Статья опубликована в журнале «Программирование». 2007. №5, с. 62–74.

УДК 681.3.06

Графическая нотация наследования автоматных классов

© Д. Г. Шопырин, А. А. Шалыто

Санкт-Петербургский государственный университет информационных технологий, механики и оптики
197101, Санкт-Петербург, Кронверкский пр. 49
E-mail: danil.shopyrin@gmail.com
Поступила в редакцию 29.05.2006

Объектно-автоматное программирование совмещает в себе основные преимущества объектно-ориентированной и автоматной технологий программирования. Его основными характеристиками являются гибкость, расширяемость и наличие мощного механизма описания сложного поведения, основанного на конечных автоматах. Недостатком объектно-автоматного программирования является отсутствие устоявшихся методов проектирования и реализации автоматных классов. В настоящей работе представлена графическая нотация для проектирования автоматных классов, совмещающая в себе возможности диаграмм классов объектно-ориентированного программирования и диаграмм поведения автоматного программирования. Предложенная графическая нотация позволяет обобщать, декомпозировать, структурировать и инкрементально расширять логику автоматных классов с помощью наследования.

Введение

Программные системы можно разделить на три класса: *преобразующие, интерактивные* и *реактивные* системы [1]. *Преобразующие* системы – это системы, завершающие свое выполнение после преобразования входных данных (например, архиватор, компилятор). *Интерактивные* системы – это системы, взаимодействующие с окружающей средой в режиме диалога (например, текстовый редактор). *Реактивные* системы – это системы, взаимодействующие с окружающей средой посредством обмена сообщениями в темпе, задаваемом средой. Реактивные системы обычно имеют следующие особенности:

время отклика *реактивной* системы задается ее окружением;
поведение *реактивных* систем детерминировано;
для *реактивных* систем характерен параллелизм;
сбои в работе *реактивных* систем крайне нежелательны.

Средств традиционного объектно-ориентированного программирования часто оказывается недостаточно для проектирования и реализации сложного поведения реактивных систем. С другой стороны, автоматное программирование предлагает мощный механизм описания и реализации сложного поведения на основе конечных автоматов. *Объектно-автоматное программирование*, являющееся синтезом объектно-ориентированной и автоматной технологий программирования, совмещает в себе их основные преимущества, такие как гибкость, расширяемость и наличие мощного механизма описания сложного поведения, основанного на конечных автоматах. Базовым понятием объектно-автоматного программирования является автоматный класс. *Автоматным классом* называется класс,

поведение объектов которого зависит от их текущего, явно выделенного, управляющего состояния и реализовано на основе конечных автоматов [2, 3].

Недостатком объектно-автоматного программирования является отсутствие устоявшихся методов проектирования и реализации автоматных классов. Наиболее распространенным подходом к проектированию и реализации автоматных классов является паттерн проектирования ***State*** [4]. В работах [5-8] описаны различные варианты развития паттерна проектирования ***State***.

В качестве средства графического проектирования поведения на основе конечных автоматов обычно используется графический язык ***Statecharts*** [9]. Многие современные подходы к проектированию реактивных систем основаны, в той или иной степени, на этом языке. Язык ***Statecharts*** является расширением традиционной модели конечных автоматов [10], в которую добавляются элементы, позволяющие описывать такие понятия, как *иерархия* и *параллелизм* [11, 12]. *Иерархия* вводится путем *вложенных состояний*, что семантически соответствует операции XOR (исключительное ИЛИ). *Параллелизм* вводится путем *ортогональных состояний*, что семантически соответствует операции AND (логическое И). Распространенными средствами проектирования систем на основе конечных автоматов являются также язык ***SDL*** [13] и язык синхронного программирования ***SyncCharts*** [14, 15].

Одним из недостатков вышеперечисленных графических языков проектирования поведения на основе автоматных классов является отсутствие средств описания объектно-ориентированной природы автоматных классов. Данный недостаток (частично) устраняется в объектно-ориентированном программировании с явным выделением состояний [16], также известном как ***SWITCH***-технология [17]. Графы переходов, используемые в ***SWITCH***-технологии, применяются совместно со схемами связей автоматов, подробно описывающими их интерфейс.

Еще одним недостатком вышеперечисленных языков является отсутствие средств описания отношения *наследования* автоматных классов. *Наследование* – это получение свойств или характеристик базового класса, обычно выполняемое с помощью введения некоторого отношения между базовым и производным классами [18]. Известны подходы, которые предоставляют возможность использования наследования при реализации автоматных классов [19, 20]. Однако вопрос использования наследования при визуальном проектировании автоматных классов до сих пор остается открытым.

В настоящей работе представлена графическая нотация для проектирования автоматных классов, совмещающая в себе возможности диаграмм классов объектно-ориентированного программирования и диаграмм поведения автоматного программирования. Предложенная графическая нотация позволяет обобщать, декомпозировать, структурировать и инкрементально расширять логику автоматных классов с помощью наследования.

1. Термины и определения

Перед тем, как перейти к описанию предлагаемой графической нотации, введем следующие термины и определения.

Автоматный класс A может быть определен тройкой $<I, S, J>$, где

- I–множество методов интерфейса автоматного класса;
- S–множество управляющих состояний автоматного класса;
- J–множество переходов между состояниями.

На множестве состояний S автоматного класса определена функция $beg(S) \in S$, возвращающая начальное состояние. Для каждого состояния $s \in S$ определены следующие функции:

- $den(s)$–действие при входе в состояние;
- $dex(s)$–действие при выходе из состояния;
- $dact(s)$–деятельность в состоянии.

Переход $j \in J$ может быть определен пятеркой $<from, to, ev, cond, do>$, где:

- $from(j) \in S$–начальное состояние перехода;
- $to(j) \in S$–конечное состояние перехода;
- $ev(j) \in I$ – причина перехода: вызов метода интерфейса, в случае которого *может* произойти переход;
- $cond(j) \in \{true, false\}$ – условие, выполнение которого необходимо для осуществления перехода;
- $do(j)$–действие, выполняемое во время перехода.

Для осуществления перехода j_0 необходимо выполнение следующих условий:

- текущим состоянием автоматного класса является состояние $from(j_0)$;
- вызван метод интерфейса автоматного класса $ev(j_0)$;
- выполнено условие $cond(j_0)$.

В этом случае выполняется следующая последовательность действий:

- выполняется действие $dex(from(j_0))$;
- выполняется действие $do(j_0)$;
- в качестве текущего состояния устанавливается состояние $to(j_0)$;
- выполняется действие $den(to(j_0))$.

После этого переход j_0 считается осуществленным.

1.1. Наследование автоматных классов

Наследование позволяет определять новые классы на основе уже существующих классов. При определении нового класса указываются только те свойства, которые отличают производный класс от базового класса. Остальные свойства добавляются в новый класс *неявно* и *автоматически* [21]. Формально наследование можно записать следующим образом [22, 23]:

$$R = P_1 \oplus P_2 \oplus \ldots \oplus P_n \oplus \Delta R,$$

где

- R–вновь определяемый класс;
- $P_1, P_2 \ldots P_n$–набор свойств, наследуемых от базовых классов;
- ΔR–инкрементально добавленные новые свойства, отличающие класс R;
- $\oplus$ – некоторая операция, которая позволяет скомбинировать свойства классов.

Рассмотрим механизм наследования автоматных классов. Все состояния и переходы базовых классов неявно попадают в производный класс. Производный класс может расширять и модифицировать поведение базовых классов. Модификация поведения базовых классов основана на перегрузке их состояний. Некоторые состояния базовых классов могут быть *перегружены* и переходы из таких перегруженных состояний могут быть модифицированы. В производный класс также могут быть добавлены новые состояния и переходы между ними.

Автоматный класс D является потомком автоматного класса B $(B \preceq D)$ в случае, если:

- $I_b \subseteq I_d$ – множество методов интерфейса I_b, реализуемого автоматным классом B, входит во множество методов интерфейса I_d, реализуемого автоматным классом D;
- $S_b \subseteq S_d$ – множество состояний S_b автоматного класса B входит во множество состояний S_d автоматного класса D;
- $beg(S_b) = beg(S_d)$ – начальные состояния классов B и D совпадают;
- для любого перехода j_b, входящего во множество переходов J_b автомата B, существует переход j_d, входящий во множество переходов J_d автомата D, такой, что
- $from(j_d) = from(j_b)$ – начальные состояния переходов j_d и j_b совпадают;
- $ev(j_d) = ev(j_b)$ – методы, являющиеся причиной переходов j_d и j_b, совпадают;
- $cond(j_d) = cond(j_b)$ – условия переходов j_d и j_b совпадают.

Переход j_d автоматного класса D *перегружает* переход j_b автоматного класса B, в случае когда:

- $from(j_d) = from(j_b)$ – начальные состояния переходов j_d и j_b совпадают;
- $ev(j_d) = ev(j_b)$ – методы, являющиеся причиной переходов j_d и j_b, совпадают;
- $cond(j_d) = cond(j_b)$ – условия переходов j_d и j_b совпадают;
- $to(j_d) \neq to(j_b)$ или $do(j_d) \neq do(j_b)$ – конечные состояния, или условия перехода, или действия на переходе *не совпадают*.

1.2. Декомпозиция и структурирование логики автоматных классов

Декомпозиция и структурирование логики автоматных классов осуществляется с помощью *групп состояний*. Группы состояний используются в языке ***Statecharts*** [9] и ***SWITCH***-технологии [24]. Группы состояний могут быть вложены друг в друга, образуя иерархию групп состояний.

Группы состояний могут иметь *групповые переходы*, также называемые *лучами*. Луч аналогичен *переходу*, за исключением того факта, что для луча не указывается начальное состояние. Луч $b \in B$ может быть определен четверкой $<to, ev, cond, do>$. Для каждого состояния $s \in S$ определена функция $beams(s)$, возвращающая множество лучей, соответствующих данному состоянию. Множество $beams(s)$ эквивалентно подмножеству переходов, имеющих s в качестве начального состояния:

$$beams(s) \equiv j \in J, from(j) = s$$

Группа $g \in G$ может быть определена тройкой $<gbeams, msub, gsub>$, где

- $gbeams(g) \subseteq B$ – множество лучей, соответствующих группе g;
- $msub(g) \subseteq S$–множество состояний, входящих в группу g;
- $gsub(g) \subset G$–множество групп, вложенных в группу g.

Для каждой группы $g \in G$ справедливы следующие утверждения:

- $\forall s \in msub(g), gbeams(g) \subseteq beams(s)$ – множество лучей состояния s, входящего в группу g, является надмножеством множества лучей, соответствующих группе g;
- $\forall g_0 \in gsub(g), gbeams(g) \subseteq gbeams(g_0)$ – множество лучей группы состояний g_0, входящей в группу g, является надмножеством множества лучей, соответствующих группе g;
- $\forall g_0 \in gsub(g), \forall s \in msub(g_0), s \in msub(g)$ – если состояние s вложено в группу g_0, вложенную, в свою очередь, в группу g, то он вложен также и в группу g;
- $\forall g_0, g_1 \in G$, *если* $g_0 \in gsub(g_1)$ *и* $g \in gsub(g_0)$, *то* $g \in gsub(g_1)$ – если группа g вложена в группу g_0, которая вложена в группу g_1, то группа g вложена также и в группу g_1.

2. Графическая нотация

При проектировании автоматных классов в настоящей работе предлагается использовать диаграммы поведения, являющиеся расширенной версией графов переходов, используемых в ***SWITCH***-технологии [3, 17]. Отличительной особенностью предлагаемых в данной работе диаграмм поведения является возможность описывать декомпозицию и структурирование логики автоматных классов с помощью наследования. Основные элементы графической нотации, используемой для построения диаграмм поведения автоматных классов, приведены на рис. 1.

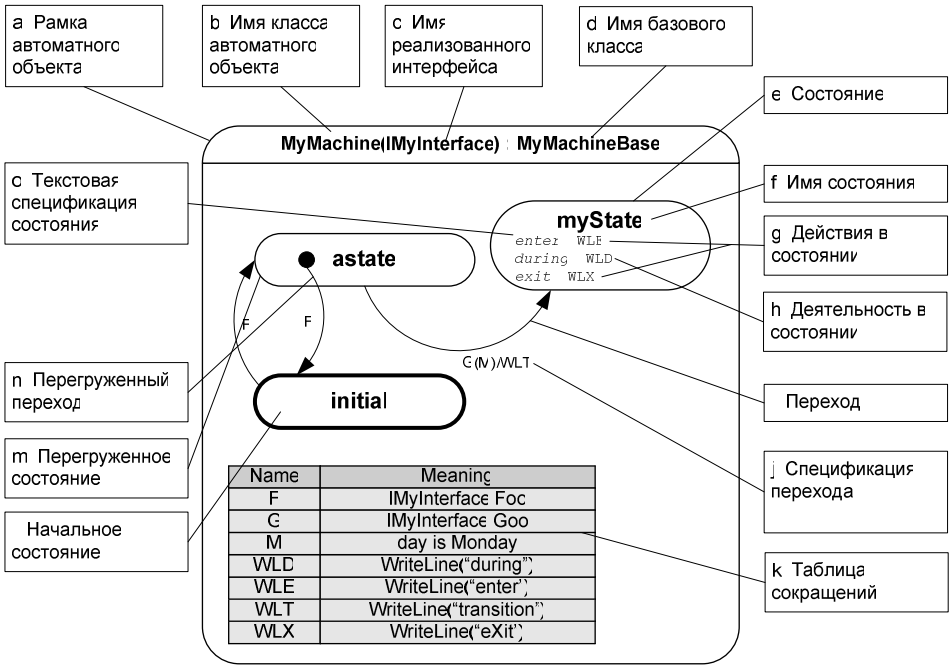


Рис. 1. Основные элементы предлагаемой графической нотации

Текстовая спецификация переходов записывается в виде
`E[(C)][/D]`,
где

- `E`–причина перехода, $ev(j_0)$;
- `C`–условие перехода, $cond(j_0)$;
- `D`–действие на переходе, $do(j_0)$.

Условие перехода и действие на переходе являются необязательными частями спецификации. Совокупность причины и условия перехода (если условие существует) называется *разрешительной* частью спецификации перехода. Допустимыми являются спецификации без условия, без действия и без условия и без действия одновременно.

Например, переход j на рис. 1 имеет спецификацию `G(M)/WLT`, где в соответствии с таблицей сокращений:

- `G` эквивалентно вызову метода `IMyInterface.Goo`;
- `M` эквивалентно условию `"day is Monday"`;
- `WLT` эквивалентно вызову метода `WriteLine("transition")`.

Поэтому переход в состояние `myState` осуществим только по понедельникам (`day is Monday`), в случае если вызван метод `Goo()` и автомат находится в состоянии `astate`. При этом перед сменой состояния в стандартный поток выводится строка `"transition"`.

2.1. Графическое представление наследования автоматов

Рассмотрим представление отношения наследования с помощью предлагаемой графической нотации. Базовый автоматный класс, в случаях, когда он существует, указывается в заголовке рамки автомата после двоеточия. Все состояния и переходы базового класса неявно переходят в производный автоматный класс.

Разрешается перегрузка состояний и переходов базового класса. Перегруженное состояние помечается жирной точкой. В случае множественного *n*-арного наследования изображается по одной пронумерованной *точке* для каждого из базовых классов, в порядке их упоминания. Точка с номером `i` соответствует базовому классу с порядковым номером `i`. Состояние, помеченное более чем одной точкой, является объединением и расширением одноименных состояний, присутствующих в нескольких базовых классах (рис. 2). Отметим, что в случае использования множественного наследования возможно возникновение разнообразных противоречий, присущих множественному наследованию. Подробный формальный анализ возможных противоречивых ситуаций выходит за рамки настоящей работы и является темой дальнейших исследований.

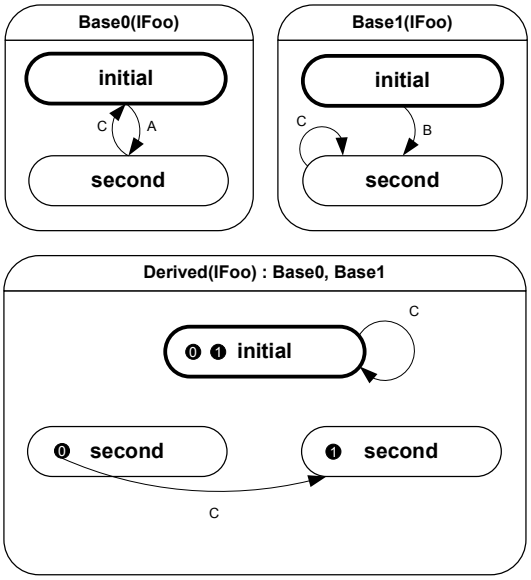


Рис. 2. Множественное наследование автоматных классов

Перегрузка перехода осуществляется по *разрешительной* части спецификации перехода. В результате перегрузки может быть изменено действие на переходе. Перегруженный переход берет свое начало в жирной точке, изображающей соответствующий базовый класс. Так, на рис. 2, класс `Derived` перегружает переход по причине `C` из состояния `second` базового класса `Base0` и в качестве конечного состояния указывает состояние `second` базового класса `Base1`.

Отметим, что для отображения отношения наследования автоматных классов также могут применяться диаграммы классов языка ***UML*** [25]. Диаграммы классов позволяют отобразить статический аспект наследования автоматных классов, однако они не предоставляют графического синтаксиса, позволяющего подробно отобразить динамический аспект наследования. Пример диаграммы классов приведен на **Рис. 3**.

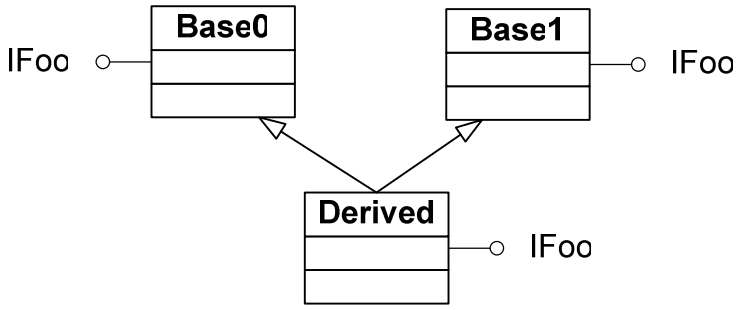
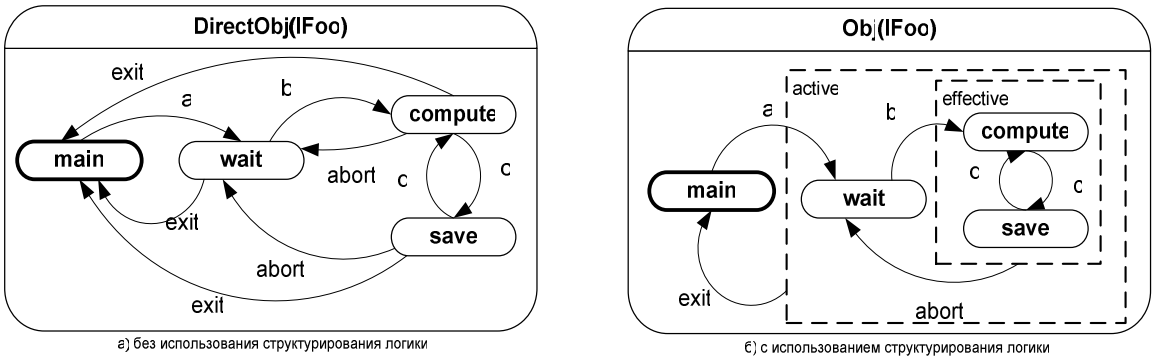


Рис. 3. Статическая диаграмма автоматных классов

2.2. Графическое представление структурирования логики автоматных классов

Рассмотрим диаграмму поведения автоматного класса `DirectObj`, изображенную на рис. 4, а. В классе `DirectObj` не используется структурирование, поэтому его диаграмма содержит дублирующиеся переходы.

Структурирование логики автоматных классов осуществляется с помощью *групп состояний*, которые изображаются в виде пунктирных прямоугольников (например, группы состояний `active` и `effective` на рис. 4, б). Группы состояний позволяют обобщать поведение, общее для нескольких состояний. В результате может быть сокращено дублирование переходов.



а) без использования структурирования логики

б) с использованием структурирования логики

Рис. 4. Пример использования структурирования логики автоматных классов

Группы состояний могут содержать групповые переходы. Групповой переход может быть совершен, когда автоматный класс находится в любом из состояний, входящих в данную группу.

Состояния, входящие в одну группу, являются потомками одного базового класса состояния. Группы состояний могут вкладываться друг в друга, образуя иерархии. Диаграмма классов состояний [26], соответствующая автоматному классу, изображенному на рис. 4, представлена на рис. 5.

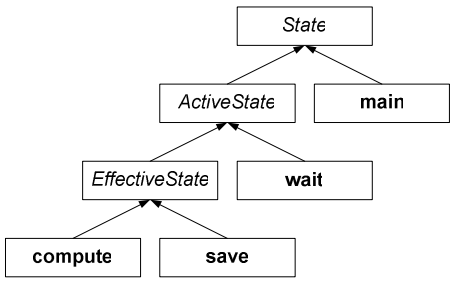


Рис. 5. Иерархия состояний

Структурирование логики может использоваться совместно с наследованием автоматных классов. Все группы состояний, определенные в базовом классе, неявно переходят в производный класс. Группы состояний базового класса, упоминаемые в производном классе, также как и состояния базового класса, помечаются жирной точкой. Производный автоматный класс может перегружать поведение в группах своего базового класса.

В качестве примера рассмотрим класс `DerivedObj`, который является потомком класса `Obj` (рис. 4, б). Диаграмма поведения класса `DerivedObj` изображена на рис. 6, а.

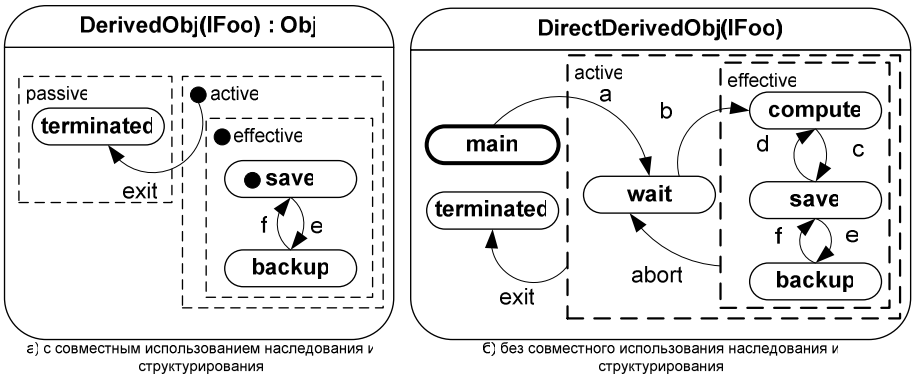


Рис. 6. Пример совместного использования наследования и структурирования логики автоматных классов

В автоматном классе `DerivedObj` упоминаются группы состояний `active` и `effective` базового класса `Obj`. Класс `DerivedObj` добавляет также новую группу состояний–`passive`. Кроме этого, автоматный класс `DerivedObj`:

- перегружает переход из группы `active` в состояние `main`, устанавливая в качестве конечного состояния состояние `terminated`, определяемое в классе `DerivedObj`;
- добавляет в группу состояний `effective` состояние `backup` и связывает его переходами с состоянием `save`.

Диаграмма поведения автоматного класса `DirectDerivedObj`, аналогичного классу `DerivedObj` (рис. 6, а), но построенного без совместного использования структурирования и наследования логики автоматных классов, изображена на рис. 6, б.

Структурирование логики автоматных классов с помощью группировки состояния обычно позволяет *значительно* сократить дублирование и повысить читаемость, как при проектировании автоматных классов, так и при их реализации.

3. Пример использования

Рассмотрим пример, иллюстрирующий применения предлагаемой графической нотации. Пусть существует семейство классов, предоставляющих доступ к файлу:

- доступ на чтение (автоматный класс `ReadFile`);
- доступ на запись (автоматный класс `WriteFile`);
- доступ на чтение, запись и чтение/запись (автоматный класс `ReadWriteFile`).

Рассматриваемые классы имеют автоматную природу (с состояниями « Закрыт », «Открыт на чтение» и т.д.). Диаграммы поведения этих автоматных классов приведены на рис. 7.

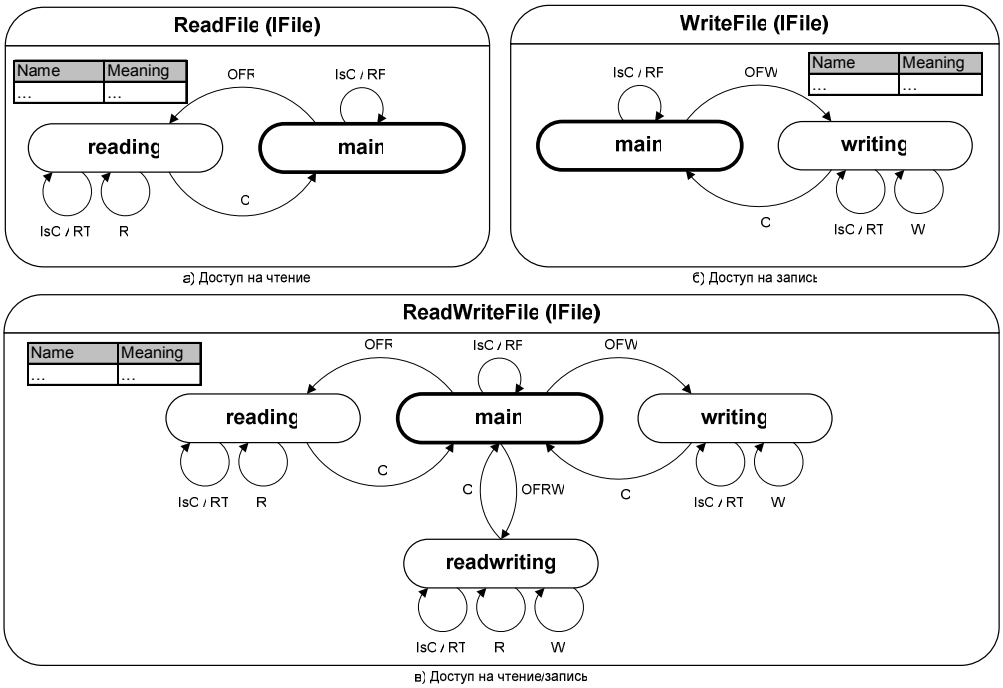


Рис. 7. Диаграммы поведения семейства классов доступа к файлу без использования наследования

Поведение описанных классов может быть обобщено (выделены одинаковые компоненты) и структурировано с помощью наследования. Эти классы образуют иерархию, показанную на рис. 8.

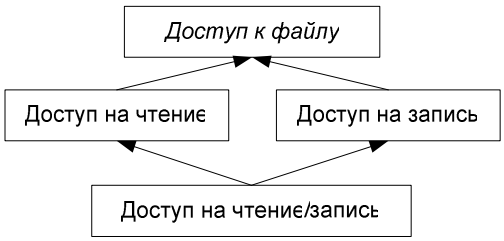


Рис. 8. Иерархия классов доступа к файлу

Корневым элементом предлагаемой иерархии является абстрактный класс, обобщающий некоторые аспекты доступа к файлу. Диаграмма поведения этих классов, построенная с использованием наследования, приведена на рис. 9.

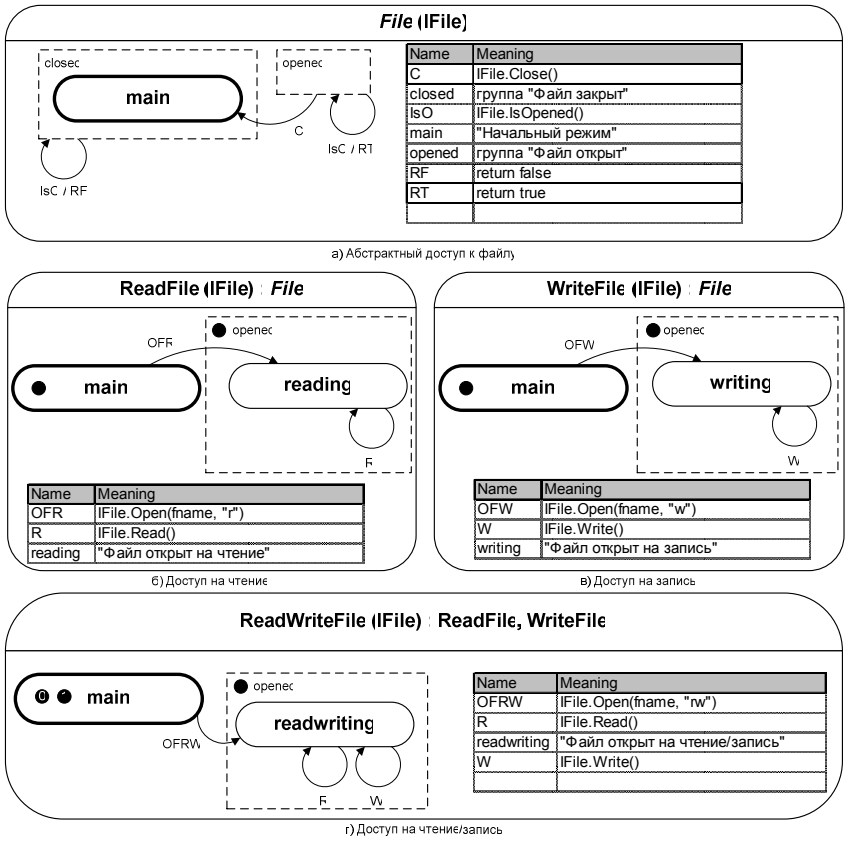


Рис. 9. Диаграммы поведения классов доступа к файлу с использованием наследования

Поведение любого из автоматных классов может быть расширено с помощью наследования. На рис. 10 приведена диаграмма поведения автоматного класса `AppendFile`, расширяющего логику автоматного класса `ReadWriteFile`, добавляя в него еще одно состояние – `appending`. Расширение происходит *инкрементально* – без изменения уже существующих автоматных классов.

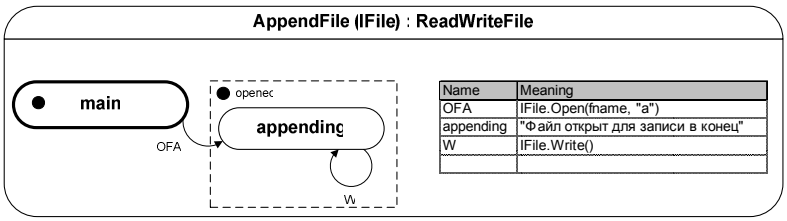


Рис. 10. Диаграмма поведения класса `AppendFile` с использованием наследования

Диаграмма поведения класса `AppendFile`, приведенная на рис. 10, эквивалентна диаграмме поведения класса `AppendFile` (рис. 11), которая построена без использования наследования. Отметим, что использование наследования позволило *значительно* сократить дублирование состояний и переходов.

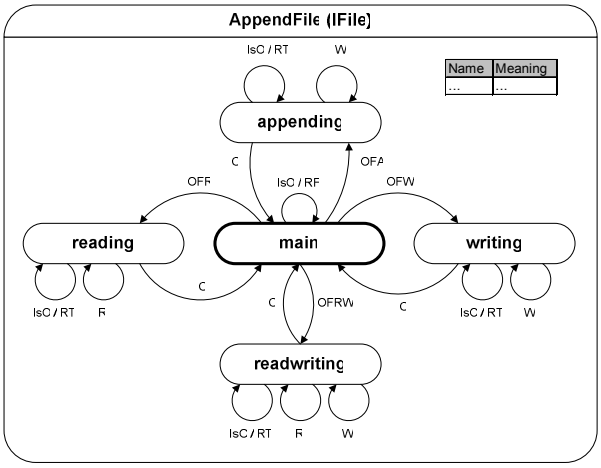


Рис. 11. Диаграмма поведения класса `AppendFile` без использования наследования

Произведем подсчет использованных состояний, групп состояний и переходов при проектировании поведения автоматных классов с использованием декомпозиции и структурирования их логики с помощью наследования (рис. 9, 10) и без него (рис. 7, 11). Результаты подсчета приведены в таблице.

Таблица. Сравнение способов проектирования

	Без использования наследования и структурирования логики	С использованием наследования и структурирования логики
Число состояний	13	5
Число перегруженных состояний	–	4
Число групп состояний	–	2
Число перегруженных групп состояний	–	4
Число переходов	42	12

Как следует из приведенной таблицы, предложенный в настоящей работе способ декомпозиции и структурирования логики автоматных классов может позволить значительно сократить количество используемых переходов, что происходит за счет устранения их дублирования. При этом в диаграмму добавляются новые сущности, такие как перегруженные состояния и группы состояний.

Заключение

В заключение отметим, что для описанной графической нотации предложено два метода реализации автоматных классов:

- на основе виртуальных методов [27];
- на основе виртуальных вложенных классов [28].

Оба предложенных метода полностью соответствуют основным принципам объектно-ориентированного программирования и позволяют изоморфно отобразить предложенную выше графическую нотацию при реализации автоматных классов.

Практическая ценность предложенной графической нотации и взаимосвязанных методов реализации подтверждается результатами внедрения в практику программирования в компании *Transas Technologies*.

Описанная графическая нотация позволяет обобщать, декомпозировать, структурировать и расширять логику автоматных классов с помощью наследования. Декомпозиция и структурирование логики автоматных классов может позволить значительно сократить дублирование, как при проектировании, так и при реализации систем, поведение которых описывается с помощью конечных автоматов.

Список литературы

1. **Harel D., Pnueli A.** On the development of reactive systems //In "Logic and Models of Concurrent Systems". NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985. pp. 477–498.
2. **Шалыто А. А., Туккель Н. И.** От тьюрингова программирования к автоматному //Мир ПК. 2002.№2, с. 144–149. http://is.ifmo.ru/works/turing/
3. **Шалыто А. А., Туккель Н. И.** SWITCH-технология - автоматный подход к созданию программного обеспечения «реактивных» систем. //Программирование. 2001, №5, с.45–62. http://is.ifmo.ru/works/switch/1/
4. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. 368 с.
5. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns // The 10th Conference on Pattern Languages of Programs, 2003.
6. **Шалыто А. А., Наумов Л. А.** Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов // Искусственный интеллект. 2004. № 4, с. 756–762. http://is.ifmo.ru/works/_aut_oop.pdf
7. **Adamczyk P.** Selected Patterns for Implementing Finite State Machines /The 11th Conference on Pattern Languages of Programs, 2004.
8. **Odrowski J., Sogaard P.** Pattern Integration - Variations of State /Proceedings of PLoP96. http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz.
9. **Harel D.** Statecharts: A visual formalism for complex systems //Sci. Comput. Program. 1987. Vol. 8, pp. 231–274.
10. **Automata Studies** / Shannon C.E., McCarthy J. Princeton University Press, 1956.
11. **Harel D., Naamad A.** The Statemate Semantics of Statecharts // ACM Trans. Softw. Eng. Methodology, 1996, vol. 5, pp. 293–333.
12. **Mikk E., Lakhnech Y., Petersohn C., Siegel M.** On Formal Semantics of Statecharts as Supported by STATEMATE // Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, 1997.
13. **Specification and Description Language** (SDL), International Engineering Consortium. http://www.iec.org/acrobat.asp?filecode=125
14. **Benveniste A.** The Synchronous Languages 12 Years Later. Proceedings of the IEEE, vol. 91, 2003,№1, pp. 64–83.
15. **André C.** SyncCharts: A Visual Representation of Reactive Behaviors /Tech. Rep. RR 95–52, I3S, Sophia-Antipolis, France, 1995.

16. **Шалыто А. А.**, **Туккель Н. И.** Танки и автоматы // BYTE/Россия. 2003. № 2, с. 69–73. http://is.ifmo.ru/works/tanks_new/

17. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 628 с.

18. **Danforth S., Tomlinson C.** Type theories and object-oriented programming //ACM Comput. Surv. 1988,№1, pp. 29–72.

19. **Sane A., Campbell R.** Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity /OOPSLA '95. http://choices.cs.uiuc.edu/sane/home.html.

20. **Lee J., Xue N., Kuei T.** A note on state modeling through inheritance // *SIGSOFT Softw. Eng. Notes.* 1998, №1.

21. **Taivalsaari A.** On the notion of inheritance //*ACM Comput. Surv.* 1996,№3, pp. 438–479.

22. **Bracha G., Cook W.** Mixin-based inheritance //In OOPSLA/ECOOP'90 Conference Proceedings, ACM SIGPLAN Not. 1990,№10, pp. 303–311.

23. **Wegner P., Zdonik S.** Inheritance as an incremental modification mechanism or what Like is and isn't like. //ECOOP '88 Conference Proceedings, Springer-Verlag, 1988, pp. 55–77.

24. **Шалыто А. А., Туккель Н. И.** Реализация автоматов при программировании событийных систем. //Программист. 2002. № 4, с. 74–80. http://is.ifmo.ru/works/evsys/

25. **Буч Г., Рамбо Дж., Джекобсон А.** UML. Руководство пользователя. М. ДМК 2000. 432 с.

26. **Заякин Е. А., Шалыто А. А.** Метод устранения повторных фрагментов кода при реализации конечных автоматов // Мир ПК (диск). 2005. № 8. http://is.ifmo.ru/projects/life_app/

27. **Шопырин Д. Г.** Объектно-ориентированная реализация конечных автоматов на основе виртуальных методов // Информационно-управляющие системы. 2005, № 3, с. 36–40. http://is.ifmo.ru/works/runewstate/

28. **Шопырин Д. Г.** Метод проектирования и реализации конечных автоматов на основе виртуальных вложенных классов //Информационные технологии моделирования и управления. 2005. №1(19), с. 87–97. http://is.ifmo.ru/works/ruvstate/