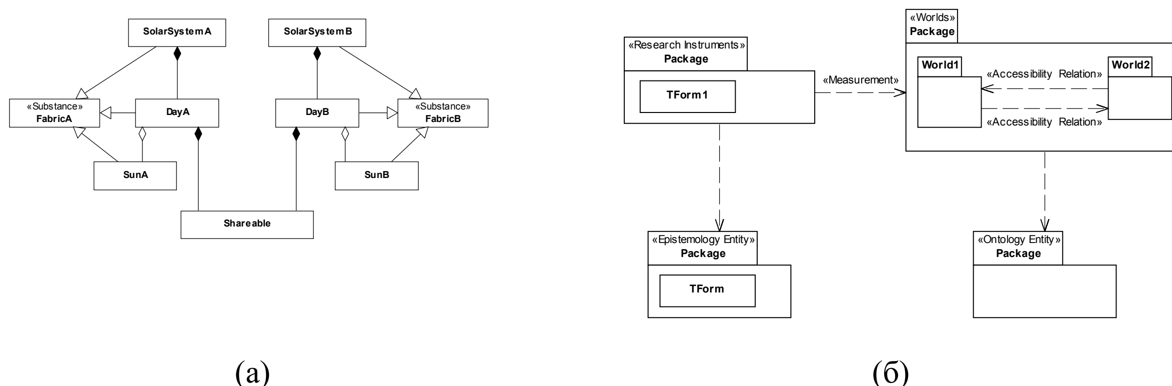
Мир получается как бы вывернутый на изнанку. Контекстом становится внутренний мир агента и рефлексирующий агент как *Исследователь* получает возможность посылать сообщения объекту класса Day и тем самым исследовать его.

Посылая одно сообщение exist, мы получим рефлексию 1-го ранга. Мы получим рефлексию 2-го ранга, если еще раз пошлем сообщение exist. Рефлексия 2-го ранга – агент представляет себе мир, в котором есть агент (он сам), который в свою очередь имеет образ мира. Продолжая посылать сообщение exist, мы получим рефлексию любого n-го ранга, т.е. потенциально бесконечный процесс познания, что, казалось бы, подтверждает известный тезис философии о бесконечности познания. В действительности, возможно, в этом нет необходимости. Агенту достаточно сравнить образ мира в рефлексии 1-го ранга с образом мира в рефлексии 2-го ранга. Если они совпадут, то мы получим информационное равновесие и процесс познания можно считать завершенным.

3. Взаимопроникающие миры. Есть еще один вариант структуры пакета миров. Пусть миры имеют разные субстанциональные классы. Тем самым они не могут

взаимодействовать друг с другом путем передачи сообщений, см. рис. 21. Однако миры могут иметь некую общую структуру, которая описывается общими переменными. Необходимое – общая переменная. Возможное – все остальное. В качестве примера можно назвать Янус-космологию Лефевра [37].

Необходимость выйти за пределы традиционного системного анализа наиболее наглядно видна в анализе *проблемных ситуаций*. Рассмотрим в качестве примера задачу о двух заместителях, см. рис. 22. Пусть поток документов обрабатывается двумя заместителями. Документ должен быть рассмотрен обоими заместителями прежде, чем он будет рассмотрен ЛПР (Лицо Принимающее Решение). Каждый заместитель имеет собственные критерии оценки документа и поэтому в некоторых случаях их мнения по одному и тому же документу могут не совпадать. Окончательное решение принимает ЛПР.



(а) (б)

Рис. 21. Диаграмма классов (а) и архитектура (б) для моделей Sun с общей переменной

Проблема, связанная с моделированием проблемных ситуаций, состоит в том, что возникают трудности с корректным применением стереотипа *World*. С одной стороны, ЛПР должен получить сообщение от своих заместителей в форме документа. С другой стороны, каждый заместитель должен оценить документ согласно своему контексту («картине мира», в терминологии Чекленда [65]), а если задан контекст, то определен *мир,* и вне *мира* уже ничего наблюдаться не может. Последнее надо понимать как запрет на обмен сообщениями между объектами, моделирующими миры, и различать понятия «ненаблюдаемый объект» и «ничто». Разрешение данной логической трудности, на наш взгляд, возможно на основе включения в онтологию уровня профиля модальной логики Сола Крипке [23].

Вычислительную семантику стереотипа *Accessibility Relation* определим как отношение зависимости, обусловленное *разделяемыми переменными* (в терминологии Smalltalk). Переменные, доступные более чем одному объекту, объединяются в группы, называемые *пулами.* Класс имеет два или более пулов, доступных его экземплярам. Один пул, содержащий глобальные переменные (global variable), доступен всем объектам системы. Каждый класс может иметь также второй пул переменных (class variable), который содержит переменные класса и доступен только его экземплярам. Классу могут быть доступны специально созданные пулы переменных (pool variable), которые доступны нескольким классам. Достижимость между мирами, в нашем случае, может быть описана, в частности, посредством переменных типа pool variable.

Введение *пакета миров* позволяет применить методологию «мягких» систем Чекленда [65], которая включает семь этапов, для организации плана вычислительного эксперимента. Как видно, задача о двух заместителях решается в два прогона этого семеричного цикла. Хотелось бы подчеркнуть, что введение логики Крипке в онтологию уровня профиля связано с внутренней структурой профиля, а не только с задачами

анализа проблемных ситуаций. Как показывает опыт построения объектных моделей, «жесткие» системы скорее исключение, чем правило.

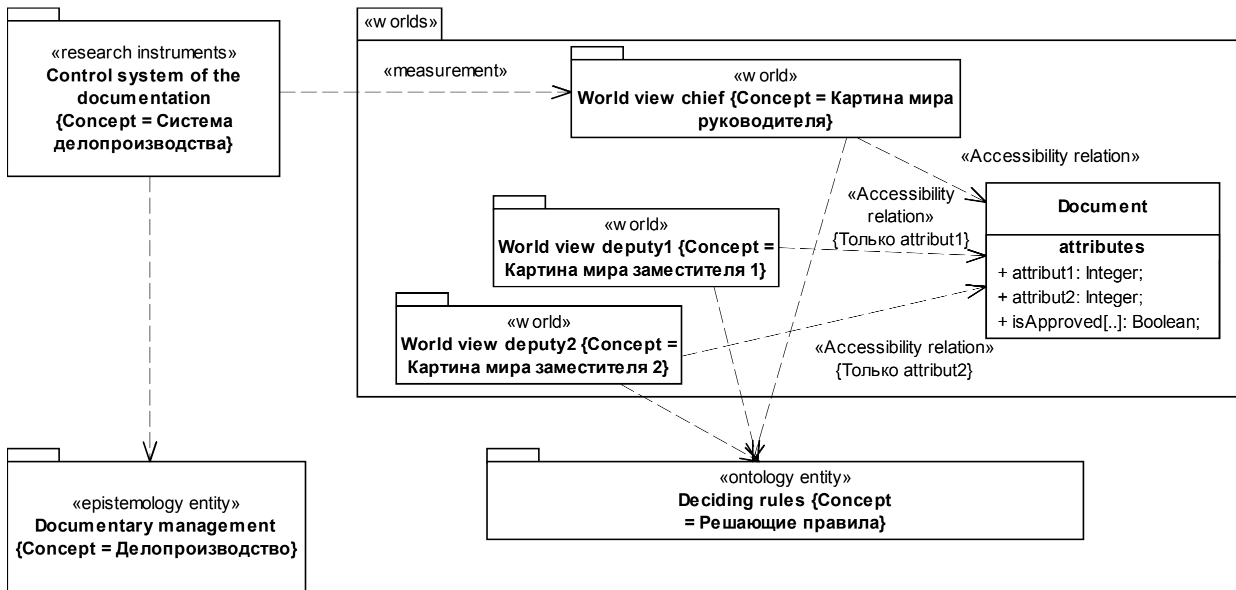


Рис. 22. Архитектура пакета Analysis System (*Research Analysis Model)* для задачи двух заместителей

9. Моделирование математических объектов. Особый интерес представляет объектное моделирование математических объектов. Имитационное моделирование позволяет по новому посмотреть на конструктивную математику. В качестве иллюстрации приведем объектную модель дискретной группы вращения равностороннего треугольника. Далее, мы будем предполагать, что объектная модель реализована на Smalltalk.

«*Research Use Case Model*» определяется четырьмя ролями «*Researcher*» (по одному для каждого прецедента) – конфигурировать систему, вычислить алгебраическое выражение, документировать результат вычисления, документировать процесс вычисления.

«*Research Analysis Model*» содержит объекты aAlgebraicExpression (онтологический контекст; определяет процесс вычисления алгебраического выражения и в общем случае определяется через понятие «математический дискурс» [22]), aElementA, aElementB, aElementI (элементы группы a, b, I, где $aa^{-1}=I$), датчик результата и датчик операций. Классы всех объектов являются потомками абстрактного класса AlgebraicObject, который имеет метод с селектором multiplyBy: aAlgebraicObject (см. рис. 23). Элементы группы различаются контроллерами умножения, каждый из которых реализует соответствующую строку таблицы умножения (табл. 3).

Таблица 3

Таблица умножения

	I	a	b
I	I	a	b
a	a	b	I
b	b	I	a

Элементы группы aElementA и aElementI является атомарными объектами. Элемент aElementB представляет собой составной элемент, имеет класс ComplexElement и выражается через образующий элемент $b = a^2$. Кооперация объектов задается паттерном поведения Strategy.

По приведенной схеме могут быть построены многие математические объекты. В частности, это верно для многочленов, которые, как известно, являются одним из основных математических объектов компьютерной алгебры. Системы компьютерной алгебры, даже если используется объектно-ориентированное программирование, основаны на символьных вычислениях. Вычисления с объектами не являются символическими вычислениями, а являются обобщением численных методов на объекты нечисловой природы. Формально методы вычислений (или, как говорят, компьютинга) объектов могут быть описаны на языке аппликативных вычислений [12].

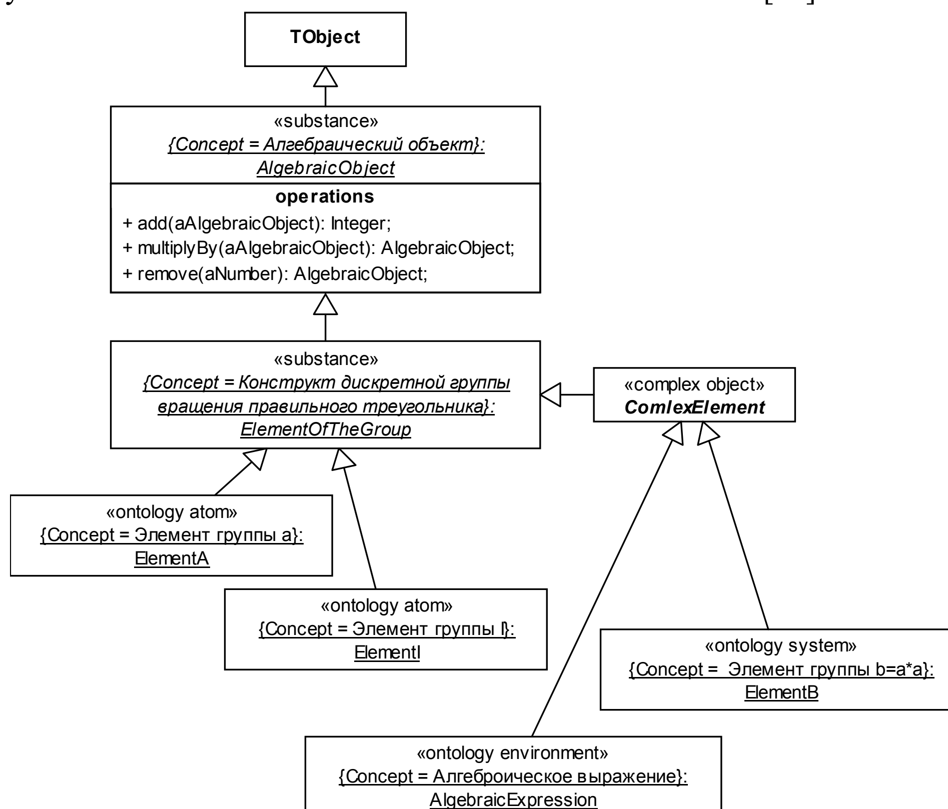


Рис.23. Диаграмма классов для моделирования дискретной группы вращения правильного треугольника

Для модели характерно рассмотрение элементов без каких-либо свойств (единственное, что можно сказать, так это то, что они принадлежат разным классам), уникальность которых определяется исключительно способом их взаимодействия друг с другом. Так как обмен сообщениями – это связь элементов, то описанный выше подход выражает философию структурализма (см. обзорную часть [22]). Заметим, что альтернативный подход – определение математических элементов с набором свойств – приводит к компьютерной алгебре.

Предметная область математики лежит в идеальной сфере. Математика имеет дело с абстрактными объектами. Такие объекты обладают некоторым набором отношений, по которому их можно сопоставить с объектами физического мира (и тем самым определить математическую модель). Для действительного мира модель будет компьютерной, а непосредственно для ментальной сферы – имитационной.

В книге [55] приведены объектные модели большинства объектов классической математики. С данным материалом можно ознакомиться также по электронным курсам на INTUIT.RU (http://www.intuit.ru/studies/courses/671/527/info).

10. Фрактальные структуры

Во многих областях науки и общественной практики возникают задачи моделирования процессов развития сложных систем, где большое значение имеет понимание сущности фрактальных структур. В этом разделе мы определим основные концепты (уровня предметной области) теории фракталов для классических геометрических фракталов. Далее, мы будем предполагать, что объектная модель реализована на Smalltalk. Подробнее см. [16].

Концепты модели зададим согласно принятой математической терминологии. Валидация должна выполняться на некотором языке, который не является языком предметной области или языком *Scientific Profil.* Мы считаем возможным для этой цели, воспользоваться исчислениями CCS (R. Milner) или CSP (C.A.R. Hoare); см., например, [39], [54]. Для валидации будем использовать понятие сильной эквивалентности, понимая под бисимуляцией процессов P и P^* следующее. P^* есть симуляция операций P над математическими объектами, а P есть математическая модель симуляции P^*.

Конструктивный процесс Rewriting = (S, s_0, R) построения фрактала определим следующим образом [58]. Зададим произвольною ломаною с конечным числом звеньев, называемою *инициатором* (начальное состояние s_0). Далее, заменим в ней каждый отрезок *генератором*. Склеим генераторы (если необходимо). В получившейся ломаной ($s_n \in$ S) вновь заменим каждый отрезок генератором, пересчитанным с коэффициентами подобия r>0. Продолжая до бесконечности, в пределе получим фрактальную кривую.

Стереотип *Research Use Case Model* определяет спецификацию имитирующей программы для *Исследователя*. Допустим, что модель содержит два прецедента: «Приготовить фрактал» и «Измерить хаусдорфову размерность». Определим следующие роли *Исследователя – Preparator* и *Observer*. Спецификация прецедентов может быть редуцирована на логику Хоара.

Прецедент «Приготовить фрактал»
{fractal isNil} fractal:=Initiator?new. Выполнить rewrite *n* раз. fractal!. Wait {fractal isFractal}
Прецедент «Измерить хаусдорфову размерность»
{fractal имеет *n* ярусов} fractal?hausdorffDimension. fractal!D_H. Wait {D_H – размерность Хаусдорфа для объекта fractal}

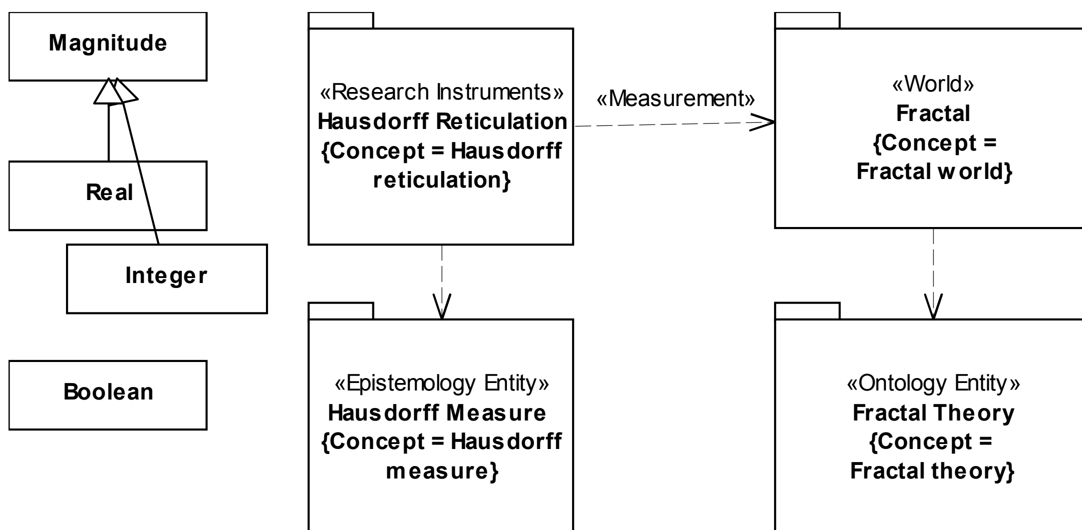


Рис. 24. Архитектура пакета Analysis System

Анализ прецедентов выполняется в модели анализа (рис. 24). Пакеты со стереотипами *Epistemology Entity* и *Ontology Entity* общего уровня задают методологии проведения измерений и правила построения фракталов в стереотипах UML (классы TopologicalSpace, MetricSpace, Initiator, Generator). Рассмотрим каждый пакет более детально.

Пакет Fractal «*Word*». Содержит реализацию прецедентов в виде классов конкретных фракталов (например, KochSnowflake, KochCurve). Этот пакет использует сущности из пакета Fractal theory.

Пакет Fractal theory «*Ontology Entity*». Пакет Fractal theory определяет таксономию геометрических фракталов на одномерном метрическом пространстве (см. рис. 25; чтобы не перегружать рисунок, мы будем считать, что концепты можно восстановить по названиям пакетов). Его вычислительная семантика определяет систему пользовательских типов. Мы будем для определения типов рассматривать интерфейсы классов как типы. Типы будем называть тем же именем, что и классы, добавляя букву T в начале имени.

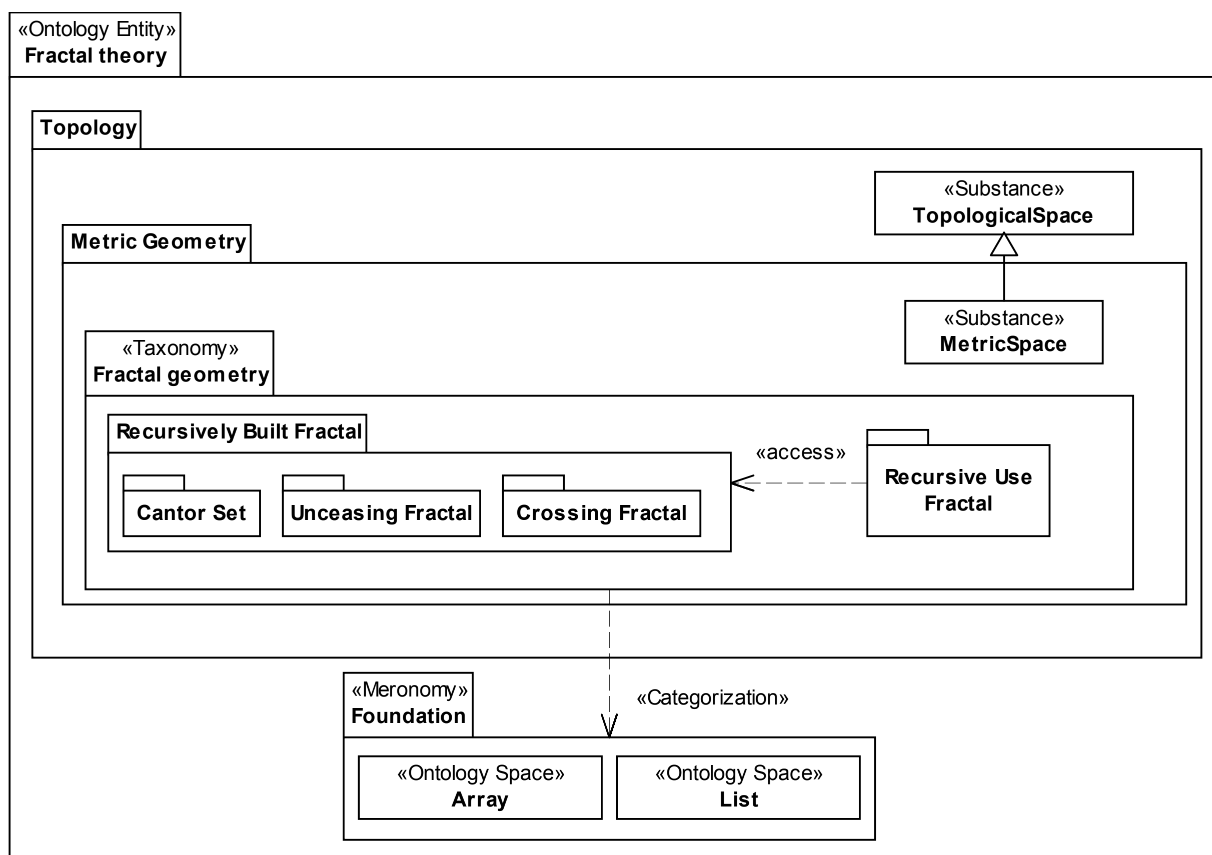


Рис. 25. Таксономия геометрических фракталов

Обычно разделяют рекурсивное определение и рекурсивное использование. Пакет Recursively Built Fractal состоит из подпакетов Cantor Set, Unceasing Fractal (непрерывные фракталы) и Crossing Fractal (фракталы с самопересечением), классы которых определяют абстрактный метод rewrite класса Generator. Пакет импортирует классы пакета Faundation (контейнер, пространство-носитель), которые определяют тип поля lowerTier в классе Generator. В общем случае в этот пакет может быть включено довольно богатое множество классов, построенных на основе структурных паттернов. Однако мы решили ограничиться классами Array и List, поскольку предметную семантику остальных классов идентифицировать не удается. Далее мы будем чаше всего рассматривать вырожденный случай, когда тип lowerTier совпадает с типом Generator (т.е. экземпляр Generator является также и ячейкой пространства-носителя и тем самым относится к типу TList).

В случае Recursively Built Fractal в теле определения класса используется определяемый класс либо непосредственно, либо через определения других классов. На рис. 25. показан случай, когда используется только один класс Generator. В общем случае на диаграмме следует указывать пакет, содержащий несколько классов-генераторов. Редукция этого пакета на типизированное λ-исчисление приводит к известному определению для *взаимной (косвенной) рекурсии*. Пакет Recursive Use Fractal содержит фракталы, использующие рекурсивное использование. В случае рекурсивного использования объект обменивается сообщением с другим объектом, который для вычисления возвращаемого значения в свою очередь посылает сообщение первому. Редукция этого подпакета на типизированное λ-исчисление приводит к *параллельной (удаленной, частичной) рекурсии*.

Рассмотрим имитационную модель, заданную диаграммой классов, представленной на рис. 26.

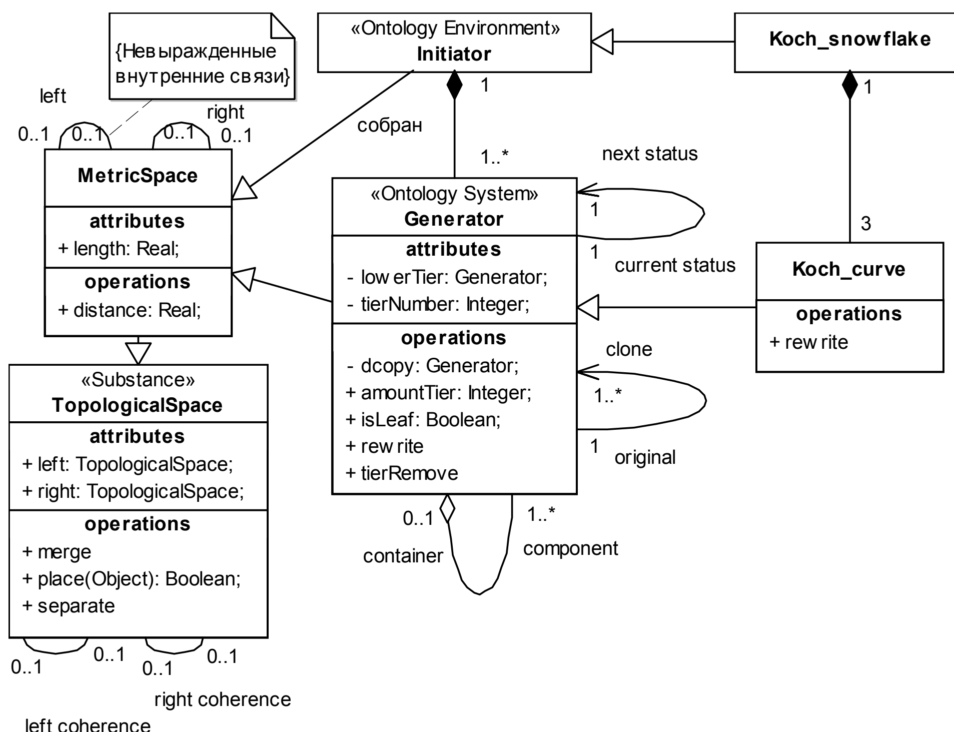


Рис.26. Диаграмма классов, моделирующая геометрические фракталы

Псевдопеременной nil сопоставим концепт «Пустое множество ∅» В качестве *субстанционального класса* выберем класс TopologicalSpace. Класс имеет свойства left и right, которые хранят указатели на левого и правого соседа и тем самым задают правую и левую связанность. Кроме того, он имеет метод separate (и обратная операция merge), который обеспечивает деление пространства на подпространства с сохранением связанности. Метод place обеспечивает установление отношения агрегации с загружаемым объектом. Экземплярам этого класса сопоставим концепт «Окрестность точки топологического пространства». Сопоставим концепт «Компактное топологическое (одномерное) пространство» объектам, каждый из которых представляет собой динамический список, всякий элемент которого – экземпляр TopologicalSpace. Динамический список может быть nil. Концепт «Полное и компактное метрическое пространство» присвоим списку, каждый элемент которого – экземпляр MetricSpace. На список наложено следующее ограничение: все внутренние связи списка не есть nil и элементы имеют согласованную связанность (будем говорить, что экземпляры соответствующих ассоциаций не выроджены). Кроме того, задается ограничение на метод place относительно отношения «больше» и «меньше».

Контекст фрактала (стереотип *Ontology Environment*) зададим классом Initiator. Класс Generator используется для создания конструктивного процесса фрактала и определяется методами rewrite и tierRemove, позволяющих создавать или удалять ярус фрактала. Методы amountTier: TInteger (гносеологическая сущность) и isLeaf: TBoolean позволяют получить информацию о глубине вложенности фрактала и идентифицировать объект как лист. Класс имеет два поля. Поле tierNumber: TInteger фиксирует номер яруса (начиная с 0, гносеологическая сущность), а lowerTier есть указатель на элемент агрегации. Поле lowerTier имеет стереотип *Ontology Space* ({Concept= Ontology space}) и в общем случае может быть само агрегатной переменной (см. пакет Faundation).

В объектной модели заданы три рефлексивные ассоциации. Агрегация, заданная парой ролей container-component, реализует связи для передачи сообщений между ярусами. Экземпляры ассоциации original-clone определяют связи для сообщений копирования или уничтожения. Ассоциация current status - next status определяет отношения между разными состояниями одного и того же объекта, в результате которого происходит смена ролей (т.е. разновидность каузальных связей объекта).

Метод rewrite. Это абстрактный метод, который замещается в классах пакета «*Word*»**.** Процедура метода может быть представлена конструкцией Decision (ветвление). Одна ветвь выполняется тогда, когда текущий ярус не является листом; тогда сообщение rewrite отправляется объектам нижележащего яруса. В противном случае создается новый ярус. Так же, как и в процессе Rewriting, создание элементов яруса может быть выполнено двумя способами – созданием экземпляров класса Generator (так как *генератор* – это шаблон) или клонированием текущего узла. Рассмотрим второй способ. Создание яруса включает: (а) сохранение состояния объекта, (б) вычисление состояния для нового яруса, (в) клонирование текущего объекта и размещение копий в поле lowerTier и (г) восстановление состояния объекта и изменение его статуса.

Процесс «Размещение копий в поле lowerTier» представляет собой процесс, который кроме непосредственного размещения объектов в контейнере предполагает восстановление связанности. Уточнение способа восстановления производится в подпакетах Recursively Built Fractal (на рис. 25 эта более тонкая градация не показана). Для некоторых фракталов надо склеивать экземпляры генератора. Наиболее естественно воспользоваться указателями left, right верхнего яруса и организовать пересылку указателей через узел. Для этого используется процедура restoreCoherence. Фракталы с самопересечением можно моделировать, если переданный указатель разместить в текущем динамическом списке. Для определения подобных конструкций необходимо пространство-носитель задавать явно.

Валидация конструктивной процедуры. Определим процесс Rewriting* = (S*, s_0*, R*). Начальное состояние s_0* – лес (экземпляр Initiator), каждое дерево которого составлено из экземпляров класса Generator. Все объекты существуют параллельно и инкапсулируют параллельные процессы, находящиеся в состоянии ожидания. Процесс для экземпляра Initiator есть Rewriting_wood = in?rewrite. down!rewrite. Rewriting_wood. Для экземпляров Generator, выступающих в роли узла, можем написать Send_downwards = up?rewrite.down!rewrite.Send_downwards. Для экземпляров Generator, выступающих в роли листа, запишим Rewrite_tier = up?rewrite. Clone. Rewrite_tier, где процесс Clone – описанный выше процесс создания объекта lowerTier. Тем самым Rewriting*= Rewriting_wood || Send_downwards || Rewrite_tier. Последовательность ярусов определяет множество состояний S*. Роль *Preporator* (после создания fractal) представим как Hand_generated = in?fractal. fractal!rewrite.in!fractal. Hand_generated. Бисимуляция с процессом Rewriting становится очевидна, если учесть то, что параллельные процессы Rewriting* взаимодействуют по рандеву и потому находятся в трассовой эквивалентности с аналогичным последовательным процессом. Применяя к спецификации прецедента «Приготовить фрактал» процесс New.Rewriting*, выполним верификацию по этому прецеденту.

Пакет Hausdorff reticulation «*Research Instruments*» определяет процедуры измерений в случае конкретного математического объекта и представлен отдельными методами или даже отдельными фрагментами кода. В классах, определяющих модель фрактала, эти методы выделены в отдельную категорию observer. Пакет использует сущности пакета Hausdorff measure. Метод hausdorffDimension определен для потомков класса Initiator; вызывает метод toCover и метод giveScale потомков класса Generator. Визуализация результатов измерений осуществляется на основе паттерна *MVC*. Факт зависимости классов пакета «*Research Instruments*» от классов пакета «*World*» фиксирует стереотип *Измерение* («Measurement»).

Пакет Hausdorff measure «*Epistemology Entity*» определяет общие принципы измерения фрактальной размерности (мы ограничимся только мерой 1-го порядка) и содержит определения измерительных процедур и методы обработки измерений. Вычислительная семантика – библиотека базовых классов для пакета Hausdorff reticulation. Предметная семантика определяется заданием процедур для проверки эквивалентности и отношения порядка (определением измерительных шкал; далее рассмотрим только количественные шкалы). Пакет включает диаграммы состояний, описывающие процессы измерений.

Процесс Hausdorff_dimension. Прямым способом измерения величины D_H (хоусдорфова размерность) является измерение $N(\varepsilon)$, ε и вычисление размерности по формуле

$$D_H = \lim_{\varepsilon \to 0} \frac{\ln N}{\ln(1/\varepsilon)}.$$

Процесс Hausdorff_dimension* = fractal!giveScale. fractal?ε. fractal!toCover(ε).fractal?N. out?D_H. Hausdorff_dimension* \ Calculation. Концепт «Хоусдорфова размерность» присвоим методу hausdorffDimension, где и будем вычислять D_H. Определим роль Observer как процесс Measurement = fractal! hausdorffDimension. fractal?D_H. Measurement. Применяя к спецификации прецедента «Измерить хаусдорфову размерность» процесс Hausdorff_dimension*, выполним верификацию по этому прецеденту. Концепт «Хоусдорфова размерность», определенный выше, и математическое понятие «хоусдорфова размерность» совпадают для случая однородных фракталов. Однако вопрос о валидации для фракталов с несколькими масштабами остается открытым.

Из вышеизложенного следует, что класс Magnitude, заданный в корневом пакете, должен иметь, по крайней мере, базовые классы Integer и Real.

Последняя модель – **«*Research Design Model*»**, - определяет способ описания модели на конкретном языке имитационного моделирования. В модели обосновывается аппроксимация сущностей модели анализа. Аппроксимация обусловлена в наибольшей степени тем, что параллельные процессы следует представлять как последовательные. Обычно для этого достаточно воспользоваться одной из процессных эквивалентностей.

Рассмотрим несколько конкретных фракталов.

Пример 1. «Фрактальная сфера». Этот фрактал получается из диаграммы классов, если кардинальное число агрегации на стороне component равно 1. Геометрический прообраз можно представить как бесконечную последовательность концентрических сфер, причем радиус самой внешней сферы равен 1.

Пример 2. Множества Кантора. Пакет «*Word*» содержит класс CantorDust (потомок класса Initiator) и класс CantorStrainer (потомок класса Generator). Ниже приведен код процедуры метода rewrite для арифметической пыли Кантора (триадное канторовское множество).

```
rewrite
| l lengthSave t e |
```

```
(lowerTier isNil) ifTrue: [l:= tierNumber. tierNumber:= tierNumber+1.
lengthSave:= length. length:= length /3. "делим отрезок на 3 части"
t := Array new: 2. " создаем пространство-носитель"
t at:1 put:(self dcopy). t at:2 put:(self dcopy). " добавляем 2-а отрезка"
lowerTier:= t. tierNumber:= l. length:= lengthSave.]
ifFalse: [1 to:2 by:1 do: [:each | e:= lowerTier at: each. e rewrite]].
```

Для объектов, моделирующих элементы канторовского множества нет необходимости восстанавливать связность. Поэтому мы помещаем эти элементы в контейнер (массив) с доступом по индексу. Другие канторовские множества (в т.ч. двухмасштабные) строятся подобным образом.

Пример 3. «Снежинка Коха». Пакет «*Word*» содержит реализацию прецедентов в виде классов KochSnowflake (потомок класса Initiator) и KochCurve (потомок класса Generator). Процесс метода rewrite создает цепочку (динамический список) из 4-х клонов объекта, которая моделирует сегменты кривой Коха. Тут же восстанавливается локальная топология. Если объект не является нижним ярусом, то сообщение rewrite передается на более глубокий уровень. Процедура restoreCoherence склеивает сегменты кривой Коха и выполняется в том случае, если на нижележащем уровне был создан новый ярус.

В заключение приведем еще один изящный пример фрактала – два зеркала, поставленных один напротив другого, который столь поэтично отражен в живописи (см., например, картину Константина Васильева «Гадание»). Можно построить объектную модель формирования коридора изображений. Этот оптический фрактал есть аналог процессов в системах, состоящих из рефлексирующих агентов, которые мы будем рассматривать позднее.

Итак, в данном пункте рассмотрены объектные модели геометрических фракталов. Основное внимание уделялось методологическим вопросам объектного моделирования фракталов, нежели исследованию самих фрактальных структур. Поэтому мы ограничились весьма узким кругом классических геометрических фракталов. Распространение модели на более широкий класс фрактальных структур встречает некоторые трудности. Если определение конструктивной процедуры, например, для мультифракталов довольно очевидно, то определение измерительных процедур, по-видимому, потребует обобщения изложенного выше подхода.

Коалгебра и коданные. Определение структур, подобных фракталам, во многих случаях удобно давать не в терминах конструктивных процедур, а в терминах коалгебры. Пусть надо дать определение бесконечного динамического списка. Допустим, что у нас такой список уже есть. Мы будем его рассматривать как потенциально бесконечный. Тогда определение будет такое: если мы выдадим команду rewrite, то в результате получится список, равный исходному (в пределах заданной точности).

Бильярд Синая и сценарии перехода к хаосу. Одно из центральных понятий современной науки – это понятие хаоса. Разберем возможные пути моделирования хаоса.

Рассмотрим движение бильярдного шара в двухмерном бильярде; трение учитывать не будем. Движение шара можно характеризовать как последовательность событий – соударение шара с бортиком стола. Бильярд Синая – это бильярд, в котором один из бортиков является выпуклым. Доказано, что в этом случае движение шара становится хаотическим (нарушается симметрия по обращению времени). Двухмерный массив, моделирующий бильярдный стол, можно представить как одномерный массив. Тогда движение шара будет скачкообразным и в простейшем случае – периодическим. Каким должен быть алгоритм, который задает полностью хаотическое движение?

Самый простой способ – это воспользоваться *линейным конгруэнтным методом*, который лежит в основе большинства стандартных функций random языков программирования. Впервые этот метод был предложен Лехмером в 1949 году. Выбирается 4 числа: модуль m (m>0); множитель a ($0 \le a < m$); приращение c ($0 \le c < m$); начальное значение X_0 ($0 \le X_0 <$m). Последовательность получается с использованием

следующей рекуррентной формулы: $X_{n+1}=(aX_n+c) \bmod m$. Линейный конгруэнтный метод, как и всякое отображение конечного множества на себя, даёт повторяющиеся последовательности. Таким образом, необходимо максимально удлинить уникальную часть последовательности; для этого специальным образом подбирают *m, a, c* и X_0. Можно также рассмотреть *логистическое уравнение*. Уравнение имеет вид $x_{i+1} = \lambda x_i(1-x_i)$, причем номер ячейки $k = xN$, где N – количество ячеек. По мере роста λ наблюдаются бифуркации удвоения периода; этот процесс называется сценарием перехода к хаосу [49]. Действительно случайные последовательности можно получить только на аппаратных генераторах случайных чисел, которые позволяют «снимать» случайные процессы физического мира; последние называются *источниками энтропии*.

Как следует из вышесказанного, всякий процесс через достаточно большое число тактов существования повторяется. Тем самым время имитационной модели является циклическим. Время реального мира, по мнению философов, таковым не является. Следовательно, если мы не допустим «просачивания» энтропии из физического мира, модель не будет валидной.

Перемешивание. Каким должен быть алгоритм перемешивания колоды карт? Эта задача известна как задача о тасовании карт. Один из возможных алгоритмов – преобразование пекаря [58].

На наш взгляд моделирование истинного индетерминизма возможно на основе ситуации гонок. *Гонки* (race condition) – это ситуация, при которой несколько параллельных потоков конкурируют за некоторый ресурс. Например, когда два потока одновременно производят запись в одну глобальную переменную. Невозможно сказать, что будет, в конечном счете, записано в данную переменную. Скорее всего, одно значение переменной будет записано поверх другого. С точки зрения логики изменения ситуация гонок - это ситуация в которой нарушается закон исключения третьего.

11. Классические игры. В качестве примера рассмотрим позиционную игру, состоящую из двух ходов. Позиционные игры достаточно хорошо изучены в теории игр; данная теоретико-игровая задача позаимствована из книги [57]. Игрок *A* делает первый ход: он выбирает одну из двух возможных альтернатив: черную ($x = 1$) или белую ($x = 2$) фишку. На ход игрока *A* игрок *B* отвечает своим ходом: $x = 1$ или $x = 2$. Функция $W(x,y)$ выплат игроку А за счет игрока *B* пусть имеет вид: $W(1,1) =1$, $W(2,1) = -2$, $W(1,2) =-1$, $W(2,2) = 2$. Данная игра может быть нормализована и сведена к матричной игре. Решение существует в чистых стратегиях. Для приведенного выше количественного примера оптимальная стратегия будет: игрок *A* на 1-м ходе выбирает $x = 1$, а игрок *B* на 2-м ходе выбирает $y = 2$. Цена игры $y = -1$.

На рис. 27 представлена объектная модель для данной активной системы. Мы предполагаем, что используется субпрофиль *Scientific Profile for Active Systems*. Абстрактные классы Subject, Agent и ActiveSystem – общие классы для активных систем и определяются в пакете со стереотипом *Ontology Entity*. Класс Totalizator определяет контекст игры и, в частности, определяет функцию выплат по сделанным ставкам. Класс Gambler определяет игроков и имеет поля paymentFunction (Concept = Функция выплат), playgroundImage (Concept = Образ игровой доски); свойство role (Concept = Первый ход); методы getPaymentFunction (Concept = Получить функцию выплат), go (Concept = Сделать ход), look (Concept = Осмотреть игровую доску). Ментальность агентов (процедура management) будем моделировать алгоритмом выбора хода. Класс Game моделирует систему – игру – и имеет поля gamblerFirst (Concept = Игрок А), gamblerSecond (Concept = Игрок В), payment (Concept = Ставки), playground (Concept = Игровая доска); метод takeRates (Concept = Принять ставки) с аргументом rates. Процедура management случайным образом определяет роль каждого агента и доводит до их сведения функцию выплат. Процедура business моделирует сам процесс игры. Постулируем, что процессы management и business параллельные и взаимодействуют по рандеву.

Заметим, что если необходимо моделировать другую систему, например, ядерный конфликт двух стран, достаточно переопределить только концепты. Во втором примечании мы уже говорили о большом значение понятия «метафора». Мы будем использовать метафору «Тотализатор», как, впрочем, и другие метафоры, понимая под метафорой тот или иной класс моделей. Как и в рассматриваемом примере, для определения новой модели достаточно только переопределить концепты. Тем самым в отличие от других моделей в метафорических моделях концепты не совпадают с именами программных сущностей.

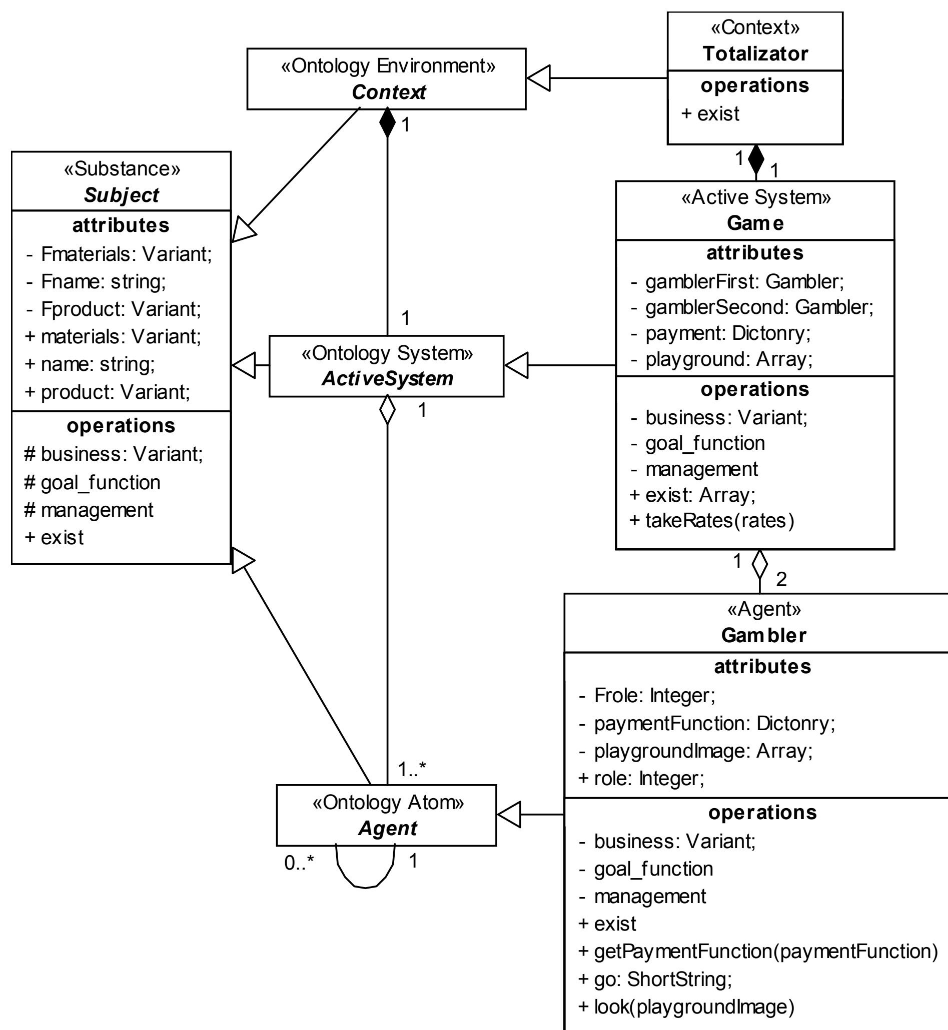


Рис. 27. Диаграмма классов для имитационной модели Game

12. MDA и Scientific Profile. В практике имитационного моделирования так же, как и в практике проектирования информационных систем, возникает желание автоматизировать процесс разработки ПО, связав UML-модель и код программы. Эта идея выражена в MDA. Аббревиатура MDA расшифровывается как Model Driven Architecture – архитектура, управляемая моделью.

В архитектуре MDA рассматриваются три модели: CIM, PIM и PSM. *Вычислительно-независимая модель (Computation Independent Model, CIM)* описывает общие требования к системе, словарь используемых понятий и окружение. CIM рекомендуется выполнять с использованием унифицированного языка моделирования UML. *Платформенно-независимая модель (Platform Independent Model, PIM)* описывает состав, структуру, функционал системы. Для создания модели также используется UML. *Платформенно-зависимая модель (Platform Specific Model, PSM) описывает* состав,

структуру, функционал системы применительно к вопросам ее реализации на конкретной платформе. Модель платформы используется для доработки PSM в соответствии с требованиями платформы.

Этапы разработки хорошо соответствуют рабочим потокам *Требования*, *Анализ* и *Проектирование*, в результате которых создаются модели со стереотипами *Research Use Case Model*, *Research Analysis Model* и *Research Design Model*. Как и в общем случае, именно переход от PIM к PSM вызовет наибольшие трудности. И это будет связано с алгоритмами построения квазипараллельного процесса и определения модельного времени. В п.1.1 мы предложили использовать *обратный инжиниринг*. Это позволяет на практике неким образом обойти эти трудности благодаря тому, что в исходном тексте можно поместить *марки*. *Марки* (mark) используются для связывания; это самостоятельные структуры данных, принадлежащие не моделям, а схемам преобразования и содержащие информацию о созданных связях.

В этой монографии мы используем методы программной инженерии для создания имитационных моделей. Хотелось бы привести хотя бы один факт обратного влияния. В ООАП в большинстве случаев предполагается искать аналоги классов создаваемой программной системы в модели предметной области. В этом плане имитационная модель может быть использована следующим образом. Построим имитационную модель и убедимся в ее адекватности. Ограничим часть предметной области, которая должна быть автоматизирована. Тем самым мы получим работающею версию разрабатываемого ПО, а оставшаяся часть имитационной модели представляет собой набор модульных тестов. Такое нетрадиционное использование ИМ демонстрирует один из подходов к программированию, известный как *test-first programming* (программирование на основе тестирования).

2.2. Особенности моделирования социально-экономических систем

Как уже было сказано выше, *UML SP* позволяет создавать субпрофили, которые позволяют конкретизировать предметную область моделирования. Далее рассматриваются имитационные модели активных систем, заданные на языке субпрофиля *Scientific Profile for Active Systems.*

2.2.1. Субпрофиль Scientific Profile for Active Systems

На рис. 28. представлена диаграмма классов, определяющая объектные модели активных систем, и пример классов для одной из конкретных моделей активных систем. Субстанциональный класс Subject фиксирует определяющие качества активных систем – активность, управление, целенаправленность, и имеет метод exist «*Exist*» (он определяет единицу дискретно-событийного времени), свойства product {Concept = Продукт}, materials {Concept = Материалы} и внутренние процедуры managment {Concept = Управление} и business {Concept = Бизнес-процесс}. С точки зрения вычислительной семантики данный класс задает пользовательский тип TSubject, который определяет общий интерфейс для всех элементов модели. Активность будем моделировать «внутренними часами» субъектов, которые изменяют свое состояние, получая сообщение «*Exist*», и генерируют последовательность внутренних событий.

Метамодель субпрофиля следующая. Теория активных систем предлагает два базовых способа организации совместных действий (бизнес-процесса) активных элементов – механизм стимулирования и механизм планирования [42]. Экземпляр потомка класса Context моделирует окружающую среду изучаемой системы и определяет ее системную динамику посредством понятия *эффективность функционирования*

активной системы. Экземпляр потомка класса ActiveSystem моделирует саму активную систему (которая обычно рассматривается в роли центра). Экземпляры потомка класса Agent моделируют агентов, составляющих активную систему и обладающих свойством активности, в том числе способностью свободы выбора своего состояния. Агенты могут обмениваться сообщениями через экземпляры агрегации между классами Agent и ActiveSystem. Агрегация также используется как канал связи между центром и агентами. Рефлексивная ассоциация для класса Agent отражает возможность существования прямых связей между агентами. Ментальность агентов будем моделировать исходя из гипотез рационального поведения и благожелательности. Для многих задач также необходимо выделение отдельного класса рефлексирующего агента.

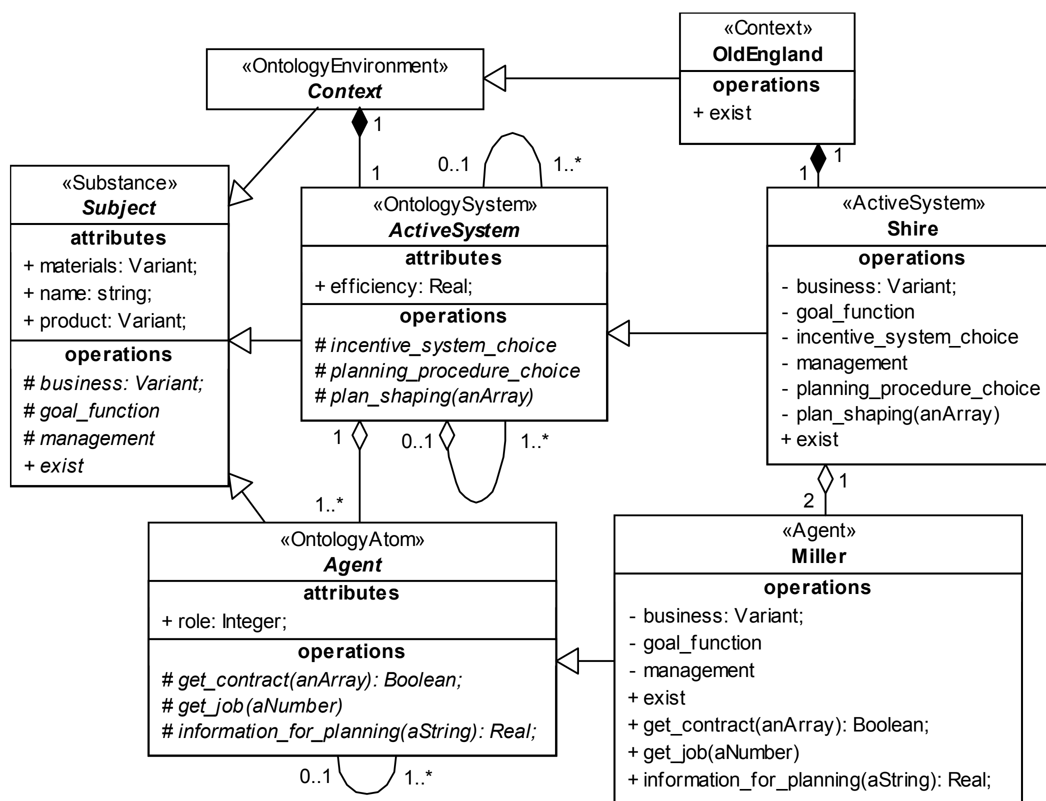


Рис. 28. Диаграмма классов для имитационной модели активной системы

Имитационная модель конкретной активной системы может иметь и другие механизмы управления, которые добавляются к базовым механизмам в классах-потомках.

Мы предполагаем, что для модели использован субпрофиль. Напомним, что стереотипы субпрофиля находятся в отношении обобщения со стереотипами профиля. Для определения концептов субпрофиля использован глоссарий из книги [42]. В п.2.1.2, проводя параллель с математикой, мы уже сравнивали диаграмму классов с уравнением, а диаграмму кооперации – с решением уравнения. В развитых точных науках, таких как физика, уравнения модели получают из общих уравнений, выражающих те или иные принципы этих наук. Рис. 28 как раз и демонстрирует, как из общей формулировки можно сразу получить диаграмму классов конкретной модели.

Модели агента. В случае общественных систем таковыми могут быть: индивидуальная личность, социальная группа, массовая совокупность. В случае экономических систем – различные экономические субъекты. Как уже сказано, агенты – это атомарные объекты и они моделируются схематически. Тем не менее, их свойства достаточно сложны. Подробнее о моделях агентов см. [42], [52].

Модели общественных систем. Теорию активных систем мы будем применять в основном для моделирования экономических и общественных систем. Имитационное моделирование довольно популярно в социологии [24]. Например, в ряде статей и книг, изданных на рубеже столетия коллективом авторов (А.К. Гуц, В.В. Коробицын, А.А. Лаптев, Л.А. Паутова, Ю.В. Фролова) предложены такие модели, как «искусственная жизнь на сахарных холмах», «гендерные отношения в искусственной жизни», этногенеза, социализации индивида, коллективных рефлексов и др. Сейчас кратко рассмотрим в качестве активных систем общественные системы.

Какие общественные системы возможны? Мы будем конструировать системы из субъектов коммуникации, задавая разные типы коммуникационных действий, опираясь на работу [51]. Приведем цитату из этой книги.

В качестве коммуникантов и реципиентов могут выступать *три субъекта*, относящиеся к разным уровням социальной структуры: индивидуальная личность (И), социальная группа (Г), массовая совокупность (М). Под социальной группой будем понимать множество людей, обладающих одним или несколькими общими социальными признаками и осознающих свою общность, выражая ее местоимением «мы». Массовая совокупность – множество случайно собравшихся людей – уличная толпа, пассажиры транспорта, массовая читательская (телевизионная) аудитория, население, общество в целом. Перечисленные субъекты могут взаимодействовать друг с другом (например, И – И, Г – Г, М – М) или между собой (например, И – Г, И – М, Г – М и т. д.). Тем самым, получается 9 видов общественных коммуникаций.

В обществе возможны *три типа коммуникационных действий*: подражание, диалог и управление. Коммуникационное действие – завершенная операция смыслового взаимодействия, происходящая без смены участников коммуникации. Под *подражанием* понимается воспроизведение реципиентом движений, действий, повадок коммуниканта. Подражание может быть произвольным и непроизвольным (бессознательным). Реципиент целенаправленно выбирает коммуниканта и использует его в качестве источника смыслов, которые он хотел бы усвоить. Коммуникант при этом зачастую не осознает своего участия в коммуникационном действии. Подражание – это такое объект-субъектное отношение, где активную роль играет реципиент, а коммуникант — пассивный объект для подражания. *Диалоговая коммуникация* представляется как последовательность высказываний участников, сменяющих друг друга в роли коммуниканта и реципиента. *Управление* – такое коммуникационное действие, когда коммуникант рассматривает реципиента как средство достижения своих целей, как объект управления. В этом случае между коммуникантом и реципиентом устанавливаются субъект-объектные отношения. Управление отличается от диалога тем, что субъект имеет право монолога, а реципиент не может дискутировать с коммуникантом, он может только сообщать о своей реакции по каналу обратной связи. Управленческий монолог может быть: в форме приказа (коммуникант имеет властные полномочия, признаваемые реципиентом); в форме внушения (суггестии), когда используется принудительная сила слова за счёт многократного повторения одного и того же монолога (реклама, пропаганда, проповедь); в форме убеждения, аппелирующего не к подсознательным мотивам, как при внушении, а к разуму и здравому смыслу при помощи логически выстроенной аргументации.

Комбинируя 9 видов социальных коммуникаций с тремя видами коммуникационных действий, получим пятнадцать допустимых комбинаций:

1.	микро межлич.	И п И	копирование образца
2.	микро межлич.	И д И	беседа
3.	микро межлич.	И у И	команда
4.	микро групп.	И п Г	референция (референтная группа)
5.	микро групп.	И у Г	руководство коллективом
6.	микро масс.	И п М	социализация

7.	микро масс.	И у М	авторитаризм
8.	миди групп.	Г п Г	мода
9.	миди групп.	Г д Г	переговоры
10.	миди групп.	Г у Г	групповая иерархия
11.	миди масс.	Г п М	адаптация к среде
12.	миди масс.	Г у М	руководство обществом
13.	макро масс	М п М	заимствование достижений
14.	макро масс.	М д М	взаимодействие культур
15.	макро масс.	М у М	информационная агрессия

Из вышесказанного видно, что конкретная модель активной системы должна отражать одну (или несколько) форм коммуникационной деятельности из этих пятнадцати.

Макроэкономическая модель Дж. М. Кейнса

В таких областях, как психология, социология, экономика и т.п., где применение традиционного математического аппарата вызывает известные сложности, можно ожидать наибольшего эффекта от применения *Scientific Profile*. Это ожидание можно объяснить тем, что классы можно использовать как фреймы (рамки), посредством которых можно структурировать сложные системы.

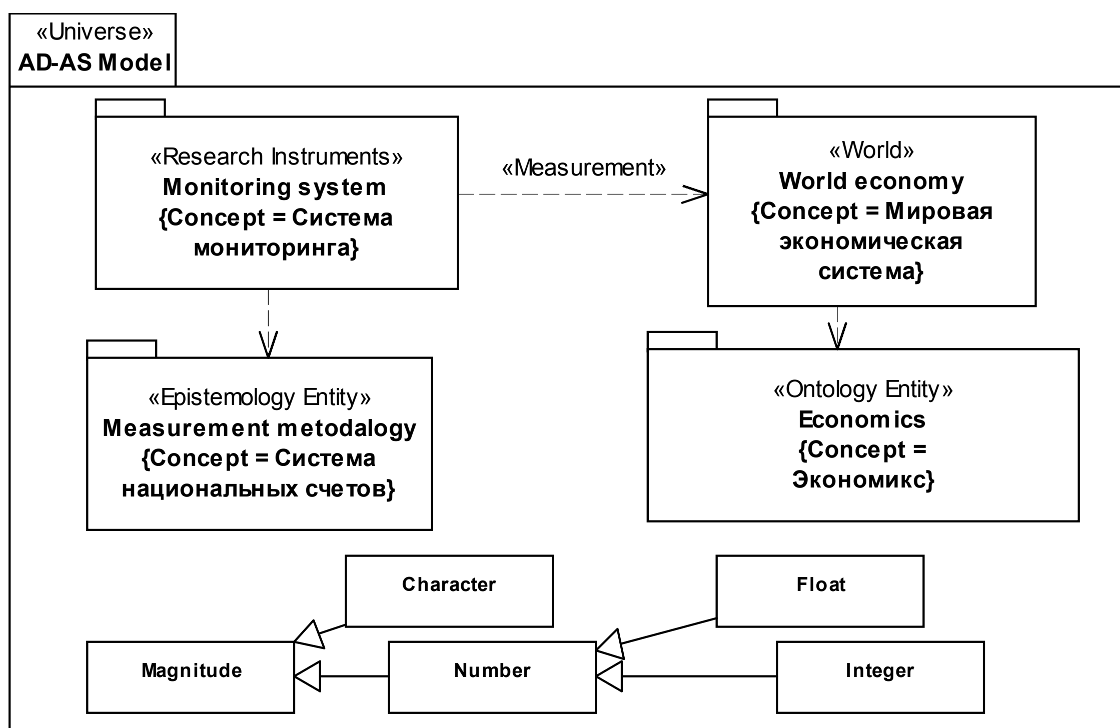


Рис.29. Архитектура модели Дж. М. Кейнса

В качестве типичного примера разработки имитационной модели в экономике рассмотрим обобщенную макроэкономическую модель Дж. М. Кейнса [36]. Далее мы будем предполагать, что модель Кейнса представлена онтологией. Пусть до некоторого момента времени T экономика находилась в состоянии равновесия, т.е. совокупный спрос равен предложению: $Y_D(t) = Y_S(t)$. Допустим, что в момент T совокупный спрос изменится. Цель *Исследователя* – определить новое равновесное состояние. Отметим, что это не динамическая, а квазистационарная задача.

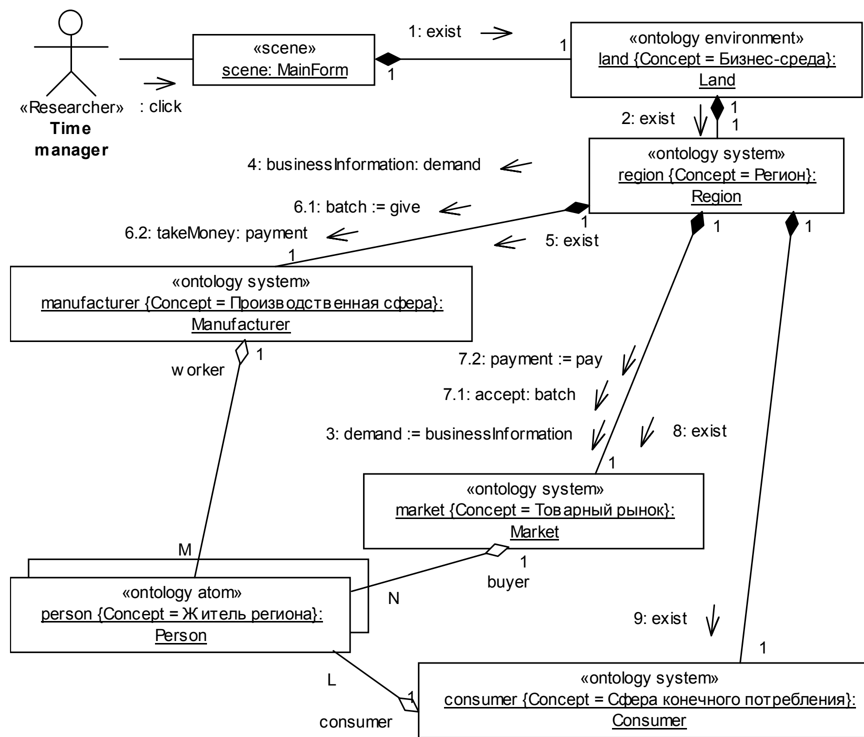


Рис. 30. Диаграмма кооперации, моделирующая субъектно-объектную структуру общественного воспроизводства

Research Analysis Model, представленная пакетом Analysis System, показана на рис. 29. Пакет Economics определяет законы функционирования СЭС и представлен классами Subject, ComplexSubject., Person и четырьмя активностями со стереотипом «*ontology activity*», моделирующими фазы общественного воспроизводства: *Production* ({Concept = Производство}), *Distribution* ({Concept = Распределение}), *Exchange* ({Concept = Обмен}) и *Consumption* ({Concept = Потребление}). Постулируется, что все активности являются параллельными.

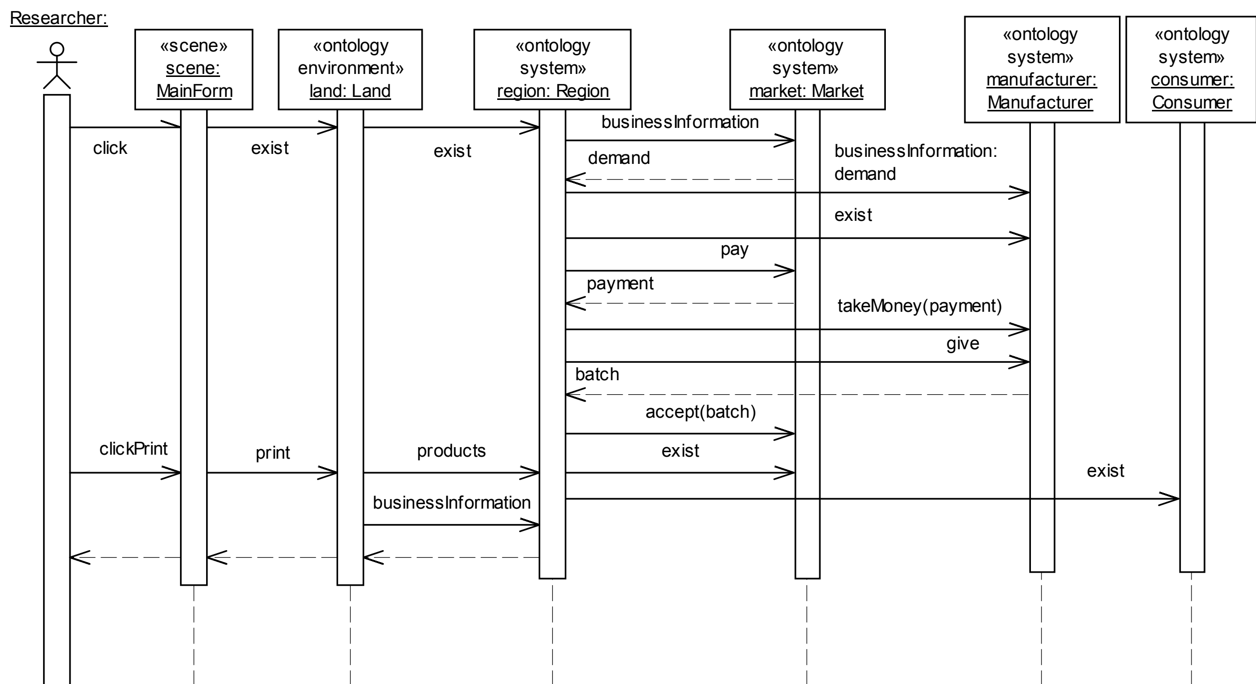


Рис. 31. Диаграмма последовательности действий, представляющая сценарий функционирования социально-экономической системы

Все классы являются потомками одного абстрактного класса Subject. Композиция этого класса содержит структуру space («*Ontology Space*», {Concept = Структура субъекта}). Пакет World Economy содержит реализации прецедентов «Приготовить начальное состояние», «Вычислить следующее состояние» и классы Land, Region, Manufacturer, Market и Consumer. Одноименные экземпляры классов используются для создания симуляции. Изучаемую систему моделирует объект region, контекст которой задается объектом land. Композиция объекта region включает объекты, моделирующие основные подсистемы: manufacturer (сфера производства), market, consumer (сфера потребления). Атомарный объект person моделирует жителя региона. Деньги, документы и продукты производства, в целях упрощения, моделируются объектами, имеющими класс Integer. Кооперация объектов, реализующих прецедент «Вычислить следующее состояние» показана на рис. 30 (в более полной модели также следует рассмотреть *Правительство* и *Банк*, которые здесь не показаны). Каналом передачи сообщений является внутренняя среда объекта region.

Research Design Model определяет аппроксимацию параллельных активностей *Production*, *Distribution, Exchange* и *Consumption* в виде последовательных деятельностей. Для анализа устойчивости удобно рассмотреть итерационное соотношение для Y_S и ввести новую переменную $y_t = Y_S(t) - Y_E$, которая равна отклонению текущего значения национального дохода от его нового равновесного значения Y_E. Если производственная функция и функция спроса линейны, поведение этой переменной описывается формулой геометрической прогрессии: $y_{t+1} = cy_t$, где $0 < c < 1$, т.е. в этом случае имеем бесконечно убывающую последовательность, модуль которой стремится к нулю. Ввиду линейности соотношения подобным же образом ведет себя начальная ошибка δy_t , т.е. алгоритм устойчив. Однако, как показано в монографии [36], в случае нелинейности возможно возникновение детерминированного хаоса, а алгоритм может стать неустойчивым.

Конечным артефактом имитационного моделирования будет дерево сценариев, представленное совокупностью вложенных пакетов диаграмм последовательности действий. Как правило, аналитическое решение можно получить на основе анализа диаграммы кооперации (рис. 30) только для некоторых случаев (см. рис. 31).

Имитационные модели макроэкономических систем описаны в учебных пособиях [11], [50].

Системная динамика Дж. Форрестера. Идеи кибернетики нашли свое выражение в системной динамике (СД). Эта методология настолько интересна, что мы решили проанализировать основные положения СД с точки зрения объектно-ориентированного моделирования (ООМ) более детально. Затем определим правила сопоставления основных конструкций системной динамики СД и ООМ. Применяя данные правила, можно строить объектные модели, опираясь на многочисленные известные валидные модели системной динамики.

По данной проблематике большая часть работ опубликована в первом десятилетии века, и почти все они обсуждают вопрос о сопоставлении моделей системной динамики с агентными моделями. В работе [5] (см. сайт [40]) показано, как модель СД может быть конвертирована в агентную модель. В качестве примера рассмотрена модель распространения нового продукта (Bass Diffusion model). В приложении приведено соотношение типичных конструкций СД и соответствующих агентных моделей. Хотя агентное моделирование и ООМ – это разные подходы к имитационному моделированию (они различаются методологиями: моделирование *снизу вверх* и *сверху вниз*), тем не менее, многие агентные решения могут почти без изменений использоваться в ООМ. Приведенные соотношения [5], к сожалению, недостаточны для создания моделей ООМ, поскольку в них отражен только процессный аспект. Отдельного внимания заслуживает работа [26] (см. сайт [40]). В книге рассматривается преобразование системно-динамической модели в код языка программирования типа Pascal или С. Показано, что

основная проблема преобразования модели связана с переходом от параллельной композиции процессов к квазипараллельному моделированию. Для решения этой проблемы авторы предлагают специальную методику, основанную на анализе диаграмм зависимостей переменных. Приведенные в этой работе результаты ориентированы на процедурное программирование, однако допускают распространение данного метода и на объектно-ориентированное моделирование.

Определим правила сопоставления элементов моделей СД и программных конструкций ООМ. Системная динамика Дж. Форрестера базируется на ряде концепций [61]. Основные положения СД мы группируем следующим образом.

Концепция вещественных и информационных сетей. Эта концепция проводит четкое разграничение между материальными и информационными потоками. Для материальных потоков характерен закон сохранения. Напротив, для информации характерна возможность накопления и тиражирования. Информационные связи управляют вещественными потоками. В ООМ материальные потоки моделируются потоковыми классами (TThread) или классами, в которых копирующий конструктор объявлен приватным (для квазипараллельного моделирования).

Уровни и темпы. Динамику поведения сколь угодно сложного процесса можно свести к изменениям некоторых фондов или, как говорят, уровней, а сами изменения регулируются темпами наполняющих или исчерпывающих фонды потоков. Темп характеризует потоки, входящие в резервуары или выходящие из резервуаров. Изменение уровня определяется только темпами потока. Мы будем исходить из того, что системная динамика в ООМ проявляется в следующем. Объект инкапсулирует структуру подсистем, предоставляя другим объектам для взаимодействия только интерфейс. Этот интерфейс и есть отражение уровней (свойства) и потоков (методы).

Функции решений, которые управляют темпами. Величина темпов регулируется информацией об уровнях. Темп не может непосредственно воздействовать на другой темп, так же как и уровень не может воздействовать на другой уровень. Как подчеркивает Дж. Форрестер, эти функциональные зависимости обычно являются существенно нелинейными. В моделях ООМ многие подобные соотношения получаются естественным образом из алгоритмов взаимодействия объектов.

Концепция петли обратной связи. В самом общем случае петля обратной связи моделируется циклом. В теории динамических систем петле обратной связи соответствует понятие предельного цикла, причем цикл может быть как аттрактором, так и репеллером. Тем самым, можно говорить о петле обратной связи с положительной или отрицательной обратной связью. Специальным случаем петли обратной связи в ООМ будет диалог – коммуникативный акт с ответом. В отличие от СД коммуникативный акт является дискретным. Моделируется вызовом метода класса с некоторым возвращаемым значением.

Запаздывание. В самом общем случае понятие *запаздывания* в ООМ связано с таким понятием, как синхронизация. Для синхронизации потоков обычно используется приостановка вычислений потока. Однако в ряде случаев такое описание будет неадекватным и следует использовать *активное ожидание*. Поток не приостанавливается, а непрерывно проверяет выполнение некоторого условия. Когда данное условие будет выполнено, поток продолжает вычисления. Типичным примером является паттерн *Producer*/*Consumer*. Поток *Producer* пишет в буфер (буфер обычно реализован как очередь), *Consumer* читает из буфера. Если в буфере нет данных, то поток Consumer ожидает поступления данных, и используется для этого активное ожидание. В некоторых реализациях этого паттерна поток Producer также проверяет, что данные были считаны из буфера. Тем самым, элемент СД *запаздывание* в ООМ реализуют такие программные конструкции, как *Producer*/*Consumer*. Запаздывание может иметь место как для вещественных потоков, так и для информационных связей. Информационное запаздывание иногда можно моделировать линиями задержки.

Продемонстрируем вышесказанное на следующей типичной модели системной динамики. Гидравлическая система состоит из двух емкостей A и B, каждая из которых характеризуется уровнем воды; начальные значения l_A и l_B. Емкости соединены двумя трубами. По первой трубе вода подается насосом из емкости A в емкость B. Поток постоянен и составляет q литр в секунду. По второй трубе вода поступает из емкости B в емкость A и поток пропорционален l_B. Коэффициент пропорциональности d устанавливается краном. Кроме того, предусмотрена задержка во времени на t_d секунд управляющего воздействия. Заметим, что с гидравлической аналогией необходимо обращаться осторожно. Приведенный далее код описывает работу насоса с управляемой мощностью, а не свободное истечение жидкости. Из внешней среды в емкость A поступает постоянный приток воды q_e литр в секунду. Требуется определить значения l_A и l_B через t секунд.

Для представления модели СД использовалась система имитационного моделирования Vensim (Ventana Systems, Inc.). Соответствующая модель представлена на рис. 32.

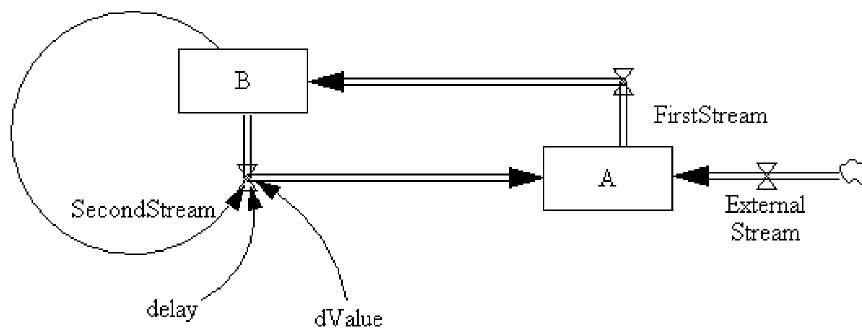


Рис. 32. Модель системной динамики

Обратимся теперь к объектной модели. Модель задается диаграммой классов рис. 33. Гидравлическая система моделируется классом Pipeline. Ограничимся моделью «толстого слоя» ($l >> 0$). Класс Surroundings моделирует окружение системы и определяет начальные и граничные условия для системы Pipeline (ExternalStream). Емкости A и B моделируется разделяемыми объектами класса Barrel – это поля A и B класса Pipeline. Подсистемы моделируются двумя классами FirstStream и SecondStream, объекты этого класса конфигурируются объектами A и B. Атомарные объекты – слой воды – моделируются экземплярами класса Drop и владеют собственными вычислительными потоками (thread). Класс Drop обладает свойствами upper и under, которые отражают свойство связанности слоев воды друг с другом (слипание).

Прямое моделирование параллельности предполагает только синхронизацию потоков. Действительно, каждый из объектов Barrel взаимодействуют с потоками FirstStream и SecondStream так, что один поток ставит в очередь, а второй поток – читает из очереди объекты Drop. Тем самым ситуации гонок не возникает. Единственно, что необходимо сделать – это синхронизировать интенсивность этих процессов. Для этого используется барьер, выполненный на основе разделяемого счетчика. Поток, выполнив цикл чтения/записи, понижает счетчик на единицу и приостанавливает себя. Когда значение счетчика становится равным нулю, поток Pipeline перезапускает потоки.

С переходом к квазипараллельному описанию модель существенно упрощается. Можно не инкапсулировать процессы течения в объектах и тем самым обойтись без классов FirstStream, SecondStream и ExternalStream. Метод Exist класса Pipeline тогда следующим образом определит единицу дискретно-событийного времени системы:

```
Drop *d;
for (int i = 0; i < 1; i++) {
```

```
d = this->A->Remove(); this->B->Add(d);
};

int d_level = DelayLine(B->level, delay); // запаздывание
int control = d_level / dValue; // управлеиие
for (int i = 0; i < control; i++) {
d = this->B->Remove(); this->A->Add(d);
}; .
```

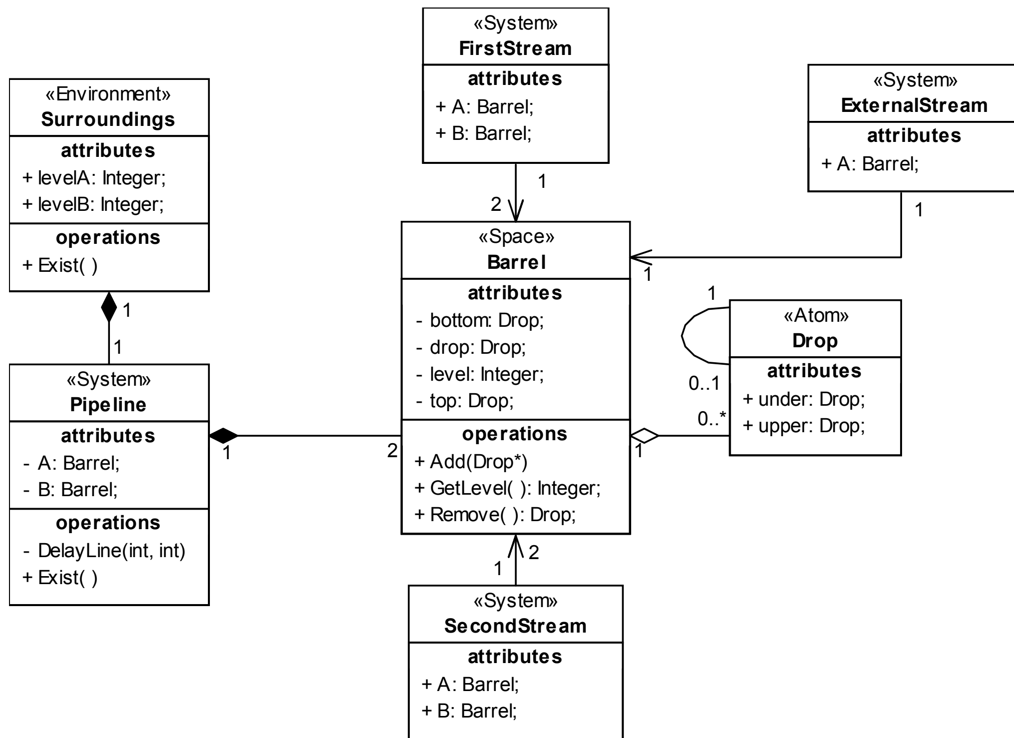


Рис. 33. Объектная модель

Первый оператор for моделирует поток воды из A в B. Второй оператор for моделирует поток воды из B в A. Интенсивность второго потока определяется величиной d_level, которая вычисляется в линии задержки DelayLine. В следующей строке кода на основе информации d_level и коэффициента пропорциональности dValue вырабатывается управляющая информация control.

Сравним обе модели. Оба цикла for моделируют материальные потоки, для которых должен выполняться некоторый инвариант цикла Exist. В частности, таковым инвариантом является величина $l_A + l_B. = const$ (закон сохранения количества жидкости). Этот закон выполняется, если внешний поток равен нулю. В модели системной динамики эти процессы изображены двойными стрелками. Данный инвариант используется для определения уравнений баланса имитационной модели. Кроме материальных потоков, имеют место два информационных процесса. Один процесс передает значение уровня емкости B в переменную управления второго цикла for. Другой информационный процесс – это процесс запаздывания. В модели системной динамики эти связи изображены сплошными стрелками. Функция решения в данном случае – вычисление величины control. Обозначается в СД как вентиль.

Сравнительный анализ объектной модели и модели системной динамики позволяет сделать вывод о гомоморфном отображении моделей ООМ на модели системной динамики. Объектные модели требуют привлечения дополнительных предположений, что делает их более содержательными. В частности, многие «загадочные» нелинейные зависимости СД в рамках ООМ получают естественное выражение как алгоритмы работы с объектами. Напомним, что в популярной системе AnyLogic есть возможность

совмещения агентных моделей и моделей системной динамики. Приведенные выше сопоставления говорят о том, что действительно возможно сопряжение обоих разновидностей моделирования, причем на глубинном теоретическом уровне.

2.2.2. Моделирование сложного поведения систем

В дискретно-событийных имитационных моделях разнообразие поведения модели достигается за счет использования условного выбора в коде программы. Однако если необходимо моделировать сложное поведение, как, например, процессы развития систем, то соответствующий подход вызывает определенные затруднения. Это связано с тем, что в процессе развития система проходит ряд стадий, которые могут кардинально изменить процессы, протекающие в системе. Под сложным поведением систем мы далее будем понимать, прежде всего, неоднородное поведение систем во времени. В качестве типичного примера приведем функции Эйри **Ai**(x) и **Bi**(x) (см. рис. 34), являющиеся решением уравнения $y'' - xy = 0$ и характеризующиеся разным поведением в отрицательной и положительной областях аргумента x (этот пример предложен Н.А. Ращепкиной). Удачным подходом может оказаться модель времени, рассматриваемая в теории хронотопов. В этом разделе рассматривается модель, реализованная на основе паттерна *State*, и которая позволяет описывать существенно разнородное поведение систем, как во временном, так и в пространственном срезе.

Термин *Хронотоп*, введён А.А. Ухтомским в контексте его физиологических исследований. Затем, во многом благодаря М.М. Бахтину, это понятие получило широкое распространение в гуманитарной сфере. «Ухтомский исходил из того, что гетерохрония есть условие возможной гармонии: увязка во времени, в скоростях, в ритмах действия, а значит и в сроках выполнения отдельных элементов, образует из пространственно разделенных групп функционально определенный «центр» («В литературно-художественном хронотопе имеет место слияние пространственных и временных примет в осмысленном и конкретном целом. Время здесь сгущается, уплотняется, становится художественно-зримым; пространство же интенсифицируется, втягивается в движение времени, сюжета, истории. Приметы времени раскрываются в пространстве, и пространство осмысливается и измеряется временем. Этим пересечением рядов и слиянием примет характеризуется художественный хронотоп» М. М. Бахтин). В данном разделе рассмотрим возможность применения объектных моделей хронотопа для имитационного моделирования сложного поведения систем.

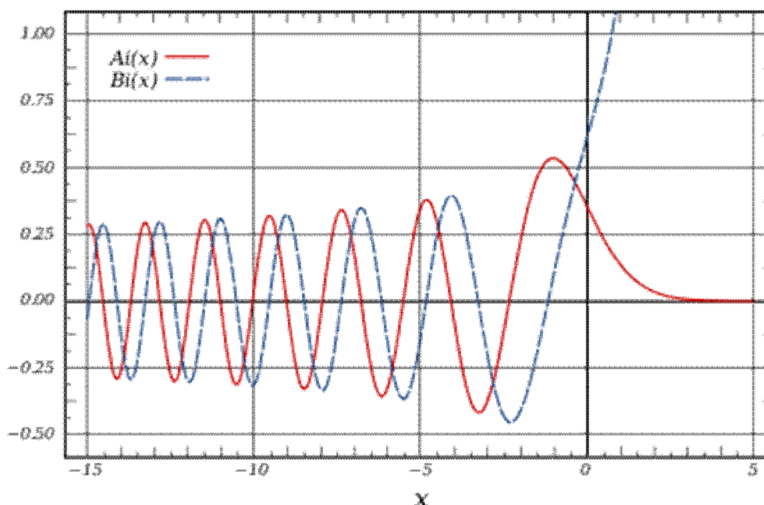


Рис. 34. Функции Эйри

Напомним способы моделирования пространства и времени в *UML SP*. Пространство моделируется динамическими списками из объектов класса SCell. Поле prestoredObject хранит указатель на некий объект, который расположен в данной ячейке пространства. Класс имеет свойства right и left, которые содержат указатели на правого и левого соседа (топология пространства). Класс имеет свойства get и put:aObject, позволяющие извлекать и помещать указатель на хранимый объект. Время в SP

рассматривается исключительно как дискретно-событийное, и продвижение по времени осуществляется посылкой сообщения со стереотипом Exist.

Для того чтобы ввести понятие хронотопа в имитационном моделировании, рассмотрим следующий пример.

Модельная задача «Грузовые перевозки». Пусть в пункте Shop производится погрузка товаров в грузовой автомобиль. Товары доставляются в пункт Home, где автомобиль разгружается. Процесс движения автомобиля рассматривать не будем. Объект Shop находится в двух состояниях: «Погрузка» и «Простой». Объект Home также находится в двух состояниях – «Простой» и «Разгрузка». Вся система в целом находится в одном из двух состояниях: ⟨«Погрузка», «Простой»⟩ или ⟨«Простой», «Разгрузка»⟩.

На рис. 35 приведена диаграмма классов для объектной модели. *Исследователь* инициирует моделируемый процесс, посылая сообщение со стереотипом *Exist*, которое последовательно передается всем объектам модели по экземплярам агрегации.

Далее предполагается, где это возможно, что концепты совпадают с именами классов. Класс TransportNetwork моделирует контекст модели; он определяет системную динамику – свойства и методы класса CargoTransportation. Экземпляр класса CargoTransportation моделирует изучаемую систему. Экземпляр класса MotorVehicle моделирует грузовой автомобиль. В композицию класса CargoTransportation входит динамический список из двух элементов класса SCell; экземпляры этого класса моделируют Shop и Home. Указатель на экземпляр класса MotorVehicle хранится в поле prestoredObjects. Процессы, инкапсулированные в объектах классов TransportNetwork, CargoTransportation, SCell и MotorVehicle, считаются параллельными. Синхронизация между ячейками Shop, Home и объектом класса MotorVehicle осуществляется путем обмена сообщениями.

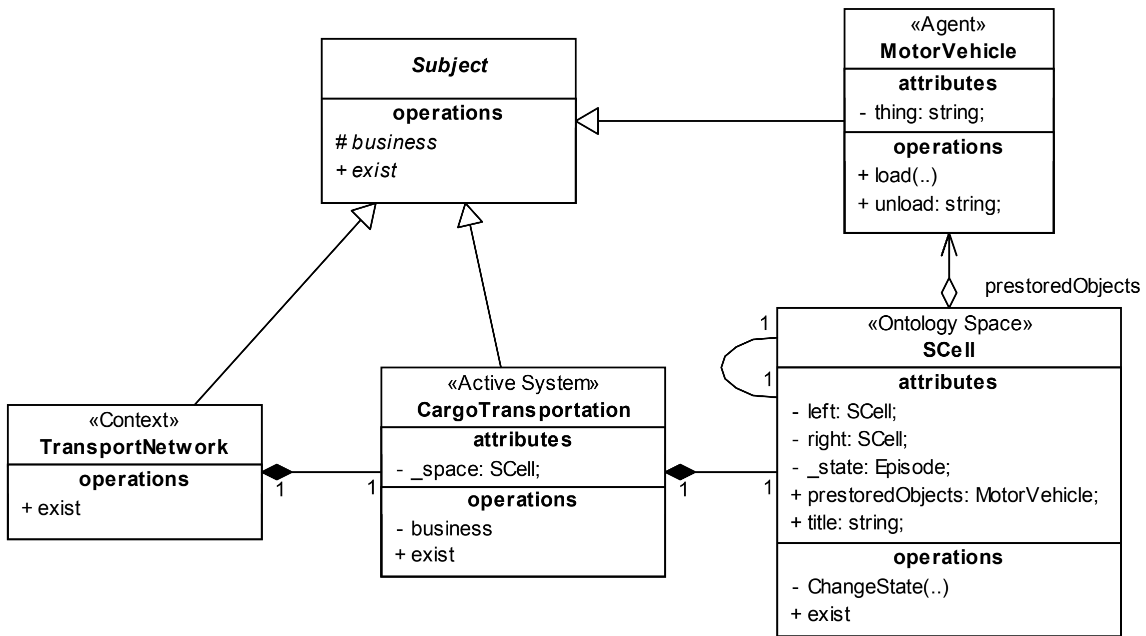


Рис. 35. Диаграмма классов для модели «Грузовые перевозки»

Выполнение методов load и unload зависит от того, в какой ячейке находится объект класса MotorVehicle и определяется в процедуре business метода exist класса CargoTransportation. Выбор поведения можно осуществить путем проверки значения поля title и условия prestoredObjects == NULL.

Модель хронотопа. Рассмотрим альтернативный способ описания активности модели, используя для создания композиции паттерн *State*. Композиция SCell показана на рис. 36, роли классов расписаны в соответствии с описанием паттерна в книге [14].

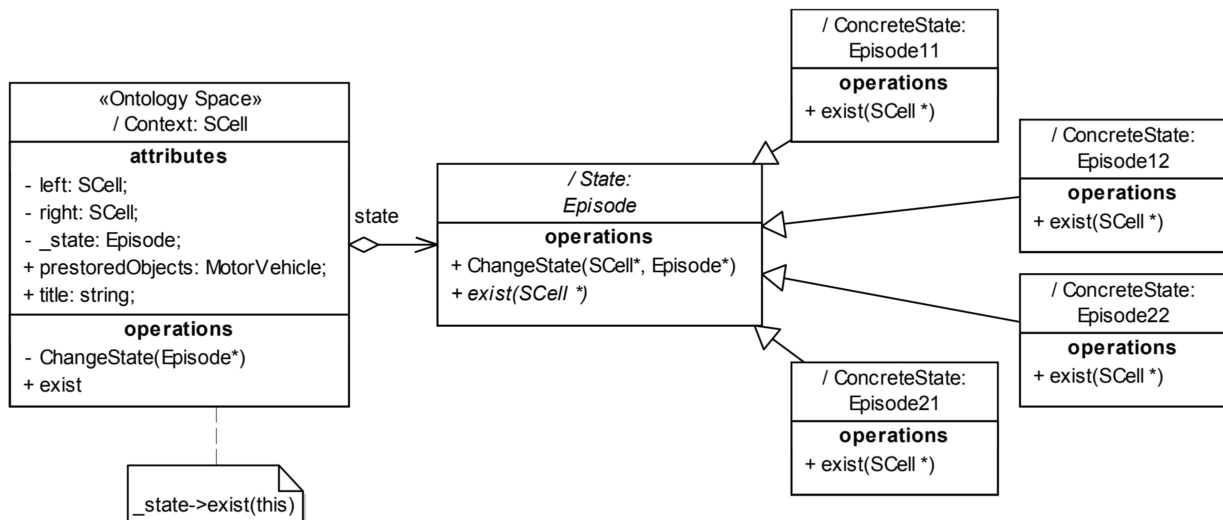


Рис. 36. Паттерн State для модели хронотопа

Класс SCell, моделирующий ячейку пространства, выступает в роли контекста паттерна и имеет поле _state класса Episode. Класс Episode {Concept = Эпизод} объявлен другом класса SCell, что дает ему привилегированный доступ к приватным полям и процедурам SCell. Это позволяет манипулировать с объектами, входящими в композицию класса SCell, а также вызывать процедуру ChangeState(Episode*), которая изменяет поле _state. Подклассы Episode реализуют поведение в конкретных ячейках пространства – Episode11 и Episode12 в ячейке Shop, а Episode21 и Episode22 в ячейке Home. Каждая процедура exist подклассов Episode заканчивается переходом в следующее состояние. Например, для Episode11 соответствующий фрагмент кода будет иметь вид:

```
Episode * tNext = new Episode12;
ChangeState(j, tNext);
```

где j – указатель на объект класса SCell. Неустойчивые состояния системы моделируются условным выбором.

Чаще всего хорошим приближением является циклическая смена состояний, как в рассмотренном примере. Однако в некоторых задачах необходимо обеспечить уникальность эпизодов. В этом случае можно воспользоваться паттерном *Singleton*.

«*Research Analysis Model*» не должна зависеть от конкретного языка программирования; различия проявляются только в «*Research Design Model*». Рассмотренная выше программная конструкция была реализована как на C++, так и на Smalltalk. Паттерн *State* по-разному реализуется на этих языках, однако диаграмма классов Рис. 28 будет в обоих случаях одинаковой.

Задача преобразования композиции параллельных процессов «*Research Analysis Model*» в квазипараллельный процесс для «*Research Design Model*» может быть решена одним из двух способов – опираясь на концепцию событийно-ориентированного или процессно-ориентированного моделирования. Например, в нашем случае, можно определить фиктивное свойство isEnabled в классе MotorVehicle и перед передачей объекта в смежную ячейку устанавливать isEnabled = false, что позволяет исходную активность разбить на совокупность независимых деятельностей. Можно показать, что между процессами имеет место отношение *наблюдаемой конгруэнции* по Р. Милнеру. В этом же рабочем потоке определяется модельное время, которое позволяет синхронизировать состояния объектов.

Предметная семантика. Предметная семантика рассмотренных конструкций определяется понятием *хронотоп*, понятие, которое нашло широкое применение в естественных, и особенно в гуманитарных науках. Под хронотопом мы понимаем последовательность эпизодов, связанных с одной и той же пространственной областью. В нашем случае существуют хронотопы Home, Shop и системы в целом.

Основанием для подобной интерпретации будет следующее. Существенной особенностью хронотопа является тесная связь пространства и времени. Особенно четко это видно на описанной выше модели. В пространстве кинотеатра возможен процесс «Просмотр», в пространстве кафе возможен процесс «Прием пищи». В пространстве пустыни возможен процесс «Идти» и невозможен процесс «Пить воду». Набор объектов манипулирования определяет структуру ячейки пространства. Последовательность действий эпизода определяет временную структуру. Программная конструкция поддерживает единство пространства и времени путем согласования пространственной и временной структуры, причем объекты манипулирования выступают в качестве посредников этого согласования. С одной стороны, поскольку метод exist определен в классе SCell, можно создавать модели с разнородным темпом времени в разных ячейках пространства. Тем самым появляется возможность моделировать системы, разные части которой имеют разные темпы развития, что характерно для больших систем. С другой стороны, поскольку эпизоды хронотопа связаны с одной ячейкой пространства, а ячейки пространства все имеют класс SCell, имеет место преемственность – дискретный аналог непрерывности пространства и времени.

Выбор механизма продвижения модельного времени в значительной степени предопределен моделью времени, которая задается в «*Research Analysis Model*». Проиллюстрируем это на материале книги [2, с.97-101]. Ф. Варелла предложил шкалу актов восприятия для человека, которая состоит из интервалов продолжительностью 0.1, 1 и 10 с. События 0.1 с. он называет моментными когнитивными актами; в нашем случае это будет соответствовать выполнению отдельных операторов методов со стереотипом *Exist*. Акт восприятия 1 с. характеризуется целостностью, соответствует коротким и законченным движениям и моделируется обычным квантом дискретно-событийного времени системы, а механизм продвижения модельного времени имеет список событий, составленный из выделенных точек этих движений. Мгновенный снимок состояния системы после выполнения процедур методов со стереотипом *Exist* можно назвать *кадром* состояния системы. Акт восприятия продолжительностью 10 с. уже предполагает осознание прошлого и будущего. Для моделирования времени может быть введена некоторая переменная, общая для всех ячеек пространства, и которой может быть приписан концепт темпорального характера (тем самым переменная становится онтологической сущностью) – «если было выполнено действие *a*, то возможно действие *b*». Механизм продвижения модельного времени должен определяться этой переменной, т.е. особым событием продвижения времени становится *кадр*. Это переходная форма времени между дискретно-событийным временем и «дискретно-кадровым» временем эпизода.

Модель хронотопа предполагает более протяженные интервалы, в течение которых выполняются некоторые законченные действия (как например, погрузка автомобиля) и эти действия уже привязаны к конкретному месту. Механизм продвижения модельного времени должен в качестве особых событий использовать некоторые выделенные кадры. Время становится двухуровневым – «быстрым», дискретно-событийным между двумя кадрами и «медленным», дискретно-кадровым в эпизоде. В хронотопе, на третьем уровне, время можно определить как «историческое»: особые события – это все или некоторые финальные кадры эпизодов. Таким образом, модель хронотопа – это модель иерархического дискретно-событийного времени. Причем в этой модели именно «историческое» время является определяющим. Хронотоп системы не однозначен: в зависимости от целей исследования в качестве эпизода можно выбирать действия разного

временного масштаба. Заметим, что в контексте модели (класс TransportNetwork) время остается по-прежнему дискретно-событийным.

Некоторые примеры. Приведем еще несколько примеров моделей хронотопов, разбив их по категориям (следуя М.М. Бахтину). Эти модели показывают то, как уже было сказано выше, что в хронотопах акцент делается на «историческом» времени.

Хронотоп неравномерного времени. Для каждой пространственной ячейки один эпизод, но в качестве аргумента конструктор класса получает некоторое целое число $H_i = kH_{i-1}$, где H_{i-1} – текущее значение H ($H_0 = 1$), а $k \neq 0$ – некоторый целый коэффициент. Число $H>0$ определяет количество повторений процесса эпизода для одного сообщения «*Exist*»; если $H<0$, то некоторые сообщения «*Exist*» игнорируются.

Хронотоп циклического времени. Один эпизод многократно повторяется, объект хранения имеет собственную линейную историю. Выход из цикла будет иметь место только тогда, когда будет выполнено некоторое условие. Эта модель интересна тем, что требуется формальное доказательство того, что данное условие будет выполнено когда-либо в будущем или, напротив, никогда не будет выполнено.

Хронотоп дороги. Ячейка пространства моделирует средство передвижения – каюту, купе, автомобиль. Перемещение моделируется специальным сообщением, изменяющим состояние системы и источником которого является агент. Эта модель иллюстрирует то, что модель пространства зависит от выбора системы отсчета. А. Брудный так, например, говорит о дороге: «В рамках хронотопа дороги возникают новые информационные контакты». Т.е. модель дороги рассматривается как коммуникационный процесс.

Хронотоп когерентного времени (лат. cohaerentia – сцепление, связь). Пусть есть две пространственные ячейки и два эпизода для каждой из них (e_{11}, e_{12} и e_{21}, e_{22}). Смена эпизодов выполняется по следующему правилу: если для первой ячейки e_{11}, то для второй ячейки следующим эпизодом будет e_{21} и наоборот, если для второй ячейки e_{21}, то для первой ячейки следующим эпизодом будет e_{11}. То же верно для пары (e_{12}, e_{22}). Эта модель интересна тем, что выбор эпизода зависит от состояния смежных ячеек и выборы согласованы между собой. Пример – когерентность экономических процессов в двух соседних странах.

Хронотоп исторического времени. Рассмотрим процесс в одной пространственной ячейке, такой, что эпизод e_0 сменяется эпизодом e_1 или e_2, а затем опять e_0. Выбор альтернативы зависит от предыстории: $e_1 \Rightarrow e_0\mathbf{T}e_2$ и $e_2 \Rightarrow e_0\mathbf{T}e_1$ ($a\mathbf{T}b$ – «a и затем b»). Модель интересна тем, что демонстрирует сильную зависимость от истории. Пример - динамика политической жизни в двухпартийной политической системе.

Примеры и пояснения

1. Модели диалогов. Одним из интереснейших направлений моделирования является моделирование диалогов. Перечислим некоторые простые модели диалогов:

- модель танца – четыре пространственные ячейки, движение агентов навстречу и обратно;

- модель спортивной игры (футбол, волейбол, теннис) – три ячейки, в двух крайних ячейках – команды (игроки); игроки поочередно меняют состояние мяча, который в центре;

- модель нематериальной деятельности – два агента последовательно пересылают друг другу образ нематериального предмета;

- модель материальной деятельности – два агента поочередно воздействуют на предмет, придерживаясь определенного алгоритма (например, чтобы была последовательность чисел);

- модель экономических отношений. Обмен объектами, имеющими ценность. Обмен товара на деньги;

- многочисленные правила этикета.

2. Модель распространения слухов [45]. В основе модели лежат следующие положения. Слух предполагает однократною воспроизводимость перед данным слушающим. Второй раз одному и тому же человеку данный слух не пересказывается. Затем роли меняются. Слушающий становится говорящим, и этот слух передается дальше. Подобные сообщения называют *самотрансляционными*. К подобным сообщениям относятся также анекдоты и «письма счастья» (Chain Letters). Один из приемов рекламы – «идея-вирус». Для слухов должно выполняться следующее условие. Как правило, слух содержит информацию, умалчиваемую средствами массовой коммуникации. Обратно: слух никогда не повторяет того, о чем говорят средства массовой коммуникации. В этой модели СМИ моделируются паттерном *Observer*, а атомарные агенты имеют список контактов.

3. «Поход Чингисхана». В настоящее время, и особенно в древности в военных действиях использовались слухи. Известно, что войска Чингисхана опережали рассказы об их невероятной жестокости, тем самым подрывая моральный дух их противников. Напрашивается аналогия со снежной лавиной, впереди которой идет ударная волна. Из примеров недавнего прошлого можно назвать практику применения слухов советскими войсками в Афганистане. Тем самым слухи становились мощным информационным оружием (точнее поражающим фактором). В этой задаче есть два интересных момента. Имитационная модель, которая позволяет оценить поражающий эффект слухов и паттерны, моделирующие войско (например, паттерн *Facade* [14]; модель игры «Морской бой»).

4. Мода — основанная на подражании передача в социальном пространстве вещественных форм, образцов поведения и идей, эмоционально привлекательных для социальных групп [51]. В модели один объект запрашивает у второго объекта значение некоторого свойства и затем устанавливает для себя то же значение.

На групповом уровне возможна референция, то же подражание, но не отдельному человеку, а социальной группе, с которой индивид желает себя идентифицировать. В отличие от моды копируемый признак далее используется как идентификатор в схеме «свой-чужой».

5. Модель привратника [45]. Идея «привратника» (gatekeeper; сторож у ворот, у входа куда-либо) принадлежит Курту Левину. «Привратником» признается тот, кто контролирует поток новостей, может изменять, расширять, повторять, изымать информацию. Исследования Д. Уайта показали, что реально используется только 10% новостных сообщений.

Рассмотрим следующую модель. Новости – случайные числа от 0 до 9. Объект Масс-медиа принимает их и передает аудитории – нескольким объектам. Пусть задан алгоритм поведения агентов, который зависит от значения этих чисел. Появляется информационная потребность – знать эти числа. Для рефлексирующих агентов – корректировать информационную картину мира.

6. Информационная безопасность [45]. В Европе есть страны, в которых информационные пространства государств пересекаются. Возникает проблема точки зрения, а не только фактического наполнения информацией этого пространства. По ключевым событиям могут возникнуть противоположные виды интерпретаций.

Пусть государство B находится в информационном поле государства A. Пусть происходит некоторое событие P (например, присвоение некоторой переменной P некоторого значения). Образ P' этого события для граждан B будет отличаться от желаемого для правительства B образа P_0, как впрочем, и от образа P'' граждан A. В основе модели использован паттерн *Observer*, где имеет место два класса ConcreteSubject. Если вернуться к модели *Прогулки Иммануила Канта*, то колокольня будет извещать об одном времени, циферблат о другом, а астрономическое время будет третьим.

7. Модель разведки [45]. В классическом смысле под разведкой понимают сбор информации, оценку ее достоверности и объединение отдельных фактов в аналитическую

картину. Работа разведки во многом сводится к поиску необходимой информации. Главным же образом она концентрируется на обработке информации. Классическим примером этого являются результаты, когда исследователи спокойно устанавливали весь военный состав командования вермахта с помощью объявлений о браке или о смерти в газетах.

Пусть имеется некоторая древовидная структура, узлы которой именованы. Пусть имеется также образ этой структуры, узлы которой не именованы. В процессе своего существования исходная структура создает некоторый информационный шум; необходимо обработать его так, что бы именовать все узлы образа.

8. «Друг моего друга». Пусть есть агенты *a, b, c*. Агент *b* общается с агентом *a* по субботам. Агент *c* общается с *a* по воскресеньям. Агенты *b* и *c* общаются между собой в остальные дни недели. Агент *b* создает фантомного агента *a*' по рассказам *c* об *a*. Агент *c* создает фантомного агента *a*'' по рассказам *b* об *a*. При определенных условиях всегда будет иметь место $a' \neq a''$. Более того, агент *a* узнает об *a*' и *a*'' и, возможно, никогда не догадается о том, что это он сам.

9. Точка встречи. Достаточно широкий класс моделей, который довольно часто обыгрывается в художественной литературе. Рефлексирующие агенты могут выполнить коммуникативный акт или даже совместные действия, не договариваясь заранее. Подобные задачи довольно подробно рассмотрел Т. Шеллинг (Scbelling T. The Strategy of Conflict, Cambridge (Mass.) 1960). Рассмотрим двух рефлексирующих агентов. Пусть один из них узник, заключенный в некую темницу. Его партнер находится вне темницы и желает помочь узнику. В каком месте они должны попытаться услышать друг друга? Будем полагать, что стена всюду имеет одинаковую толщину, однако достаточную, чтобы услышать звуки ударов. Если темница в плане есть окружность, то задача не имеет решения. Для полуокружности есть два решения – точки пересечения отрезка диаметра и окружности. Допустим, что центр отрезка мы несколько сместим в сторону от центра окружности, так, чтобы получилась ломаная. Казалось бы, будет три решения, поскольку будет три угла. Однако понятно, что в данном случае будет только одно решение – «странный» угол около центра. Шеллинг назвал такие места фокальными точками.

Вместо реального обмена сообщениями, агенты обмениваются сообщениями с фантомными агентами и, тем не менее, получают реальный результат (например, встречаются, или даже проламывают стену, как в задаче узника). По мнению В. Лефевра эта задача моделирует ситуацию контакта земной и внеземной цивилизаций [37].

10. «Мистер и миссис Смит». Агенты *a* и *b* участвуют в процессе *S*, для выполнения которого необходимы фантомные агенты *a*' и *b*', которых они старательно и играют. Не зная о том, они участвуют также в процессе *S*'.

11. Неповторимость. Если мы хотим моделировать уникальные временные кадры, то для этого надо использовать паттерн *Singleton* [14]. В нашем примере в классах-потомках SCell для этого зарезервирована процедура static SCell* Instance(), которая позволяет создавать объекты соответствующих классов.

12. Циклическое время. Последнее связано с существованием таких циклов, как суточный, месячный (лунный) и годовой. Циклы возникают и тогда, когда имеет место сильная связь между процессами по причинно-следственным отношениям – завершение одного процесса приводит к запуску другого по кругу. Многие модели хронотопов (как и в рассмотренных примерах) с хорошей точностью описываются циклическими процессами. Здесь уместно вспомнить известную притчу о человеке, пожелавшего повторить последний день; результат – бесконечное циклическое время. Ацикличность и однонапрвленность времени можно моделировать, если некий объект будет иметь собственную историю. В качестве примера приведем следующую фантастическую историю.

Хронотоп «День сурка» (Groundhog Day*)*. Название происходит от фильма «День сурка» – фантастической комедии режиссёра Гарольда Рэмиса, созданной по истории

Дэнни Рубина. День сурка – это вид временной петли, попадая в которую, персонаж вынужден вновь и вновь проживать один и тот же день, пока не найдет выхода, исправив причину появления аномалии.

Модель будет следующей. Хронотоп моделируется одним экземпляром класса Episode, который инкапсулирует следующую активность. Агент *a* пытается угадать слово (или букву некоего алфавита), загаданное агентом *b*. Посылает сообщение (вариант догадки) и получает ответ (Yes, No). Если ответ No, эпизод возобновляется. Все объекты, кроме агента *a*, размещенные в поле prestoredObjects, инициализируются. Объект, моделирующий эпизод и находящейся в поле _state, уничтожается, а затем создается заново. После этого все повторяется.

13. Хронотоп яхты (класс Yacht). Приведем еще один, более сложный пример модели. Пусть имеется два путешественника, каждый из них имеет отдельную каюту (stateroom). Оба путешественника могут также общаться и принимать пищу в кают-компании. Для простоты не будем рассматривать технические помещения и персонал судна. Пространство определим динамическим списком из трех ячеек, крайние ячейки имеют связность с центральной ячейкой. Создадим для каждой ячейки-каюты по два потомка класса Episode: «Время сна» (sleep) и «Время ожидания» (expect) (хронотопы кают; классы Episode11, Episode12 для первой каюты и Episode31, Episode32 – для второй каюты); для ячейки, моделирующей кают-компанию, также создадим эпизоды: «Время ожидания» и «Время приема пищи» (eat) (хронотоп кают-компании, классы Episode21, Episode22). В начальный момент оба агента находятся в каютах, затем переходят в кают-компанию. Хронотоп яхты состоит из двух состояний – из конфигурации «Время сна» – «Время ожидания» – «Время сна» переходим к конфигурации «Время ожидания» – «Время приема пищи» – «Время ожидания». Затем опять возвращаемся к первоначальной конфигурации.

14. Хронотоп «Путешествие». Типичный сюжет приключенческой литературы – путешествие (например, Ж. Верн, «Дети капитана Гранта»). В хронотопе путешествия смена эпизодов согласована с перемещением путещественника по ячейкам пространства (Здесь время как бы вливается в пространство и течет по нему (образуя дороги). М.М. Бахтин). Рассмотрим модель, в которой имеется три пространственных ячейки (Англия, Южная Америка, Австралия), образующих закольцованный список route. Экспедицию будем моделировать массивом expedition, в каждой ячейке которого поместим агента, и будем указатель на него помещать в поле prestoredObjects. Определим по три эпизода для каждой пространственной ячейки. Два эпизода будут описывать нечто, происходящее вне сюжета, а один – события сюжета (События в Англии, Путешествие по Южной Америке, Путешествие по Австралии). Объект expedition смешается по списку route всякий раз, когда в ячейку sCell посылается сообщение exist. Перемещение выполняет сам объект expedition. Во всех трех ячейках и в объекте expedition процессы параллельные. Систему представим классом Journey.

15. Еще одна модель времени. Возможна другая программная конструкция, также использующая паттерн *State*. В этом случае в качестве контекста паттерна выбирается класс CargoTransportation, для которого вводится поле _state. Это поле – массив из двух элементов класса Episode. Объекты класса SCell хранятся в поле _manipulatesObjects класса Episode и передаются от одного эпизода к другому (вариант паттерна *Memento*). Однако для этой модели времени предметная семантика неизвестна.

2.3. Моделирование организационных систем

На парадигме агентного моделирования основаны многие системы имитационного моделирования. Проблема организации совместных действий агентов подробно рассматривается в агентном программировании [52]. Тем самым существует значительный опыт создания мультиагентных систем. Нашей же основной задачей

является моделирование существующих систем, поэтому мы основное внимание уделим тем методам организации кооперативных действий, которые изучаются в теории активных систем.

Процессы организации совместных действий, т.е. бизнес-процессов, обычно рассматриваются как теоретико-игровые ситуации. Обычно выделяют иерархические, кооперативные и рефлексивные игры [42].

Иерархические игры. Иерархические игры, как правило, состоят из двух ходов. Центр, который всегда имеет право первого хода, предлагает агентам некоторую информацию. В ответ на это агенты делают свой ход – выбирают одну из нескольких альтернатив поведения. После чего начинается бизнес-процесс с участием агентов.

В качестве примера проанализируем механизм распределения ограниченного ресурса. Для конкретизации деталей объектной модели рассмотрим задачу «Лорд и два мельника». Предположим, что центр должен распределить дефицитный ресурс (воду) между двумя агентами (водяными мельницами) наиболее выгодным для себя способом. Протокол обмена сообщениями моделирует систему документооборота активной системы, см. рис. 37. Последовательность функционирования следующая: центр выбирает процедуру планирования и сообщает ее агентам (процедура planning_procedure_choice), агенты при известной процедуре планирования сообщают центру информацию (метод information_for_planning), на основании которой и формируются планы (процедура plan_shaping), которые затем доводятся до агентов (метод get_job), см. рис. 38. Центр принимает решения на основе сообщений агентов, которые могут сообщить недостоверную информацию (тем самым возникает ситуация игры). Пусть существует две процедуры планирования. Процедура планирования p_1 предполагает, что агенты сообщают центру оценки идеальных точек r_i своих функций предпочтения. Процедура планирования p_2 предполагает, что решение об объеме выделяемого ресурса принимается центром непосредственно на основании заявок агентов.

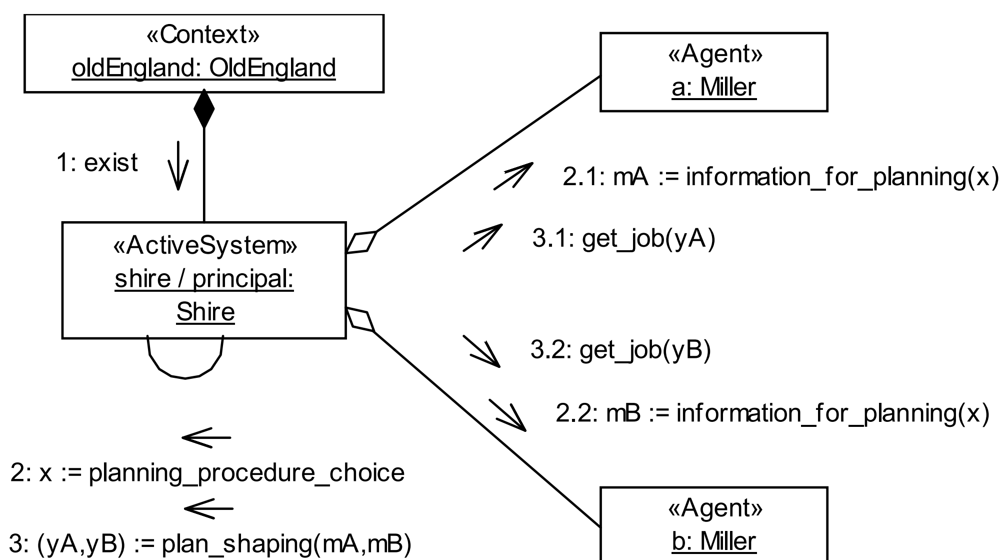


Рис. 37. Диаграмма кооперации для модели «Лорд и два мельника»

Затем бизнес-процесс выполняется и вычисляется эффективность управления. Имитационная модель учитывает стимулирующие, институциональные, мотивационные механизмы управления и несколько механизмов распределения ресурса (в примере их два). Все процессы являются параллельными и взаимодействуют между собой посредством обмена сообщениями. Подробнее об этих механизмах управления см. [36]. Целью *Исследователя* является поиск решения игры с учетом одновременного воздействия всех механизмов управления и некоторых других факторов.

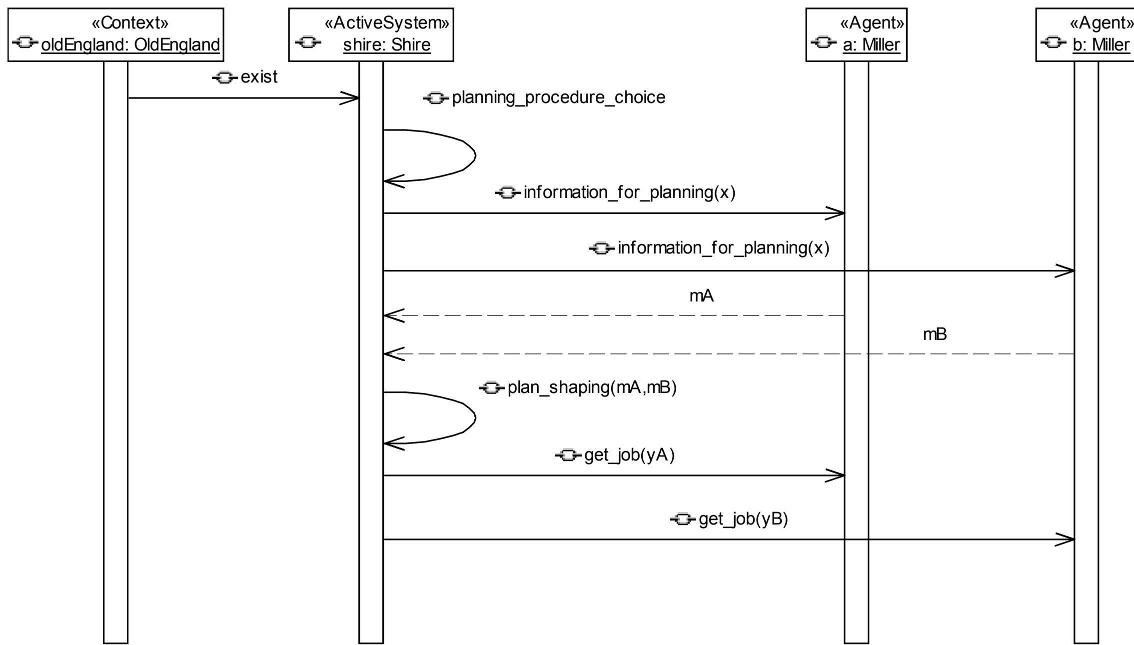


Рис. 38. Сценарий поведения системы «Лорд и два мельника»

Рефлексивные игры. Моделирование общественных систем, как правило, предполагает моделирование рефлексии. Человек, как существо способное к рефлексии, перманентно живет в созданной им самим виртуальной реальности и общается с фантомами, а не с реальными людьми. Это верно даже тогда, когда происходит непосредственный диалог (поэтому некоторым кажется, что все вокруг – лжецы). Эта идея отражена так же в модели коммуникативного акта Р.О. Якобсона. Мы будем строить модели общественных явлений исходя из этой концепции. Поясним основные понятия рефлексивных игр на следующем примере (процетируем [42]).

Как человек переходит «зебру»? Пусть 0 – стоять, 1 – идти (ехать), см. рис. 39. Рефлексия первого уровня – пешеход знает, что водитель намерен остановиться (выберет действие 0). Рефлексия второго уровня – пешеход знает, что водитель знает, что пешеход знает, что водитель остановится. Имеет место формирование команды – неподвижной точки отображения. По этой схеме происходит большинство совместных действий людей.

Рассмотрим игру, в которой участвуют агенты из множества $N = \{1, 2\}$ (пешеход и водитель). В ситуации присутствует неопределенный параметр $q \in \Omega$ (ПДД – правила дорожного движения); *структура информированности* i-го агента включает в себя следующие элементы. Во-первых, представление i-го агента о параметре q – обозначим его q_i, $q_i \in \Omega$. Во-вторых, представления i-го агента о представлениях других агентов о параметре q – обозначим их q_{ij}, $q_{ij} \in \Omega$, $j \in N$. И так далее.

Аналогично задается *структура информированности I игры* в целом – набором значений $q_{i1 \dots il}$, где l пробегает множество целых неотрицательных чисел,$j_1, \dots, j_l \in N$, а $q_{ij1 \dots jl} \in \Omega$.

Рефлексивной игрой Γ_I называется игра, описываемая следующим кортежем:

$$\Gamma_I = \{N, (X_i)_{i \in N}, \tilde{f}(.)_{i \in N}, \Omega, I\},$$

где N – множество реальных агентов, Xi – множество допустимых действий i-го агента (0 – стоять, 1 – идти(ехать)), fi (.) – его целевая функция (авария, без аварии), Ω – множество возможных значений неопределенного параметра (ПДД1 – первым переходит дорогу пешеход, ПДД2 – первым проезжает автомобиль), I – структура информированности.

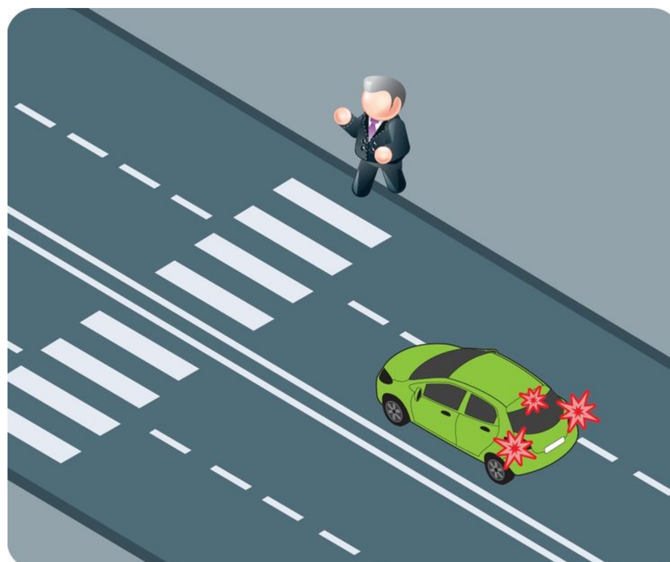


Рис. 39. Пешеходный переход

Наряду со структурами информированности Ii, $i \in N$, можно рассматривать структуры информированности Iij (структура информированности j-го агента в представлении i-го агента) – *фантомные агенты*.

Набор действий x_t*, $t \in \Sigma_+$, назовем *информационным равновесием*, если выполнены следующие условия:

(1) в рефлексивной игре участвует конечное число реальных и фантомных агентов.

(2) одинаково информированные агенты выбирают одинаковые действия.

(3) рациональное поведение агентов – каждый из них стремится выбором собственного действия максимизировать свою целевую функцию, подставляя в нее действия других агентов, которые оказываются рациональными с точки зрения рассматриваемого агента в рамках имеющихся у него представлений о других агентах.

Для формального изложения дополним кортеж, задающий рефлексивную игру, набором функций w_i (.) – *функцией наблюдения* i-го агента. Будем считать, что функции наблюдения являются общим знанием среди агентов. Это наблюдение пешехода за машиной и водителя за пешеходом.

Пусть в рефлексивной игре существует информационное равновесие x_t*, $t \in \Sigma_+$. Зафиксируем $i \in N$ и рассмотрим i-го агента. Он ожидает в результате игры пронаблюдать величину w_i $(q_i, x_{i1}, \ldots, x_{i,\ i-1}, x_i, x_{i,\ i+1}, \ldots, x_{in})$. На самом же деле он наблюдает величину w_i $(q_i, x_{i1}, \ldots, x_{i,\ i-1}, x_i, x_{i,\ i+1}, \ldots, x_{in})$. Поэтому требование стабильности для i-агента означает совпадение этих величин.

Имитационная модель. Контекст – городская дорога. Система – ситуация на пешеходном переходе. Рефлексирующие агенты – пешеход и водитель. Фантомные агенты – образ водителя, образ пешехода. Цель исследователя – найти набор действий информационного равновесия. Квант действия – выбирает действие и наблюдает за контрагентом, если действие подтверждается (неподвижная точка), то выполняет.

Объектная модель строится точно так же, как это было показано в п. 8 примечаний к п.1.2, т.е. вводится поле worldImage для класса агента. Как правило, в моделях с рефлексирующими агентами структура данных для этого поля задается более подробно. В нашем случае надо предусмотреть поля для q_i, $q_i \in \Omega$, q_{ij}, $q_{ij} \in \Omega$, $j \in N$ и более высоких рангов, если они ожидаются (иногда удобнее отказаться от рекурсии). Кроме того, в классе агента определяются методы посылки сообщения другому агенту и наблюдения.

Примеры и пояснения

1. ***Объектные модели и кибернетика***. В 1948 году Норберт Винер [Norbert Wiener, Cybernetics or Control and Communication in the Animal and the Machine, (Hermann & Cie Editeurs, Paris, The Technology Press, Cambridge, Mass., John Wiley & Sons Inc., New York, 1948)] предложил термин «кибернетика» в современном понимании – как науку об общих закономерностях процессов управления и передачи информации в машинах, живых организмах и обществе. На наш взгляд одним из важных моментов успеха кибернетики было то, что это был первый научный подход, в котором был применен информационный взгляд на объект изучения. В книге [*Kelly, Kevin Out of control: the new biology of machines, social systems and the economic world. — Boston: Addison-Wesley, 1994*] сказано, что кибернетика фокусирует внимание на том, как объект обрабатывает информацию, реагирует на неё и изменяется или может быть изменен, для того чтобы лучше выполнять первые две задачи.

С точки зрения объектно-ориентированного имитационного моделирования кибернетика представляет собой редукцию объектных моделей управления на функциональные модели (IDEF0). От диаграммы кооперации переходим к диаграммам деятельности (т.е. к процессам), а затем определяем процессы как функции. Хотя возможно сущность кибернетики глубже.

Многие идеи кибернетики могут быть обобщены до объектных моделей и представляют несомненный интерес. Для этого необходимо попытаться представить кибернетическую модель в виде совокупности коммуникационных актов.

Приведем несколько примеров информационного управления.

2. ***«Ведение объекта»***. Агент из пункта *A* прибывает в пункт *B*, а из него в *C* или *D*. Агента надо направить именно в пункт *C*, для этого организуется некоторое действие в *B*.

3. ***«Доброволец»*** (вольный пересказ одной фантастической истории С.В. Лукьяненко). Рассмотрим следующую историю. … «Конечно же – только не я». Нам нужен доброволец – сказал капитан, – кто пойдет? Все посмотрели на меня. Все, вместе с капитаном, ждали моего ответа. «Я», – сказал я.

Эта модель имеет два аспекта – психологический и «рациональный». В последнем случае агент меняет свои представления под воздействием экспертов, вполне рационально полагая, что когда несколько экспертов имеют единое мнение по какому-либо вопросу, то оно, скорее всего, правильное. Эта ситуация также обыгрывается в известном психологическом опыте, в котором испытуемый дает ответ о цвете предмета. Возможность высказаться ему предоставляется последним в цепочке.

4. ***Информационное оружие***. С. Расторгуев дает следующее определение информационного оружия: «Информационное оружие представляет собой средства, позволяющие целенаправленно активизировать в информационной системе определенные процессы, т.е. те процессы, в которых заинтересован субъект, применяющий оружие» [48 с. 155]. В отличие от задачи двух центров, модель должна отражать структурную динамику. Для этого можно использовать, например, идиому делегирования [32].

5. ***Информационные войны***. Информационные атаки с помощью средств массовой информации и глобальной сети Интернет, это может быть психологическое воздействие на все население с целью изменить общественное мнение по тем или иным вопросам.

Рассмотрим следующую задачу. Сражение проиграно стороной *A* стороне *B*; внушить *B*, что все наоборот. Сторона *A* вообще не участвует в сражении, убедить *B*, что сражение было и *B* его проиграло. За основу можно взять как модель Observer с двумя классами ConcreteSubject, так и модель распространения слухов (см. п. 2.2). В последнем случае решение очевидно. Напротив, для открытого общества необходимы значительно более изощренные решения.

2.4. Моделирование бизнес-процессов

Анализ бизнес-процессов. Отправной точкой по разработке *Scientific Profile* послужил *Ericsson-Penker Profile* [66]. Ряд современных методов моделирования бизнес-процессов основан на использовании языка UML. Среди таких методов наиболее известными являются метод Ericsson-Penker и метод, реализованный в технологии Rational Unified Process (RUP). Общим в *Scientific Profile* и в профиле *RUP Business Modeling* является использование нескольких моделей (методика моделирования RUP предусматривает построение двух моделей: Business Use Case Model и Business Analysis Model). Основным отличием *Scientific Profile* от обоих профилей является то, что в нашем случае объект исследования моделируется в контексте познавательной ситуации. Бизнес-среда определяется в пакете со стереотипом *World* как онтологический контекст.

Модель производственного предприятия

Модель предприятия довольно подробно рассмотрена почти во всех учебниках по имитационному моделированию (см., например, [3]).

Приведем типовую модель предприятия. Диаграмма классов показана на рис. 40.

Контекст модели задается классом ProductionSphere {Concept = Производственная сфера общественного воспроизводства}. Изучаемую систему – предприятие, – моделирует поле _enterprise класса Division. Буферный класс Corporation вводится как для моделирования корпорации (и тогда он будет иметь много общего с Division), так и для моделирования части окружающей среды. В данной задаче мы будем полагать, что Corporation – это часть окружающей среды предприятия.

Композиция объекта _enterprise определяется паттерном *Composite* (см. [14]). Этот паттерн используется в имитационных моделях (да и вообще в программировании) достаточно часто и применяется тогда, когда необходимо моделировать иерархические структуры.

Класс StructuredUnit выступает в роли *Component* (компонент), который
- объявляет интерфейс для компонуемых объектов;
- предоставляет подходящую реализацию операций по умолчанию, общую для всех классов (в нашем случае – это конструктор с параметром);
- объявляет интерфейс для доступа к потомкам и управления ими (метод add);
- определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его.

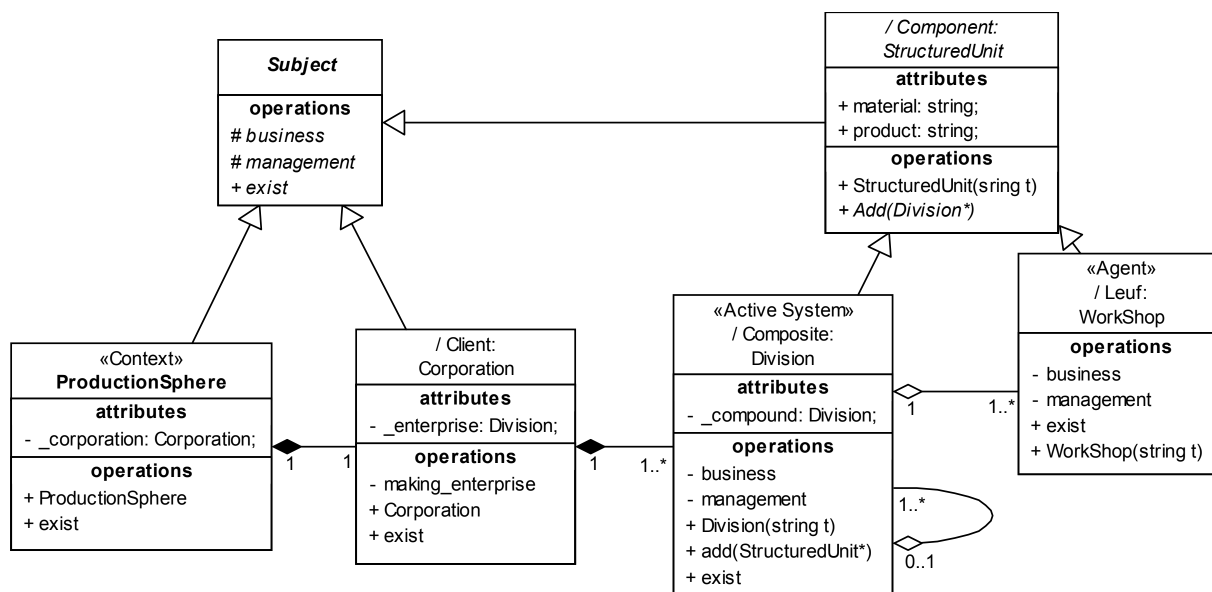


Рис. 40. Диаграмма классов для модели предприятия

Поскольку предприятие рассматривается в контексте ProductionSphere, мы наделяем класс StructuredUnit свойствами material и product.

Класс Division (его экземпляры моделируют само предприятие и подразделения предприятия) выступает в роли *Composite* (составной объект) и
- хранит компоненты-потомки (поле _compound, т.е. *состав департамента*);
- определяет поведение компонентов, у которых есть потомки (метод exist):

```
void Division::exist() {
for (int i=1; i<=compound_count; i++) {
_compound[i]->material = this->material;
_compound[i]->exist();
this->product = _compound[i]->product;
}
```

- реализует относящиеся к управлению потомками операции (мы ограничимся только методом add) в интерфейсе класса Component

```
void Division::Add(StructuredUnit * t) {
++compound_count;
_compound[compound_count] = t;
} .
```

Класс Division конкретизирует процедуры business и management абстрактного класса Subject. Обе процедуры вызываются из метода exist.

Атомарные объекты (в нашем случае – отделы и цеха предприятия) моделируем классом WorkShop. В паттерне этот класс выступает в роли Leuf (лист):
- представляет листовые узлы композиции и не имеет потомков;
- определяет поведение примитивных объектов в композиции.

В роли *Client* **(**клиент) выступает класс Corporation; он манипулирует объектами композиции через интерфейс Component. В частности, создает модель предприятия, например, так:

```
void Corporation::making_enterprise() {
_enterprise = new Division("Предприятие");
_enterprise->Add(new WorkShop("Склад"));
Division * manufacture = new Division("Производство");
_enterprise->Add(manufacture);
manufacture->Add(new WorkShop("Цех1"));
manufacture->Add(new WorkShop("Цех2"));
}  .
```

На рис. 40 явно не показаны потоки управления. Это информационные потоки, которые в простейшем случае обеспечивают передачу плановой информации и контроль выполнения плана между _enterprise и объектами класса WorkShop. Для их моделирования необходимо задать дополнительные свойства в классе StructuredUnit и определить процедуры management в классах Division и WorkShop. Протокол обмена сообщениями, содержащими управленческую информацию, образует модель документооборота предприятия. Если необходимо выделить администрацию (бухгалтерия, отдел кадров и т.д.) и производство, то для них нужно определять свои методы exist и определять отдельные классы Administration (составной элемент называется Department) и Manufacture,

Модель предприятия – это квазифрактальный объект (см. комментарии к п. 2.1). В нашем случае фрактал имеет три уровня. В качестве генератора выступает класс Corporation. Так как между уровнями имеет место подобие, то и между процессами exist в нашем случае также имеет место подобие. Если мы выделим администрацию и производство как отдельные классы, то это уже будет неоднородный фрактал.

Примеры и пояснения

1. ***Что такое бизнес-процесс?***

В системе протекают разнообразные процессы, и их следует отличать друг от друга. Если пользоваться терминологией системного анализа, то бизнес-процесс – это процесс, который обеспечивает выполнение основной функции системы. Основная функция определяется ролью, которую играет система во взаимодействии с окружающим миром. Традиционные приложения имитационного моделирования – логистика и теория очередей [56], [53].

Кроме того, существуют процессы управления, которые обеспечивают уже сам бизнес-процесс. Эти процессы могут быть «встроены» в бизнес-процесс, как в случае самоорганизации, так могут быть и обособлены. Существуют процессы развития, которые приводят к изменениям в самой системе. Под процессами развития мы будем понимать процессы разрешения противоречий. Наконец, следует выделять процессы эволюции. Последние предполагают наличие механизмов самовоспроизведения и изменчивости.

2. ***«Железнодорожная ветка».*** Сначала рассмотрим типовую задачу, решаемую методами объектного моделирования. Железная дорога на участке *AB* является одноколейной. В узлах *A* и *B* имеется стрелка и семафор, которые позволяют регулировать движение встречных поездов. Продолжительность движения состава на участке *AB* составляет T_{AB} часов. Частота движения поездов в течение суток есть *N*.

3. ***«Пассажирские перевозки 1».*** Имеется замкнутый маршрут, по которому курсирует *N* автомобилей городского транспорта. На маршруте имеется *M* остановок, на которых собираются пассажиры с интенсивностью *I* пас/час. Маршрутное такси, прибывшее на остановку, забирает всех пассажиров. Для того чтобы автопредприятие получило максимальную прибыль при заданном *N*, необходимо поддерживать строгий интервал между автомобилями. Для этого каждое маршрутное такси регистрируется в диспетчерском пункте (или пунктах) с целью выдержать график движения.

4. ***«Пассажирские перевозки 2».*** Та же самая задача. Только теперь пункт диспетчера отсутствует. Водители автомобилей сами договариваются о скорости движения путем обмена сообщениями (чаще на расстояние прямой видимости). Данная модель – есть простейшая модель процессов *самоорганизации.*

Эта и предыдущая модель интересны тем, что позволяют определить различие между двумя механизмами управления, а также поискать способы трансформации одного механизма в другой.

5. ***«Полет пчелы».*** Для таких систем, как пчелиный рой, характерны сложные коллективные явления. Мы рассмотрим модель сбора нектара. Пусть пчела движется от цветка к цветку и собирает нектар. Часть нектара пчела тратит на движение. Пчеле приходится обработать множество цветков, пока она наполнит свой медовый желудочек нектаром. Чтобы собрать килограмм нектара с гречихи, пчелы должны посетить около двух миллионов цветков. В хороших условиях пчелиные семьи могут собрать в день по 20 – 25 кг. нектара. Модель можно построить с «точки зрения пчелы», тогда маршрут можно описать как внешний итератор на динамическом списке (последний моделирует последовательность посещаемых цветов). Можно также построить модель с «точки зрения цветка». Сравнение этих моделей позволяет определить инварианты способов описания.

2.5. Адаптация, развитие и эволюция

2.5.1. Адаптация

Под *гомеостазом* обычно понимают, в соответствии с формулой К. Бернара, «сохранение *постоянства внутренней среды* организма при наличии возмущений во внешней среде».

Основная идея гомеостазиса – это эффект «склеивания» противоположностей [47]. Между антагонистами устанавливается отрицательная обратная связь. Типичным примером является шарик на дне лунки или материальная точка между двумя пружинами.

Моделирование гомеостазиса впервые было предпринято Р. Эшби, предложившим модель, названную им гомеостатом [63]. Бир сделал определенные попытки по использованию некоторых гомеостатических принципов при практической разработке организационных структур управления, где пытался провести некоторые кибернетические аналогии между живой системой и сложным производством (см. Бир С. Кибернетика и управление производством. – М.: Наука. 1965. – 391 с.). В настоящее время накоплен огромный фактологический материал, описывающий различные проявления гомеостаза в живых, технических, социальных и экологических системах. В большинстве работ рассматривается гомеостат в предположение, что воздействие внешней среды сводится к изменению интенсивности возмущающего сигнала. Математические и кибернетические модели гомеостазиса достаточно хорошо изучены. Однако и в живых и социальных системах приходится иметь дело не только с изменением интенсивности воздействия, но и с качественными изменениями характера этого воздействия. Здесь мы заметим только то, что объектные модели гомеостата предоставляют новые возможности моделирования. Например, паттерн *Adapter* [14] позволяет учитывать изменение интерфейса гомеостата.

2.5.2. Развитие систем

Одной из разновидностей развивающихся систем являются конфликты в общественных системах. В данной статье рассматривается *социальный конфликт*, имея в виду процесс, в котором два (или более) индивида или группы активно ищут возможность помешать друг другу в достижении определенной цели. В качестве примера рассматривается имитационная модель конфликта, описанная Уильямом Шекспиром в драме «Гамлет». Выбор художественного образа в качестве объекта исследования обусловлен тем, что текст художественного произведения доступен и понятен всем членам научного сообщества, не может измениться, а также содержит всю возможную информацию об объекте моделирования.

Фабула (Wikipedia, Гамлет). Рядом с Эльсинором, королевским дворцом Дании, солдаты несколько раз видели призрака, похожего на недавно погибшего короля. Встреча Гамлета с призраком приводит его в смятение – призрак рассказал ему о том, что его дядя, нынешний король, убил его, и завещает сыну месть. Гамлет поражён и растерян настолько, что решает притвориться сумасшедшим. В это время в Эльсинор приезжает труппа бродячих актёров. Гамлет просит их поставить пьесу «Убийство Гонзаго», вставив туда несколько строк своего сочинения. Король внимательно следит за действием пьесы и уходит после того, как в пьесе Гамлета происходит убийство. Король понимая, что Гамлет для него опасен, отсылает его в Англию, чтобы его сразу же по приезде казнили. Принц спасается от этой участи и возвращается в Данию. Дядя прибегает к уже испытанному приёму – яду. Гамлет умирает, перед смертью убивая короля. Датский престол переходит к Фортинбрасу, норвежскому правителю.

Объектная модель системы. На рис. 41 представлена объектная модель, описывающая данный конфликт.

Контекст модели задан классом Danish {Concept = Дания, XIV век}. Система моделируется классом RoyalFamily {Concept = The royal family}. В композицию класса входит поле _castle, представляющее собой ячейку класса SCell «Ontology Space» и моделирующее пространство системы. Атомарные объекты класса Character моделируют участников конфликта.

Конфликт складывается, по меньшей мере, из трех фаз: зарождение конфликта, развитие конфликта и разрешение конфликта. Для описания интенсивности противоречия

предлагается «трехцветная модель». Участники противостояния имеют три состояния: green (спокойствие), yellow (состояние тревожности) и red (опасность). Состояние системы в целом определяется сочетанием этих состояний. Далее предполагается, что все переменные подобного рода рассматриваются как лингвистические переменные.

Модель агента. Агент, имеющий класс Character, это атомарный объект. Поэтому нам достаточно только точно описать его взаимодействие с системой. Поле state определяет состояния (green, yellow, red) агента. Поле action является приватным и описывает намерение агента, т.е. будущее действие. Поле deformation описывает изменение структуры и организации агента в результате внешнего воздействия. В случае конфликта – это обычно негативное воздействие; но может быть и позитивным. Изменение значений полей происходит в результате обмена сообщениями. Агенты между собой и с системой обмениваются сообщениями, имеющими класс AgentMessage. Этот класс имеет единственное поле content (типа String). Метод perceive(AgentMessage) моделирует получение сообщения агентом (оно изменяет поле deformation), а метод AgentMessage affected() – сообщение, посылаемое агентом (оно определяется полем action).

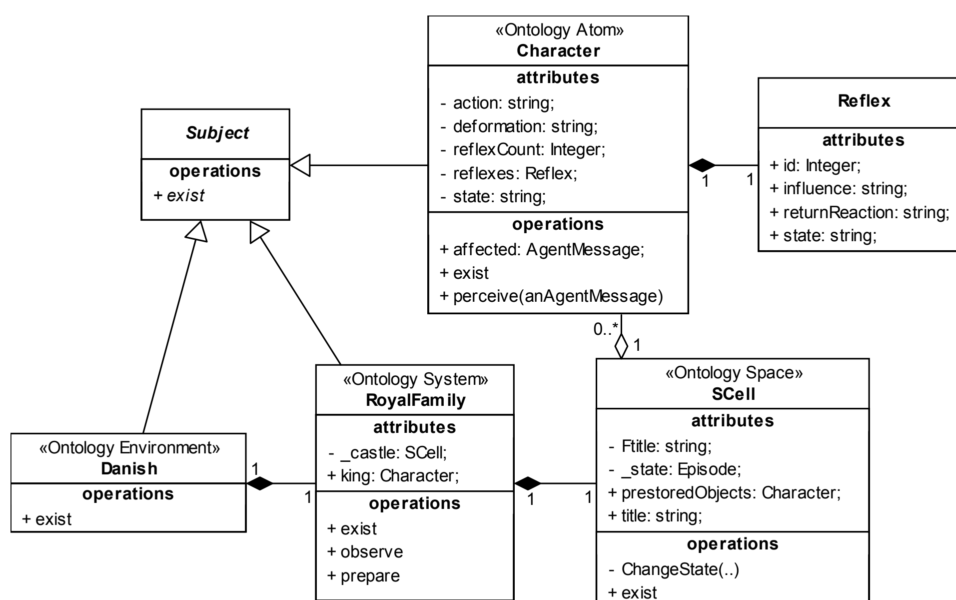


Рис. 41. Объектная модель системы

Метод exist() описывает квант существования агента. Ниже приведен упрощенный вариант кода этого метода.

```
void Character::exist()
{bool done = true; int i = 1;
while (done) {
if (reflexes[i].state == state && reflexes[i].influence ==
deformation)
{done = false;};
i++; if (i>reflexCount) {done = false;} }; i = i - 1;

if (reflexes[i].id == 0) {state =
reflexes[i].returnReaction;};
if (reflexes[i].id == 1) {action =
reflexes[i].returnReaction;}; }
```

Поле reflexes – массив класса Reflex, определяет таблицу рефлексов – действий, зависящих от состояния воздействия или изменений, вызванные воздействием. Поле int id определяет тип реакции агента: 0 – изменить внутреннее состояние, 1 – выполнить внешнее действие. Заметим, что сам по себе рефлекс моделируется иначе (не как поиск в

таблице, а как поиск по дереву), однако, так как это атомарный объект, нам достаточно, чтобы совпадал только конечный результат.

Множество рефлексов рассматриваем как множество равновероятных альтернатив. Это состояние неустойчиво. Для выбора альтернативы надо ввести правило нарушения симметрии:
- спонтанное нарушение симметрии;
- детерминированное, т.е. выбираемое по критериям;
- синергетическое, когда последовательность действий образует некоторую систему;
- надситуационное, например моральные или философские соображения агента.

Хронотоп модели. Для описания разнородного поведения системы уже не достаточно условных выборов в коде программы. В нашей модели используется модель хронотопа, основу которой составляет паттерн State. Композиция SCell показана на рис. 42, роли классов расписаны в соответствии с описанием паттерна в книге [14].

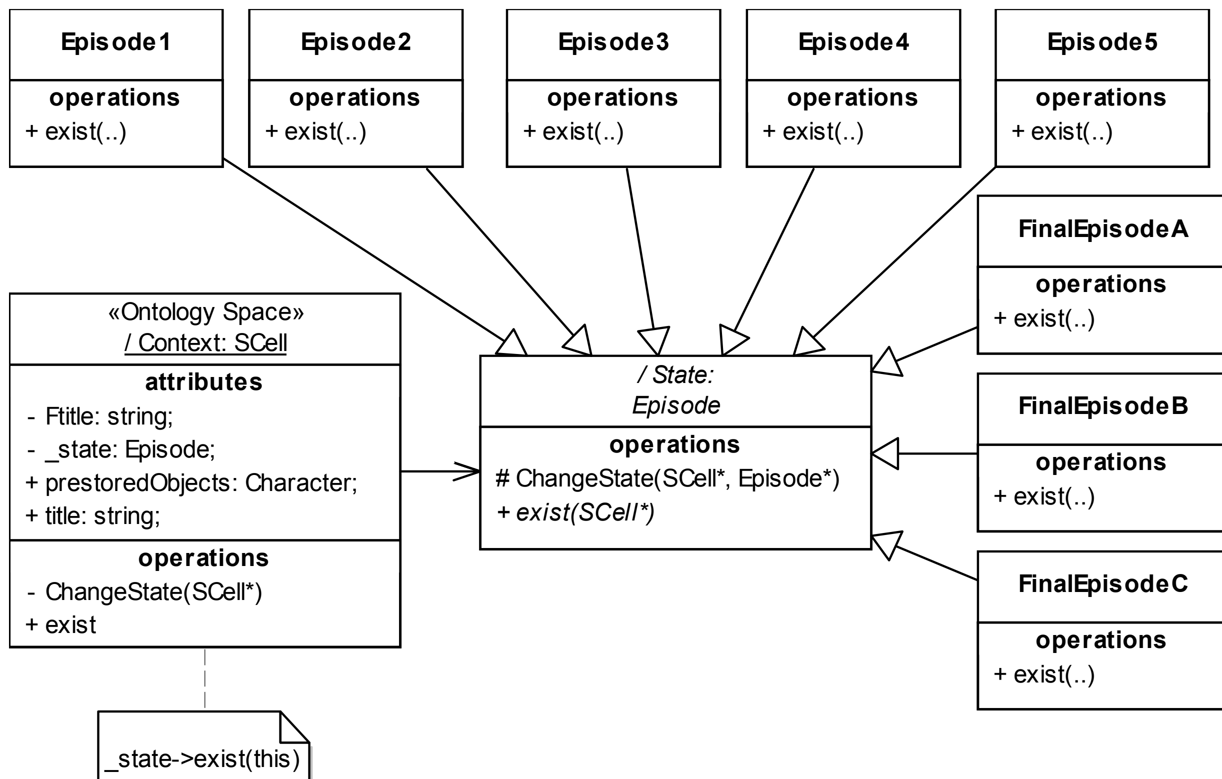


Рис. 42. Паттерн State для хронотопа модели «Гамлет»

Класс SCell, моделирующий ячейку пространства, выступает в роли контекста паттерна и имеет поле _state класса Episode. Класс Episode {Concept = Эпизод} объявлен другом класса SCell, что дает ему привилегированный доступ к приватным полям и процедурам SCell. Это позволяет манипулировать с объектами, входящими в композицию класса SCell, а также вызывать процедуру ChangeState(Episode*), которая изменяет поле _state. Подклассы Episode реализуют поведение в конкретных ячейках пространства – в нашем случае есть только одна ячейка _castle. Каждая процедура exist подклассов Episode заканчивается переходом в следующее состояние (см. рис. 42). Например, для Episode1 соответствующий фрагмент кода будет иметь вид:

```
Episode * tNext = new Episode2;
ChangeState(j, tNext);
```

где j – указатель на объект класса SCell. Неустойчивые состояния системы моделируются условным выбором.

На рис. 43 приведена диаграмма деятельности для данной модели. Каждой части соответствует акт трагедии. Приведем краткое содержание каждого действия.

1 часть – завязка, пять сцен первого действия. Встреча Гамлета с Призраком.

2 часть – развитие действия, вытекающее из завязки. Гамлету необходимо усыпить бдительность короля, он притворяется сумасшедшим. Клавдий предпринимает меры, чтобы узнать о причинах такого поведения. В результате – смерть Полония, отца Офелии, невесты принца.

3 часть. Кульминация: а) Гамлет окончательно убеждается в вине Клавдия; б) сам Клавдий сознает, что его тайна раскрыта; в) Гамлет раскрывает глаза Гертруде.

4 часть: а) отправка Гамлета в Англию; б) приход Фортинбраса в Польшу; в) безумие Офелии; г) смерть Офелии; д) сговор короля с Лаэртом.

5 Часть – развязка. Дуэль Гамлета и Лаэрта. Смерть Гертруды, Клавдия, Лаэрта, Гамлета.

Мы специально приводим достаточно подробное описание каждой части трагедии, поскольку каждая сцена имеет свои уникальные особенности моделирования. Для примера на рис. 44 показана диаграмма кооперации для эпизода-развязки.

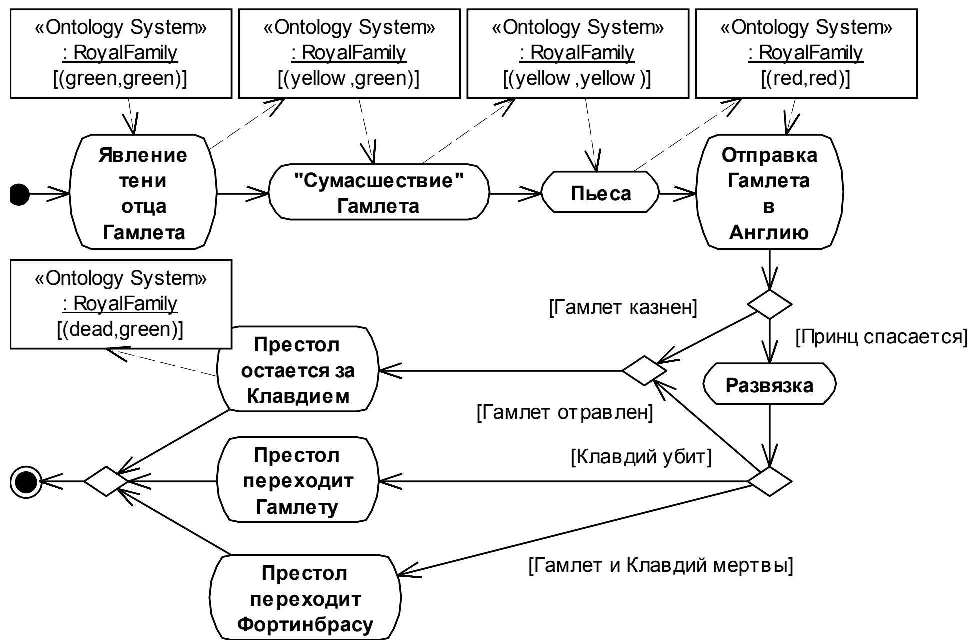


Рис. 43. Диаграмма деятельности для модели «Гамлет»

Описанная выше программная конструкция интерпретируется как хронотоп (см. п. 2.1.2). Под хронотопом мы понимаем последовательность эпизодов, связанных с одной ячейкой пространства. Эпизод можно рассматривать как протокол обмена сообщениями между агентами. В простейшем случае – это просто последовательность операторов. Однако часто приходится использовать более гибкие конструкции. Хронотоп состоит из пяти эпизодов (в трагедии пять актов; удивительно, что они совпадают с эпизодами) и трех финальных эпизодов (FinalEpisodeA, FinalEpisodeB, FinalEpisodeC), в которых переопределяется поле king.

Точность модели, описанной выше, может быть значительно повышена, вплоть до точного совпадения с художественным образом. В этом плане имитационные модели можно использовать для создания художественных произведений (что наблюдается в отношении компьютерных игр). Еще одна близкая идея высказана С. Лемом. В одном из своих фантастических произведений он описывает ситуацию, когда имитационная модель используется для записи происходящих событий. Такая технология будет затребована, если необходима реконструкция событий.

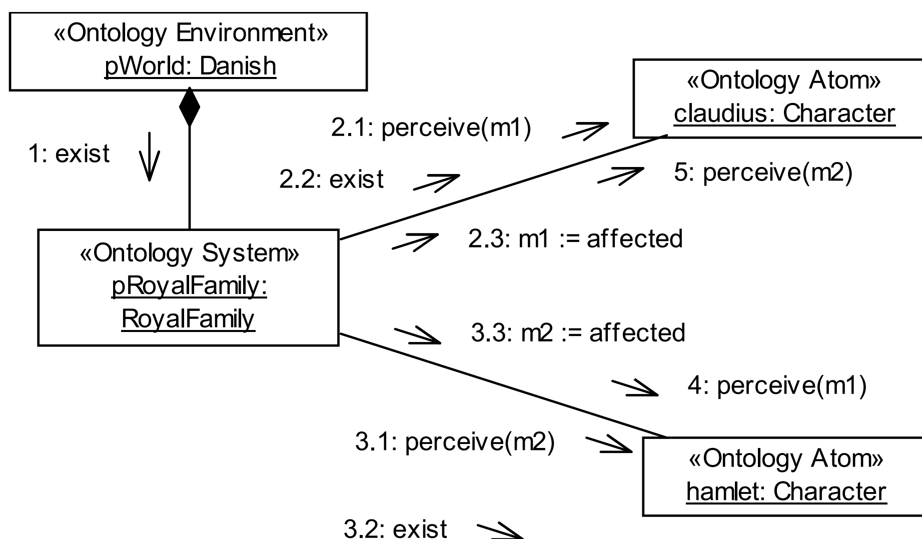


Рис. 44. Диаграмма кооперации для пятого эпизода

Итак, в данном разделе рассмотрены объектные модели развивающихся систем. Типичный пример развивающейся системы – конфликт в общественной системе – предполагает, по меньшей мере, три фазы развития: зарождение конфликта, развитие конфликта и разрешение конфликта. Для описания смены фаз развития можно использовать программную конструкцию, которая интерпретируется как модель хронотопа. Показано, что для моделирования интенсивности противоречия достаточно трехбалльной шкалы. Модель агента – участника конфликта – предполагает описание состояния агента, определяемое тремя полями: внутренним состоянием агента, деформацией агента и намерением агента. Предложенные решения обладают достаточной общностью и пригодны для моделирования широкого класса развивающихся систем.

2.5.3. Эволюция

Эволюция имеет место там, где существует тот или иной механизм самовоспроизведения, некий механизм изменчивости и механизм отбора. Наследственность представляет консервативность живых организмов, выступая в неразрывной связи с **изменчивостью,** обусловленной воздействием внешней среды и жизненным опытом индивидуальных особей. В общем случае можно назвать два механизма: создание системы по шаблону и формирование системы по прототипу. Рассмотрим эти модели подробнее.

Клонирование. В данном разделе рассматривается одна из имитационных моделей процесса самовоспроизведения, характерная для экономических субъектов.

Типичным примером механизма самовоспроизведения является франчайзинг. Франчайзинг – разновидность отношений между экономическими субъектами, когда одна сторона (франчайзер) передаёт право на определённый вид бизнеса и, что важно для нас, проверенную бизнес-систему (франшизу) другой стороне (франчайзи).

Рассмотрим объектную модель самовоспроизведения фирмы (рис. 45). Данная модель есть модель предприятия (вычислительная семантика – паттерн *Composite*), дополненная новыми свойствами и методами, обеспечивающими процесс самовоспроизведения. Методы exist, business, management, add, making_enterprise предназначены для моделирования фирмы, и мы их рассматривать не будем.

В классе контекста вводится новый метод clone, который для нашей задачи будет иметь стереотип *Exist*. Он вызывает метод clone_enterprise класса Corporation{Concept =

Интегрированная бизнес-группа (ИБГ)}, что в свою очередь, порождает цепочку процессов, приводящих к созданию клона предприятия. Поле _enterprise отводится под оригинал, а поле _franchise {Concept = Франшиза} – под клон.

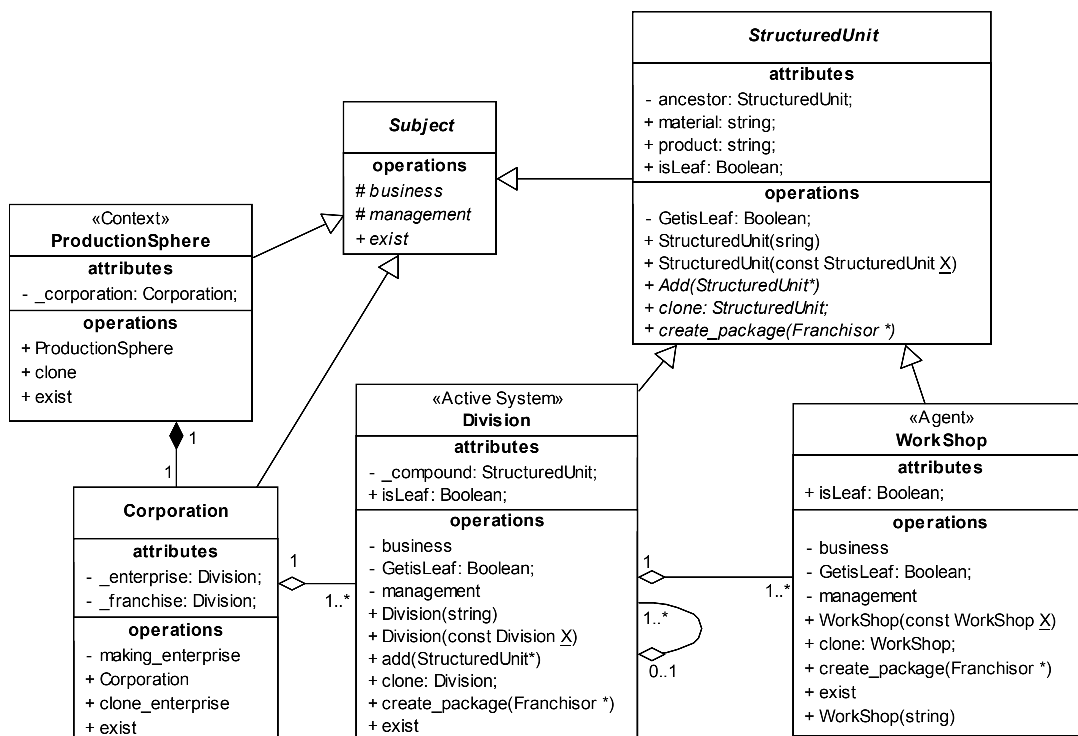


Рис. 45. Диаграмма классов для модели самовоспроизведения

В абстрактном классе StructuredUnit объявлены абстрактные методы clone и create_package. Метод clone позволяет объекту создавать клон самого себя, для чего используется копирующий конструктор StructuredUnit(const StructuredUnit &X), который затем определяется в классах Division и WorkShop. Метод virtual void create_package(Franchisor *) предназначен для создания копии структуры текущего узла; эту работу выполняет *агент клонирования*. Для успешной работы *агента клонирования* вводятся поле ancestor {Concept = Входимость} и свойство isLeaf, которое принимает значение false для узла и true для листа дерева структуры.

Для процедуры create_package класса Division алгоритм обхода элементов структуры есть префиксный левосторонний (или правосторонний) обход:

```
void Division:: create_package (Franchisor * t) {
t->make(this);
for (int i=1; i<=compound_count; i++) {_compound[i]-> create_package (t);}
t->backtrack(); }  .
```

Метод create_package класса WorkShop проще; он сводится к обработке листа: void WorkShop:: create_package (Franchisor * t) { t->make(this); }.

Концепт «Создать франчайзинговый пакет» назначим методу create_package. Руководство ИБГ издает распоряжение (сообщение create_package) для всех подразделений фирмы о документировании бизнеса и назначает исполнителя (экземпляр класса Franchisor). В распоряжении установлен алгоритм, согласно которому руководитель подразделения должен проконтролировать выполнение данной работы исполнителем в своем отделе и довести данное распоряжение до подчиненных ему структурных единиц.

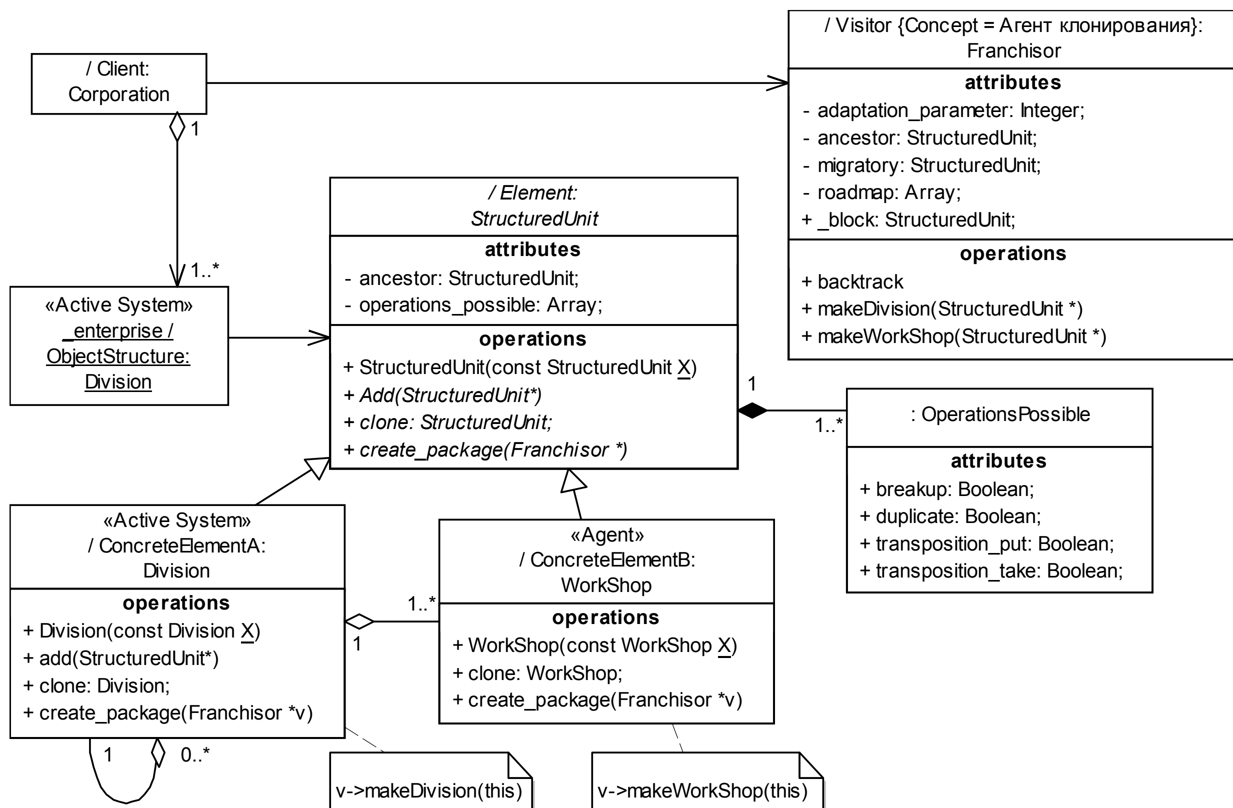


Рис. 46. Паттерн *Visitor* для модели самовоспроизведения

Объект класса Franchisor {Concept = Агент клонирования} является ролью франчайзера и решает задачу создания клона фирмы для передачи партнеру (франчайзи). Класс Franchisor имеет поля _block {Concept = Сборка клона} (public), ancestor {Concept = Непосредственный предок} и roadmap{Concept = Дорожная карта} (вычислительная семантика – стек узлов). Процедура void backtrack() сокращает стек на один элемент. Процедура void make(StructuredUnit *) создает элемент структуры клона (объект-оригинал передается в процедуру в качестве аргумента) и имеет, например, такой вид:

```
void Franchisor::make(StructuredUnit *t) {     StructuredUnit   *sUnit   =   t->clone(); // t- прототип
      // sUnit ->Initialize(); // если необходимо
      sUnit->ancestor = this->ancestor;  // предок в дереве клона
      if (this->ancestor != NULL) this->ancestor->Add(sUnit);  // агрегация
      else _block = sUnit; // это корень дерева
      // для следующего уровня
      if (!t->isLeaf()) {     this->ancestor = sUnit; _count++;
roadmap[_count] = sUnit;};
}  .
```

Обратимся теперь к изучению *изменчивости*. Франчайзер, как правило, организует жесткий контроль над выполнением стандартов бизнеса. Тем не менее, существует необходимость внесения тех или иных адаптивных изменений в бизнес-систему. Мы рассмотрим адаптацию франшизы под размеры бизнес-системы. Для этого нам пройдется обобщить рассмотренную выше модель самовоспроизведения.

Вычислительная семантика. Модель самовоспроизведения фирмы в самом общем случае задается паттерном *Builder* (см. рис. 46.). Данная модель есть объектная модель предприятия (паттерн *Composite*; классы StructuredUnit, Division и WorkShop). Мы ограничимся «мягким» вариантом паттерна *Builder* – будем полагать, что класс, играющий роль *строителя*, один, но зато он может быть параметризирован. Процедура clone_enterprise класса Corporation{Concept = Интегрированная бизнес-группа} создает

агента клонирования, конфигурирует им метод create_package объекта _enterprise и по завершению процедуры принимает клон бизнес-системы. Вычислительная семантика поведения агента определяется паттерном *Visitor* (см. рис. 46); роли классов расписаны согласно [14].

Класс Franchisor (в роли *Visitor*) объявляет и реализует методы makeDivision и makeWorkShop, которые выполняют клонирование текущего элемента и его связей. Агент получает элемент в качестве аргумента и обращается к элементу через интерфейс, посылая сообщение clone. Процесс клонирования описывается паттерном *Prototype*. Кроме того, агент выполняет копирование связей элемента или модифицирует их, работая с полем ancestor. Агент выполняет элементарные операции, руководствуясь переменной operations_possible (см. рис. 46) – списком допустимых операций. Переменная задана в каждом элементе структуры и параметрически зависит (зависимость моделируется массивом) от переменной adaptation_parameter, известной агенту. Всего возможно 16 значений переменной operations_possible, однако все они могут быть сведены к следующим: тождественное отображение (¬duplicate ∧ ¬breakup ∧ ¬transposition_take ∧ ¬transposition_put = true), начать перемещение элемента (¬duplicate ∧ breakup ∧ transposition_take ∧ ¬transposition_put = true), начать перемещение дубля (duplicate ∧ ¬breakup ∧ ¬transposition_take ∧ ¬transposition_put = true), завершить перемещение (¬duplicate ∧¬breakup ∧ ¬transposition_take ∧ transposition_put = true), дублировать элемент (duplicate ∧ ¬breakup ∧ ¬transposition_take ∧ transposition_put = true), повторно использовать элемент в техпроцессе (¬duplicate ∧ ¬breakup ∧ transposition_take ∧ transposition_put = true). Для дублирования элемента создается новый экземпляр класса Franchisor. Перемещаемые элементы или их дубли помещаются в поле migratory.

Предметная семантика. Для процессов клонирования характерна высокая устойчивость к изменениям. Основное ограничение – необходимость сохранения технологии бизнес-процессов – придает дополнительную «жесткость» процессу клонирования. Приведем определение основных концептов онтологии модели.

Концепт «Адаптировать и документировать структурную единицу» для методов makeDivision и makeWorkShop. Агент клонирования производит документирование и адаптацию бизнес-системы, придерживаясь инструкций, которые могут быть формально описаны следующим образом. Пронумеруем вершины дерева p в порядке их обхода, начиная с нуля. Обозначим параметр адаптации, как ξ. Зададим в каждой вершине функцию $I(p,\xi)$ (operations_possible), которая определяет список операций, не нарушающих технологию. Введем понятие *квазилокального оператора H над лесом g*: преобразование, порождаемое функцией $I(p,\xi)$ в вершине p, посредством которого лесу g с выделенной вершиной p сопоставляется лес g^*. Функцию $I(p, \xi)$ можно назвать *булевым образом* оператора H для вершины p. Как следует из вышесказанного, можно задать шесть квазилокальных операторов, которые формально будем записывать в следующем виде: $H(p, I(p, \xi)): g \to g^*$.

Концепт «Конфигурирование бизнес-системы». Данный концепт назначается всей программной конструкции, описанной выше. Обозначим суперпозицию операторов, как $H \oplus G$ и будем понимать под этой записью последовательное применение квазилокальных операторов – сначала G, а затем H. Таким образом, процедура клонирования может быть представлена в следующем виде $J(\xi) = \sum H_{n-p}(p, I(p, \xi))$, где p изменяется от 0 до n. Задавая разные функции $I(p,\xi)$ можно определять различные операторы $J(\xi)$, в том числе такие, которые позволяют: (а) переместить элемент, (б) дублировать элемент и переместить дубль. Перемещение элемента структуры возможно как *по*, так и *против* направления обхода структуры, а также на другой уровень.

Для аналитического исследования программной симуляции можно воспользоваться λ-исчислением. Можно показать, что уже во втором поколении клонов имеет место

вырождение – происходит упрощение технологического процесса и утрата структурной информации; последнее снижает адаптивность.

Итак, в данном пункте рассмотрен механизм самовоспроизведения, такой, что клон системы создается одновременно с обходом агентом структуры оригинала. Предложены операторы преобразования графа структуры, совместимые с клонированием. Удачные решения фиксируются в структуре фирмы и составляют часть интеллектуальной собственности франчайзера. Тем самым показано, что механизм самовоспроизведения, основанный на клонировании, допускает изменчивость, что обеспечивает адаптацию бизнес-систем.

Самоорганизация. В настоящее время существует два четко выраженных подхода к кибернетическому моделированию биологии развития (морфогенезу) живых организмов [1]. Один известен как клеточные автоматы (однородные структуры), другой – как L-системы. L-системы, предложенные А. Линденмайером, формально описываются как параллельные развивающиеся грамматики и моделируют деление клеток и смену состояния клетки как следствие предшествующего ее состояния. В моделях клеточных автоматов состояние клетки изменяется в результате взаимодействия с другими клетками организма. Оба подхода прочно заняли свое место в теории морфогенеза. Однако оба подхода имеют общий недостаток – отсутствуют средства описания иерархической организации живых организмов.

Покажем, что в рамках *Scientific Profile* удается объединить оба описанных выше подхода и решить проблему описания иерархической структуры многоклеточных организмов. Корректно параллельные грамматики могут быть введены на фракталах.

В комбинации с рекурсией определения рекурсия использования позволяет строить новые типы фракталов. Одним из интересных вариантов рекурсии использования являются квазифракталы, способные моделировать процессы самоорганизации. Такие фракталы могут быть построены на основе формальных грамматик, которые условно можно назвать «*двухуровневыми (в общем случае – n-уровневыми) параллельными грамматиками*».

На компактном топологическом пространстве Ω можно определить формальную грамматику (на метрическом пространстве – математический анализ), например, следующим образом. Разобьем пространство на два подмножества σ_1 и σ_2, таких, что $\sigma_1 \cap \sigma_2 = \varnothing$ и $\sigma_1 \cup \sigma_2 = \Omega$. Пусть фрагмент кода некоего метода имеет вид:

```
String lm = this->leftMessage; String rm = this->rightMessage;
String m = lm + value + rm;
if (m == "0Ay") {value = "B";}
if (m == "0By") {value = "A";}
if (m == "xB0") {value = "A";}
if (m == "xA0") {value = "B";}  .
```

В данном случае объект получает сообщение от правого и левого соседа (т.е. вычисляется некоторая функция справа и слева) и изменяется текущее состояние (value) объекта согласно следующей параллельной (т.е. применяемой к каждому объекту одновременно) грамматике:

Аксиома: объекты $\sigma_1[A]$, $\sigma_2[B]$ – модель отрезка; состоит из двух частей. Инструкции: $0Ay \Rightarrow B$, $0By \Rightarrow A$, $xB0 \Rightarrow A$, $xA0 \Rightarrow B$. Продукции: $\mathrm{A} \to \mathrm{A}$, $\mathrm{B} \to \mathrm{B}$. Где $X = \{0, x, y\}$ – алфавит сообщений, $L = \{A, B\}$ – алфавит состояний, в которых могут находиться объекты σ_1, σ_2.

Не составляет труда составить более сложные примеры параллельных грамматик. Когда подобные грамматики не зависят от сообщений, приходим к грамматикам Линденмайера как частному случаю. Если изменение состояний зависит от получаемых сообщений и не зависит от состояния, то приходим к грамматикам

самовоспроизводящихся автоматов. Таким образом, объектные модели квазифракталов с параллельными грамматиками можно рассматривать как обобщение клеточных автоматов и L-систем. Характерной особенностью фрактала является его иерархическая структура, которая в данной грамматике никак не отражена. Поэтому, допустим, что каждый объект текущего уровня получает сообщение не только от своих соседей, но и сообщения с вышележащего уровня (или уровней). Мы будем полагать, что эти сообщения (в нашем случае их два – правое и левое) одинаковы для всех объектов текущего уровня и действуют одновременно с локальными сообщениями.

Продемонстрируем, как *n–уровневые параллельные грамматики* могут быть применены к проблемам морфогенеза. Небольшие изменения грамматики позволяют построить объектные конструкции, обладающие всеми свойствами многоклеточных организмов: ростом и дифференциацией, регуляцией размеров, регенерацией и размножением. Контактное взаимодействие клеток моделируется связями left и right объектов класса Generator (точнее аналогом этого класса), находящимися в роли листа. Иерархия организма моделируется экземплярами класса Generator, находящимися в роли узла.

Пусть имеем динамический список из экземпляров класса Generator, находящихся в роли листа, размещенных непосредственно в объекте класса Initiator. В качестве примера рассмотрим следующую одноуровневую параллельную грамматику. Пусть A – аксиома. Инструкции: $yAx \Rightarrow A$, $xAx \Rightarrow A$, $yAy \Rightarrow B$, $yBy \Rightarrow A$, $xBy \Rightarrow D$, $xAy \Rightarrow D$. Продукции: $A \rightarrow AA$, $B \rightarrow BB$, $D \rightarrow \lambda$., где λ – пустое слово.

Алфавит сообщений $X = \{x, y\}$; x посылается клеткой вправо, y – влево. Внутреннее состояние клетки определим полем state, которое может принимать значения из алфавита $L = \{A, B, D\}$ и доступно только для чтения по свойству state. Состояния A и B – два состояния живой клетки; напротив, клетка в состоянии D погибает. Выборка инструкций осуществляется из объекта genotype класса Dictionary, где в качестве ключа используется конкатенация сообщений и состояния в виде $\alpha\Psi\beta$, причем $\alpha,\beta \in X$, $\Psi \in L$. Основанием для модели служат данные генетических исследований простейших многоклеточных организмов. Гены, вовлеченные в процесс деления клеток, являются наиболее древними «генами многоклеточности». Они дополняют гены, контролирующие деление одноклеточных животных и помогают многоклеточным организмам синхронизировать этот процесс. Самые молодые – гены апоптоза, отвечающие за уничтожение клеток, более ненужных или не способных выполнять функции, нужные для организма.

Начальное состояние организма согласно описанной выше грамматике представлено одной клеткой (зиготой), находящейся в состоянии A. Применяя продукции и инструкции, получим следующую цепочку состояний: $yAx \Rightarrow A \rightarrow A\ A \rightarrow$
$yAy\ xAx \Rightarrow B\ A \rightarrow BB\ AA \rightarrow yBy\ xBy\ xAy\ xAx \Rightarrow A\ D\ D\ A \rightarrow A\ A$.

Конечное состояние организма идентично состоянию по завершению фазы роста ($yAx \Rightarrow A \rightarrow A\ A$). Поэтому рост организма останавливается, а дальнейшее существование организма поддерживается устойчивым циклом (в математической терминологии – предельный цикл, по М. Эйгену – гиперцикл).

Клетки могут взаимодействовать друг с другом на расстоянии с помощью сигнальных белков, передаваемых через межклеточное вещество. Это является основанием для применения двухуровневых грамматик, рассмотренных выше, и позволяет моделировать процессы формирования и функционирования органов многоклеточных организмов. Однако обсуждение имитационных моделей биологических организмов выходит за рамки данной темы.

Примеры и пояснения

1. ***Модель дуополии***. Описанная выше модель социального конфликта характерна для большинства моделей конфликтов. Для сравнения приведем расширенный вариант

экономической модели, известной как модель дуополии. Пусть в начальный момент времени фирмы *A* и *B* занимают какие-то доли рынка продукта. Это состояние устойчивое, малое отклонение устраняется соответствующими механизмами компенсации. *Фаза зарождения конфликта* – фирма *A* узнает, что фирма *B* готовит серию мероприятий (например, новую модификацию продукта), которое может привести к существенному перераспределению рынка сбыта. *Фаза развития конфликта*. Фирма *A* сначала планирует, а затем и начинает осуществлять комплекс мероприятий, которые должны помешать фирме *B* в реализации ее планов. Фирма *B*, узнав об этом, начинает предпринимать контрмеры. Когда одна из конфликтующих сторон переходит к активным действиям, наступает *фаза разрешения конфликта* – обе фирмы реализуют свои мероприятия. Заключительный эпизод описывается одной из моделей дуополии, например моделью Ф. Басса, и описывает процесс конкуренции на рынке. Финальный эпизод заканчивается установлением нового состояния равновесия. Даже если деятельность фирм не попадает под определение конфликта, то все равно последовательность фаз разрешения противоречия будет такой же.

2. ***Структурные изменения в процессах развития.*** Процессы развития сопровождаются изменениями структуры и организации системы. Рассмотрим некоторые из этих процессов.

Интенсивное развитие системы. В процессе развития связи системы с окружающей средой возрастают, в результате чего часть окружающей среды становится частью системы. Эта ситуация моделируется паттерном *Adapter*. При этом происходит изменение интерфейса системы.

Рассмотрим следующую модель (рис. 47). Пусть некое предприятие (Enterprise) производит продукцию (свойство product), для чего потребляет некоторый материал (свойство material). Пусть в окружающей среде (Production {Concept = Сфера производства}) существует фирма, которая поставляет этот материал (класс MaterialB{Concept = Материалы группы B}), вырабатывая его из сырья (класс MaterialA{Concept = Материалы группы A}).

С течением времени эта фирма становится основным поставщиком материала для предприятия и постепенно сворачивает остальные свои связи. Естественно рассматривать эту фирму в составе данного предприятия. Такие «плавающие» границы можно моделировать паттерном *Adapter* так, как это показано на рис. 47. Благодаря «расслоению» окружающей среды, *Исследователь* по необходимости может производить измерение либо в контексте Production, либо в контексте Corporation.

Фрактализация. Еще один способ (возможно наиболее характерный) образования иерархической структуры – образование квазифрактальной системы. Особенность – на каждом уровне иерархии повторяется структура верхнего уровня.

Экстенсивное развитие. Усложнение системы происходит за счет усложнения атомарных объектов. Сама схема моделирования точно такая, как и в случае фрактализации. Однако теперь новые уровни образует атомарный объект. Тем самым он выступает как в роли атома, так и в роли подсистемы. Процесс соответствует известному методу построения организации путем делегирования полномочий.

Организационная перестройка. Изменение функций элементов системы. Моделируется паттерном Strategy. Пример – изменение дисциплины обслуживания заказов (первый пришел – первый ушел, первый пришел – последний ушел).

3. ***Жизнь как информационный процесс****.* Большинство живых организмов формируется за счет процессов самоорганизации, которые управляются генотипом. Этот процесс представляет интерес и для социально-экономических систем (заметим, что именно имитационное моделирование может сделать этот подход интересным для практики).

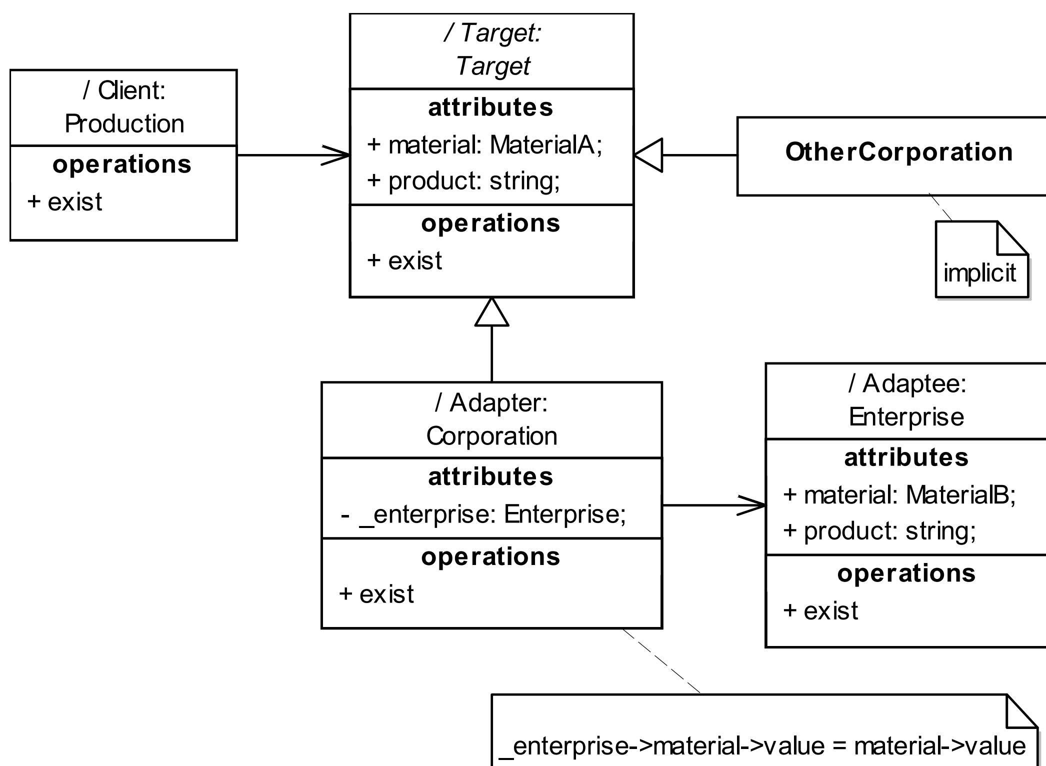


Рис. 47. Диаграмма классов для модели «плавающих» границ

С точки зрения объектного моделирования, жизнь – довольно простое и даже ординарное явление. О жизни можно говорить всякий раз, когда возникает устойчивый процесс самовоспроизведения. Некоторые из простейших моделей были рассмотрены выше.

Возможен более интересный случай. Если элементы системы обладают свойством рефлексии, то возможно самовоспроизведение процесса без самовоспроизведения объекта. В обществе за время существования цивилизации должны были возникнуть подобные устойчивые процессы. Они могут передаваться от одного человека к другому или от одной общественной группы к другой или даже сами порождать общественные группы. Например, привычку к курению можно рассматривать как простейший информационный вирус.

Разум не обязательно связан с жизнью. В книге [2], которую мы уже упоминали, биологический вид человека рассматривается как система управления биологической популяцией и выдвигается гипотеза о когнитивной активности биологического вида. Для нас особенно интересна предлагаемая модель коммуникации биологического вида с особями. Коммуникации крайне затруднены ввиду значительного различия в темпах восприятия.

*4. **Эволюция Социальных Систем***. Выше мы рассмотрели один из механизмов самовоспроизведения организации. То, что франчайзинг не только воспроизводит организацию, но и имеет механизмы изменчивости, позволяет говорить о некой эволюции.

Подобная точка зрения не оригинальна. В 1990-м году в США вышла в свет книга Питера Сенге «Пятая дисциплина», посвященная новой концепции в теории управления – теории самообучающейся организации. Сегодня эта книга стала классикой, цитируется едва ли не в каждой статье, посвященной современному менеджменту, и является одним из лучших описаний реальных проблем, с которыми сталкиваются крупные корпорации. Схожая точка зрения представлена в книге И.В. Бощенко «Эволюция Социальных Систем», вышедшей в 2005 г. Автор проводит аналогию между организацией и нейроном и предлагает ряд гипотических эволюционных моделей развития социума.

Эволюционная точка зрения на социальные процессы в той или иной мере выражена в целом ряде современных направлений исследования. Приведем в связи с этим только два высказывания. В трактовке М. Хаммера и Дж. Чампи, термин «реинженириng бизнеса» означает «создание компании заново» (см. книгу Хаммер М., Чампи Дж. Реинжиниринг корпорации: манифест революции в бизнесе. – СПб.: Издательство С.-Петербургского университета, 1997). По мнению С.П. Расторгуева, «Порой проще убивать элементы (формальные нейроны) и рождать их заново, чем корректировать …» [48].

5. ***Хоум-Сити***. Строим дом. Когда готов новый этаж, заселяем его. Используем паттерн строитель. Эффективность сначала растет. Затем наступает фаза эксплуатации объекта. Затем начинаются затраты на ремонт. Когда соотношение «качество-цена» жильцов не устраивают, они покидают дом и дом разрушается из-за недостатка финансирования.

Это простейшая модель жизненного цикла системы.

6. ***Самоорганизация***. Самоорганизация довольно распространена в природе. Самый простой пример – это озеро, в которое впадает множество рек, а вытекает всегда только одна река. Эта модель подобна модели [18].

7. ***Роевой интеллект***. Следующая модель интересна тем, что демонстрирует пример коллективного поведения животных. Для стаи птиц характерно следующее: (а) в воздухе птицы почти никогда не сталкиваются; (б) стая перемещается скоординировано. Однако известно, что у многих птиц нет вожака. Стая представляет собой пример *роевого интеллекта*. Другие известные примеры – колония муравьев или пчел. Коллективное поведение есть результат индивидуального поведения каждой отдельной особи.

Компьютерная модель стаи птиц (модель Boids) создана Крейгом Рейнольдсом (Craig Reynolds) в 1986 году. В имитационной модели запрограммировано поведение каждой из птиц в отдельности и взаимодействие птиц. В модели Рейнольдса использованы три правила. Во-первых, каждая птица стремится избежать столкновений с другими птицами. Во-вторых, каждая птица двигалась в том же направлении, что и находящиеся рядом птицы. В-третьих, птицы стремились двигаться на одинаковом расстоянии друг от друга. Визуально, на экране компьютера, стая птиц выглядела крайне правдоподобно. Птицы сбивались в группы, уходили от столкновений и хаотично метались. В развитие модели можно определить правила поведения птиц, так, чтобы в результате взаимодействия формировалась та или иная геометрическая фигура, например, журавлиный клин. Для описания возможных конфигураций стаи мы использовали паттерн *Flyweight* [14].

8. ***Куча песка***. В развитии систем большое значение имеют периоды быстрых изменений и катастроф (бифуркаций).

Имеется решетка LxL, в ячейках которой находятся песчинки. Ячейки с 1, 2, 3 и 4 песчинками считаются устойчивыми, а более – неустойчивыми. Неустойчивые ячейки опрокидываются, передавая 4 песчинки своим соседям (имеющим общую сторону). На границе решетки песчинки теряются. В начальный момент вся решетка устойчива. Затем одна из ячеек выводится из состояния устойчивости. Возникает лавина. Требуется определить размер лавины (полное число опрокидываний) и площадь лавины (число клеток хотя бы раз опрокинувшихся). Данная модель демонстрирует так называемую *самоорганизованную критичность* (П. Бак), которая позволяет описать множество сложных явлений, таких как финансовые кризисы и природные катастрофы.

Мы рассматривали модели с *обрушением* (активные среды), типичным примером которых является модель «Эффект домино». Модель допускает многочисленные нетривиальные обобщения и имеет прикладное значение – модель лесных пожаров (два яруса), модель массовых столкновений на скоростных автомагистралях, модель формирования доменов (несколько точек возбуждения) и др.

Глава III. Изучение программных симуляций

Конечный результат изучения программной симуляции (артефакт имитационного моделирования) – дерево сценариев с указанием количественных значений критических характеристик. Сценарии функционирования, развития или эволюции систем могут быть представлены в форме диаграмм последовательности действий. Вопросы познания – это вопросы гносеологии. В этой главе рассмотрим вопрос о содержимом двух архитектурных пакетов гносеологического раздела. Условно материал можно разграничить следующим образом. Аналитические методы, рассматриваемые в разделе 3.1, определяют содержание пакета «*Epistemology Entity*», а модели экспериментальных установок и приборов - «*Research Instruments*» - рассмотрим в разделе 3.2.

Традиционно в имитационном моделировании превалирует экспериментальный метод. Однако моделирующая программа представляет собой просто компьютерную программу, и поведение программной симуляции может быть исследовано аналитическими методами, если обратится к формальным моделям программ. Чтобы различать математические, имитационные модели и формальные модели программ, в последнем случае мы предлагаем термин *модели объяснения*. Такой выбор термина связан с тем, что формальный анализ отвечает на вопрос «Почему?», в то время как имитационный эксперимент дает ответ на вопрос «Как?». Оба метода исследования дополняют друг друга. После разработки имитационной модели, на основе данных имитационных экспериментов составляется модель объяснения, а ответы на все вопросы исследователь получает аналитическим путем, изучая модель объяснения. Подобная стратегия исследования довольно конструктивна, поскольку позволяет создавать хорошо обоснованные методики исследования имитационных моделей. Мы сначала рассмотрим формальные методы изучения программных симуляций, а затем вернемся к экспериментальному методу, чтобы показать, как производится планирование эксперимента.

В *computer science* известны многочисленные формальные модели программ, развитие которых в значительной степени обусловлено проблемами верификации (проверки правильности программ). Обычно выделяют два подхода – алгебраический и логический. Для наших целей в алгебраическом подходе представляют интерес такие исчисления, как *CCS* (R. Milner), *CSP* (C.A.R. Hoare), π-исчисление (R. Milner) и некоторые другие. В логическом подходе – метод *model checking*. Рассмотрим возможности этих методов в качестве моделей объяснения.

3.1. Аналитические методы

Алгебраический подход. Идея этого подхода заключается в том, чтобы операции над процессами рассматривать как алгебраические операции. Это позволяет формальным способом доказывать эквивалентность двух и более процессов (см. [39], [54], [65]). Хотя в данной книге мы отдаем некоторое предпочтение CCS, для глубокого понимания алгебры процессов мы все же рекомендуем очень популярную среди специалистов книгу А. Хоара [65].

Calculus of Communicating Systems (R. Milner). Исчисление CCS (алгебра взаимодействующих процессов) было разработано Р. Милнером в 1980 г. в Эдинбургском университете. Алгебра процессов – это язык спецификации взаимодействующих процессов, функционирующих параллельно. Отражает только параллелизм и взаимодействие процессов по именованным каналам, но не мобильность процессов.

Алгебра – это множество объектов одной природы и набор операций над этими объектами, результат которых – объект того же множества. Можно рассматривать

параллельную композицию процессов как процесс. После выполнения действия, процесс то же остается процессом. Следовательно, процессы имеют алгебраическую структуру, их можно рассматривать как элементы некоторой *алгебры процессов*.

Обозначим: a?x – прием сообщения, посланного по каналу a, в переменную x; a!v – посылка сообщения v по каналу a. Операции алгебры процессов:

1. Префикс: a.E – процесс выполняет действие a, а затем ведет себя как процесс E.
2. Параллельная композиция: E | F.
3. Сумма: E+F – процесс ведет себя как процесс E или как процесс F.
4. Ограничение: E \L – процесс имеет скрытый процесс L.
5. Переименование: E [a_1/b_1, a_2/b_2, …, a_n/b_n] – вводим другие имена для процессов.

Часы Хоара. Продемонстрируем использование CCS на следующем простом примере. Пусть есть класс Clock, который имеет два поля – pendulum: Boolean и time: Integer. Свойство time показывает текущее значение поля time. Метод exist задает квант существования часов и представлен следующим кодом

```
if (pendulum) { pendulum = false;} else {pendulum = true; time = time + 1;}
```

Рассмотрим *Исследователя* в роли *Наблюдателя контекста*. Спецификацию прецедента запишем в стиле логики Хоара:

Прецедент: Показать время
Предусловие: pendulum:= true, time = 0
Сценарий: Observation
Постусловие: time = n.

Редуцируем объектную модель на процессную модель (тут можно воспользоваться диаграммой деятельности). Запишем следующую систему

Clock:= (x + y). Clock
x = tick. y
y = tock. x,

где tick – процесс, в котором изменяется состояние переменной pendulum с true на false, tock – переменная pendulum изменяется с false на true, а переменная time увеличивается на единицу.

Опишем наблюдателя контекста Observation = !exist. !exist. ?time. Observation. Вся экспериментальная ситуация будет Observation | Clock.

Что будет наблюдать исследователь? Пусть n=1; построим последовательность

Clock:= ?exist. tick. ?exist. tock. !time. = ?exist. ?exist.!time. / tick. tock.

В этом случае мы будем наблюдать натуральный ряд чисел 1, 2, 3, … . Поскольку все результаты получены из системы (1), то эта система и будет *моделью объяснения*.

Рассмотрим часы несколько иной конструкции, известные как «маятник Ньютона» или «колыбель Ньютона». На нитях подвешены два шарика. Левый шарик ударяет по неподвижному правому и останавливается. Правый шарик ведет себя подобным образом. Пусть имеется два параллельных процесса x и y. Движение левого шарика – это процесс x, правого – y. В этой системе имеет место синхронизация путем обмена сообщениями. Пусть переменная time доступна обоим процессам. Процесс x, послав сообщение, ведет себя как процесс tick и не может изменить time. Процесс y, получив сообщение, ведет себя как процесс tock и изменяет time на единицу, после чего посылает сообщение процессу x. Затем роли процессов меняются.

Clock:= (x | y). Clock

x = !time. tick. ?time. tock
y = ?time. tock. !time. tick.

Можно организовать синхронизацию через общие переменные.

Модель договорной цены. Рассмотрим процесс, моделирующий достижение некоторой договоренности, например цены в процессе сделки (или единого мнения двух экспертов). Этот процесс интересен тем, что это один из массовых процессов. Процесс заканчивается достижением информационного равновесия, т.е. достижением равенства представления агента о другом агенте с действительным типом контрагента.

bargain = seller | buyer
seller = estimate. speech!price1. speech?price2 (0.+ seller)
buyer = estimate. speech!price2. speech?price1 (0.+ buyer).

Можно преобразовать процесс в диалог с точки зрения покупателя, продавца или вообще стороннего наблюдателя.

buyer = speech!price2. speech?price1. speech!price2. speech?price1.0 / estimate. estimate.

Это и есть то, что будет фиксировать *Исследователь*. Система процессов тем самым и есть *модель объяснения* для данной имитационной модели.

CCS эффективно там, где необходимо анализировать стационарные процессы или предельные циклы. Пример – стационарное существование многоклеточного организма. Этот предельный цикл представляет собой смену информационной фазы на фазу реструктуризации.

π-исчисление. CCS не описывает мобильность процессов и изменение конфигурации систем. π-исчисление (π-calculus) – это расширение исчисления CCS, предложенное самим Р. Милнером для того, чтобы работать с системами процессов с изменяемой конфигурацией (мобильными процессами и реконфигурируемыми связями).

В дополнение к CCS, в π-исчислении можно создавать новые имена каналов и передавать их другим процессам при коммуникации параллельных процессов. Процесс, принявший имя канала, может взаимодействовать по этому новому каналу с окружением.

Введем следующие обозначения:

0 – пустой процесс, Nil;
P, Q, ... – имена процессов;
P|Q – параллельная композиция;
$\underline{a}$(x).P или a(x)!.P – вывод по каналу a значения x;
a(y).P или a(y)?.P – ввод по каналу a нового значения для y;
!P – репликация P (порождение нового параллельного процесса P);
{v/x} P – в терме P все свободные вхождения x заменяются на v;
{new x} P – в процессе P имя x является локальным.

Рассмотрим процесс самоорганизации логистических цепочек [18]. Данный процесс иллюстрирует еще один массовый способ достижения информационного равновесия. Принципиальное отличие от модели договорной цены состоит в том, что представления агента о контрагенте не корректируются, а продолжается поиск другого контрагента требуемого типа $t_r \in L$ на множестве N. В этом процессе имеет место динамическое изменение связей. Рассмотрим эту активность с точки зрения π-исчисления.

Дадим краткое описание объектной модели. Цепочку купли-продажи моделируем динамическим списком, каждый объект которого имеет класс Participant (Concept = Участник). Класс Participant имеет свойства left и right. Массив contingent – контейнер для всех фрагментов цепочек; первоначально это есть множество N. Активность *selecting_the_contractor for: p from: i* позволяет подобрать для агента p (активный агент),

расположенного в ячейке i контрагента q (пассивный агент). Данная активность показана на рис. 48. Основу процесса составляет цикл по всем ячейкам массива contingent. Действие *passive_agent_choose* выбирает крайне левого агента в цепочке. Поэтому действие сводится к q:= contingent at:j. Действие *observe* возвращает переменную found типа Boolean, которая имеет значение true, если типы агентов совпали, и false – в противном случае. Действие, обозначенное на рис. 48 как p right: q, представляет собой активность, которая приводит к «зацеплению» агента q к фрагменту цепочки (метод p right: q) и освобождению ячейки j (contingent at:j put: nil.).

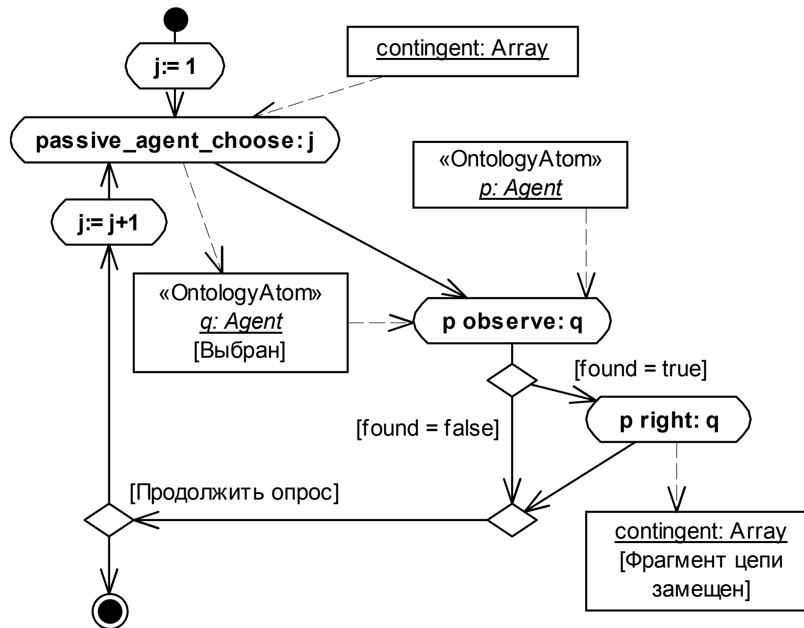


Рис. 48. Диаграмма деятельности для декомпозиции действия selecting_the_contractor

Пусть имеется *N* параллельных процессов x_i, инкапсулированных в объектах класса Participant. Пусть активным агентом является *i*-й агент и ищет контрагента типа w_i. Рассмотрим систему параллельных процессов selecting_the_contractor | x_1 | x_2 | … | x_n

selecting_the_contractor = p(q)!. (0 + selecting_the_contractor)
xi = p(z)?. z!. z(v)?. comparison. (grap. havoc. x_i + x_i)
xj = q?. q(w_j)!. 0, $i \neq j$,

где z – канал связи с пассивным агентом, который назначается динамически и поэтому первоначально не определен, v – неопределенная величина типа контрагента.

Процесс selecting_the_contractor (рис. 49) в качестве канала связи использует указатель p активного агента, представленного процессом x_i. По нему он передает указатель q очередного пассивного агента. Процесс x_i получает по каналу p имя канала q, после чего по каналу q запрашивает у процесса x_j значение переменной w_j (посылая ему сообщение с селектором agent_type). Процесс x_j по каналу q пересылает ему переменную w_j. После этого процесс x_i производит сравнение w_j с w_i (comparison) и, если типы совпали, происходит «зацепление» пассивного агента (grap) и освобождение ячейки j (havoc, опустошение); далее процесс ведет себя как x_i.

Формализуем приведенные выше рассуждения (сократим название процессов до первой буквы).
p(q)!. 0 | p(z)?. z!. z(v)?. c. g. h. 0 | q?. q(w_j)!. 0
1. Первые два процесса взаимодействуют по каналу p.
0 | q!. q(v)?. c. g. h. 0 | q?. q(w_j)!. 0
2. Второй и j-й процессы теперь могут взаимодействовать по каналу q.

0 | q(v)?. c. g. h. 0 | q(w_j)!. 0
3. j-й процесс по каналу q передает значение w_j второму процессу
0 | {wj/v} c. g. h. 0 | 0
, где {x/v} P – в терме P все свободные вхождения v заменены на x

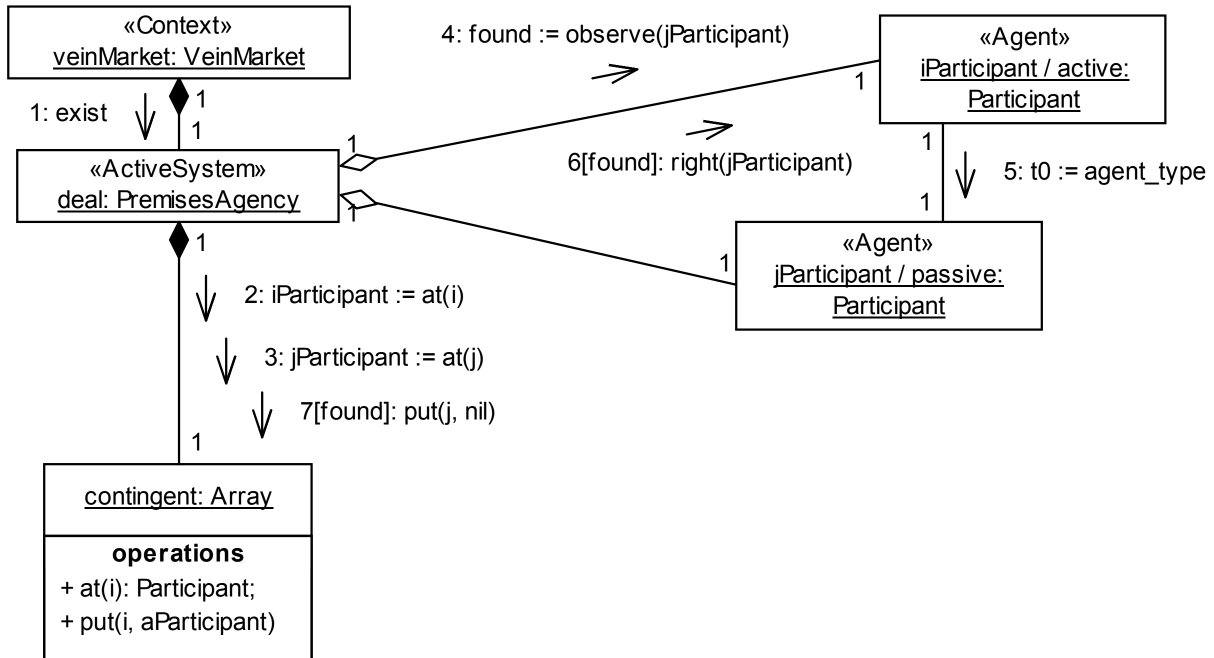


Рис. 49. Диаграмма кооперации для активности selecting_the_contractor

Логический подход. Темпоральная логика CTL*. В проблеме верификации программ эффективными оказались методы *Model Checking* [30]. Мы решили попробовать применить эти идеи для изучения программных симуляций, а также для фиксации результатов имитационных экспериментов. Дадим краткое описание этого подхода.

Для описания свойств программной системы используются темпоральные логики, которые задаются на моделях программных систем. Темпоральные логики – это модальные логики, которые строятся добавлением к логике высказываний новых знаков, отражающих свойства времени. Известны логики Прайора, фон Вригта, Леммона и др. Пнуели предложил логику для верификации компьютерных программ. Логика содержит дополнительные знаки **F** (когда-нибудь будет), **G** (всегда будет), **X** (в следующий момент будет) и ряд аксиом. Мы используем логику CTL* (логика деревьев вычислений), которая отличается от логики Пнуели дополнительными темпоральными операторами **U** (до тех пор, пока) и **R** (высвободить). CTL* имеет два полезных подмножества: CTL (логика ветвящегося времени) и LTL (логика линейного времени).

Изучение программных симуляций методами темпоральной логики является универсальным подходом. Если система состоит из пассивных элементов, то адекватным математическим языком будет теория динамических систем. Для математических моделей теории динамических систем имитационная модель может быть построена для символического образа динамической системы. Можно составить формулы CTL* для критических точек, окрестности критических точек, аттракторов, репеллеров и точек орбит. Однако в качестве примера мы рассмотрим более сложный случай – когда модель составляется для описания теоретико-игровой ситуации. Это обусловлено широким классом моделей – имитационным моделированием организационных систем. Для активных систем адекватным математическим аппаратом является теория игр.

Обратимся к примеру активной системы п. 2.3. На первой итерации можно выделить ведущий механизм управления – механизм распределения. Пусть множество переменных есть $V = \{p, m_1, m_2\}$, которые приобретают значения на домене интерпретации D = $\{0, 1, 2\}\times\{0, 1, 2\}\times\{0, 1, 2\}$, где 0 – игрок еще не сделал хода, 1 – выбрал первую альтернативу, 2 – выбрал вторую альтернативу. Определим множество атомарных высказываний $AP = \{p = 0, p = 1, p = 2, m_1 = 0, m_1 = 1, m_1 = 2, m_2 = 0, m_2 = 1, m_2 = 2\}$. Высказывания будут следующие: $p = 1$ – центр предложил процедуру планирования, при которой агенты сообщают свои идеальные точки r_j, $j = 1,2$ функций предпочтения, $p = 2$ – центр предложил процедуру планирования, при которой агенты сообщают заявки, $m_j = 1$ – j-й агент сообщает недостоверную информацию, $m_j = 2$ – j-й агент сообщает достоверную информацию. Функция разметки L: $S \to 2^{AP}$ и отношение переходов R показаны на рис. 50. Множество начальных состояний $S_0 = \{(0, 0, 0)\}$.

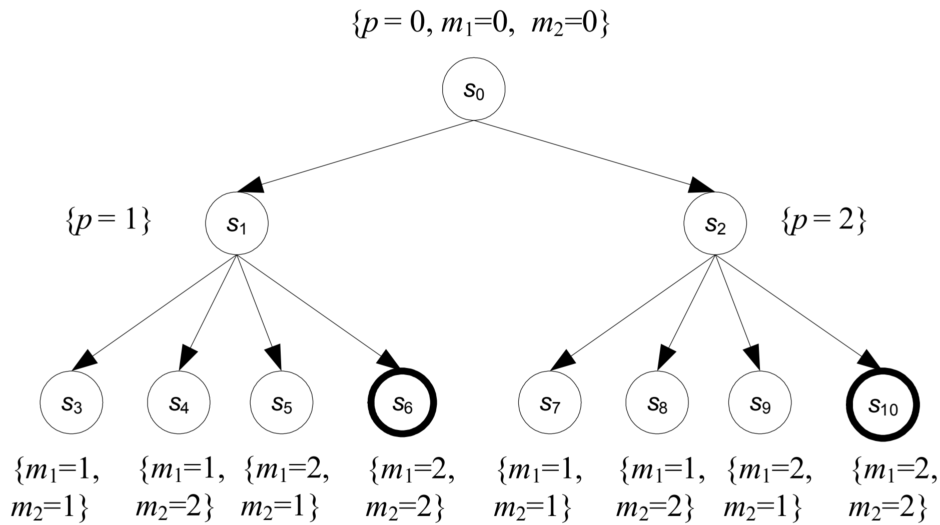


Рис. 50. Структура Крипке для программной симуляции

Приведем теперь некоторые формулы CTL* для теоретико-игровых моделей. Формулы CTL* строятся из логических операций $\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, темпоральных кванторов **A** («на всех путях»), **E** («на некоторых путях») и темпоральных операторов **X** (neXt), **F** (Future), **G** (Global), **U** (Until), **R** (Release). В нашем случае игра будет иерархической и для этой ситуации можно выписать следующие формулы.

1. Начальное и терминальное состояние (окончание игры):

$$f_0 = (p=0) \wedge (m_1=0) \wedge (m_2=0),\ f_1 = \neg (p=0) \wedge \neg (m_1=0) \wedge \neg (m_2=0)$$

2. Предтерминальное состояние:

$$\neg f_1 \wedge \mathbf{EX} f_1$$

3. Равновесие Нэша. Составим формулу, которая позволяет определять состояния равновесия в теоретико-игровом смысле. Равновесие в смысле Нэша – это такое состояние, одностороннее отклонение от которого не выгодно ни одному из агентов. Пусть w^* – экстремум целевой функции центра, а v_j^* – соответствующее значение функции предпочтения j-го агента. Расширим множество атомарных высказываний AP, добавив в него высказывания $w<w^*$ и $w=w^*$, а также $v_j<v_j^*$, $v_j=v_j^*$, $j = 1,2$. На основе данных имитационных экспериментов выполним разметку для терминальных состояний (на рис.50 выделены состояния, в которых высказывание $w=w^*$ истинно).

Составим следующую формулу для начального состояния s_0:

$$\varphi = \mathbf{EF}(\varphi_1 \wedge \mathbf{EX}\varphi_2) \wedge \mathbf{EG}\ \text{isOptimumPath}.$$

Потребуем, чтобы подформула $\varphi_1 \wedge \mathbf{EX}\varphi_2$ была истинной в некотором предтерминальном состоянии, а φ_2 – в некотором терминальном состоянии. Формулы для φ_1 и φ_2 запишем в следующем виде:

$$\varphi_1 = \text{isOptimumPath} \wedge \neg f_1 \wedge \mathbf{AX}\,(\delta \wedge f_1),$$

$$\varphi_2 = \text{isOptimumPath} \wedge f_1 \wedge (w{=}w^*) \wedge [(v_1{=}v_1^*) \wedge (v_2{=}v_2^*)],$$

где $\delta = [(w<w^*) \vee (w{=}w^*)] \wedge [(v_1<v_1^*) \vee (v_1{=}v_1^*) \wedge (v_2<v_2^*) \vee (v_2{=}v_2^*)]$; т.е. если рассматриваемый путь оптимальный, то на следующем шаге все состояния, кроме равновесного, менее (точнее – не более) выигрышны для каждого игрока.

Рассмотренные формулы допускают непосредственное обобщение на некоторые модели, представляющие практическую ценность. В модель анализа прецедентов, в пакет со стереотипом *Epistemology Entity* необходимо добавить библиотеку процедур для вычисления **EX**, **EG**, **EU** и процедуры заполнения таблицы выполнимости подформул. Сама таблица и средства визуализации определяются в пакете со стереотипом *Research Instruments*.

Приведенные выше формулы позволяют доказывать разного рода общие утверждения о свойствах программной симуляции. В рассматриваемом примере можно доказать (вычислением), например, следующее утверждение. Существует два решения игры, а именно – метаигрок выбирает либо первую, либо вторую процедуру планирования, а оба игрока в обоих случаях сообщают истинное значение плановой информации.

Таблица 4

Свернутая таблица истинности для пути $s_0s_1s_6$

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
f_0	*true*										
f_1				*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
$\neg f_1$ **EX** f_1		*true*	*true*								
isOptimumPath	*true*	*true*					*true*				
φ_1		*true*									
φ_2							*true*				
φ	*true*										

В нашем примере все вычисления можно выполнить непосредственно в таблице истинности. Все формулы CTL* – формулы состояния, поэтому таблицу истинности можно представить так, как показано в Табл. 4. Из таблицы после завершения вычислений можно удалить атомарные высказывания, все «ординарные» состояния (точки орбит) и подформулы. Таблица истинности тогда даст компактное и стилизованное представление множества состояний. Для нашего примера два пути будет обращать φ в истину ($s_0s_1s_6$ и $s_0s_2s_{10}$). Заметим, что если из модели ментальности агентов исключить гипотезу благожелательности (заменив ее, допустим, на противоположную), и $r_1 < 1/2$, $r_1 + r_2 > 1$, то решением игры будет только единственный путь $s_0s_1s_6$.

На последующих итерациях рассматривается влияние других механизмов управления на полученный результат. Такой анализ позволяет объяснить, почему одни механизмы распределения внезапно становятся неманипулируемыми, а другие, напротив, теряют это качество.

Примеры и пояснения

1. Рефлексивная игра. Активные системы с рефлексируюшими агентами – это как раз та область моделирования, где аналитические методы исследования наиболее востребованы. Это связано с тем, что всякий раз приходится иметь дело с потенциальной бесконечностью. Напомним, что два рефлексирующих агента имеют много общего с двумя параллельными зеркалами.

Вернемся к модели пешеходного перехода п. 2.3. Казалось бы, где здесь взяться бесконечности? Однако мы сразу можем увидеть эту возможность, если обратимся к модели объяснения.

Приведем структуру Крипке для ситуации этой рефлексивной игры (рис. 51). В начальном состоянии s_0 агенты имеют разные представления r_i о ПДД. Получая сообщение m_i, агенты могут изменить свои представления и достигнуть информационного равновесия. Однако если оба агента одновременно получают сообщения, они вынуждены рефлексировать. На пути $s_0s_5s_6$ возможен цикл с бесконечной рефлексией. И для этой модели потребуется формальное доказательство, что этот цикл будет, по меньшей мере, конечным.

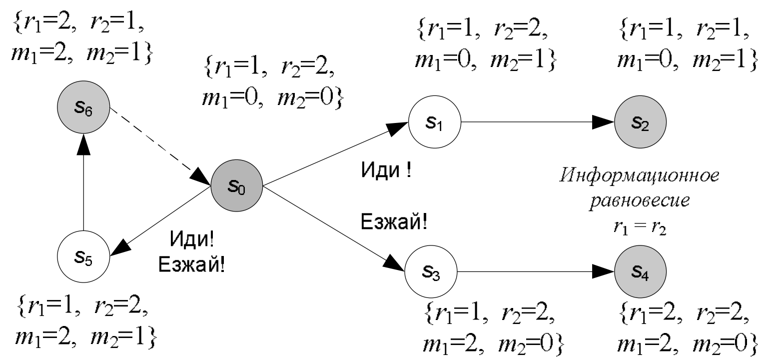


Рис. 51. Структура Крипке для модели «Пешеходный переход»

Чтобы не было бесконечного цикла, необходимо нарушить симметрию альтернатив. Это может быть светофор или спонтанное нарушение симметрии. Знание ПДД – тоже нарушение симметрии.

2. Аттракторы и репеллеры

Для обыкновенных дифференциальных уравнений объектная модель позволяет задавать *символический образ* фазового пространства динамической системы. Впервые кодировка траекторий последовательностями символов для описания глобального поведения геодезических на поверхностях отрицательной кривизны была применена Ж. Адамаром в 1898 г. Основателем методов *символической динамики* является Х. Морс. Термин «символическая динамика» ввели Х. Морс и Ж. Хедлунд. В 1983 году Г.С. Осипенко ввел понятие *символического образа* динамической системы [43].

Пример. Рассмотрим динамическую систему $dx/dt = \beta + au^2$, где $a>0$. Как известно, при $\beta<0$ имеется два положения равновесия – устойчивое и неустойчивое (см. рис. 52, точки B и D). При $\beta=0$ возникает негрубая система. При $\beta>0$ точки равновесия исчезают.

Онтология *Scientific Profile* определяет концепт времени как дискретно-событийное время. Поэтому в объектной модели можно ввести ассимтотические переходы, т.е. переход обозначается явно даже в том случае, если для достижения данного состояния необходимо бесконечно большое время. Каждое положение точки моделируется полем value (см. рис. 52). Это позволяет ввести вместо фазового пространства символьные ячейки, которые в целом образуют *символьный образ* динамической системы [37]. Предложенный подход в принципе отличается от общепринятых методов дискретизации

динамических систем (в т.ч. от [37), и, на наш взгляд, больше соответствует качественной теории дифференциальных уравнений.

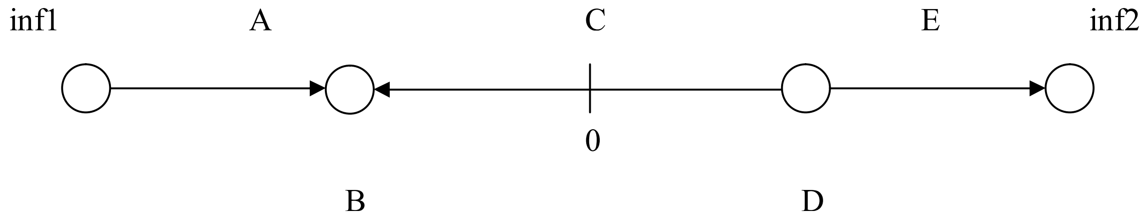


Рис.53. Моделирование фазовой плоскости состояниями

Рассмотрим объектную модель для нашего примера. Пусть объект dynamicSystem класса DynamicSystem имеет состояния $S = <l, A, B, C, D, E, r>$. Для элементов S задано некоторое отношение порядка. С++ код имеет вид

```
char value;
switch (value) {
    case 'l': value = 'A'; break;
    case 'A': value = 'B'; break;
    case 'B': value = 'B'; break;
    case 'C': value = 'B'; break;
    case 'D': value = 'D'; break;
    case 'E': value = 'r'; break;
}  .
```

Представим систему процессов структурой Крипке $M = (S, S_0, R, L)$ над множеством атомарных высказываний AP (в нашем случае это множество утверждений вида value='l', value='A', … по всем S; эти утверждения для разных состояний могут быть либо истинными, либо ложными), где S-множество состояний (два или четыре экземпляра состояний L), S_0 – множество начальных состояний, R – отношение переходов (инструкции и продукции, причем продукции описывают переходы между разными экземплярами состояний, рис. 57), L: $S \rightarrow 2^{AP}$ функция разметки.

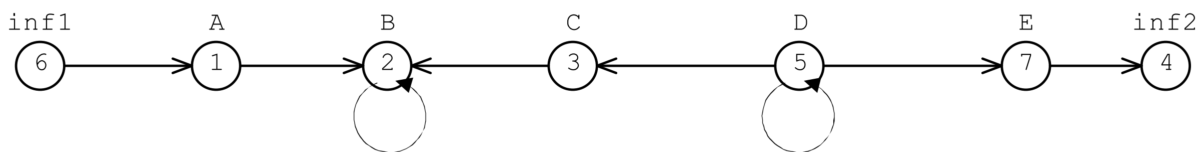


Рис. 53. Граф переходов

В CTL* зададим формулы для критических точек и орбит.

Критические точки. Существуют пути, такие, что из состояния s^* следующее состояние тоже s^* (рефлектирующие переходы). Формула будет следующая:

$$f_1 = \mathbf{EG}(s^* \rightarrow \mathbf{X}\, s^*)$$

или для предельного цикла $\mathbf{EG}(s^* \rightarrow \mathbf{AF}\, s^*)$.

Малое возмущение. Пусть s – некоторое состояние, для которого истинно атомарное высказывание p. Состояние s' является малым возмущением s ($s, s' \in S_0$), если истинным является выражение

$$f_2 = \mathbf{E}[(\mathbf{X}p \wedge \neg p\mathbf{U}p) \vee (\mathbf{X}\neg p \wedge p\mathbf{U}\, \neg p)].$$

*Аттрактор s**. Существуют только входящие в состояние s* переходы. Пусть для s* истинно атомарное высказывание p (в нашем примере таковым высказыванием будет

value=B). В следующем состоянии пути точка будет критической. Определим этот критерий формулой

$$f_3 = \mathbf{E}[\mathbf{X}p \wedge \neg p\mathbf{U}p] \wedge f_1.$$

*Репеллер s**. Для s* истинно атомарное высказывание *p* (в нашем примере таковым высказыванием будет value=D). Все переходы из *s** – исходящие. Зададим формулу

$$f_4 = \mathbf{E}[\mathbf{X}\neg p \wedge p\mathbf{U}\ \neg p]\ ] \wedge f_1.$$

Орбита. Состояние *s* принадлежит орбите, если истинно

$$f_5 = \neg\ f_1.$$

Для каждого обыкновенного дифференциального уравнения мы можем построить таблицу истинности, которая дает *стилизованный образ* динамической системы (стилизованные изображения – это упрощенные изображения предметов мира, которые точно передают характерные черты изображаемого предмета). Например, для рассмотренного выше уравнения получим таблицу истинности представленную в табл. 5.

Таблица 5

Таблица истинности для логического образа

	l	A	B	C	D	E	r
f_1			*true*		*true*		
f_2		*true*		*true*		*true*	
f_3			*true*				
f_4					*true*		
f_5	*true*	*true*		*true*		*true*	*true*

Таблица истинности дает достаточно подробное представление о фазовом портрете динамических систем, однако практически ничего не сообщает об орбитах (кроме того, что состояние принадлежит какой-то орбите). Более подробные сведения об орбитах можно получить, если использовать LTL (логику линейного времени). Как CTL, так и LTL являются подмножествами CTL*. Последняя позволяет описывать еще более тонкие свойства динамических систем.

Анализ фазового пространства динамической системы тесно связан с проблемой обращения времени. Явно инверсию по времени можно задать двумя способами. Первый способ состоит в следующем. Пусть в структуре Крипке $M = (S, S_0, R, L)$ множество начальных состояний совпадает с множеством репеллеров $S_0 = S_R$, а множество аттракторов обозначим как S_A. Тогда можно ввести инверсную структуру $\underline{M} = (S, S_A, \underline{R}, L)$, где $\underline{R}$ – инвертированное отношение переходов. Темпоральную логику CTL* можно задать на этой двойственной структуре. Второй способ состоит в том, что в CTL* вводится оператор инверсии *I*, который меняет порядок пути с прямого на обратный. Однако в этом случае мы получаем новую темпоральную логику, которая требует своего обоснования.

3. Другие формальные методы. За пределами рассмотрения остались множество других, интересных и хорошо зарекомендовавших себя методов исследования программных симуляций. Мы ограничимся только следующими замечаниями.

Судя по всему, основой применения различных формализмов может стать модальная логика. *Логика изменения* лежит в основе формализмов процессных алгебр и темпоральных логик. Особое значение имеет *логика причинности*. В рамках коммуникационной парадигмы мы предположим, что причиной изменения является один или несколько коммуникационных актов. Напомним, процесс коммуникации моделируется диаграммами классов и диаграммами кооперации. При этом допущении логика причинности могла бы стать научной базой теории коммуникации. Для

аналитического изучения процессов коммуникации, по-видимому, можно воспользоваться формализмом *алгоритмических сетей* (В.В. Иванищев, 1986).

3.2. Имитационный эксперимент

Сущности гносеологического раздела являются моделью эксперимента или наблюдения, в частности моделью экспериментальной или измерительной установки. Предметная семантика пакетов гносеологического раздела определяется заданием процедур для проверки эквивалентности и отношения порядка (определением измерительных шкал). Пакет включает диаграммы деятельности, моделирующие процессы измерений. Большинство концептов гносеологического раздела образовано на базе понятий философии науки. Напомним некоторые из них.

Научное наблюдение – деятельность, предполагающая восприятие объекта и последующее документирование результатов наблюдения. В отличие от созерцания научное наблюдение предполагает замысел, цель и средства. Для документирования результатов наблюдений используются шкалы.

Классификационные шкалы (шкалы наименований, номинальные шкалы). Мы будем говорить, что объекты относятся к одному типу, если в процедуре валидации посылаются равные сообщения и возвращаемые сообщения также равны. В противном случае эксперимент приводит к вызыву исключительной ситуации. Таксономию мы будем рассматривать как коллекцию образцов объектов, сравнивая с которыми некоторый объект, мы сможем отнести его к тому или иному таксону. Более строго таксономию можно определить, если задать категорию на алгебраической структуре. Другой способ определения. Равенство двух объектов на каждом шаге одинаковых конструктивных процедур. Конструктивные процедуры равны, если равны соответствующие сообщения.

Сравнительные (порядковые) шкалы, включая оценочные и ранговые шкалы. Таксономия ничего не говорит о различии между объектами, кроме того, что они разные. Для того чтобы увидеть различие, необходимо задать сравнительные шкалы. Шкалы порядка позволяют не только разбивать объекты на классы, но и упорядочивать классы по возрастанию (убыванию) некоего признака.

Количественные шкалы (шкалы интервалов и отношений). В рамках одного типа можно ввести отношение порядка, сравнивая объекты по некоторой величине. Т.е. предполагается существование некоторой единицы измерения, позволяющей определять, насколько значение признака у одного объекта больше или меньше, чем у другого.

Для реализации операциональных процедур введем класс Magnitude (объекты, допускающие сравнение по величине) и его подклассы Character (256 ASCII) и Number (Число). Они позволят построить три типа шкал – классификационные, сравнительные (стратификационные) и количественные. Тем самым появляется возможность определить такие процедуры, как наблюдение, сравнение, счет, измерение и некоторые другие. Действие приборов может быть основано на законах природы, выходящих за рамки изучаемой предметной области. Поэтому классы гносеологических сущностей не обязаны входить в иерархию классов онтологических сущностей. Однако иерархия классов Magnitude должна быть доступна как гносеологическим, так и онтологическим объектам. Тем самым определяется понятие наблюдаемых объектов (так как в этом случае становится возможным обмен сообщениями между датчиками и изучаемой системой). В ограничениях (общий механизм UML) задается запрет использовать гносеологические сущности в онтологических за исключением потомков Magnitude.

Для *наблюдения* мы должны, как правило, осуществлять некоторые материальные операции, применять инструменты и так далее. Наиболее важные из них — это *сравнение, измерение* и *эксперимент*.

Процедура ***сравнения*** предполагает существование такого отношения, в котором сравниваемые предметы объективно выступают как качественно однородные и никакие другие свойства данных предметов не играют для указанного отношения никакой роли.

Эквивалентность. Для построения измерительных шкал мы будем использовать несколько разновидностей эквивалентности. Прежде всего, определим вычислительную семантику (аксиоматическая, операционная (как?) и денотационная (что?)) для основных понятий – тождества и равенства.

В операционной семантике рассматривается некоторая абстрактная машина, которая, в частности, имеет память (с прямой адресацией). Мы будем считать два объекта *тождественными*, если указатели ссылаются на одну и ту же область памяти. Это соответствует тому, что мы разными буквами обозначаем один и тот же математический объект (например, константу 1 можно обозначить как a и как b, тогда a≡b). Несколько сложнее определить равенство. Операционная семантика *равенства* – это побитное равенство двух разных непересекающихся участков памяти абстрактной машины. Можно также сказать так: копия объекта и копируемый объект находятся в отношении равенства.

Более строгое обоснование основано на процессных алгебрах. Пусть L – некоторый алфавит (s_0). Обозначим математический объект x некоторым подмножеством букв $X \subseteq L$, после чего определим $L' = L \cap X$ ($s \in S$). Этот процесс будем называть Refer_to_as = (S, s_0, R), т.е. *именованием*. Будем говорить, что для $\forall a,b \in X$, $a \equiv b$. Рассмотрим теперь вычислительный процесс Refer_to_as*, который определим следующим образом. Мы будем считать два объекта *тождественными*, если указатели ссылаются на одну и ту же область памяти. Нетрудно убедится, что Refer_to_as ~ Refer_to_as*. Также просто доказывается отношение бисимуляции для рефлективности, симметричности и транзитивности тождества. Единственно, что следует отметить, так это то, что вычислительная (операционная) семантика рефлективности означает запрет (ошибка трансляции) на повторное определение указателя с одним и тем же именем в абстрактной машине.

Измерение – процедура, фиксирующая не только качественные характеристики объектов и явлений, но и количественные аспекты. Оно предполагает наличие в средствах деятельности некоторого масштаба (единицы измерения), алгоритма (правил) процесса измерения и измерительного устройства. Способ измерения предполагает три главных момента: 1) выбор единицы измерения и определение производных мер; 2) установление правила сравнения измеряемой величины с мерой и правило сложения мер; 3) описание процедуры сравнения. Типичный пример – измерение длины. Численное значение измеряемой величины выражено отвлеченным числом, напротив, результат измерения всегда является наименованным числом.

При разработке имитационной модели обоснование валидности измерительной процедуры – это одна из серьезных проблем. Упомянем, например, проблему валидности для измерительной процедуры «Хаусдорфова размерность», рассмотренную в п. 1.3. Для решения этой задачи можно использовать те же формальные подходы, которые были рассмотрены в предыдущем разделе.

Эксперимент. Эксперимент можно представить как наблюдение, которому предшествуют некие предварительные действия – подготовка объекта изучения. Существуют два типа экспериментальных задач: 1) *исследовательский эксперимент,* который связан с поиском неизвестных зависимостей между несколькими параметрами объекта, и 2) *проверочный эксперимент,* который применяется в случаях, когда требуется подтвердить или опровергнуть те или иные следствия теории.

Приборы. Назначение прибора заключается в том, что бы взаимодействие прибора с объектом изучения привело к формированию макрообраза, воспринимаемого исследователем. Сам человек «физически, как орудие исследования, представляет собой макроскопический прибор».

В зависимости от того, как тот или иной прибор выполняет данную функцию, все они могут быть разделены на три типа: 1) преобразователи, 2) анализаторы, 3) усилители.

Приборы-преобразователи. Человек воспринимает только часть физического мира; этот тип приборов служит для взаимодействия исследователя с другой частью физической реальности, непосредственно не воспринимаемой органами чувств (электромагнитные волны, ультразвук). Важная особенность этих приборов – преобразование информации с одних носителей на другие. Частным случаем приборов такого типа являются приборы-индикаторы, функция которых – давать сведения о присутствии либо отсутствии искомого явления в исследуемой среде.

Приборам-преобразователям (детекторам) соответствуют фрагменты кода и процедуры-функции, которые генерируют первичную информацию, обрабатывая конкретную ситуацию в конкретном месте.

Приборы-анализаторы. Примерами этих приборов будут такие приборы как спектроскоп, химические, газовые анализаторы, анализаторы сигналов. Принцип действия этих приборов основан на том, чтобы оказать воздействие на объект изучения, и преобразовать его в новый объект или объекты.

Это наиболее сложная компонента пакета «*Research Instruments*». Чаше всего анализатор представлен фрагментами кода, распределенными по всем классам объектной модели. Для описания таких приборов можно использовать диаграммы развертывания. Такая распределенная система должна давать целостный образ программной симуляции. Одна из проблем, возникающих в этом случае – обеспечение контролируемого нарушения инкапсуляции для передачи данных наблюдений. Среди решений назовем следующие: введение фиктивных свойств и методов, обработка особых ситуаций, применение паттерна *MVC*, программные агенты, методы аспектного программирования. Программные агенты удобны тем, что требуют минимального вмешательства в классы модели, однако они отличаются высокой сложностью. Методы аспектного программирования не всегда могут быть реализованы.

В качестве примера приведем конструкцию прибора (программного агента) для примера п. 2.4, где рассматривается модель предприятия. Модель предприятия описывается паттерном *Composite*, и существует необходимость обойти эту древовидную структуру.

Для обхода структуры определим класс Pathfinder

```
class Pathfinder {
public:
      Pathfinder() {}
      void step(String p, StructuredUnit *t){
            ShowMessage("Шаг "+p); //<- наблюдение
            и, если необходимо, можно изучить t
      }  };  .
```

А в классе StructuredUnit определим метод investigate(Pathfinder *), который реализуем в классах WorkShop и Division. Соответсвенно, код процедур будет иметь вид:

```
void WorkShop::investigate(Pathfinder * t) { t->step(_name, this); }
```

и

```
void Division::investigate(Pathfinder * t){ t->step(_name, this);
for (int i=1; i<=compound_count; i++){_compound[i]->investigate(t);} }.
```

Напомним, что поле _name задано в классе StructuredUnit и определяет название структурной единицы, а массив _compound содержит составляющие подразделения.
Для наблюдения структуры Исследователю достаточно послать сообщение investigate_enterprise, процедура для которого имеет вид:

```
void Corporation::investigate_enterprise(){
Pathfinder *pathfinder = new Pathfinder;
_enterprise->investigate(pathfinder);
}   .
```

Приборы-регистраторы. Их основная функция – регистрация и хранение полезной информации в форме, допускающей последующее ее восприятие (в том числе с помощью приборов-усилителей), анализ, сравнение и измерение. Самый типичный пример – фоторегистрация на чувствительной эмульсии.

Приборы-усилители. Приборы данного типа применяются в тех случаях, когда идущие от объекта сигналы остаются в обычных условиях за порогом ощущений или когда особенности среды затрудняют их непосредственное отражение.

В имитационном моделировании это есть модули пакета «*Research Instruments*», которые выполняют операции по *визуализации* результатов наблюдений. В настоящее время разработаны многочисленные технологии визуализации данных, и мы не будем на них останавливаться. Однако стоит обратить внимание на следующую проблему.

Далеко не всегда понятно, где заканчивается объект моделирования, а где начинается визуализация. Например, формы многоклеточных животных во многих случаях есть результат визуализации, а сами многоклеточные организмы – двухмерные и даже одномерные существа. В математике такая процедура называется преобразованием к «правильной» системе координат.

Методы имитационного эксперимента. Как уже было сказано во введении к этому пункту, на наш взгляд, имитационный эксперимент должен быть направлен на построение *модели объяснения*. Поэтому методы и соответствующие методики должны быть увязаны с методами аналитических подходов. Примером может служить метод *model checking*, рассмотренный в предыдущем пункте.

На этапе разработки имитационной модели метод *model checking* сохраняет свою традиционную роль – как инструмент верификации. Однако на этапе эксплуатации его роль кардинально меняется, поскольку *model checking* можно рассматривать как метод исследования. Покажем, как это можно сделать для имитационных моделей активных систем.

Метод *model checking* применяется итерационно. Последовательность действий на каждой итерации может быть следующей.

1. Определяется (или модифицируется) множество атомарных высказываний AP и множество начальных состояний S_0. Исследуется множество состояний S, множество переходов R и определяется функция разметки L для программной симуляции посредством серии имитационных экспериментов. Генерируется структура Крипке $M = (S, S_0, R, L)$, после чего модель Крипке следует упростить. Цель этого упрощения – понижение размерности задачи.

2. В зависимости от задачи составляется (или уточняется) целостная система высказываний, адекватно отображающая теоретико-игровую ситуацию (позиционная, кооперативная, иерархическая, рефлексивная игра). Каждое высказывание записывается в виде формул CTL* и преобразуется затем к CTL или LTL.

3. Формулируется некоторая гипотеза о свойствах программной симуляции. Посредством стандартных процедур производятся вычисления на структуре Крипке. Результат вычислений представляется в форме таблицы истинности.

Итерации выполняются до тех пор, пока не будет получено объяснение результатов имитационных экспериментов. Проведение подобных манипуляций с имитационной моделью невозможно или ограничено, так как это может поставить под сомнение валидность модели.

Планирование эксперимента и метод Монте-Карло приведены в учебниках по имитационному моделированию [3], [11], [50]. Не потеряла актуальности и классическая работа [56].

Примеры и пояснения

*1. **Подбрасывания монеты***. Классический пример из теории вероятностей – подбрасывание монеты.

Контекст опишем классом Gambling {Concept = Мир азартных игр}; саму изучаемую систему зададим классом Tossup {Concept = Жеребьевка, toss – подбрасывать, толчок}, который имеет свойства top и bottom (тип Integer), а также метод

```
void Tossup::Exist()
{ int i, n = 0; Randomize(); n = Random(100);
for (i = 0; i < n; i++) {pCoin->Exist();}
if (pCoin->state) {top = 0; bottom = 1;}else {top = 1; bottom = 0;}
}  .
```

Заметим, что свойства top и bottom не существуют, пока экспериментатор не поймает монету рукой. Эту ситуацию неопределенности определим как top = -1 и bottom = -1.

Процедура метода моделирует процесс вращения монеты pCoin и событие выпадания одной из сторон монеты (пусть 0 – решка, а 1 – орел). Класс Coin «Ontology Atom» {Concept = Монета} имеет свойство state (тип boolean) и метод

```
void Coin::Exist()
{ if (state) {state = false;} else {state = true;}} ,
```

который описывает процесс вращения монеты вокруг оси.

Процесс измерения может быть проведен как в контексте задачи, так и в самой системе. В последнем случае в методе Gambling::Exist() нужно послать сообщение observe = pTossup->Observe(), а в систему Tossup необходимо встроить датчик (переменная k), который производит измерение свойств в результате выполнения процесса

```
int Tossup::Observe()
{ int k = top; ShowMessage( " Выпала сторона "+IntToStr(k)); return k;}.
```

С таким же успехом мы можем провести измерение и непосредственно в контексте, например, так:

```
observe = pTossup->top;   // установка состояния прибора по состоянию системы
ShowMessage( "Выпала сторона "+IntToStr(observe));
```

Приведем малоизвестный факт, относящейся к жизни Клода Элвуда Шеннона, создателя теории информации. Этот замечательный ученый обладал не только математическим даром, но и не менее умелыми руками. Прекратив заниматься математикой, он превратил свой дом в настоящий музей механических игрушек, сделанных своими руками. Большая часть из них была так или иначе связана с жонглированием. Он создал специальный подбрасыватель монет, который позволял устанавливать число оборотов монеты в воздухе, и, тем самым, на практике доказал, что движение монеты – процесс детерминированный.

Измерение – это модель того, как часть нашего мира взаимодействует с другой его частью. Если монетка квантовомеханическая, то процедура установки состояния прибора observe = pTossup->top уже становится неадекватной. Это связано с тем, что прибор – это макросистема, а объект pTossup – микросистема. Вопрос имитационного моделирования квантовомеханических систем остается открытым. Если рассмотреть противоположный

случай – подбрасывание монеты в мегамире, – то в этом случае наблюдение должно производиться из самой системы, а прибор должен быть врезан в процедуру Exist класса Tossup. Как и в случае квантовой механики, мы не знаем, какое описание будет адекватным. Возможно, придется признать, что для этих масштабов абсолютные темпомиры – это объективная физическая реальность. Тем не менее, поскольку *UML SP* – это универсальный язык моделирования, мы не можем обойти круг вопросов, связанных с квантовой механикой и теорией относительности.

2. ***Измерения в квантовой механике***. Как можно описать квантовые явления на *UML SP*? Рассмотрим некоторые гипотические модели, которые иллюстрируют то, как могут быть построены модели микромира.

Отличие квантовой механики от классической обычно определяют тремя аспектами: необъективностью, нелокальностью и индетерминизмом. Первая особенность определяется следующим образом: акт измерения не только изменяет состояние системы, но, что значительно важней, создает это состояние. Нелокальность можно определить так: квантовая система, каким то образом «чувствует» свою часть независимо от расстояния. Индетерминизм выражается в способе описания – вместо обычных вероятностей приходится использовать комплексные амплитуды.

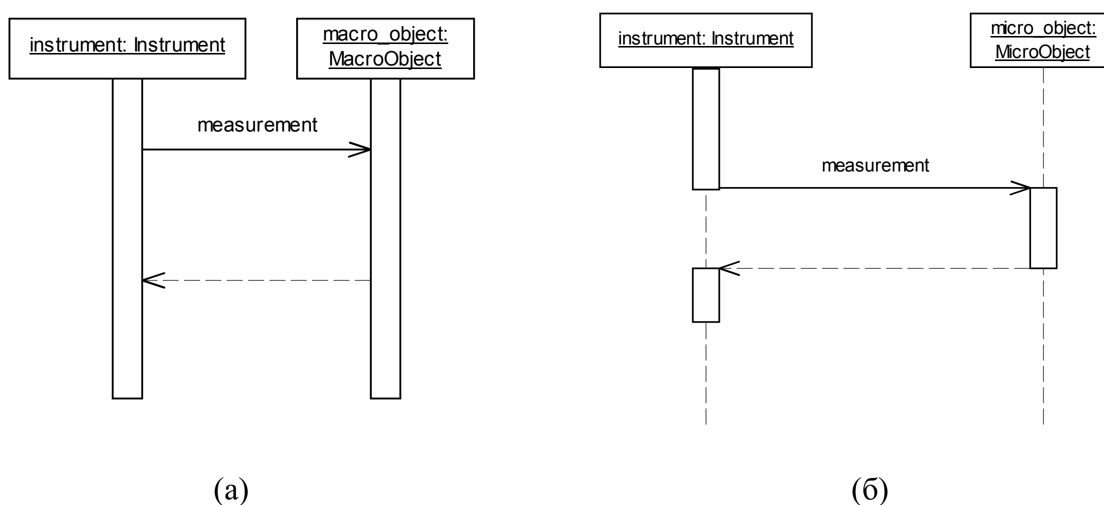


Рис. 54. Процедура измерения для (а) макросистемы и (б) микросистемы

Мы допустим, что макромир и микромир – это абсолютные темпомиры (см. рис. 54). С точки зрения экспериментатора, измерение в микросистеме порождает абсолютный темпомир, который есть событие и не может быть разложен на более тонкие действия. Измерение в макросистеме также рассматривается как абсолютный темпомир, но это только прием мышления; процессы остаются параллельными в момент измерения. При желании мы можем отследить порядок событий, происходящих в этот период (см. рис. 54а). Напротив, для микросистемы это сделать в принципе невозможно. Мы моделируем это тем, что прерываем линию жизни измерительного прибора (и вообще всех макрообъектов); см. рис. 54б.

Таким образом, квантовомеханическое описание – это более простой процесс; это последовательный вычислительный процесс, который моделируется простой передачей фокуса управления между процессами. Ясно, что в этом случае вопрос о временной структуре квантовой системы лишен смысла, как лишен смысла вопрос о структуре точки. Тем не менее, это не исключает возможность познания квантовых процессов, поскольку принцип наименьшего действия остается в силе.

Нелокальность квантовых явлений хорошо моделируется несколькими указателями на один и тот же объект. Именно так можно построить модель эксперимента к парадоксу Эйнштейна – Подольского – Розена (ЭПР). Отсюда, в частности, следует, что для макромира должны использоваться «умные» указатели, которые одному объекту сопоставляют один указатель.

Сущность эффекта ЭПР состоит в следующем. Измерение, проводимое над одной из частиц, автоматически изменяет состояние второй частицы, даже если они удалены друг от друга на очень большое расстояние. Тут мы имеем дело с квантовой нелокальностью, поскольку экспериментально доказано, что никаких агентов взаимодействия не существует.

Следуя Б.Б. Кадомцеву, рассмотрим мысленный эксперимент с шарами черного и белого цвета (Кадомцев Б.Б. Динамика и информация, 1997 г.). Пусть пространство моделируется динамическим списком с очень большим количеством ячеек. Создадим класс Instrument, который имеет поле _space для нашего списка, и два метода leftColor и rightColor. Методы возвращают цвет шара (класс Ball, свойство color), расположенного соответственно в крайне левой и крайне правой ячейке. Кроме того, определим метод prepare для приготовления начального состояния системы.

Рассмотрим сначала классическую систему. Метод prepare выполняет следующие действия. Шары сначала создаются – один черный и один белый, - затем взбалтываются (например, указатели переставляются случайным образом) и, не проводя измерения, указатели помещаются в крайние ячейки. Экспериментатор сразу определит цвет второго шара, как только узнает цвет первого. Это и есть классический аналог эффекта ЭПР. Цвет шара – это скрытый (неизвестный экспериментатору) параметр.

У квантовых систем никаких скрытых параметров нет. Мы предлагаем следующую имитационную модель для квантовых шаров. Пусть в начальный момент создается один шар; создадим два указателя и хорошо их перемешаем. После этого, также как и в классическом случае, разместим указатели в крайних ячейках. Квантовый шар не имеет цвета до момента измерения, так как он находится в «небытие» (см. рис. 54). В момент измерения (например, в левой ячейке) устанавливается цвет шара. Допустим, что цвет шара меняется на противоположный в момент обращения к свойству color. У шаров имеется квантовая корреляция: по предположению они находятся в запутанном состоянии, так что появление черного цвета у шара в левой ячейке мгновенно приводит к окрашиванию шара в правой ячейке в белый цвет. Возникает иллюзия того, что правый и левый шар взаимодействуют со сверхсветовой скоростью. Пусть вместо шаров мы проводим эксперимент с атомом водорода. Каким бы ни было расстояние, всякий раз мы будем иметь дело с одним и тем же объектом – атомом водорода.

Удобным инструментом имитационного моделирования квантовых явлений, возможно, может стать концепция *Мультиверса*, составляющая основу *многомировой интерпретации* (Many-worlds interpretation) квантовой механики, ввиду того, что она оперирует с дискретными понятиями. Эта идея принадлежит Хью Эверетту III, которую он изложил в своей докторской диссертации в 1957 г. Сам термин принадлежит Брайсу Девитту, внесшему существенный вклад в развитие этих идей в семидесятые годы прошлого века.

Заключение

В этой книге сделана попытка изложить имитационное моделирование с точки зрения алгоритмического подхода с учетом достижений программной инженерии последних лет. Рассмотрен профиль UML для имитационного моделирования, который разрешает использовать UML-средства для моделирования систем. Мы не думаем, что метамодель профиля надо рассматривать как нечто застывшее и окончательное. Были рассмотрены методология MSP – методология разработки имитационных моделей – и некоторые новые методы изучения программных симуляций. Отличительной чертой нашего подхода является то, что мы отделяем имитационную модель (*Research Analysis Model*) от способа реализации модели (*Research Design Model*). Это позволяет рассматривать вопросы адекватности модели независимо от таких вопросов, как методы вычислений, устойчивость и точность алгоритмов, техническая реализация.

Один мой знакомый математик в течение некоторого времени пытался искренне понять суть той работы, которая изложена в этой книге. Ему казалось невозможным, что модели могут быть описаны на каком-то другом языке, кроме языка математических уравнений. В лучшем случае он соглашался рассматривать эти модели как объектно-ориентированную версию численных методов. Надо думать, это не единичное мнение. Поэтому в заключение монографии хотелось бы коснуться этого вопроса.

Как уже было неоднократно сказано, мы опираемся на *коммуникационную парадигму* имитационного моделирования. В рамках UML SP эта парадигма необходима для обеспечения непротиворечивости языка как формальной системы. Тем не менее, представляется, что область применения данной парадигмы значительно шире. Любой объект реальности – физической или идеальной – можно рассмотреть как совокупность коммуникативных актов между компонентами объекта. Эта абстракция подобна математической абстракции – как совокупности отношений между элементами множества. Однако абстракция процесса коммуникации более содержательна и не может быть сведена к математической абстракции. Можно говорить о науке, которая изучает данные абстракции. В этом плане имитационное моделирование весьма близка к информатике, если понимать информатику как науку об информационном взаимодействии. Мы предположим, что фундаментальные законы природы следует формулировать на языке дискретности, языке коммуникативных актов. Тогда законы природы, определенные в традиционной форме, следует рассматривать как приближенное описание, некую асимптотику дискретных уравнений. Надежду на то, что эта программа реализуема, дает конструктивный подход в математике, который позволяет разрешить многочисленные парадоксы традиционной математики. Подтверждением высказанного утверждения может служить история развития объектно-ориентированного подхода в программировании. Формирование основных понятий и принципов ООАП – это процесс осознания, абстрагирования и определения объективно существующих в природе дискретных отношений.

«Оцифровка» физического мира и реальности в целом, сведение взаимодействия к коммуникативным актам, а законов природы – к законам процессов коммуникации – основная идея этой книги.

Библиографический список

1. Аладьев В.З. Дискретное моделирование в биологии развития // Математическая биология развития. – М.: «Наука», 1982. – С. 194-203.
2. Алюшин А.Л., Князева Е.Н. Темпомиры: Скорость восприятия и шкалы времени. – М.: Издательство ЛКИ, 2008. – 240 с.
3. Аристов С.А. Имитационное моделирование экономических систем: Учеб. пособие. – Екатеринбург: Изд-во Урал.гос.экон.ун-та. 2004.
4. Арлоу Д., Нейштадт И. UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование, 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2007. – 624 с.,
5. Борщев А.В. Практическое агентное моделирование и его место в арсенале аналитика // Exponenta Pro. – 2008. – № 3,4.
6. Бродский Ю.И. Модельный синтез и модельно-ориентированное программирование // ВЦ РАН, Москва, 2013. – 141 с.
7. Беркович С.Я. Клеточные автоматы как модель реальности: поиски новых представлений физических и информационных процессов // М.: Изд-во МГУ, 1993. – 112 с
8. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / перевод с англ. под редакцией И. Романовского и Ф. Андреева ВТОРОЕ ИЗДАНИЕ Rational Санта-Клара, Калифорния
9. Буч Г. Рамбо Д. Джекобсон А. Язык UML. Руководство пользователя. – ДМК Пресс, 2004. – 432 с.
10. Варфоломеев В.И. Алгоритмическое моделирование элементов экономических систем. - М.: Финансы и статистика, 2000. – 208 с.
11. Власов М.П., Шимко П.Д. Моделирование экономических процессов: Учеб. пособие. / М. П. Власов, П. Д. Шимко. – Ростов н/Д : Феникс, 2005. – с. 409
12. Вольфенгаген В.Э. Методы и средства вычислений с объектами. Аппликативные вычислительные системы. М.: JurInfoR Ltd., АО "ЦентрЮрИнфоР", 2004. – 789 с.
13. Вяльцев А. Н. Дискретное пространство-время. – М.: КомКнига, 2007. – 400 с.
14. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.
15. Гурьянов В.И. Специальный UML-профиль для моделирования сложных систем // Информационные технологии моделирования и управления. – Воронеж: Изд-во «Научная книга», 2010. – № 3(62). – С. 356-362. (см.. http://econf.rae.ru/article/5385)
16. Гурьянов В.И. Объектное моделирование фрактальных структур // Математические модели и их приложения: сб. науч. тр. Вып. 13. – Чебоксары: Изд-во Чуваш. ун-та, 2011.- С. 148-159
17. Гурьянов В.И. Обратная задача распараллеливания в имитационном моделировании // АВСИ-2012.. – Коломна: МГОСГИ, 2012. – С.187-191.
18. Гурьянов В.И., Семенова А.Н. Имитационная модель процессов самоорганизации логистических цепочек // Филиал СПбГИЭУ в г. Чебоксары, УЧЁНЫЕ ЗАПИСКИ - Выпуск 2 (9), - 2012. – С. 114-118.
19. Гурьянов В.И. Модели конструктивной физики в классической механике материальной точки // Математические модели и их приложения: сб. науч. тр. Вып. 15. – Чебоксары: Изд-во Чуваш. ун-та, 2013. – С. 148-159.
20. Гурьянов В.И. Дискретный аналог уравнения колебаний струны // Математика. Образование. : материалы 21-й Междунар. Конф. Чебоксары: Изд-во Чуваш. ун-та, 2013. – С. 356-357.
21. Гурьянов В.И. Моделирование классификаций на визуальном языке имитационного моделирования UML SP // Сборник докладов шестой всероссийской научно-практической конференции «Имитационное моделирование. Теория и практика» (ИММОД-2013). Том 1. // Издательство «ФЭН» Академии наук РТ, Казань, 2013, – С. 128-132.
22. Гутнер Г.Б. Онтология математического дискурса. Структура и сущность в математическом рассуждении. М.: Издательство Московского Культурологического Лицея, 1999.
23. Гуц А.К. Математическая логика и теория алгоритмов: Учебное пособие. Омск. Издательство Наследие. Диалог-Сибирь, 2003. – 108 с.

24. Давыдов А.А. Компьютационная теория социальных систем //Социологические исследования. 2005. – № 6. С. 14-24.
25. Дойч Д. Структура реальности. – Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001, – 400 стр.
26. Духанов, А. В. Имитационное моделирование сложных систем: курс лекций / А. В. Духанов, О. Н. Медведева; Владим. гос. ун-т. – Владимир: Изд-во Владим. гос. ун-та, 2010. – 115 с.
27. Емельянов А.А. и др. Имитационное моделирование экономических процессов: Учеб. пособие / А.А. Емельянов, Е.А. Власова, Р.В. Дума; Под ред. А.А. Емельянова. - М.: Финансы и статистика, 2002. -368 с:
28. Замятина Е.Б., Лядова Л.Н., Сухов А.О. Программные и языковые средства для создания адаптируемой к конкретной предметной области системы имитации // Сборник докладов шестой всероссийской научно-практической конференции «Имитационное моделирование. Теория и практика» (ИММОД-2013). Том 1. // Издательство «ФЭН» Академии наук РТ, Казань, 2013, – С.337-342.
29. Кашкин В.Б. Введение в теорию коммуникации: Учеб. пособие. – Воронеж: Изд-во ВГТУ, 2000. – 175 с.
30. Кларк Э. М., Грамберг О., Пелед Д. Верификация моделей программ: Model checking. – М: МЦНМО, 2002. - 416 с.
31. Колин К. К. Философия информации и структура реальности: концепция «четырех миров» // Знание. Понимание. Умение. 2013. № 2. С. 136–147.
32. Коплиен Дж. Программирование на C++. Классика CS. СПб.: Питер, 2005. – 479 с
33. Князева Е.Н., Курдюмов С.П. Основания синергетики. Режимы с обострением, самоорганизация, темпомиры. – СПб: Алетейя, 2002. – 414 с.
34. Лавров С., Программирование. Математические основы, средства, теория. - БХВ-Петербург, 2001. – 314 с.
35. Ларман К. Применение UML и шаблонов проектирования. 2-е издание.: Пер.с англ. - М.: Издательский дом «Вильямс», 2004. – 624 с.
36. Лебедев В.В. Математическое моделирование социально-экономических процессов, М.: Изограф, 1997. – 224 с.
37. Лефевр В.А. Конфликтующие структуры. Издание второе, переработанное и дополненное. — М.: Изд-во «Советское радио», 1973. — 158 с.
38. Минский М. Фреймы для представления знаний. - М., Энергия, 1979.
39. Миронов А.М., Теория процессов, М.: МГУ им.М.В.Ломоносова, 2009.- 344 с. (см. http://intsys.msu.ru/staff/mironov/processes.pdf)
40. Национальное Общество Имитационного Моделирования [Электронный ресурс] URL: http://simulation.su/ru.html
41. Нейлор Т. Машинные имитационные эксперименты с моделями экономических систем. - М.: Мир, 1975. - 392 с.
42. Новиков Д.А. Теория управления организационными системами. – М.: МПСИ, 2005. – 584 с.
43. Осипенко Г.С. «О символическом образе динамической системы», сб. Граничные задачи, Пермь, 1983
44. Ожигов Ю.И. Конструктивная физика.- НИЦ «Регулярная и хаотическая динамика», 2010. – 440 с.
45. Почепцов Г.Г. Теория коммуникации — М.: «Рефл-бук», К.: «Ваклер» — 2001. — 656 с.
46. Поппер К.Р. Знание и психофизическая проблема: В защиту взаимодействия. – М.: Изд-во ЛКИ, 2008. – 256 с.
47. Прангишвили И.В. Системный подход и общесистемные закономерности. Серия "Системы и проблемы управления". – М: СИНТЕГ, 2000. – 528 с.
48. Расторгуев С.П. Информационная война. — М: Радио и связь, 1999. — 416 С.
49. Розенталь И. Л. Геометрия, динамика, Вселенная. – М.: Наука, 1987. – 144 с.
50. Снетков Н.Н. Имитационное моделирование экономических процессов: Учебно-практическое пособие. – М.: Изд. центр ЕАОИ, 2008. – 228 с.
51. Соколов А. В. Общая теория социальной коммуникации: Учебное пособие. — СПб.: Изд-во Михайлова В. А., 2002 г. — 461 с.
52. Тарасов В. Б. От многоагентных систем к интеллектуальным организациям: философия, психология, информатика. – М.: Эдиториал УРСС, 2002. - 352 с.

53. Труб И.И. Объектно-ориентированное моделирование на C++: Учебный курс – Спб.: Питер, 2006. – 411 с.
54. Хоар Ч. Взаимодействующие последовательные процессы: Пер. с англ.– М.: Мир,1989. – 264 с.
55. Шамин Р.В. Современные численные методы в объектно-ориентированном изложении на C#, - 2011.
56. Шеннон Р. Имитационное моделирование систем. Искусство и наука. – М.: Мир, 1978. – 417 с.
57. Шикин Е. В., Чхартишвили А. Г. Математические методы и модели в управлении: Учеб. пособие. – 2-е изд.. испр. – М.: Дело. 2002. – 440 с.
58. Шредер М. Фракталы, хаос, степенные законы. Миниатюры из бесконечного рая. — Ижевск: «РХД», 2001. – 528 с.
59. Шрейдер Ю.А., Шаров А.А. Системы и модели. - М: Радио и связь, 1982 – 152 с. (Кибернетика)
60. Цехмистро И.З., Штанько В.И. и др. Концепция целостности / Харьков: Изд-во Харьковского гос. ун-та, 1987. – 223 с.
61. Форрестер Дж. Основы кибернетики предприятия (Индустриальная динамика) /перевод: Л.А. Балыков, Л.Е. Балясный, А.И. Госман, В.Ю. Невраев, Н.А. Палатников, В.Э. Рексин. – М.: Прогресс, 1970. – 340 с.
62. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. : Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 512 с.
63. Эшби, У. Р. Введение в кибернетику: пер. с англ. – М.: Иностранная литература, 1959.
64. Alpert S., Brown K., Woolf B. The Design Patterns Smalltalk Companion. 1st Edition. - Addison-Wesley Professional, 1998. – 464 p.
65. Checkland P.B., Scholes I. Soft Systems Methodology in Action. Chichester: Wiley, 1990.
66. Eriksson H., Penker M. Business Modeling with UML: Business Patterns at Work. – John Wiley & Sons, 2000. – 459 p.
67. Milner R. Calculus of Communicating Systems. Lecture Notes in Computer Science. v.91, 1980
68. Natalya F. Noy and Deborah L. McGuinness. «Ontology Development 101: A Guide to Creating Your First Ontology». Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.
http://protege.stanford.edu/publications/ontology_development/ontology101.html

Приложение I

Листинг 1

Unit1.h

```
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

#include "Unit2.h"
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
      TButton *Button1;
      TEdit *Edit1;
      TEdit *Edit2;
      TLabel *Label1;
      TLabel *Label2;
      TEdit *Edit3;
      TLabel *Label3;
      TButton *Button2;
      void __fastcall Button1Click(TObject *Sender);
      void __fastcall Button2Click(TObject *Sender);

private:
      Market * pWorld; // мир модели

public:           // User declarations
      __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Unit1.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"


//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
      : TForm(Owner)
{
}
```

```
//---------------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
// Приготовить начальное состояние

Edit1->Text = "1000";
pWorld = new Market;
int r =  StrToInt(Edit1->Text);
pWorld->Prepare(r);
}

//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
pWorld->Exist();  // квант существования
bool d = pWorld->observe; // <- наблюдение
if (d) {
Edit2->Text = "Согласен с ценой";
Edit3->Text = "Товар куплен";
}
else {
Edit2->Text = "Не согласен с ценой"; // <- анализ данных
Edit3->Text = "Товар не куплен";  // <- анализ данных
}

}
//---------------------------------------------------------------------------
```

Листинг 2

Unit2.h

```
#ifndef Unit2H
#define Unit2H
//---------------------------------------------------------------------------

#include "Unit3.h"
#include "Unit5.h"
//---------------------------------------------------------------------------
class Market: public Fabric
{
Buying * pBuying;   // система "Покупка"

public:
int observe;  // <- наблюдение

      Market()
      {
      pBuying = new Buying;
      }
      void Prepare(int);
      void Exist();

};
//---------------------------------------------------------------------------
#endif
```

Unit2.cpp

```
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Unit2.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)

void Market::Prepare(int val)
// Приготовить начальное состояние
{
pBuying->Prepare(val);
}

void Market::Exist()
// Существовать
{
pBuying->Exist();
observe = pBuying->deal; // <- измерение
}
```

Листинг 3

Unit3.h

```
#ifndef Unit3H
#define Unit3H
//---------------------------------------------------------------------------

#include "Unit4.h"
#include "Unit5.h"
//---------------------------------------------------------------------------
class Buying: public Fabric
// Класс для системы  Buying
{
      Agent * pSeller;
      Agent * pBuyer;
      char goods;

public:
bool deal; // свойство deal

      Buying()    {
      // Конструктор
      pSeller = new Agent;  // продавец
      pBuyer = new Agent;  // покупатель
      goods  = 'A'; // товар
}
      void Prepare(int); // приготовит начальное состояние
      void Exist(); // квант существования системы

};
//---------------------------------------------------------------------------
#endif
```

Unit3.cpp

```
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Unit3.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

void Buying::Prepare(int val)
// Приготовить начальное состояние
{
pSeller->Estimate(val); // оценить товар
}
//---------------------------------------------------------------------------
void Buying::Exist()
// Существовать
// Процесс покупки
{
int m = pSeller->Offer();  // продавец предлагает цену
pBuyer->Listen(m);  // покупатель слушает предложение
bool d = pBuyer->Declare_its_decision(); // покупатель дает ответ
if (d) { pBuyer->Buy(goods); // покупатель покупает товар
deal = true; }
else { deal = false; }
}
```

Листинг 4

Unit4.h

```
//---------------------------------------------------------------------------

#ifndef Unit4H
#define Unit4H
//---------------------------------------------------------------------------
#include "Unit5.h"
//---------------------------------------------------------------------------
class Agent: public Fabric
{
int price;   // цена
char goods; // товар

public:
      void Listen(int);
      void Buy(char);
      int Offer();
      bool Declare_its_decision();
      void Estimate(int);

};
//---------------------------------------------------------------------------
#endif
```

Unit4.cpp

```
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Unit4.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

void Agent::Listen(int val)
// Слушать
{
price = val;
}
int Agent::Offer()
// Предлагать
{
//price = 1000;
return price;
}
bool Agent::Declare_its_decision()
// Принять решение о покупке и объявить его
{
boolean b = true;
if (price>1500) {
b = false; ShowMessage("Не согласен с ценой"); //<- наблюдение
}
else {ShowMessage("Согласен с ценой"); //<- наблюдение
}
return b;}

void Agent::Buy(char val)
// Купить товар
{goods = val;}

void Agent::Estimate(int val)
// Оценить товар
{price = val;}
```

Листинг 5

Unit5.h

```
//---------------------------------------------------------------------------

#ifndef Unit5H
#define Unit5H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <StdCtrls.hpp>

#include <string.h>

class Fabric
// Субстанциональный класс
{
private:
      Fabric(const Fabric &f) {}
public:
      void Exist();

};

class Goods
// Класс для товара
{
public:
      AnsiString value;

};
class Message
// Класс для сообщений
{
public:
      AnsiString content;

};
//---------------------------------------------------------------------------
#endif
```

Unit5.cpp

```
//---------------------------------------------------------------------------
#pragma hdrstop
#include "Unit5.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

void Fabrics::Exist()
// Существовать
{ }
```

Приложение II

Параллельное программирование

Разработка имитационных моделей с использованием параллельного программирования сложна и связана со значительными затратами времени. Если применяется параллельное программирование, то только для того, чтобы повысить производительность вычислений. Тем не менее, в ряде случаев прямое моделирование параллельности следует считать оправданным. Например, тогда, когда необходимо разобраться в тонкостях процесса коммуникации. В этом приложении мы рассмотрим (естественно, кратко) параллельное программирование на примерах из раздела 2.1.2.

Язык программирования C++ непосредственно не поддерживает параллельное программирование. Однако большинство конкретных реализаций этого языка имеют соответствующие расширения, и все эти реализации имеют много общего. В частности, Borland C++ Builder предлагает ряд средств параллельного программирования, которые расположены в модуле SyncObjs (#include <SyncObjs.hpp>).

Для того чтобы создать поток, необходимо, чтобы классы были потомками класса TThread. Класс имеет особый метод void __fastcall Execute(), который задает процесс вычисления. Именно этот метод мы помечаем стереотипом «Exist». Метод Resume() делает поток активным, метод Suspend() – приостанавливает выполнение, а метод Terminate() завершает поток.

Для моделирования перемещения объекта по конвейеру создадим абстрактный класс Fabric – потомок класса TThread. Конструктор этого класса должен определяться как __fastcall Fabric::Fabric(bool CreateSuspended) : TThread(CreateSuspended) {}. Определим класс Shop со стереотипом «Environtment», потомок класса Fabric. И будем запускать поток из главного потока процедурой

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ w = new Shop(true);
w->Resume();
} ,
```

где true в конструкторе означает, что поток создается приостановленной (это может потребоваться в дальнейшем).

Зададим квант существования мира модели следующим кодом

```
void __fastcall Shop::Execute()
{ FreeOnTerminate = true; // освободить занятую потоком память по окончании его работы

pTransporter->Resume();
bool b = true;
while (b) { if(Terminated) break;
Sleep(5);
Synchronize(UpdateCaption); // вывести на главную форму
if (tick == 1000) {
 b = false; pTransporter->Terminate();
 };
tick++;
};
}  .
```

Условие tick == 1000 завершает поток. Поток также завершается, если будет вызван метод Terminate() из главного потока.

Перемещение объекта обработки по конвейеру осуществляет транспортер, который моделируется классом Transporter и помечается как «System».

```
class Transporter: public Fabric {
public:

 int tick;
 Counter *count;
 Cell *conveyer_belt;
 TCriticalSection *cs;
 CertainObject *co;

 __fastcall Transporter(bool);
 __fastcall ~Transporter();

 void __fastcall Execute();
 void __fastcall Transporter::UpdateCaption();// вывести на главную форму

};     .
```

В конструкторе создаем общие объекты

```
__fastcall Transporter::Transporter(bool v): Fabric(v)       {
        tick = 0;
        Cell *a = new Cell(true); Cell *b = new Cell(true);
        a->next = b; b->next = a;
        a->id = 1; b->id = 2;
        cs = new TCriticalSection ;
        a->cs = cs; b->cs = cs;
        count = new Counter;
        a->count = count; b->count = count;

        conveyer_belt = a;
}    .
```

Квант существования системы Transporter определим следующим образом:

```
void __fastcall Transporter::Execute() { FreeOnTerminate = true;

co = new CertainObject; co->characteristic = 0;
conveyer_belt->certainObject =  co;
count->count = 2; // счетчик активных ячеек
conveyer_belt->Resume(); conveyer_belt->next->Resume();

bool b = true;
while (b) { if(Terminated) break;
Sleep(10);

cs->Enter(); try {
if (count->count <= 0) {
 count->count = 2;
 Synchronize(UpdateCaption);
 conveyer_belt->Resume(); conveyer_belt->next->Resume();
 tick++;
};
} __finally {cs->Leave();};

};
}    .
```

Объект conveyer_belt представляет собой динамический список из нескольких (в примере – двух) объектов класса Cell, способных хранить объекты обработки co. В

данном методе организуется бесконечный цикл while, в котором в критической секции cs проверяется условие count = 0; это условие того, что все ячейки отработали свой квант существования. Если это так, то устанавливается count = 2 (две ячейки), после чего потоки ячеек перезапускаются.

Перемещение объекта обработки осуществляется в объектах класса Cell.

```
class Cell : public TThread {
protected:
 void __fastcall Execute();
public:
 int id, tick;
 Counter *count;
 CertainObject *certainObject;
 Cell *next;
 __fastcall Cell(bool CreateSuspended);
 void __fastcall Cell::UpdateCaption();
};   ,
```

где квант существования имеет вид:

```
void __fastcall Cell::Execute()
{ FreeOnTerminate = true;
do { Sleep(10);
 if (certainObject != NULL) {
       certainObject->characteristic++; // обработка объекта
       bool done = true;
       do { // ждать, пока отработает соседняя ячейка
       if (next->Suspended) {
             next->certainObject = this->certainObject;
             this->certainObject = NULL;
             done = false;
       }
       else { Sleep(10); done = true; };
       } while (done);

 };

 tick++;
 count->count = count->count - 1;
 this->Suspend();

} while (!Terminated);
}  .
```

Поток ждет, пока соседняя ячейка выполнит свой квант существования и приостановится (next->Suspended == true), после чего перемещает объект в эту соседнюю ячейку. Уменьшает счетчик активных ячеек count на единицу и приостанавливает свое выполнение.

В приведенном выше коде функция Sleep используется с целью избежать перегрузки процессора. Метод Synchronize посредством процедуры UpdateCaption позволяет передать данные из потока в главный поток. Их использование с точки зрения моделирования не принципиально.

Рандеву. Рассмотрим стандартный метод синхронизации двух потоков посредством двух разделяемых объектов класса TEvent на примере задачи *Прогулки Иммануила Канта* из п. 2.1.2. Поток Setting имеет следующий код:

```
void __fastcall Setting::Execute() {
FreeOnTerminate = true;

TWaitResult r;
```

```
TEvent *e1 = new TEvent(NULL,true,false, "e1", true);
TEvent *e2 = new TEvent(NULL,true,false, "e2", true);
Koenigsberg *k = new Koenigsberg (false); k->event1 = e1;  k->event2 = e2;

do {
      r = e1->WaitFor(INFINITE);
      if (r == wrSignaled) {
                  k-> season = illuminance(season); // освещенность
                  e1->ResetEvent();  e2->SetEvent();
} while (!Terminated);
}   .
```

Поток Koenigsberg имеет код:

```
void __fastcall Koenigsberg::Execute() {
FreeOnTerminate = true;
TWaitResult r;
do {
      // это физический процесс - вращение земли вокруг своей оси
      if (day) day = false; else day = true;
      if (tick == 2) { // полночь
                  tick = 0;
                  this->event1->SetEvent();
                  r = this->event2->WaitFor(INFINITE);
                  event2->ResetEvent();
      }
tick++;
} while (!Terminated);
}    .
```

Синхронизация осуществляется следующим образом. В начальный момент оба события e1 и e2 сброшены. Если поток Setting первым приходит в точку рандеву, то поток ждет установки события e1. Когда поток Koenigsberg приходит в точку рандеву, то он устанавливает событие e1 и ждет установки события e2. Поток Setting продолжает свое выполнение, сбрасывает событие e1 и устанавливает событие e2. Поток Koenigsberg продолжает свое выполнение и сбрасывает событие e2.

Если поток Koenigsberg первым приходит в точку рандеву, то он устанавливает событие e1 и поток Setting без ожидания сразу продолжает выполнения. Далее все происходит так же, как описано выше.

Приложение III

Стереотипы UML SP

Метамодель профиля. *UML SP* использует все элементы UML. Профиль содержит определения стереотипов (см. пример на рис.1). Все термины *UML SP* выделены наклонным шрифтом.

Важным аспектом применения профиля является возможность создания субпрофилей, конкретизирующих стереотипы. На рис.1. показано отношение обобщения между стереотипами Ontology System и Active System.

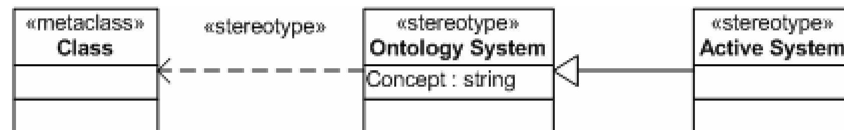


Рис.1. Пример определения стереотипов Ontology System и Active System

Accessibility Relation "metaClass" Dependency (из Core)

Предметная семантика. Отношение достижимости в модельной структуре Крипке.

Вычислительная семантика. Нарушение инкапсуляции пакетов «Word». *Помеченные значения.* Категория: Attribute; Имя: Concept; Тип: String; Документация: Название отношения достижимости. *Ограничения.* Отношение между пакетами в пакете «*Words*»

Categorization "metaClass" Dependence (из Core). *Предметная семантика*: функтор; сопоставляет таксону соответствующий архетип. *Вычислительная семантика*: зависимость, отражающая использование классов пакета «*Meronomy*» в классах пакета «*Taxonomy*». *Помеченные значения*: Concept = Наименование классификации; наименование функтора; принцип сопоставления. *Ограничения*: применяется только к пакетам «*Meronomy*» и «*Taxonomy*»; пакет «*Taxonomy*» зависит от пакета «*Meronomy*».

Epistemology Entity "metaClass" Package (из Model Management)

Предметная семантика. Методологии проведения измерений, выраженные в процедурах измерений и обработки данных. В частности, включает методы статистической обработки данных. *Вычислительная семантика.* Иерархия абстрактных классов, пригодных для повторного применения. *Помеченные значения.* Категория: Attribute; Имя: Concept; Тип: String; Документация: Название системы мер.

Ограничения. Пакет используется пакетом «*Research Instruments*»

Exist "metaClass" Operation (из Core)

Предметная семантика. Стереотип помечает те операции класса, которые определяют единицы дискретно-событийного времени имитационной модели. Этот стереотип выражает принцип, который может быть назван *принципом объектно-темпоральной декомпозиции*: на каждом шаге объектной декомпозиции необходимо определять деятельность, которая задает единицу дискретно-событийного времени для каждого из выделенных объектов.

Вычислительная семантика. Метод класса. *Помеченные значения*. Нет. *Ограничения.* Применим к операциям классов, входящих в пакеты *World* и *Ontology Entity*.

Make "metaClass" Operation (из Core)

Предметная семантика. Прагматика сообщения. *Вычислительная семантика.* Метод класса. Стереотип, применяемый ко всем операциям классов по умолчанию. Применим ко всем операциям, которые не помечены стереотипом *Exist*.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Имя прагматики сообщения. *Ограничения.* Применим к операциям классов, входящих в пакеты «*World*» и «*Ontology Entity*».

Measurement "metaClass" Dependency (из Core)

Предметная семантика. Взаимодействие исследовательской установки и объекта исследования. *Вычислительная семантика*. Контролируемое нарушение инкапсуляции классов пакета *«World» Помеченные значения.* Категория: Attribute; Имя: Concept; Тип: String; Документация: Название эксперимента или наблюдения.

Ограничения. Стереотип *Measurement* применяется к отношениям зависимости, связывающим пакеты *Research Instruments* и *World*. Пакет *Research Instruments* зависит от *World*.

Meronomy "metaClass" Package (из Model Management).

Предметная семантика. Мерономия, моделирует архетипы таксонов. Схема строения систем. *Вычислительная семантика*: классы, определяющие типы пользователя. *Помеченные значения*: Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование мерономии. *Ограничения*: содержит только классы «*Ontology Space*». Входит в «*Ontology Entity*».

Ontology Activity "metaClass" State (из Core, Activity – состояние-действие).

Предметная семантика. Описывает ситуацию, когда в каком-либо контексте некоторый процесс занимает пренебрежимо малое время и может рассматриваться как событие. С точки зрения наблюдателя контекста процесс представляет собой событие, не имеющее продолжительности. С точки зрения наблюдателя процесса, контекст представляет собой статическую картину, а изменение доступных переменных контекста запрещено.

Вычислительная семантика. В языках программирования активность моделируется процедурой; создание активности – это вызов процедуры из текущей процедуры

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Имя процесса. *Ограничения.* Только в пакетах «*World*» и «*Ontology Entity*».

Ontology Atom "metaClass" Class (из Core).

Предметная семантика. Конечный уровень декомпозиции изучаемой системы на подсистемы. *Вычислительная семантика.* Программные классы.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование атомарного объекта. *Ограничения.* Программным сущностям, определяющим строение классов с этим стереотипом, нельзя назначать концепты.

Ontology Category "metaClass" Class (из Core).

Предметная семантика. Таксон. Классы, помеченные стереотипом *Ontology Category*, определяются как абстрактные, и значит, они не могут иметь экземпляры; этот стереотип моделирует множество видов. *Вычислительная семантика*. Абстрактные классы. *Помеченные значения*. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование таксона. *Ограничения*: входят только в пакет «*Taxonomy*» .

Ontology Environment "metaClass" Class (из Core)

Предметная семантика. Определяет функцию изучаемой системы относительно окружающей среды. В простых случаях достаточно задать граничные и начальные условия.

Вычислительная семантика. Программный класс.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование контекста исследуемой системы.

Ограничения. Класс «*Ontology Environment*» находится в отношении агрегации с классом «*Ontology System*».

Ontology Entity "metaClass" Package (из Model Management)

Предметная семантика. Теория моделируемого объекта. Пакет, как правило, имеет сложную структуру. *Вычислительная семантика.* Иерархия абстрактных классов, пригодных для повторного использования в линейке имитационных моделей.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование классификационной системы, в которую входит объект исследования или наименование теории.

Ограничения. Пакет используется пакетом «*World*». Должен содержать, по меньшей мере, два класса помеченных «*Substance*» и «*Ontology Space*».

Ontology Space "metaClass" Class (из Core).

Предметная семантика. Мерон. Ячейка фазового пространства системы. Позволяет моделировать пространство состояний системы.

Вычислительная семантика. Иерархия классов, задающая пользовательские типы.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование мерона классификационной системы. *Ограничения.* Только для внутренних переменных классов «*Substance*» и классов пакета «*World*».

Ontology System "metaClass" Class (из Core).

Предметная семантика. Изучаемая система. *Вычислительная семантика.* Программный класс. Классы, помеченные стереотипами *Ontology Environment, Ontology System* и *Ontology Atom*, образуют тройку объектной декомпозиции и находятся в отношении «часть-целое». Декомпозиция на подсистемы может применяться рекурсивно.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование изучаемой системы. *Ограничения.* Класс «*Ontology System*» находится в отношении агрегации с классом «*Ontology Atom*».

Research Analysis Model "metaClass" *Model* (из Model Management)

Предметная семантика. Имитационная модель в широком смысле.

Вычислительная семантика. Описывает реализацию *прецедентов*, моделируя взаимодействие между компонентами имитационной модели.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Название имитационной модели.

Ограничения. Не может содержать элементы, входящие в *Research Design Model*.

Research Design Model "metaClass" *Model* (из Model Management) и ***Research Design Model Realization*** "metaClass" *Collaboration* (из *Collaboration*).

Семантика. Модель анализа не зависит от конкретного языка программирования. Модель дизайна определяет способ описания модели анализа для выбранного языка реализации. Композиция параллельных процессов модели анализа представлена в модели дизайна как квазипараллельный процесс. *Ограничения.* «*Research Design Model*» требует обязательной реализации взаимосвязи с «*Research Analysis Model*».

Research Instruments "metaClass" Package (из Model Management)

Предметная семантика. Описывает программные сущности, моделирующие средства наблюдения и измерений. *Вычислительная семантика.* Пакет для конкретных классов.

Помеченные значения. Категория**:** Attribute; Имя**:** Concept; Тип**:** String; Документация**:** Название измерительной системы, системы мониторинга или экспериментальной установки.

Ограничения. Не может содержать элементы, входящие в пакеты «*World*» *или* «*Epistemology Entity*».

Researcher "metaClass" Actor (из Use Case)

Семантика. Кто-то или что-то вне имитационной модели, взаимодействующий с имитационной моделью с целью исследования имитационной модели. Среди ролей *Researcher* наиболее важными являются роли *Наблюдатель контекста* и *Наблюдатель системы*. *Ограничения.* Может быть ассоциирован только с «*Research Use Case*»

Research Use Case "metaClass" *Use Case* (из Use Case)

Семантика. Каждый экземпляр прецедента описывает последовательность действий, связанных с изучением моделируемой системы.

Ограничения. Только «*Researcher*» может взаимодействовать с «*Research Use Case*»

Research Use Case Model "metaClass" Model (из Model Management)

Семантика. *Research Use Case Model* - это модель предполагаемых функций имитационной модели. Используется для определения части функциональных требований.

Помеченные значения. Нет

Ограничения. Модель должна содержать «*Researcher*» и «*Research Use Case*».

Research Use Case Realization "metaClass" *Collaboration* (из *Collaboration*)

Предметная семантика. Схема экспериментальных действий.

Вычислительная семантика. Определяет реализацию прецедентов «*Research Use Case*» с точки зрения программной системы.

Ограничения. Требует обязательной реализации взаимосвязи с «*Research Use Case*».

Substance "metaClass" Class (из Core).

Предметная семантика. Каркас модели. «Ткань», на которой «рисуется» система. *Вычислительная семантика.* Абстрактные классы архитектурного паттерна. Это может быть единственный класс, задающий общий интерфейс для конкретных классов.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Наименование каркаса системы. *Ограничения.* Только в пакете «*Ontology Entety*», но не в пакетах «*Taxonomy*» и «*Meronomy*».

Taxonomy "metaClass" Package (из Model Management).

Предметная семантика. Таксономия. *Вычислительная семантика*. Группировочная сущность для абстрактных классов, объявляющих интерфейс. *Помеченные значения*. Категория**:** Attribute; Имя**:** Concept; Тип**:** String;Документация: Название таксономии. *Ограничения*. Содержит классы «*Ontology Category*». Входит в «*Ontology Entity*».

Universe "metaClass" Package (из Model Management).

Предметная семантика. Все множество сущностей связанных с данным исследованием. *Вычислительная семантика*. Представление «*Research Analysis Model*».

Помеченные значения. Категория**:** Attribute; Имя**:** Concept; Тип**:** String; Название имитационной модели. *Ограничения.* Должен содержать элементы UML SP.

World "metaClass" Package (из Model Management).

Предметная семантика. Стереотип *World* (мир модели или *модельный мир*) отражает исследуемую систему и ее окружение и содержит описание имитационной модели в традиционном понимании (имитационная модель в узком смысле слова).

Вычислительная семантика. Конкретные классы данной имитационной модели.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Название модельного мира. *Ограничения.* Пакет должен содержать «*Ontology Environment*», «*Ontology System*», «*Ontology Atom*».

Worlds "metaClass" Package (из Model Management).

Предметная семантика. Пакет миров. Описывает несколько вариантов одной имитационной модели, между которыми нет обмена сообщениями.

Вычислительная семантика. Совокупность компьютерных программ.

Помеченные значения. Категория: Attribute; Имя: Concept; Тип: String; Документация: Название модельной структуры Крипке. *Ограничения.* Пакет должен содержать только пакеты «*World*», между которыми возможны только отношения «*Accessibility Relation*».

Научное издание

Гурьянов Василий Иванович

ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ НА UML SP

Монография

Редактор В.В. Степанов

Подписано в печать 03.02.2014. Формат 60х84/16. Бумага писчая. Печать оперативная. Усл. печ. л. 7,3. Учетно-изд. л. 3,82. Тираж 100. Заказ ____

Филиал федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный экономический университет» в г. Чебоксары
428034, г. Чебоксары, Ядринское шоссе, 3
www.cheb-engec.ru

Отпечатано в типографии «Новое время» ИП Сорокин А.В.
г. Чебоксары, ул. Мичмана Павлова, 50/1