С. А. Ермоченко

Шаблоны распределения ответственности

Лекционные материалы по дисциплине «Дополнительные главы информатики. Шаблоны проектирования» для студентов 5 курса специальности Прикладная математика (1-31 03 03)

2011

Введение

Шаблоны распределения ответственности (GRASP – General Responsibility Assignment Software Patterns) используются в объектно-ориентированном проектировании для назначения классам и объектам различных обязанностей таким образом, чтобы обеспечить гибкость проектируемой системы. Гибкость проектируемого программного обеспечения характеризует способность быстро и с минимальными затратами внести в программный продукт какие-либо изменения. При чем внесение изменений может быть связано не только с добавлением новых функций в новую версию продукта, но и с изменением ошибок в существующей реализации.

Таким образом, система, спроектированная согласно шаблонам GRASP, позволяет:

– осуществлять поддержку программного продукта без необходимости внесения серьезных изменений в архитектуру классов;

– разрабатывать различные части системы параллельно несколькими разработчиками, уменьшая общее время разработки;

– повысить «читаемость» исходного кода программы, то есть сделать его более понятным разработчикам;

– обеспечить возможность повторного использования кода в других проектах, в которых понадобится аналогичный функционал.

В отличие от других шаблонов проектирования, шаблоны распределения обязанностей не предлагают конкретных решений для конкретных задач. Они лишь формулируют общие шаблонные принципы, согласно которым можно успешно спроектировать систему.

Шаблон «информационный эксперт»

Информационный эксперт (Information Expert) – *класс, обладающий необходимой и достаточной информацией для осуществления возложенных на него функций*.

Основной акцент в определении делается на необходимость и достаточность информации. Необходимость обозначает, что класс не должен обладать информацией, которая не нужна ему для выполнения своих обязанностей. Например, если разрабатывается класс **`ArraySorter`**, который должен сортировать некий массив (класс **`Array`**), то ему не нужно знать, откуда будет получен этот массив. То есть, если массив считывается из файла неким классом **`ArrayFileReader`**, то класс **`ArraySorter`** не должен обладать информацией об экземпляре класса **`ArrayFileReader`**, так как для сортировки массива эта лишняя информация.

В общем смысле любой класс системы выполняет некоторые функции с использованием определенной информации. Эту информацию класс получает из своих атрибутов, либо из параметров своих методов, либо из внешних источников (таких как базы данных, файлы, сокеты и так далее). Поэтому, строго следуя предложенному определению, любой класс можно назвать информационным экспертом. Обычно, когда говорят о том, что тот или иной класс обладает какой-либо информацией, считают, что эта информация содержится только в его атрибутах. Но даже в этом случае информационным экспертом можно считать любой класс, у которого кроме методов есть атрибуты. Поэтому в качестве информации выступает только информация о предметной области приложения. Например, при проектировании интернет-магазина предметная область характеризуется информацией о товарах, клиентах и заказах. В такой системе класс **`DataBaseConnection`**, реализующий функции подключения к серверу баз данных, и обладающий необходимой ему информацией о параметрах подключения, тем не менее не является информационным экспертом, так как

эта информация не характеризует предметную область приложения, а является вспомогательной для конкретной реализации функциональных возможностей системы.

Таким образом, в процессе выделения информационных экспертов необходимо выделить классы, которые будут хранить информацию о предметной области приложения. При этом, как правило, функции, которые возлагаются на информационные эксперты, сводятся к временному хранению информации в оперативной памяти и предоставлении доступа к этой информации другим классам, которые, в свою очередь, уже будут обрабатывать эту информацию или осуществлять сохранение / извлечение информации из внешних хранилищ (баз данных, файлов различных форматов и так далее).

Чаще всего информационный эксперт хранит нужную ему информацию в виде атрибутов. Методами такого класса являются только так называемые get-теры и set-теры. При этом методы, записывающие нужные значения в атрибуты классов (set-теры) могут осуществлять проверку корректности информации, хотя даже эта обязанность может назначаться другому классу, а не информационному эксперту.

Шаблон «информационный эксперт» обеспечивает повышение инкапсуляции в системе.

За рамками рассмотренного шаблона остался вопрос о том, какую информацию о предметной области хранить в том или ином информационном эксперте. Ведь, в конце концов, можно описать один класс, который будет хранить всю информацию о предметной области. Для ответа на этот вопрос используются другие шаблоны распределения ответственности, такие как Low Coupling и High Cohesion.

Шаблон «создатель»

Создатель (Creator) – *класс, в обязанности которого входит создание экземпляров других классов*.

Выделяют несколько критериев, при выполнении которых некоторому классу **A** следует назначить обязанность по созданию экземпляров некоторого класса **B**:

1. класс **A** содержит или получает данные, необходимые для создания экземпляра класса **B**. Здесь сознательно вместо термина «информация» использован термин «данные», для того, чтобы подчеркнуть, что необходимые данные не обязательно должны быть атрибутами класса **A.** Класс **A** может получать их через параметры своих методов, либо самостоятельно запрашивать из внешних источников (базы данных, файлы разных форматов, сетевые соединения по различным протоколам, и так далее).

2. класс **A** записывает экземпляры класса **B**. То есть класс **A** передает в класс **B** информацию, необходимую классу **B** для осуществления своих функций (класс **A** вызывает set-теры класса **B**). Данный критерий очень близок предыдущему, а чаще всего эта два критерия выполняются одновременно.

3. класс **A** активно использует экземпляры класса **B**. То есть классу **A** необходимо для осуществления своих функций активно использовать атрибуты и методы класса **B**. Часто в таких случаях внутри методов класса **A** создаются экземпляры класса **B** как локальные переменные. Однако это еще не делает класс **A** создателем. Такая обязанность на него возлагается в том случае, если созданный для собственных целей класса **A** экземпляр класса **B** может понадобиться другим классам. Тогда созданный экземпляр может передаваться классом **A** за пределы своих методов, например через возвращаемые значения этих методов.

4. класс **A** агрегирует или содержит в себе экземпляры класса **B**. В таком случае класс **A** имеет атрибут или атрибуты, типом которых является класс **B** или массив / коллекция экземпляров класса **B**. И агрегация, и композиция (содержание в себе) экземпляров класса характеризуются наличием такого атрибута. Однако существуют и различия между ними. Если экземпляр класса **A** агрегирует в себе экземпляр класса **B**, то это значит, что первый из них немыслим без второго. Например, класс **Car** агрегирует в себе класс **Motor**, и без экземпляра класса **Motor** экземпляр класса **Car** в принципе существовать не может. Если же атрибут класса **A**, имеющий тип класса **B** или массива / коллекции экземпляров класса **B**, может быть нулевой ссылкой или пустым массивом / коллекцией, то говорят, что класс **A** содержит в себе класс **B**. Например, класс **Car** содержит в себе экземпляр класса **Driver**, но это поле может быть пустым, если автомобиль стоит на ремонте, и водителя у него пока нет.

Четыре перечисленных признака имеют разную степень важности при назначении классу обязанностей по созданию экземпляров других классов. Так, например, выполнение первого критерия в подавляющем большинстве случаев является достаточной причиной назначить классу обязанность создателя. Выполнение второго критерия тоже в большинстве случаев приводит к появлению у класса обязанности создателя. Наличие третьего признака уже достаточно редко приводит к назначению классу обязанности создателя. Четвертый признак, если присутствует только он один, практически никогда не приводит к назначению классу ответственности создателя. Этот признак может только подтвердить обязанности создателя, обусловленные другими критериями. При этом чаще всего он «помогает» третьему признаку.

Шаблон «контроллер»

Контроллер (Controller) – *класс, отвечающий за обработку системных событий*.

Любой программный продукт должен определенным способом реагировать на внешние события. Например, консольное приложение взаимодействует с пользователем через строки, вводимые с клавиатуры. Настольное приложение реагирует на клавиатурные команды, действия мыши. Веб-приложение осуществляет взаимодействие с клиентами посредством HTTP-запросов. Кроме этого любое приложение может реагировать на события операционной системы:

– события, возникающие в определенное время (генерация ежемесячных отчетов, ежедневная антивирусная проверка и так далее);

– события, связанные с аппаратной составляющей компьютеров (показания различных датчиков и измерительных приборов, имеющих интерфейс взаимодействия с операционной системой или непосредственно с нашим приложением);

– события, генерируемые другими компьютерами, и оповещающими операционную систему по сети;

– другие.

При проектировании программного продукта все эти события можно определенным образом унифицировать, выделяя отдельные классы-источники событий, и отдельные классы-обработчики событий. Но, так как любой класс-обработчик в таком случае привязывается к конкретному источнику (или источникам), и наоборот, модификация системы становится затрудненной, если необходимо оставить обработку события такой же, а вот источник этого события изменить. Для того, чтобы убрать зависимости между большим количеством источников и обработчиков, в системе определяется специальный класс-контроллер событий, выполняющий роль диспетчера, получающего событие (или запрос на выполнение определенных

действий) из определенного источника, и перенаправляющего его нужному обработчику.

В зависимости от типа источника событий различают несколько типов контроллеров:

– внешний контроллер, получающий все запросы от неких внешних источников (по некоторому сетевому протоколу). С точки зрения самого источника событий такой контроллер представляет всю систему в целом, обеспечивая необходимый уровень инкапсуляции;

– контроллер роли, получающий все запросы от некоторого объекта реального мира (например, температурный датчик, датчик влажности в хранилище библиотеки для системы поддержания микроклимата, или датчики движения и датчики реакции на звук в системах охраны помещений);

– контроллер прецедента, обрабатывающий все внутренние события системы (например, события, генерируемые элементами управления формы в настольном приложении, или события генерируемые мышью и клавиатурой, или события таймера).

Если разрабатываемая система будет реагировать на события, кардинально различающиеся по способу генерации (например, события таймера и HTTP-запросы), для различных типов источников событий можно создавать несколько отдельных контроллеров, но если контроллеры выполняют очень схожие функции, нужно стараться унифицировать интерфейсы источников событий и создать один контроллер, их обрабатывающий. Обратной стороной такого подхода является создание раздутых контроллеров, получающих абсолютно все, даже сильно различающие по типу источника, события. Отличительным признаком таких контроллеров является наличие большого количества вспомогательных атрибутов и методов, призванных распределять разнотипные запросы. В таком случае один контроллер лучше разделять на несколько специализированных контроллеров.