УДК 681.3

А.А. НИКОНЕНКО

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ В КОМПЬЮТЕРНОЙ ЛИНГВИСТИКЕ: FACTORY METHOD, PROTOTYPE, SINGLETON

Ключевые слова: *паттерны проектирования, компьютерная лингвистика, обработка естественноязыковых текстов.*

ВВЕДЕНИЕ

Настоящая статья продолжает исследование, посвященное использованию паттернов при разработке архитектуры лингвистических информационных систем. Статья [1] содержит базовую информацию о паттернах, которая необходима для понимания изложенного материала. Поскольку общее описание паттернов, их структура и специфика использования, сильные и слабые стороны были рассмотрены ранее, в данной работе будут описаны только три порождающих паттерна: Factory Method, Prototype, Singleton, которые замыкают рассмотрение порождающих паттернов, начатое в [1]. В каталоге «банды четырех» [2], содержащем пять порождающих паттернов, можно найти общее описание данной категории паттернов и типы задач, в которых рекомендовано их применение.

Перейдем непосредственно к рассмотрению паттернов (в тексте статьи будут использованы некоторые схемы и цитаты из работы [2]).

ПАТТЕРН FACTORY METHOD (ФАБРИЧНЫЙ МЕТОД)

Назначение. Паттерн определяет интерфейс для создания объекта, при этом оставляет подклассам выбор решения относительно порождаемого класса. Фабричный метод позволяет классу делегировать подклассам порождение конкретных классов.

Суть паттерна достаточно проста — создается абстрактный класс (Creator), который содержит метод (Factory Method), порождающий объекты других классов (продукты). Все эти продукты являются наследниками абстрактного класса Product. Далее каждая из реализаций (ConcreteCreator) класса Creator замещает Factory Method так, чтобы он порождал специфичные продукты (ConcreteProduct). Абстрактный класс Creator может содержать код, реализующий фабричный метод по умолчанию.

Такой подход позволяет клиенту создавать необходимый ConcreteCreator, а конкретные классы правильного вида он произведет автоматически. Схематически паттерн изображен на рис. 1.

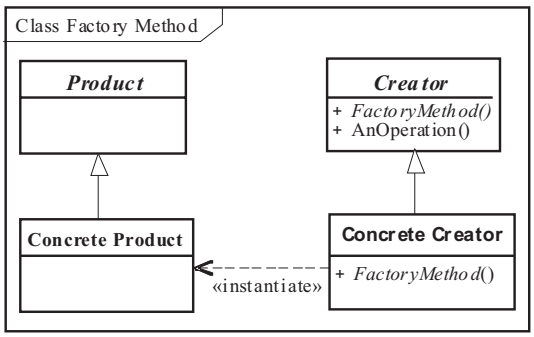

Рис. 1. Структура паттерна Factory Method

Описание функциональности паттерна. Как следует из названия, Factory Method весьма близок к паттерну Abstract Factory, который в отличие от Factory Method порождает семейства взаимосвязанных продуктов, а Factory Method порождает только один продукт. Другой особенностью паттерна является наличие в нем дополнительных операций (на рис. 1 это метод Creator->AnOperation(), однако таких методов может быть несколько). Считается, что в данных методах содержится общее поведение для обработки продуктов любого вида, а в подклассах замещается только фабричный метод.

Возникает вопрос, если паттерн Factory Method — это урезанный вариант Abstract Factory, который порождает только один продукт и дополнительно содержит набор методов, которые не замещаются в подклассах, то с какой целью это решение вынесено в отдельный шаблон? Ответ: несмотря на близкую структуру, эти паттерны решают разные архитектурные задачи.

Наибольшее сходство будут иметь паттерны, в которых один класс содержит несколько фабричных методов. При этом в отличие от Абстрактной Фабрики в данном случае нет ограничения, указывающего на необходимость замещения всех фабричных методов в подклассе. Иными словами, если Abstract Factory имеет два метода: CreateProductA и CreateProductB, то один подкласс фабрики (Factory1) будет порождать ProductA1 и ProductB1, а другой (Factory2) будет порождать ProductA2 и ProductB2, поскольку фабрика всегда порождает семейства взаимосвязанных продуктов.

Если вместо Abstract Factory используется паттерн Factory Method, то, поместив в класс Creator два фабричных метода: CreateProductA и CreateProductB, становится возможным порождение и таких подклассов Creator, в которых будет замещен только один из методов создания продукта (при условии, что Creator содержит поведение по умолчанию). Также возможно создать такой подкласс Creator, который будет порождать ProductA1 и ProductB2.

Из сказанного следует, что Factory Method — это один из методов реализации паттерна Abstract Factory.

Наиболее оправдано применение Factory Method в тех случаях, когда класс содержит сложное поведение, которое должно быть изменено в подклассах. Чтобы не замещать весь код, реализующий это поведение (при этом не всегда возможно получить исходный код того или иного класса), изменяемая часть кода выносится в отдельный класс, который является классом-продуктом. Сложное поведение, которое будет изменяться в подклассах, находится в методе AnOperation класса Creator.

Метод AnOperation использует объект класса ConcreteProduct и обращается к нему в соответствии с интерфейсом класса Product. Иными словами, теперь варьируемая часть сложного поведения из Creator->AnOperation вынесена в класс Product. Осталось решить последний вопрос: каким образом, не замещая метод AnOperation в потомках класса Creator, создавать необходимые подклассы класса Product? Ответ: с использованием паттерна Factory Method. Метод AnOperation спроектирован так, что сам он не создает объектов класса Product, а вызывает специальную операцию FactoryMethod, которая и порождает экземпляр класса Product. Чтобы ConcreteCreator в методе AnOperation использовал экземпляр класса ConcreteProduct2 вместо экземпляра класса ConcreteProduct1, достаточно произвести замещение ConcreteCreator->FactoryMethod так, чтобы он возвращал экземпляр класса ConcreteProduct2.

Таким образом, благодаря паттерну Factory Method проектировщик получает возможность изменять сложное поведение класса без замещения всего кода, формирующего это поведение.

Рассмотрим ситуацию, когда в одном классе существует несколько методов со сложным поведением, которое нужно конфигурировать. Один из методов решения — создание в этом классе столько фабричных методов, сколько существует методов с поведением. Иными словами, для класса Creator, содержащего три метода с поведением: AnOperation1, AnOperation2, AnOperation3, потребуется создание трех фабричных методов: FactoryMethod1, FactoryMethod2, FactoryMethod3.

Для уменьшения числа фабричных методов можно применить параметризованный фабричный метод. В этом случае параметр указывает на вид создаваемого продукта. В таком дизайне будет только один фабричный метод — FactoryMethod, а принимаемый им параметр указывает, для какой операции порождается продукт. Такой подход налагает определенную специфику на замещение метода FactoryMethod в подклассах. Изменения касаются ситуации, когда в одном из подклассов необходимо заменить только продукты, порождаемые, например, для AnOperation1, и оставить без изменения продукты, порождаемые для других операций. Поскольку существует только возможность замещения метода целиком, то, внеся изменения в FactoryMethod, касающиеся порождения новых продуктов для AnOperation1, теряется старое поведение, отвечающее за порождение продуктов для AnOperation2 и AnOperation3.

Целесообразный выход из данной ситуации — создание ветвления. Условием для ветвления выступает значение параметра, получаемого фабричным методом. Если параметр указывает на потребность в порождении объекта для AnOperation1, то выполняется новый код, в противном случае происходит вызов метода родительского класса FactoryMethod.

Использование паттерна в компьютерной лингвистике. Как было рассмотрено выше, паттерн используется тогда, когда разные подклассы одного родительского класса требуют выполнения над объектом одного и того же сложного действия разными способами. Типичным объектом в лингвистических приложениях является текст. Поэтому в качестве примера рассмотрим архитектуру системы для извлечения данных из текста (рис. 2).

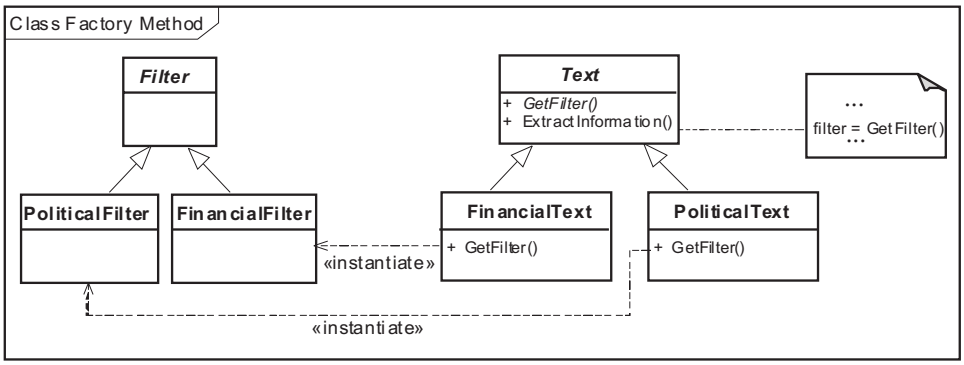

Рис. 2. Пример применения паттерна Factory Method в системе извлечения данных из текста

Предназначение данной системы — извлечение из текстов значимой информации. Для текстов финансовой тематики значимой информацией являются размеры капиталовложений, котировки акций, изменение процентных ставок, кросскурсы и т.д. Для текстов политической тематики значимая информация — действия политических лидеров, даты и места встреч, заявления, договоры и т.д.

Архитектура системы подразумевает, что существует единый класс Text, содержащий метод извлечения информации по умолчанию (ExtractInformation). Тексты конкретных тематик будут представлены в системе подклассами класса Text — FinancialText и PoliticalText. Для извлечения информации из текстов недостаточно стандартного метода ExtractInformation. Его необходимо модифицировать так, чтобы при выделении значимой информации учитывалась специфика текста.

Предположим, что алгоритм разбора текста для выделения из него информации будет стандартным для текстов любой тематики. В зависимости от тематики текста изменяться будут лишь методы фильтрации информации. Поэтому создадим специальный класс Filter, отвечающий за выделение значимой информации. В его подклассах FinancialFilter и PoliticalFilter будет осуществляться выбор из текста финансовой и политической информации.

Алгоритм разбора текста, находящийся в Text->ExtractInformation, подразумевает использование описанных выше фильтров. Однако он не создает объекты класса Filter самостоятельно, а использует для этих целей фабричный метод GetFilter. В подклассах FinancialText и PoliticalText метод GetFilter замещается так, чтобы он возвращал экземпляры классов FinancialFilter и PoliticalFilter соответственно. Если возникнет необходимость расширить систему анализатором текстов спортивной тематики, то достаточно будет создать класс SportFilter и добавить один подкласс SportText класса Text. Единственным отличием между классами SportText и, например, FinancialText, будет метод GetFilter, возвращающий фильтры разных классов.

Применимость паттерна. Фабричный метод используется в случае, когда:

- классу заранее неизвестно, объекты каких классов ему необходимо создавать;
- класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и необходимо планировать локализацию информации о том, какой класс принимает эти обязанности на себя.

ПАТТЕРН PROTOTYPE (ПРОТОТИП)

Назначение. Паттерн задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

Суть заключается в следующем. Если Client должен порождать объекты разных подклассов класса Prototype, то его параметризируют прототипом нужного подкласса. Иными словами, в качестве параметра он хранит объект одного из подклассов класса Prototype, а в случае необходимости порождения нового экземпляра он вызывает операцию Prototype->Clone, которая возвратит новый объект того же класса, что и хранимый. Схематически паттерн изображен на рис. 3.

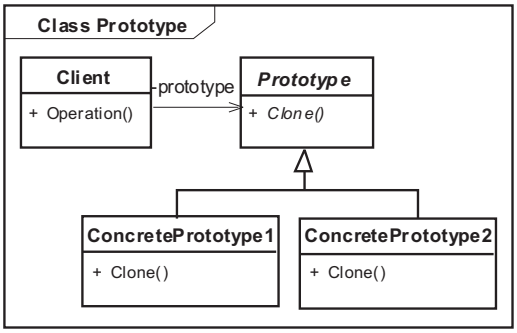

Рис. 3. Структура паттерна Prototype

Описание функциональности паттерна. Рассмотренный выше паттерн Factory Method содержит один существенный недостаток — значительный рост иерархии классов. Из последнего примера паттерна Factory Method следует, что при добавлении нового фильтра также приходится добавлять новый подкласс класса Text, причем единственная задача этого подкласса — порождение правильного фильтра. Иными словами, добавление нового подкласса-продукта обусловливает добавление нового подкласса-фабрики. Для описанного выше случая такое расширение системы было некритичным, однако если количество фильтров (в описанном примере класс Filter — это продукт) в системе возрастает до 20, то и количество подклассов класса Text (который представляет собой фабрику) также увеличивается до 20, а систему, содержащую более 40 классов, весьма сложно поддерживать.

Паттерн Prototype предлагает решение, которое позволит избежать роста связанных иерархий классов. Решение описанной выше задачи с использованием паттерна Prototype показано на рис. 4.

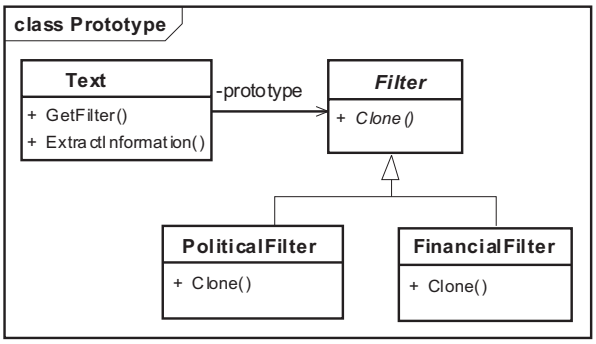

Рис. 4. Паттерн Prototype в системе извлечения данных из текста

В данном случае нет необходимости в иерархии подклассов класса Text, которую необходимо регулярно расширять при добавлении нового фильтра. Если в предыдущем примере клиент должен был породить правильный подкласс класса Text, который корректно проведет замещение операции GetFilter, то в данном случае клиент должен проинициализировать класс Text правильным прототипом фильтра, который затем будет клонирован и использован в приложении.

Приведенный выше пример демонстрирует сокращение размеров иерархии классов, которое может обеспечить паттерн Прототип, однако он не является идеальным для решения описанной задачи. Данное ранее решение с использованием паттерна Фабричный Метод более приемлем, поскольку паттерн Прототип призван решать задачи несколько иного класса. Если его применить в данном контексте, то клиенту придется порождать классы-прототипы только для передачи в класс Text, где они будут храниться в виде шаблонов. Для использования этого объекта классу Text необходимо вызывать метод клонирования. Таким об-

разом, получены две копии экземпляров одного и того же класса, одна из которых явно лишняя. В таком случае более естественным решением будет изменение метода GetFilter таким образом, чтобы он возвращал именно класс, а не его клон.

Специфичным свойством паттерна является также то, что при клонировании получаются объекты, конфигурация которых в точности совпадает с конфигурацией объекта-родителя, т.е. по умолчанию оба объекта будут иметь одинаковые значения параметров. Это обстоятельство также можно использовать для сокращения иерархии классов. Предположим, мы разрабатываем систему каталогизации текстов, которая входной поток текстов должна распределить по папкам с заданными тематиками. При этом список тематик должен быть легко расширяемым и конфигурируемым. Иными словами, необходимо создать родительский класс Thematic и породить от него подклассы, специфичные для каждой тематики, например FinancialThematic, PoliticalThematic, SportThematic, EcologicalThematic и т.д. При этом возникает две трудности.

- Хорошая система каталогизации потребует создания большого количества тематик, а значит, количество классов в такой системе будет большим и систему будет трудно поддерживать и вносить в нее изменения.
- Если система предполагает создание новых и удаление существующих тематик пользователем, то возникает вопрос о динамическом создании и удалении классов из системы.

Оба эти вопроса решаются с помощью паттерна Прототип. Для решения первого вопроса необходимо изменить структуру классов системы. Вместо создания подклассов класса Thematic для представления разных тематик будем конфигурировать класс Thematic разными параметрами. Допустим, в простейшем варианте тематика задается набором ключевых слов. Тогда для представления разных тематик создадим экземпляры класса Thematic, проинициализированные различными наборами ключей. Таким образом, получим объекты Thematic1, Thematic2, Thematic3, каждый из которых будет представлять конкретную тематику. Если возникнет необходимость в получении еще одного экземпляра, например объекта Thematic1, для создания подтематики, то применим метод Thematic1->Clone.

Второй вопрос предполагает динамическое изменение структуры системы, а значит, статический набор классов здесь неприемлем. Однако можно воспользоваться моделью, рассмотренной выше, в которой все классы создаются динамически. В этой модели не возникает сложностей с созданием новых и удалением существующих классов, поскольку в роли класса выступают экземпляры объектов (так называемые объекты-классы). Более того, такая модель позволяет легко сохранить структуру классов на диске, а при следующей загрузке считать ее и создать необходимые объекты. Таким образом, обеспечивается динамическое конфигурирование системы объектами непосредственно во время работы, что и требуется. Если система предполагает динамическую загрузку большого количества различных прототипов, то рекомендуется использовать диспетчер прототипов — специальное хранилище, которое возьмет на себя функции по загрузке, выгрузке и управлению прототипами.

Представим краткий перечень основных преимуществ, которыми обладает паттерн Прототип.

1. Добавление и удаление классов во время выполнения, т.е. динамическое создание новых классов.

2. Динамическое конфигурирование приложения классами, т.е. загрузка новых классов во время работы приложения.

3. Создание новых объектов-классов путем конфигурации параметров, т.е. замена порождения подклассов клонированием родительского класса в нужном состоянии.

4. Сокращение числа подклассов, т.е. замена параллельных иерархий классов одной иерархией продуктов (пример построения альтернативной архитектуры для системы извлечения данных из текста).

5. Создание новых классов путем группировки существующих, т.е. возможность сохранить набор объектов как один класс и далее порождать экземпляры этого составного класса, пользуясь методом Clone. Если в описанной выше системе тематики текстов задаются не списком ключевых слов, а набором текстов подобной тематики, то можно воспользоваться паттерном Прототип для создания новых тематик. В этом случае пользователь имел бы возможность выделить множество объектов типа Text и сохранить их как составной объект-тематику. Этот объект можно было бы сохранить на диске либо породить новые его экземпляры для дальнейшей работы.

Основной сложностью при реализации данного паттерна является написание метода Clone. Во-первых, если в системе используются классы, автором которых является другой человек, а исходный код их недоступен, то добавить данный метод будет весьма сложно. Кроме того, с методом Clone связан вопрос глубокого/поверхностного копирования. Суть вопроса заключается в обработке ссылок на другие объекты, которые хранит копируемый (клонируемый) объект. Поверхностное копирование означает, что объект-оригинал и его клон разделяют между собой ссылки, т.е. оба они ссылаются на один и тот же объект. Поверхностное копирование часто бывает реализовано по умолчанию в современных языках программирования (например, в языке Java). Однако часто поверхностного копирования бывает недостаточно, так как в этом случае клон зависим от оригинала, поскольку разделяет с ним компоненты.

Класс, реализующий метод Clone, должен иметь копирующий конструктор, который принимает входящим параметром объект своего класса. Данный конструктор и будет использоваться в методе Clone для создания нового экземпляра прототипа. Иногда в классы, обеспечивающие клонирование, включают метод Initialize с целью изменения значения некоторых параметров клона после клонирования.

Использование паттерна в компьютерной лингвистике. Выше были рассмотрены основные преимущества паттерна Прототип. Несмотря на весьма широкий набор полезных свойств, наиболее востребован паттерн в системах, позволяющих пользователю создавать объекты из составных частей. Причем чем более широкие возможности по конфигурированию базовых частей и построению из них объектов предполагает система, тем большую пользу принесет паттерн Прототип.

В качестве примера рассмотрим архитектуру системы для нахождения текстов с заданной информацией. Данная система позволяет пользователю конструировать свои собственные типы фильтров для дальнейшего использования в логических выражениях. Все тексты, поданные на вход системы, будут проверены на соответствие условиям выражения. Пользователю выдаются только те тексты, для которых выражение истинно.

Предполагается, что фильтром является набор неких шаблонов расположения слов в предложении. Так, например, класс TimeFilter может быть задан следующими шаблонами:

— *[0–9]{2}:[0–9]{2}:[0–9]{2}[0–9]{2}:[0–9]{2}.[0–9]4* — для времени в виде часы:минуты:секунды день.месяц.год, например 11:30:22 12.09.2010;

— *[0–9]{2}:[0–9]{2}:[0–9]{2}* — для времени в виде часы:минуты:секунды, например 11:30:22;

— *[0–9]{2}:[0–9]{2}* — для времени в виде часы:минуты, например 11:30;

— *[0–9]{2}–[0–9]{2}* — для времени в виде часы-минуты, например 11-30;

— *[0–9]{2}–[0–9]{2}.(am,pm) — для времени в виде часы-минуты am/pm, например, 11-30 am или 11-30 pm

и другие шаблоны.

Создание класса PlaceFilter потребует более широкой функциональности, совмещения регулярных выражений (как показано в шаблонах выше) с синтаксическими/морфологическими шаблонами, например:

— *«НазваниеГорода»*, где «НазваниеГорода» представляет морфологический шаблон, указывающий на все названия городов, присутствующие в морфологической базе;

— *«НазваниеМестности»*;

— *«НазваниеУлицы»*;

— *(возле, около, рядом с, поблизости от, ...) «ИмяСуществительное»*

и другие шаблоны.

Создание класса PersonFilter потребует в основном работы по различению слов с прописной буквы, применения к ним морфологических фильтров и разбора синтаксических выражений.

Создание класса ActionFilter базируется в основном на нахождении связок «глагол*существительное» (заработал*деньги, купил*дом) и непосредственно глаголов (победил, был*выбран) в тексте на различных позициях.

Предполагается, что базовая поставка системы будет включать определенное количество начальных, заранее сконфигурированных фильтров, например вышеперечисленных. Однако пользователь должен иметь возможность добавить в систему новые нужные ему фильтры, причем как верхнего уровня (например, фильтр стихийных бедствий), так и подфильтры (например, фильтр стран, как подтип PlaceFilter).

Далее, пользователь от каждого фильтра может породить конкретные классы-значения. Каждый такой класс является фильтром, который сконфигурирован списком точных значений, например класс Berlin — подкласс PlaceFilter, в котором список допустимых значений имеет вид «Берлин», «берлин», «Berlin».

Вся описанная выше информация будет использована для поиска интересующих пользователя документов. Предположим, пользователя интересуют документы относительно встречи Путина с Януковичем в Берлине в сентябре 2010 года. Используя описанные выше классы, пользователь создаст классы StartTime, сконфигурированный на 1 сентября 2010, EndTime — на 3 сентября 2010, а также создаст класс Berlin, как подкласс класса PlaceFilter, классы Putin и Yanukovich, как подклассы класса PersonFilter, и класс Meeting, как подкласс ActionFilter.

Далее, пользуясь созданными классами, следует задать логическое выражение. Для данного примера имеем:

(PersonFilter=Putin and PersonFilter=Yanukovich) and (TimeFilter>=StartTime and TimeFilter<=EndTime) and (PlaceFilter=Berlin) and (ActionFilter=Meeting).

Теперь система сможет найти все тексты с интересующей информацией.

На рис. 5 изображена не архитектура самой системы, а диаграмма классов, которые будут порождены в описанном примере. Поскольку данная система проектируется с использованием паттерна Прототип, то реальная архитектура системы будет состоять только из двух элементов, указанных на рис. 6. Остальные подклассы класса PrototypeFilter будут созданы в виде объектов-классов. Соответственно пользователь сможет сохранять созданные фильтры на диске,

добавлять в систему фильтры, созданные другими пользователями, удалять ненужные фильтры и т.д., т.е. пользоваться всеми преимуществами гибкого дизайна иерархии классов.

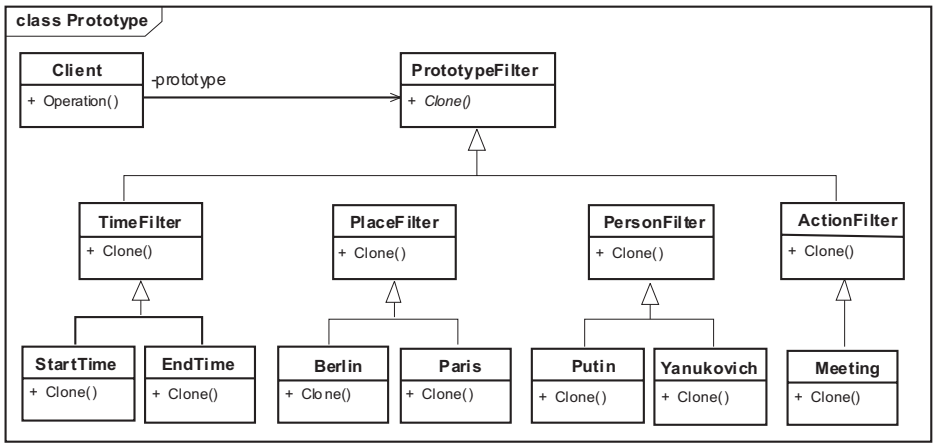

Рис. 5. Пример использования паттерна Prototype в системе фильтрации данных

Применимость паттерна. Прототип используется, когда система не зависит от способов создания, компоновки и представления продуктов, при этом:

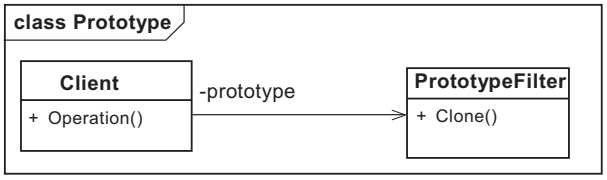

Рис. 6. Архитектура системы фильтрации данных

- порождаемые классы определяются в процессе работы программы, например с помощью динамической загрузки;
- необходимо избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
- экземпляры класса могут находиться в одном из небольшого числа состояний. Иногда более целесообразно установить соответствующее число прототипов и клонировать их, а не порождать регулярно класс вручную в требуемом состоянии.

ПАТТЕРН SINGLETON (ОДИНОЧКА)

Назначение. Паттерн гарантирует, что система имеет только один экземпляр класса и предоставляет к нему глобальную точку доступа.

Суть Одиночки состоит в следующем. Если в системе существует необходимость запретить порождение более одного экземпляра класса, то в классе объявляется специальный статический метод (Instance), предоставляющий доступ к его экземпляру. Метод Instance имеет доступ к переменной, хранящей ссылку на экземпляр класса. Если в момент доступа к экземпляру ссылка пуста, то Instance породит экземпляр и возвратит ссылку на него, иначе возвратит ссылку на уже существующий экземпляр. Конструктор класса делают защищенным, чтобы закрыть другие способы порождения экземпляров. Таким образом, получить доступ к экземпляру можно только через метод Instance.

Описание функциональности паттерна. Singleton реализует простую идею: один класс — один экземпляр, т.е. его применение рассчитано на системы, которые ограничивают число порождаемых экземпляров некоторых классов. Обычно такими классами могут быть различные менеджеры, осуществляющие управление всеми остальными элементами системы. Например, для операцион-

ной системы таким классом может быть оконный менеджер. Отметим, что Singleton, порождающий только один свой экземпляр, представляет частный случай. В действительности паттерн может быть сконфигурирован для порождения любого заданного числа экземпляров.

В некотором роде паттерн Одиночка подобен паттерну Прототип, поскольку также порождает свои экземпляры. Кроме того, как и в случае Прототипа, паттерн Одиночка базируется на структуре объект — класс, хотя и созданной иным способом. Несмотря на внешнее сходство, реализации паттернов существенным образом отличаются. Паттерн Одиночка представляет собой класс со скрытым конструктором, поэтому порождение экземпляров данного класса возможно только через специальную операцию Instance. Соответственно данный метод объявлен как статический, т.е. как метод класса. Вызов этого метода возвращает ссылку на экземпляр класса, которая хранится в статической переменной uniqueInstance. Если в момент обращения ссылка пуста (ни один экземпляр класса не был еще порожден), то для порождения экземпляра будет вызван конструктор класса. Ссылка на новый экземпляр будет сохранена в переменной uniqueInstance.

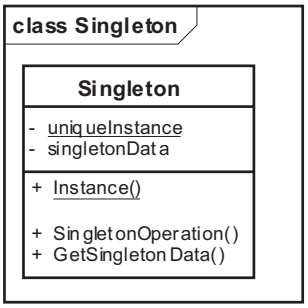

Рис. 7. Структура паттерна Singleton

Данный подход позволяет клиенту не акцентировать внимания на времени создания экземпляра класса Singleton, а использовать его так, как будто нужный экземпляр существует в любой момент работы системы. Функции порождения экземпляра возьмет на себя сам класс. Дополнительным преимуществом такого подхода (отложенной инициализации) является то, что система не перегружается неиспользуемыми объектами, поскольку экземпляр класса Singleton порождается в момент обращения, т.е. когда он гарантированно необходим в системе.

Другим положительным моментом применения паттерна является обеспечение глобальной доступности объекта без использования глобальных переменных. Это значит, что те объекты, для создания которых используется Singleton, присутствуют в системе в единичных экземплярах, поэтому наиболее логичным хранилищем ссылки на такой объект будет глобальная переменная, так как именно она обеспечивает наиболее простой метод доступа к объекту из любого места кода. Паттерн Одиночка избавляет программиста от необходимости создавать глобальные переменные для доступа к своим экземплярам. Доступ к любому экземпляру класса Singleton может быть получен непосредственно через имя класса. Этот паттерн позволяет легко изменить количество разрешенных экземпляров. Для поддержания двух и более экземпляров необходимо изменить только операцию доступа к экземпляру, т.е. метод Instance и добавить статические переменные для хранения ссылок на дополнительные экземпляры, например добавить в метод Instance параметр, указывающий номер экземпляра для получения, а в самом методе реализовать ветвление, возвращающее ссылку на экземпляр, соответствующий указанному параметру.

Более сложным является вопрос порождения подклассов Одиночки. Сложность заключается не в объявлении подкласса, а в методе порождения его экземпляров, поскольку существующий метод Instance будет порождать экземпляры класса Singleton вместо экземпляров подкласса. Существуют три способа решения данной проблемы.

1. Переопределить операцию Instance в подклассах так, чтобы она порождала экземпляры нужного класса.

2. Создать в родительском классе Singleton в операции Instance ветвление, отвечающее за порождение правильного класса. Нужную ветвь определить с помощью переменной среды (предполагается, что Singleton используется для представления какого-то глобального объекта системы, существующего в единственном экземпляре, поэтому такой метод допустим).

3. Наиболее гибкий метод — ведение реестра Одиночек, когда в системе существует список, в котором содержатся имена Одиночек и их классы. Метод Instance при порождении класса определяет его тип, используя поиск по имени в реестре Одиночек. Имя актуального Одиночки, как и в п. 2, хранится в переменной среды.

Использование паттерна в компьютерной лингвистике. Как было описано выше, паттерн Одиночка целесообразно применять в классах, представляющих крупные элементы управления, которые могут существовать только в единственном экземпляре. Также возможно его применение в сочетании с другими паттернами. В качестве примера рассмотрим возможности применения паттерна Одиночки вместе с паттерном Абстрактной Фабрики. В работе [1] в качестве примера многофункциональной лингвистической системы приведена схема паттерна AbstractFactory. В начале работы системы порождается нужный подкласс HandlersFactory, который далее производит конкретные продукты «правильных» классов. Следовательно, применять паттерн Одиночка нужно к классу HandlersFactory. Далее необходимо порождать подклассы Одиночки. Ранее было рассмотрено три возможных метода порождения подкласса Одиночки: создание ветвления в операции Instance, замещение этой операции в подклассах и использование реестра Одиночек. Применим наиболее простой подход — ветвление: в классе HandlersFactory реализуем метод Instance так, что в зависимости от значения переменной среды lang он будет возвращать либо экземпляр класса RussianHandlersFactory, либо экземпляр класса EnglishHandlersFactory. Если планируется изменение подкласса HandlersFactory во время выполнения программы, то в методе Instance при изменении значения переменной lang нужно предусмотреть механизм уничтожения старого подкласса HandlersFactory (так, в языке Java эти функции выполняет сборщик мусора) и инициализации нового подкласса.

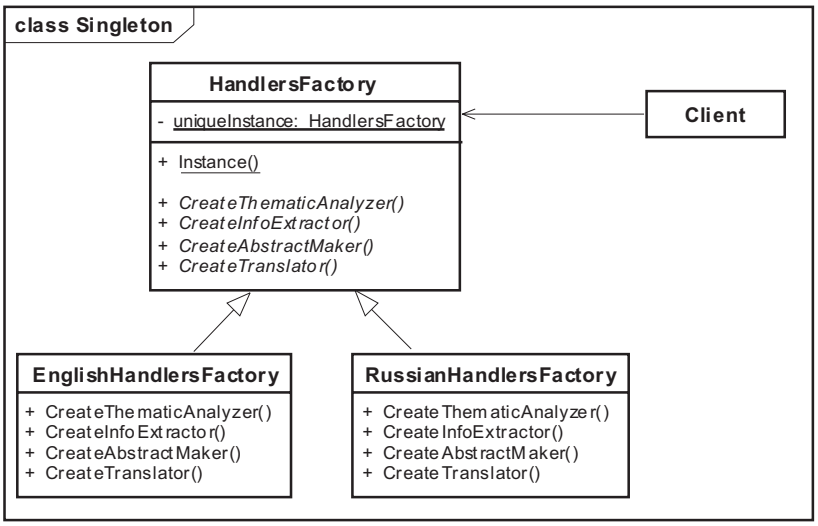

Рис. 8. Пример архитектуры многофункционального приложения с использованием паттернов Singleton и AbstractFactory

На рис. 8 дана диаграмма классов после внесения исправлений. Иерархия классов продуктов остается без изменений. Отметим, что класс HandlersFactory

уже не является абстрактным, а метод Instance становится статическим. Добавив в приложение паттерн Singleton, получим следующие результаты:

— в любой момент времени может существовать только один экземпляр фабрики, который доступен через метод HandlersFactory->Instance;

— клиенту нет необходимости порождать фабрику, достаточно установить нужное значение переменной lang; при первом обращении к фабрике необходимый экземпляр будет порожден автоматически.

Применимость паттерна. Целесообразно использовать паттерн Одиночка, когда:

• должен быть только один экземпляр некоторого класса, легкодоступный всем клиентам;

• единственный экземпляр должен расширяться путем порождения подклассов, и клиентам необходимо иметь возможность работать с расширенным экземпляром без модификации своего кода.

ЗАКЛЮЧЕНИЕ

В данной работе рассмотрены три порождающих паттерна: Factory Method, Prototype, Singleton. Детально описаны их основные особенности и способы применения. Приведенная выше информация может быть использована в качестве пособия с практическими рекомендациями для разработки дизайна прикладной системы. Примеры применения паттернов в компьютерной лингвистике являются законченными архитектурными решениями. Они могут быть использованы при создании программных систем соответствующей направленности.

Вопрос применения каждого из порождающих паттернов в решении конкретной прикладной задачи должен рассматриваться отдельно с учетом специфики создаваемой системы. Большинство из перечисленных паттернов являются взаимозаменяемыми, но способными решить задачу разными методами. Приведенная выше информация призвана помочь сделать правильный выбор паттерна с учетом неочевидных деталей.

СПИСОК ЛИТЕРАТУРЫ

1. Никоненко А.А. Использование паттернов проектирования в компьютерной лингвистике. Порождающие паттерны. Ч. I: Abstract Factory и Builder // Искусственный интеллект. — 2010. — № 4. — С. 278–286.
2. Gamma E., Helm R., Johnson R., Vlissides J. Design patterns: elements of reusable object-oriented software. — Boston: Addison-Wesley, 1995. — 518 p.

Поступила 14.10.2010