УДК 004.4

Ю.Э. Парфенов, А.В. Щербаков

Харьковский национальный экономический университет, Харьков

ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ ПРОЕКТИРОВАНИЯ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В статье проанализированы существующие шаблоны проектирования программного обеспечения, дана их классификация, указаны особенности применения шаблонов, выделены достоинства и недостатки каждого из них. Показана целесообразность использования шаблонов проектирования для повышения эффективности процесса разработки программного обеспечения. Приведены рекомендации по использованию шаблонов при разработке программного обеспечения для решения целого класса задач в различных предметных областях.

Ключевые слова: *шаблон проектирования, класс, объект, программное обеспечение, процесс разработки ПО.*

Введение

Шаблон проектирования – повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста [1].

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные [1].

На наивысшем уровне существуют архитектурные шаблоны, они охватывают собой архитектуру всей программной системы. Иногда шаблоны консервируют громоздкую и малоэффективную систему понятий, разработанную узкой группой. Когда количество шаблонов возрастает, превышая критическую сложность, исполнители начинают игнорировать шаблоны и всю систему, с ними связанную. Нередко шаблонами заменяется отсутствие или недостаточность документации в сложной программной среде.

Как показывает анализ литературы [2 – 8], часть специалистов считает, что слепое применение шаблонов из справочника, без осмысления причин и предпосылок выделения каждого отдельного шаблона, замедляет профессиональный рост программиста, так как подменяет творческую работу механической подстановкой шаблонов. Люди, придерживающиеся данного мнения, считают, что знакомиться со списками шаблонов необходимо тогда, когда программист «дорос» до них в профессиональном плане - и не раньше. Хороший критерий нужной степени профессионализма - выделение шаблонов самостоятельно, на основании собственного опыта. При этом, разумеется, знакомство с теорией, связанной с шаблонами, полезно на любом уровне профессионализма и направляет развитие программиста в правильную сторону. Сомнению подвергается только использование шаблонов «по справочнику».

Шаблоны могут пропагандировать плохие стили разработки приложений, и зачастую слепо применяются. Для преодоления этих недостатков используется рефакторинг [4].

Цель статьи - проанализировать существующие шаблоны проектирования программного обеспечения, выявить их сильные и слабые стороны, выработать рекомендации по их использованию.

Основная часть

Типы шаблонов проектирования

Шаблоны проектирования делятся на следующие категории: порождающие шаблоны, структурные шаблоны и поведенческие шаблоны [2].

Порождающие шаблоны - абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту. Эти шаблоны проектирования относятся к созданию экземпляров классов. Далее они могут быть разделены на

шаблоны создания классов и шаблоны создания объектов. В то время как шаблоны создания классов эффективно используют наследование в процессе создания их экземпляров, шаблоны создания объектов эффективно используют делегирование, чтобы выполнить эту работу.

Структурные шаблоны - определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Поведенческие шаблоны - определяют взаимодействие между объектами, тем самым увеличивая его гибкость.

Порождающие шаблоны

«Абстрактная фабрика» (Abstract Factory) - предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.

«Фабричный метод» (Factory Method) - определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

«Одиночка» (Singleton) - гарантирует, что некоторый класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему.

«Строитель» (Builder) - отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных представлений.

«Прототип» (Prototype) - описывает виды создаваемых объектов с помощью прототипа и создаёт новые объекты путём его копирования. У прототипа те же самые результаты, что у «абстрактной фабрики» (Abstract Factory) и «строителя» (Builder): он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имён. Кроме того все эти шаблоны позволяют клиенту работать со специфичными для приложения классами без модификаций.

«Пул объектов» (Object Pool) - класс, который представляет собой интерфейс для работы с набором инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

«Абстрактная фабрика» (Abstract Factory) создает экземпляр нескольких семейств классов. Цель применения шаблона - обеспечить уровень косвенности, который позволяет абстрагировать создание семейств связанных или зависимых объектов без непосредственного указания их конкретных классов. Объект "фабрика" отвечает за обеспечение сервисов создания для всего семейства платформы. Клиенты никогда не создают объекты платформы непосредственно, они просят «фабрику» сделать это для них.

Этот механизм позволяет легко обмениваться семействами продуктов, потому что определенный класс объекта «фабрики» появляется в приложении только один раз: там, где он создается.

Приложение может за один раз заменить все семейство продуктов, просто создавая различные конкретные экземпляры «абстрактной фабрики».

Так как сервис, предоставляемый «фабрикой» объектов, является всеобъемлющим, она обычно реализуется как «Одиночка».

«Абстрактная фабрика» определяет фабричный метод для каждого продукта. Каждый фабричный метод инкапсулирует оператор new и специфичные для конкретной платформы классы продуктов. Каждая "платформа" затем создается по образцу класса, производного от «фабрики».

Рекомендации по использованию:

1. Решите, является ли "независимость от платформы" и сервисы создания источником проблем в настоящее время.

2. Спланируйте матрицу "платформ" в зависимости от "продуктов".

3. Определите интерфейс «фабрики», который состоит из фабричных методов для каждого продукта.

4. Определите класс, производный от «фабрики», для каждой платформы, который инкапсулирует все обращения к оператору new.

5. Вместо обращения к оператору new клиент должен использовать для создания объектов продуктов фабричные методы.

Шаблон «Одиночка» (Singleton) создает класс, только один экземпляр которого может существовать. Цель применения - гарантировать, что у класса есть только один экземпляр, обеспечить глобальную точку доступа к нему, а также обеспечить оперативную инициализацию или инициализацию «при первом использовании».

Сделайте класс единственного экземпляра объекта ответственным за создание, инициализацию, доступ и принудительное применение ограничений. Объявите этот экземпляр как закрытое статическое поле класса. Предоставьте открытый статический метод, который содержит весь код инициализации, а также предоставляет доступ к экземпляру. Клиент вызывает функцию доступа (используя имя класса), когда бы ни потребовалась ссылка на единственный экземпляр класса.

Использование «Одиночки» следует рассматривать только, если удовлетворены все три условия:

1. Принадлежность единственного экземпляра не может быть назначена разумным образом.

2. Желательна отложенная инициализация.

3. Глобальный доступ не обеспечивается другим образом.

Если принадлежность единственного экземпляра, порядок инициализации объекта и глобальный доступ не являются проблемами, «Одиночка» не является достаточно интересным.

Шаблон «Одиночка »может быть расширен для поддержки доступа к количеству экземпляров класса, зависящего от конкретного приложения.

Подход использования "статического метода доступа" не будет поддерживаться в классах, производных от класса «Одиночки».

Сделайте класс единственного экземпляра объекта ответственным за доступ и инициализацию «при первом использовании». Единственный экземпляр является закрытым статическим полем. Функция доступа – открытый статический метод.

Рекомендации по использованию:

1. Определите закрытое статическое поле в классе "единственного экземпляра".

2. Определите открытый статический метод доступа в этом классе.

3. Выполните "ленивую инициализацию" (создание при первом использовании) в методе доступа.

4. Определите все конструкторы как **protected** или **private**.

5. Чтобы манипулировать «Одиночкой» клиенты могут использовать только метод доступа.

«Пул объектов» (Object pool) позволяет исключить накладные расходы на создание и освобождение ресурсов путем повторного использования объектов.

Использование «пула объектов» позволяет получить значительное увеличение производительности. Является наиболее эффективным в тех случаях, когда стоимость инициализации экземпляра класса высока и в любой момент времени используется небольшое количество экземпляров.

Пул объектов позволяет "проверять" объекты в нем. Когда эти объекты больше не используются их процессами, они возвращаются в пул для повторного использования.

Однако, нежелательно, чтобы другому процессу приходилось ждать, пока «освободится» конкретный объект. Поэтому «пул объектов» создает новые объекты, когда они необходимы. Он также должен реализовать средства для периодической «очистки» неиспользуемых объектов.

Общей идеей шаблона «Пул объектов» является то, что, не обязательно создавать новые экземпляры класса, если можно повторно использовать уже существующие экземпляры.

- Reusable (повторно-используемый) - экземпляры классов в этой роли взаимодействуют с другими объектами в течение ограниченного периода времени. После этого они больше не нужны.
- Client (клиент) - экземпляры классов в этой роли используют повторно-используемые объекты.
- ReusablePool (пул повторно-используемых объектов) - экземпляры классов в этой роли управляют повторно-используемыми объектами.

Как правило, желательно, сохранять все повторно-используемые объекты, которые не используются в настоящее время, в один и тот же пул объектов для того, чтобы ими можно управлять, используя один и тот же подход. Для достижения этой цели класс ReusablePool должен быть «Одиночкой». Его конструкторы являются закрытыми, что заставляет другие классы, вызывать его метод getInstance, чтобы получить один экземпляр класса ReusablePool.

Объект класса Client вызывает метод acquireReusable объекта класса ReusablePool, когда он нуждается в повторно-используемом объекте. Объект класса ReusablePool содержит коллекцию повторно-используемых объектов, которая и является пулом.

Если при вызыве метода acquireReusable в пуле есть какие-либо повторно-используемые объекты, то он удаляет такой объект из пула и возвращает его. Если пул пуст, то, если это возможно, метод acquireReusable создает повторно-используемый объект. Если acquireReusable не может создать новый повторно-используемый объект, то он ждет, пока такой объект появится в коллекции.

Когда объекты класса Reusable больше не используются объектами класса Client, они передают их методу releaseReusable объекта класса ReusablePool. Метод ReleaseReusable возвращает такие повторно-используемые объекты в пул.

Во многих приложениях, использующих шаблон «Пул объектов», есть основания для ограничения общего количества повторно-используемых объектов, которые могут существовать. В таких случаях объект класса ReusablePool, который создает повторно-используемые объекты, отвечает за это. В данном случае класс ReusablePool должен иметь метод для указания максимального количества объектов, которые будут созданы.

Рекомендации по использованию

1. Создайте класс ObjectPool с закрытым массивом объектов внутри.

2. Создайте методы acquare и release в классе ObjectPool .

3. Убедитесь, что класс ObjectPool является «Одиночкой».

Структурные шаблоны

«Адаптер» (Adapter) - преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами. Обеспечивает совместную работу классов, которая была бы невозможной из-за несовместимости их интерфейсов.

«Декоратор» (Decorator) - динамически возлагает на объект новые функции. «Декораторы» применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов.

«Заместитель» (Proxy) - подменяет другой объект для контроля доступа к нему, то есть является заменителем другого объекта. «Заместитель» применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно при использовании простого указателя. С помощью этого шаблона при доступе к объекту вводится дополнительный уровень косвенности.

«Компоновщик» (Composite) - группирует объекты в древовидные структуры для представления иерархии типа "часть-целое". Позволяет клиентам работать с единичными объектами так же, как с группами объектов.

«Мост» (Bridge) - отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое.

«Приспособленец» (Flyweight) - использует разделение для эффективной поддержки большого числа мелких объектов. При использовании «приспособленцев» не исключены затраты на передачу, поиск или вычисление состояния объекта, особенно если раньше оно хранилось как внутреннее. Однако такие расходы компенсируются экономией памяти за счет разделения объектов-«приспособленцев».

«Фасад» (Facade) - предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой. Клиенты общаются с подсистемой, посылая запросы «фасаду». Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, «фасаду», возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы. Клиенты, пользующиеся «фасадом», не имеют прямого доступа к объектам подсистемы.

Шаблон «Адаптер» (Adapter) «соединяет» интерфейсы различных классов.

Цели применения шаблона:

• Преобразовать интерфейс класса в другой интерфейс, который требуется клиентам. «Адаптер» позволяет классам с несовместимыми интерфейсами работать вместе.

• «Обернуть» существующий класс в новый интерфейс.

• «Подогнать» старый компонент к новой системе.

Повторное использование всегда был трудным и неочевидным. Одной из причин была необходимость разработки чего-то нового, повторно используя что-то старое.

Как известно, между старым и новым всегда существуют противоречия. Это могут быть физические размеры, несогласованность, синхронизация, неудачные допущения или конкурирующие стандарты.

Таким образом, необходимо создать абстракцию-посредника, преобразующую устаревший компонент для работы в новой системе. Клиенты вызывают методы объекта-«адаптера», который перенаправляет их в вызовы к устаревшему компоненту. Эта стратегия может быть реализована с использованием наследования или агрегации.

«Адаптер» работает как «обертка» или модификатор существующего класса. Он обеспечивает различное или преобразованное представление этого класса.

Рекомендации по использованию

1. Определите компоненты, которые хотят быть размещенны (т.е. клиентов), и компонент, который нуждается в адаптации (т.е. адаптируемый).

2. Определите интерфейс, который требует клиент.

3. Спроектируйте класс-«обертку», который может "подключать" адаптируемого к клиенту.

4. «Адаптер» («обертка») "содержит" экземпляр адаптируемого класса.

5. «Адаптер» («обертка») сопоставляет интерфейс клиента и интерфейс адаптируемого.

6. Клиент использует свой интерфейс.

Шаблон «Фасад» создает единственный класс, который представляет всю подсистему.

Цели:

• Обеспечить унифицированный интерфейс для набора интерфейсов подсистемы. «Фасад» определяет интерфейс более высокого уровня, что делает подсистему проще в использовании.

• «Обернуть» сложную подсистему в более простой интерфейс.

Этот шаблон связан с инкапсуляцией сложной подсистем в единственном интерфейсном объекте. Это позволяет сократить время обучения, необходимое для успешного использования подсистемы. Он также способствует независимости подсистемы от потенциально многих клиентов.

С другой стороны, если «фасад» является единственной точкой доступа к подсистеме, то это будет ограничивать возможности и гибкость, которая может понадобиться "продвинутым пользователям". Объект «фасада» должен быть достаточно простым посредником.

Он не должен стать всезнающим оракулом или "божественным" объектом.

Рекомендации по использованию.

1. Определите более простой, унифицированный интерфейс для подсистемы или компонента.

2. Спроектируйте класс-«обертку», который инкапсулирует подсистему.

3. «Фасад» («обертка») должен скрывать сложность и взаимодействия компонента, и делегировать к соответствующим методам.

4. Клиент должен использовать только «фасад» .

5. Подумайте, нужны ли дополнительные «фасады».

«Заместитель» (Proxy) создает объект, представляющий другой объект.

Цели:

• Обеспечить заменитель или заполнитель для другого объекта для управления доступом к нему.

• Использовать дополнительный уровень абстракции для поддержки распределенного, управляемого или интеллектуального доступа.

• Добавить «обертку» и делегацию для защиты реального компонента от излишней сложности.

Спроектируйте «заменитель» или «заместитель», т.е. объект, который создает реальный объект, как только клиент впервые делает запрос к «заместителю», запоминает отличительные черты этого реального объекта и перенаправляет запрос к этому реальному объекту. Затем все последующие запросы просто пересылаются непосредственно инкапсулированному реальному объекту.

Существует четыре общих ситуаций, в которых применим шаблон «Заместитель»:

1. «Виртуальный заместитель» является заполнителем для «дорогих для создания" объектов. Реальный объект создается только, когда клиент впервые делает запрос к объекту.

2. «Удаленный заместитель» обеспечивает местное представительство для объекта, который находится в другом адресном пространстве.

3. «Защитный заместитель» управляет доступом к «чувствительному» главному объекту. Объект-«заменитель» проверяет права доступа абонента до перенаправления запроса.

4. «Умный заместитель» выполняет дополнительные действия, когда происходит доступ к объекту.

Рекомендации по использованию:

1. Определите «аспект», который лучше всего реализуется как «обертка» или «заменитель».

2. Определите интерфейс, который сделает «заместителя» и исходный компоненты взаимозаменяемыми.

3. Рассмотрите определение «фабрики», которая может инкапсулировать решение о том желателен ли заместитель или исходный объект.

4.Класс-«обертка» должен содержать указатель на реальный класс и реализовать указанный интерфейс.

5. Указатель можно инициализировать в конструкторе или при первом использовании.

6. Каждый метод «обертки» должен обращаться к «завернутому» объекту.

Поведенческие шаблоны

«Интерпретатор» (Interpreter) - для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.

«Шаблонный метод» (Template Method) - определяет «скелет» алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры. «Шаблонные методы» - один из фундаментальных приемов повторного использования кода. Они особенно важны в библиотеках классов, поскольку предоставляют возможность вынести общее поведение в библиотечные классы. Шаблонные методы приводят к инвертированной структуре кода, которую иногда называют принципом Голливуда, подразумевая часто употребляемую в этой кино-империи фразу "Не звоните нам, мы сами позвоним". В данном случае это означает, что родительский класс вызывает операции подкласса, а не наоборот.

«Итератор» (Iterator) - дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

«Команда» (Command) - инкапсулирует запрос в виде объекта, позволяя тем самым параметризовывать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций.

«Наблюдатель» (Observer) - определяет зависимость типа «один ко многим» между объектами, так что при изменении состояния одного объекта все зависящие от него объекты получают извещение и автоматически обновляются. Шаблон позволяет изменять субъекты и наблюдатели независимо друг от друга. Субъекты разрешается

повторно использовать без участия наблюдателей и наоборот. Это дает возможность добавлять новых наблюдателей без модификации субъекта или других наблюдателей.

«Посетитель» (Visitor) - представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

«Посредник» (Mediator) - определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия. Коллеги посылают запросы посреднику и получают запросы от него. Посредник реализует кооперативное поведение путем переадресации каждого запроса подходящему коллеге (или нескольким коллегам).

«Состояние» (State) - позволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что поменялся класс объекта.

«Стратегия» (Strategy) - определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется.

«Хранитель» (Memento) - позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.

«Цепочка обязанностей» (Chain of Responsibility) - позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запросом начинает обрабатываться один из нескольких объектов. Объекты-получатели связываются в цепочку, и запрос передается по цепочке, пока кокой-то объект его не обработает.

«Посредник» (Mediator) определяет упрощенную связь между классами

Цель: определить объект, который инкапсулирует, как взаимодействует набор объектов. Посредник обеспечивает слабую связь, предотвращая явные ссылки объектов друг на друга, и позволяет независимо изменять их взаимодействие.

Например, в операционной системе Unix разрешение на доступ к системным ресурсам регулируется на трех уровнях детализации: мир, группа и владелец. Группа представляет собой набор пользователей, предназначенных для моделирования некоторой функциональной принадлежности. Каждый пользователь в системе может быть членом одной или нескольких групп, и каждая группа может иметь ноль или более пользователей.

Разделение системы на многие объекты в целом увеличивает степень повторного использования, но сопутствующее увеличение количества взаимосвязей между этими объектами стремится уменьшить его. Объект посредника включает в себя все взаимосвязи, выступает в качестве концентратора, отвечает за управление и координацию взаимодействия своих клиентов, и способствует ослаблению связи, предотвращая явные ссылки объектов друг на друга.

Шаблон «Посредник» повышает отношение «сеть многие ко многим» до «полной информация о состоянии объекта». Моделирование взаимосвязей с объектом улучшает степень инкапсуляции, а также позволяет изменять или расширять поведение этих взаимосвязей с помощью наследования.

Пример, в котором полезен посредник, это проектирование пользователей и групп в операционной системе. Группа может иметь ноль или более пользователей, а пользователь может быть членом ноль или более групп. Шаблон «Посредник» обеспечивает гибкий и ненавязчивый способ присоединения и управления пользователями и группами.

Коллеги (или участники) не связаны друг с другом. Каждый общается с посредником, который, в свою очередь взаимодействует с ними. Соответствие "многие ко многим" между коллегами, которое могло бы существовать в противном случае, было повышено до "полной информация о состоянии объекта".

Рекомендации по использованию:

1. Определите совокупность взаимодействующих объектов, которые выиграют от взаимной развязки.

2. Инкапсулируйте эти взаимодействия в новом классе.

3. Создайте экземпляр этого нового класса и измените все объекты-"участники" для взаимодействия только с посредником.

4. Равномерно балансируйте принцип развязки с принципом распределения ответственности.

5. Будьте осторожны, чтобы не создать «контролер» или «божественный» объект.

«Наблюдатель» (Observer) предоставляет способ уведомления определенного количества классов о некотором изменении.

Цели:

• Определить зависимость «один ко многим» между объектами, таким образом, что, когда один объект изменяет свое состояние, все его зависимые объекты получают уведомление и обновляются автоматически.

• Инкапсулировать компоненты ядра (или обычные компоненты) в абстракции «субъект», а переменные компоненты (опциональные компоненты, компоненты пользовательского интерфейса) - в иерархии наблюдателя.

Определите объект, который является «хранителем» модели данных или бизнес-логики (субъект). Делегируйте всю функциональность «представления» для отделения объектов наблюдателя. Наблюдатели регистрируют сами себя в субъекте. Всякий раз, когда в субъекте происходят изменения, он извещает об этом всех зарегистрированных наблюдателей.

Субъект представляет общую абстракцию. «Наблюдатель» представляет переменную абстракцию. Субъект подсказывает объектам «наблюдателей», делать свое дело. Каждый наблюдатель может выполнять «call back»- вызов по мере необходимости.

Рекомендации по использованию:

1. Различайте основную (или независимую) функциональность и дополнительную (или зависимую) функциональность.

2. Смоделируйте независимую функциональность с помощью абстракции «субъект».

3. Смоделируйте зависимую функциональность с помощью иерархии «наблюдателей».

4. Субъект должен быть связан только с базовым классом наблюдателя.

5. Клиент настраивает количество и тип наблюдателей.

6. Наблюдатели регистрируют сами себя в субъекте.

7. Субъект передает события всем зарегистрированным наблюдателям.

8. Субъект может "протолкнуть" информацию наблюдателям, или, наблюдатели могут "вытянуть" нужную им информацию из субъекта.

Выводы

Главная польза каждого отдельного шаблона состоит в том, что он описывает решение целого класса абстрактных проблем. Также тот факт, что каждый шаблон имеет свое имя, облегчает дискуссию об абстрактных структурах данных между разработчиками, так как они могут ссылаться на известные шаблоны. Таким образом, за счёт шаблонов производится унификация терминологии, названий модулей и элементов проекта.

Правильно сформулированный шаблон проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова.

Шаблоны проектирования могут ускорить процесс разработки, предоставляя испытанные, проверенные парадигмы разработки. Эффективное проектирование программного обеспечения требует учета проблем, которые могут проявиться только позднее на стадии реализации. Повторное использование шаблонов проектирования помогает избежать трудноуловимых нюансов, которые могут вызвать серьезные проблемы, и улучшает читабельность кода для программистов и архитекторов, знакомых с шаблонами.

Часто люди понимают только, как применять определенные методы проектирования программного обеспечения для конкретных задач. Эти методы трудно применимы к более широкому кругу проблем. Шаблоны проектирования обеспечивает общие решения, представленные в формате, который не требует специфики, связанной с той или иной проблемой.

Кроме того, шаблоны позволяют разработчикам общаться с помощью хорошо известных понятий. Общие шаблоны проектирования могут быть усовершенствованы со временем, что делает их более надежными, чем специализированные разработки.

Список литературы

1. *Шаблон проектирования.[Электронный ресурс] – Режим доступа: http://ru.wikipedia.org/wiki.*
2. *Мартин Фаулер. Шаблоны корпоративных приложений. – М.: «Вильямс», 2009. – 544 с.*
3. *Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: «Питер», 2007. – 366 с.*
4. *Джошуа Кериевски Рефакторинг с использованием шаблонов (паттернов проектирования) – М.: «Вильямс», 2006. – 400 с.*
5. *Марк Гранд Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML – М.: «Новое знание», 2004. – 560 с.*
6. *Крэг Ларман Применение UML 2.0 и шаблонов проектирования. – М.: «Вильямс», 2006. – 736 с.*
7. *Скотт В. Эмблер, Прамодкумар Дж. Садаладж Рефакторинг баз данных: эволюционное проектирование – М.: «Вильямс», 2007. – 368 с.*
8. *Влиссидес Дж. Применение шаблонов проектирования. Дополнительные штрихи. - М.: Издательский дом "Вильямс", 2003. – 144с.*

Надійшла до редколегії 20.03.2012

Рецензент: д-р техн. наук, проф. С.В. Лістровий, Українська державна академія залізничного транспорту, Харків.

ВИКОРИСТАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ ПРИ РОЗРОБЦІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Ю.Е. Парфьонов, О.В. Щербаков

У статті проаналізовано існуючі шаблони проектування програмного забезпечення, наведена їх класифікація, визначені особливості застосування шаблонів, виділені переваги та недоліки кожного з них. Показана доцільність використання шаблонів проектування для підвищення ефективності процесу розроблення програмного забезпечення. Наведені рекомендації щодо використання шаблонів при розробленні програмного забезпечення.

Ключові слова: *шаблон проектування, клас, об'єкт, програмне забезпечення, процес розробки ПЗ.*

USING DESIGN PATTERNS IN SOFTWARE DEVELOPMENT

Y.E. Parfyonov, O.V. Shcherbakov

The article is devoted to analysis of existing software design patterns. It relates to their classification, particularities their using, highlighting advantages and disadvantages of each one. The expediency of the use of design patterns to improve the software development process was shown. The recommendations on the use of patterns in software development were presented.

Keywords: *design pattern, class, object, software, software development process.*