2. Введение в объектно-ориентированный анализ и проектирование

2.1. OOP, OOD и OOA

Объектно-ориентированное программирование (object-oriented programming, OOP) - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

1) OOP использует в качестве базовых элементов объекты, а не алгоритмы;
2) каждый объект является экземпляром какого-либо определенного класса;
3) классы организованы иерархически.

Программа будет объектно-ориентированной только при условии соблюдения всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к OOP, а называется программированием на основе абстрактных типов данных.

В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

1) поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций, и собственные данные, с ограничением доступа к ним;

2) объекты относятся к соответствующим типам (классам);

3) классы могут наследовать атрибуты суперклассов.

Поддержка наследования в таких языках означает возможность установления отношения «is-a» («есть», «это есть», «это»), например, красная роза - это цветок, а цветок - это растение. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным, такие языки называют объектными. Согласно этому определению объектно-ориентированными языками являются Smalltalk, Object Pascal, C++ и CLOS, а Ada - объектный язык. Но поскольку объекты и классы являются элементами обеих групп языков, желательно использовать и в тех, и в других методы объектно-ориентированного проектирования.

Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование основное внимание уделяет правильному и эффективному структурированию сложных систем.

Объектно-ориентированное проектирование (object-oriented design, OOD) - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части:

1) объектно-ориентированное проектирование основывается на объектно-ориентированной декомпозиции;

2) объектно-ориентированное проектирование использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором - алгоритмами.

Объектно-ориентированный анализ направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ (object-oriented analysis, OOA) - это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Как соотносятся OOA, OOD и OOP? На результатах OOA формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии OOP.

2.2. Сложность, присущая программному обеспечению

2.2.1. Простые и сложные программные системы

Звезда в преддверии коллапса; ребенок, который учится читать; клетки крови, атакующие вирус, - это только некоторые из потрясающе сложных объектов физического мира. Компьютерные программы тоже бывают сложными, однако их сложность совершенно другого рода.

Конечно, не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Это не означает, что такие системы плохо сделаны или, что их создатели имеют низкую квалификацию. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заме-

нить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна.

Мы будем рассматривать процесс разработки сложных программных комплексов. Они применяются для решения самых разных задач, таких, например, как системы с обратной связью, которые управляют или сами управляются событиями физического мира, и для которых ресурсы времени и памяти ограничены; задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с обновлениями и запросами; системы управления и контроля за реальными процессами (например, диспетчеризация воздушного или железнодорожного транспорта). Системы подобного типа обычно имеют длительное время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. Среди промышленных программ также встречаются среды разработки, которые упрощают создание приложений в конкретных областях и программы, которые имитируют определенные стороны человеческого интеллекта.

Существенная черта промышленной программы - уровень сложности. Один разработчик практически не в состоянии охватить все аспекты такой системы. Сложность здесь неизбежна. С ней можно справиться, но избавиться от нее нельзя.

Конечно, среди разработчиков есть гении, которые в одиночку могут выполнить работу группы обычных людей и добиться в своей области успеха, сравнимого с достижениями Франка Ллойда Райта или Леонардо да Винчи. Такие люди нужны как архитекторы, которые изобретают новые идиомы, механизмы и основные идеи, используемые затем при разработке других систем. Однако, как замечает Петерс: «В мире очень мало гениев, и не надо думать, будто в среде программистов их доля выше средней» [3]. Несмотря на то, что все люди чуточку гениальны, в промышленном программировании не приходится полагаться только на гениальность. Поэтому нужно рассмотреть более надежные способы конструирования сложных систем. Но сначала ответим на вопрос: почему сложность присуща всем большим программным системам?

2.2.2. Почему программному обеспечению присуща сложность?

Брукс отмечает, что сложность программного обеспечения - отнюдь не случайное его свойство [3]. Сложность вызывается четырьмя основными причинами:

1) сложностью реальной предметной области, из которой исходит заказ на разработку;
2) трудностью управления процессом разработки;

3) необходимостью обеспечения достаточной гибкости программы;
4) неудовлетворительными способами описания поведения больших дискретных систем.

Подробнее остановимся на каждой из причин.

1. ***Сложность реального мира***. Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. Рассмотрим необходимые характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы и робота. Даже в общих чертах, достаточно трудно понять, как работает каждая такая система. Теперь прибавьте к этому дополнительные требования (часто не формулируемые явно), такие как удобство, производительность, стоимость, выживаемость и надежность. Сложность задачи и порождает ту сложность программного продукта, о которой пишет Брукс.

Эта внешняя сложность обычно возникает из-за «нестыковки» между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены на многих страницах текста, «разбавленных» немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов - схем, прототипов - и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Большая программная система - это крупное капиталовложение, и мы не можем позволить себе выкидывать сделанное при каждом изменении внешних требований. Большие системы должны эволюционировать в

процессе их использования. Встает задача о сопровождении программного обеспечения.

2. ***Трудности управления процессом разработки.*** Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса. Размер исходных текстов программной системы отнюдь не входит в число ее главных достоинств, поэтому нужно делать исходные тексты более компактными. Однако новые требования для каждой новой системы неизбежны, а они приводят к необходимости либо создавать много программ «с нуля», либо пытаться по-новому использовать существующие. Всего 20 лет назад программы объемом в несколько тысяч строк на ассемблере выходили за пределы наших возможностей. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложим ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

3. ***Гибкость программного обеспечения.*** Домостроительная компания обычно не имеет собственного лесхоза, который бы ей поставлял лес для пиломатериалов; совершенно необычно, чтобы монтажная фирма соорудила свой завод для изготовления стальных балок под будущее здание. Однако в программной индустрии такая практика - дело обычное. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она заставляет разработчика создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень трудоемким делом.

4. ***Проблема описания поведения больших дискретных систем.*** Когда мы кидаем вверх мяч, мы можем достоверно предсказать его траекторию, потому что знаем, что в нормальных условиях здесь действуют известные физические законы. Мы бы очень удивились, если бы, кинув мяч с

чуть большей скоростью, увидели, что он на середине пути неожиданно остановился и резко изменил направление движения. В недостаточно отлаженной программе моделирования полета мяча такая ситуация легко может возникнуть.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Д. Парнас [3] пишет: «когда мы говорим, что система описывается непрерывной функцией, мы имеем ввиду, что в ней нет скрытых сюрпризов. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных». С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовала на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пике. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, мы должны удовлетвориться разумным уровнем уверенности в их правильности.

Наряду с приведенными причинами сложности, работу со сложными системами усугубляют ограниченные физические возможности человека, которые связаны с объемом его краткосрочной памяти. Психологические

исследования показали, что количество структурных единиц, за которыми одновременно может следить человек, равно 7±2.

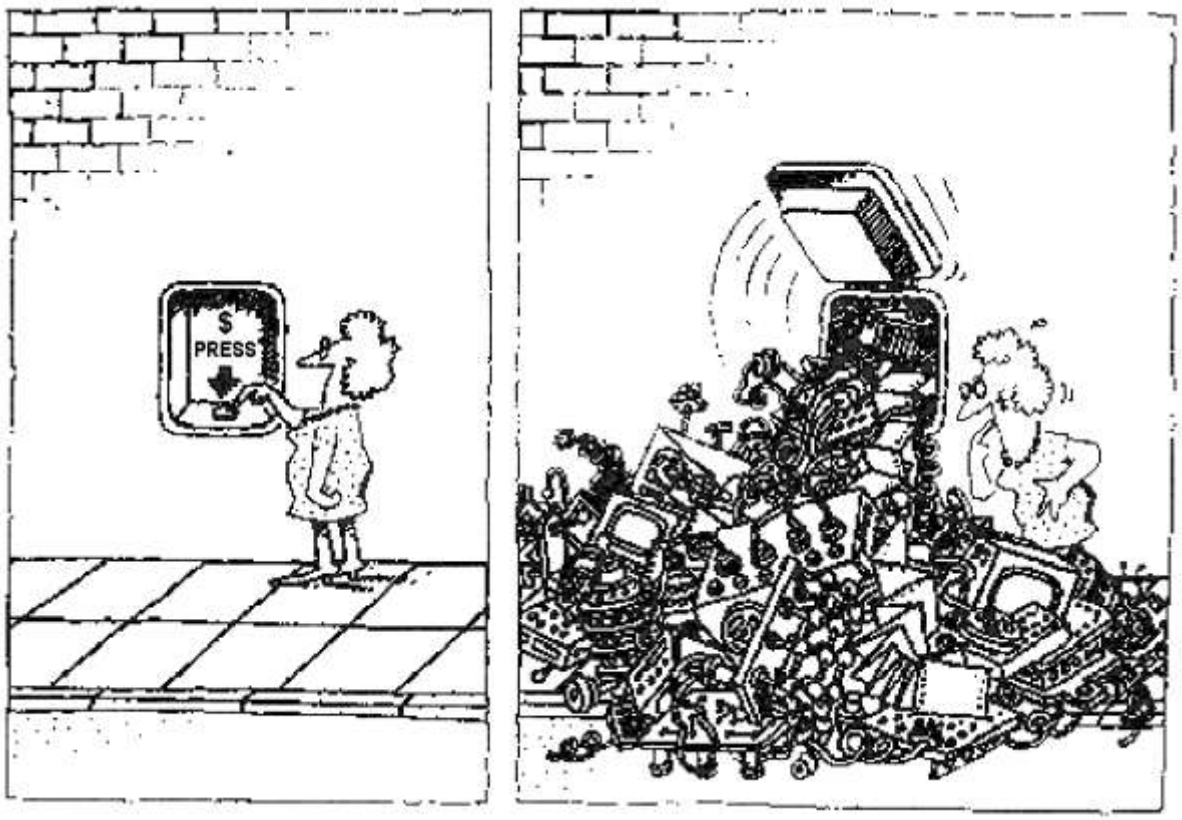

Рис. 2.1. Задача разработчиков программной системы – создать иллюзию простоты

Как же справляться со сложными системами? Можно заметить, что в других областях, например, архитектуре или управлении так же присутствуют сложные системы. Их можно встретить повсюду, и они успешно функционируют. Значит со сложностью можно справиться. Рассмотрим способы преодоления сложности.

2.2.3. Пять признаков сложной системы

Исходя из изучения структуры сложных систем, выводят пять общих признаков любой сложной системы.

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы. Это связано, скорее всего, с особенностью нашего восприятия - мы можем понять лишь те системы, которые имеют иерархическую структуру.

2. Выбор того, какие компоненты в данной системе считаются элементарными, относительно произволен и зависит от исследователя. Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого. Одна и та же роль элемента в системе может быть важной для одного и незначительной для другого исследователя.

3. Внутрикомпонентная связь в системе обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами. Это дает возможность изолированно изучать каждую часть.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных. Подобно тому, как в растении много разных клеток, но тип элементарного элемента один – клетка.

5. Любая работающая сложная система является результатом развития работавшей ранее более простой системы. Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматриваемые как сложные, делятся на элементарные, и из них строятся более сложные системы. Не возможно сразу правильно создать элементарные объекты. С ними нужно вначале поработать, чтобы больше узнать о реальном поведении системы, а затем уже совершенствовать.

2.2.4. Декомпозиция, абстракция и иерархия

Проблема ограниченности восприятия человеком сложных систем решается с помощью декомпозиции, абстракции и иерархии.

1. ***Иерархия. Каноническая форма сложной системы.*** В любой сложной системе можно выделить два типа иерархий. Первый тип – это ***иерархия «обобщение-специализация»*** (другое название «общее и частное» или «is-a»), в которой подкласс представляет собой специализированный частный случай своего суперкласса. Эта иерархия по-другому еще называется структура классов. Например, роза – это цветок, цветок – это растение. Второй тип иерархии – это ***иерархия «быть частью»*** (другое название «целое-часть» или «part-of»), когда про элемент говорят, что он часть другого. По-другому такая иерархия называется структура объектов. Например, лепестки - часть цветов.

Внесение иерархии позволяет четко классифицировать объекты и их свойства, а, значит, помогает справиться с присущей им сложностью.

Понятия структура классов и структура объектов, объединенные с пятью признаками сложных систем, называются ***канонической формой***

сложной системы (рис. 2.2). Те программные системы, в которых хорошо продумана каноническая форма, являются наиболее успешными.

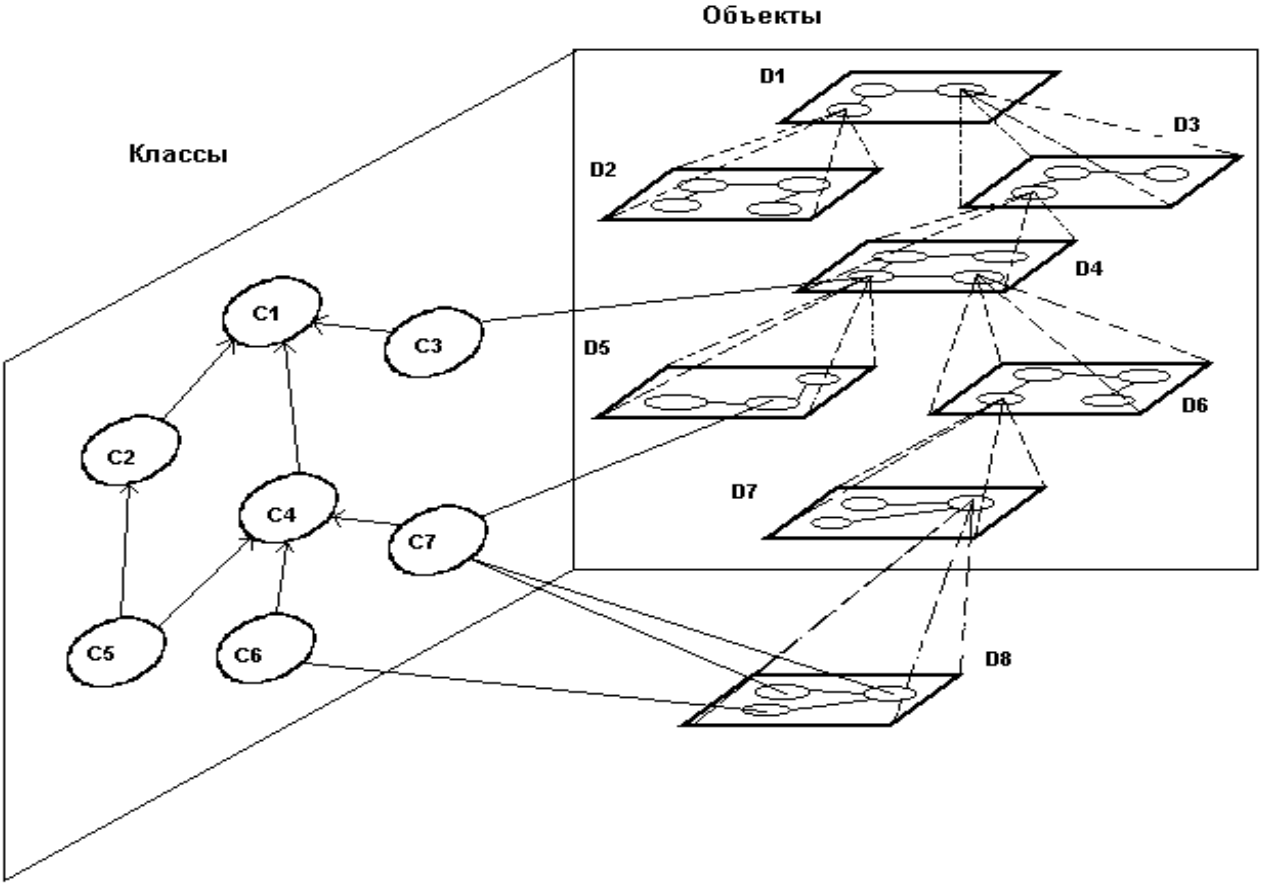

Рис. 2.2. Каноническая форма сложной системы

2. ***Декомпозиция.*** Способ управления сложными системами был известен давно, это принцип: «divide et impera» («разделяй и властвуй»). Программную систему можно разделить на подсистемы и совершенствовать независимо, при этом в уме не нужно держать информацию обо всей системе сразу.

Различают два типа декомпозиции - алгоритмическая и объектно-ориентированная. При ***алгоритмической декомпозиции*** основное внимание концентрируется на порядке происходящих событий (процессах). Примером алгоритмической декомпозиции может являться структурная схема, которая показывает связи между функциональными элементами системы (рис. 2.3). ***Объектно-ориентированная декомпозиция*** придает основное значение объектам, которые в процессе взаимодействия друг с другом определяют поведение системы. Объектно-ориентированная декомпозиция основана на объектах, а не на алгоритмах (рис. 2.4). Обе декомпозиции важны. Но начинать лучше с объектно-ориентированной.

3. ***Абстракция*** – это выделение существенных, с точки зрения наблюдателя, особенностей объекта и игнорирование несущественных.

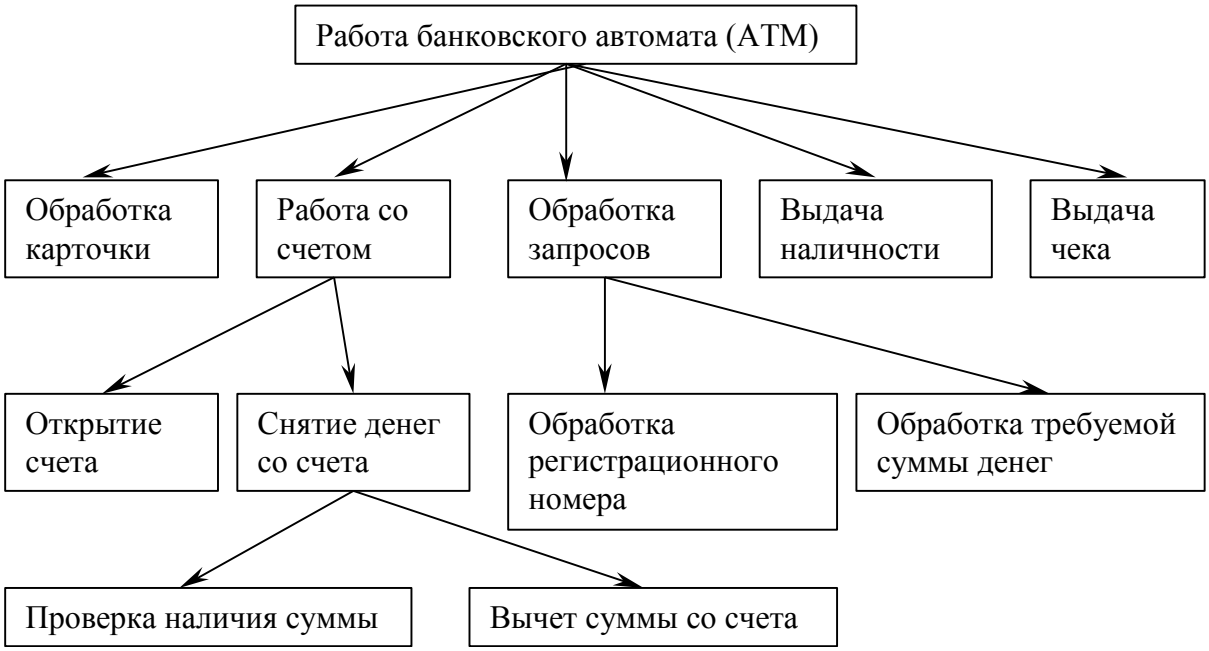

Рис. 2.3. Пример алгоритмической декомпозиции

Рис. 2.4. Пример объектно-ориентированной декомпозиции