ООД

Что такое хороший дизайн? По каким критериям его оценивать, и каких правил придерживаться при разработке? Как обеспечить достаточный уровень гибкости, связанности, управляемости, стабильности и понятности кода?

Качество ПО

- «Качество-тот неперывный стимул, которым среда вынуждает нас созидать мир, где мы живем. Полностью. До последней частицы.» *
- Мифы(легенда) –логика(рациональность)
- Поезд(коллективное собрание мифов) – рельсы(качество).
- Качество ПО – стимул труда разработчика, созидающего на основании коллективных «мифов» комфортное мироощущение.

* Роберт Пёрсиг Дзен и искусство ухода за мотоциклом с. 391

Управление качеством разработки ПО

- *Объектом управления качеством* служит процесс создания продукции, в ходе которого под воздействием субъектов управления формируется её качество.
- *Объектом менеджмента качества* (общего руководства качеством) служит не только процесс создания продукции, но и вся остальная деятельность в области качества.
- *Субъектами управления качеством* выступают руководители всех уровней управления, каждый из которых воздействует на процесс создания продукции путём реализации своих функций. Все вместе они образуют субъект управления качеством — *систему управления качеством* (систему менеджмента качества, или кратко — систему качества).

Управление


Воздействие
Субъект управления
Объект управления
Обратная связь

Паттерны программирования (локальные мифы)

В процессе разработки программного обеспечения очень часто встречаются одни и те же проблемы и задачи, очень часто они даже не зависят от того, какой конкретно язык программирования используется.

Для таких проблем существуют известные способы решения, хорошо зарекомендовавшие и ставшие обычными.

Такие решения и принято называть паттернами программирования (шаблонами проектирования и т. п.).

Паттерн программирования - это независимая от языка стратегия решения проблем при разработке средствами ООП.

О паттернах из книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования»

В общем случае паттерн состоит из четырех основных элементов:

1. ***Имя***. *Сославшись на него, мы можем сразу описать проблему проектирова*ния, ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы.
2. ***Задача***. *Описание того, когда следует применять паттерн. Необходимо сфор*мулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком дизайне. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.
3. ***Решение***. *Описание элементов дизайна, отношений между ними, функций* каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн – это шаблон, применимый в самых разных ситуациях.
4. ***Результаты*** *– это следствия применения паттерна и разного рода компро*миссы. Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна.

О паттернах

Паттерны проектирования – это не то же самое, что связанные списки или хэштаблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. Но это и не сложные, предметно ориентированные решения для целого приложения или подсистемы.

Здесь под паттернами проектирования понимается **описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте**

Грейди Буч о Книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования»

Любая хорошо структурированная объектно-ориентированная архитектура

изобилует паттернами.

На самом деле, для меня одним из критериев качества объектно - ориентированной системы является то, насколько внимательно разработчики отнеслись к типичным взаимодействиям между участвующими в ней объектами.

Если таким механизмам на этапе проектирования системы уделялось достаточное внимание, то архитектура получается более компактной, простой и понятной, чем в случае, когда наличие паттернов игнорировалось.

Важность паттернов при создании сложных систем давно осознана в других дисциплинах. ……..

В двух словах концепция паттерна проектирования в программировании – это ключ к использованию разработчиками опыта высококвалифицированных коллег.

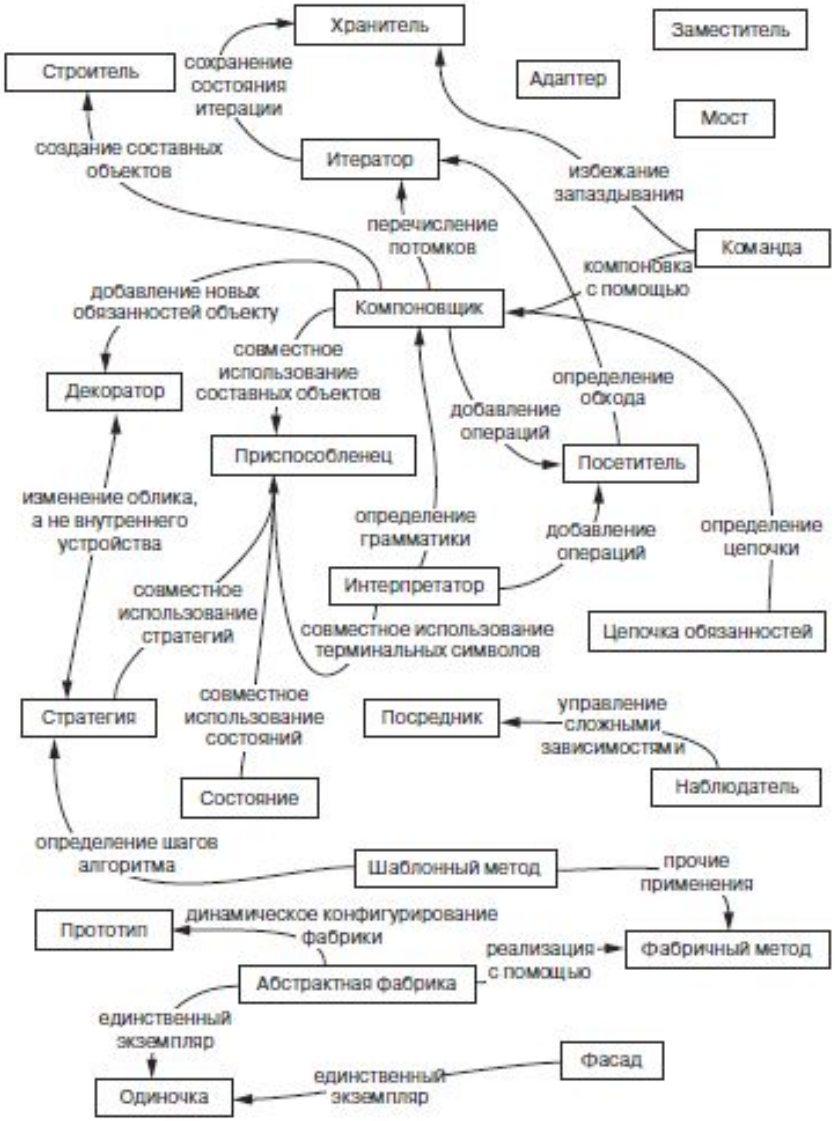
Пространство паттернов проектирования

Цель / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор
			Шаблонный метод
Объект	Абстрактная фабрика	Адаптер (объекта)	Итератор
	Одиночка	Декоратор	Команда
	Прототип	Заместитель	Наблюдатель
	Строитель	Компоновщик	Посетитель
		Мост	Посредник
		Приспособленец	Состояние
		Фасад	Стратегия
			Хранитель
			Цепочка обязанностей

критерий – *уровень – говорит о том, к чему обычно применяется пат*терн: к объектам или классам. Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования, поэтому они статичны, то есть зафиксированы на этапе компиляции.

Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения и потому более динамичны. Почти все паттерны в какой -то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что сфокусированы лишь на отношениях

Отношения между паттернами



Как решать задачи проектирования с помощью паттернов

Самая трудная задача в объектно-ориентированном проектировании – разложить систему на объекты.

При решении приходится учитывать множество факторов: инкапсуляцию, глубину детализации, наличие зависимостей, гибкость, производительность, развитие, повторное использование и т.д. и т.п.

Все это влияет на декомпозицию, причем часто противоречивым образом.

Паттерны проектирования помогают выявить не вполне очевидные абстракции и объекты, которые могут их использовать.

Например, объектов, представляющих процесс или алгоритм, в действительности нет, но они являются неотъемлемыми составляющими гибкого дизайна.

Паттерн **стратегия** описывает способ реализации взаимозаменяемых семейств алгоритмов.

Паттерн **состояние** позволяет представить состояние некоторой сущности в виде объекта

Эти объекты редко появляются во время анализа и даже на ранних стадиях проектирования. Работа с ними начинается позже, при попытках сделать дизайн более гибкими пригодным для повторного использования.

Определение степени детализации объекта

Размеры и число объектов могут сильно варьироваться.

Паттерн **фасад** показывает, как представить в виде объекта целые подсистемы, а паттерн **приспособленец** –как поддержать большое число объектов при высокой степени детализации.

Другие паттерны указывают путь к разложению объекта на меньшие подобъекты.

Абстрактная фабрика и строитель описывают объекты, единственной целью которых является создание других объектов, а **посетитель и команда** – объекты, отвечающие за реализацию запроса к другому объекту или группе

Интерфейсы

Паттерны проектирования позволяют определять интерфейсы, задавая их основные элементы и то, какие данные можно передавать через интерфейс.. Хорошим примером в этом отношении является **хранитель**.

Он описывает, как инкапсулировать и сохранить внутреннее состояние объекта таким образом, чтобы в будущем его можно было восстановить точно в таком же состоянии.

Объекты, удовлетворяющие требованиям паттерна **хранитель**, должны определить два интерфейса:

- Один ограниченный, который позволяет клиентам держать у себя и копировать хранители,
- другой привилегированный, которым может пользоваться только сам объект для сохранения и извлечения информации о состоянии их хранителя.

Паттерны проектирования специфицируют также отношения между интерфейсами.

В частности, нередко они содержат требование, что некоторые классы должны иметь схожие интерфейсы, а иногда налагают ограничения на интерфейсы классов.

Так, **декоратор и заместитель** требуют, чтобы интерфейсы объектов этих паттернов были идентичны интерфейсам декорируемых и замещаемых объектов соответственно. Интерфейс объекта, принадлежащего паттерну **посетитель**, должен отражать все классы объектов, с которыми он будет работать

Типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать

- *при создании объекта явно указывается класс. Задание имени класса привя*зывает вас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, создавайте объекты косвенно.

Паттерны проектирования: **абстрактная фабрика, фабричный метод,**

прототип;

- *зависимость от конкретных операций. Задавая конкретную операцию, вы* ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения.

Паттерны проектирования: **цепочка обязанностей, команда**;

Типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать

- *зависимость от аппаратной и программной платформ. Внешние интерфей*

сы операционной системы и интерфейсы прикладных программ (API) раз личны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Даже на «родной» платформе такую программу трудно поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости.

Паттерны проектирования: **абстрактная фабрика, мост;**

- *зависимость от представления или реализации объекта. Если клиент «знае*т», как объект представлен, хранится или реализован, то при изменении

объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений.

Паттерны проектирования: **абстрактная фабрика, мост, хранитель, заместитель;**

- *зависимость от алгоритмов. Во время разработки и последующего исполь*

зования алгоритмы часто расширяются, оптимизируются и заменяются. За висящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, вероятность изменения которых высока, следует изолировать. Паттерны проектирования: **мост, итератор, стратегия, шаблонный метод посетитель;**

Типичные причины

- ***сильная связанность***. *Сильно связанные между собой классы трудно испол*зовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать.

Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабосвязанных систем в паттернах проектирования применяются такие методы,

как абстрактные связи и разбиение на слои.

Паттерны проектирования: **абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель**;

Типичные причины

- *расширение функциональности за счет порождения подклассов. Специализ*ация объекта путем создания подкласса часто оказывается непростым делом.

С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т.д.). Для определения подкласса необходимо так же ясно представлять себе устройство родительского класса

Композиция объектов и делегирование – гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов

Паттерны проектирования: **мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия;**

- *неудобства при изменении классов. Иногда нужно модифицировать класс, но* делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов.

Благодаря паттернам проектирования можно модифицировать классы и при таких условиях.

Паттерны проектирования**: адаптер, декоратор, посетитель.**

Хороший дизайн должен быть SOLID: TOP-5 архитектурных принципов

Что такое хороший дизайн? По каким критериям его оценивать, и каких правил придерживаться при разработке?

Как обеспечить достаточный уровень гибкости, связанности, управляемости, стабильности и понятности кода?

Роберт Мартин составил список, состоящий всего из пяти правил хорошего проектирования, которые известны, как принципы SOLID.

Достичь такой лаконичности удалось, использовав небольшую хитрость: дело в том, что термин SOLID - это аббревиатура, которая в свою очередь состоит из аббревиатур, за каждой из которых прячется целый класс паттернов.

- *S - Не должно существовать более одного мотива для изменения данного класса.*
- *O – Объекты проектирования (классы, функции, модули и т.д.) должны быть открыты для расширения, но закрыты для модификации.*
- *L – Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом.*
- *I – Клиент не должен вынужденно зависеть от элементов интерфейса, которые он не использует.*
- *D - Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

Принципы объектно-ориентированного дизайна (мифология Брюса Эккеля*)

Изящество (качество) всегда вознаграждается - напутствие

Может показаться, что слишком долго искать действительно красивое решение проблемы, но когда вы сможете легко адаптировать его к новой ситуации, избежав долгих часов, дней, а то и месяцев борьбы с собственным кодом, вы будете вознаграждены(даже если со стороны это будет незаметно).

Это позволит вам создать программу, которую легко не только скомпилировать и отладить, но и понимать и изменять, что, собственно, и составляет коммерческую ценность.

Понимание этого требует некоторого опыта, поскольку, пока вы делаете фрагмент кода элегантным, кажется, что вы мало продуктивны.

Не поддавайтесь спешке и суете, они только замедлят вашу работу.

*Брюс Эккель **«Thinking in Java.»**.

1 Сначала заставь работать, потом ускоряй

Это верно, даже если вы уверены, что фрагмент кода действительно важен и будет основным узким местом в вашей системе.

Не торопитесь.

Сперва заставьте систему работать с настолько простым решением задачи, насколько это возможно.

Уже потом, если решение оказалось не достаточно быстрым, профилируйте его.

Почти всегда вы обнаружите, что ваше мнимое узкое место не проблема.

Потратьте время на более важные вещи.

2 Помните принцип «Разделяй и властвуй»

Если проблема, которой вы занимаетесь, слишком сложна, попробуйте вообразить, как должна работать программа, если некий черный ящик скроет все сложности.

Этот черный ящик — объект.

Напишите сначала код, который использует объект, а потом рассмотрите проблемный объект еще раз и инкапсулируйте его сложности в другие объекты.

3 Отделите создателя класса, от его пользователя(программиста-клиента)

Пользователь класса — это своего рода «покупатель», и ему не интересно, что происходит внутри класса.

Создатель класса должен быть экспертом в своем деле и создавать код так, чтобы даже использование его начинающим программистом, было работоспособным и эффективным.

Библиотеку использовать легко, только если способ ее использования прозрачен.

4 Когда создаете класс, постарайтесь использовать такие имена, чтобы комментарии не понадобились.

Ваша цель сделать программный интерфейс модуля концептуально простым.

Для этого используйте, по возможности, перегрузку методов.

5 Анализ и дизайн должны определить, как минимум, классы вашей системы, их открытые интерфейсы и их отношения с другими классами, в особенности — базовыми.

Если ваш способ разработки производит более того, спросите себя, все ли произведенное имеет значение на протяжении жизненного цикла программы.

Если нет, поддержка лишнего влетит вам в копеечку.

Участники групп разработки стараются не поддерживать ничего, что не способствует их продуктивности.

Фактически, многие методологии разработки этого не учитывают.

6 Автоматизируйте все

Пишите тестовый код первым(до написания самого класса) и храните его вместе с классом.

Автоматизируйте прогон тестов вместе со сборкой. Возможно, вы выберете для этого Ant — стандарт дефакто для сборки Java программ.

Таким образом, все изменения будут автоматически проверены прогоном тестового кода и вы немедленно обнаружите ошибки.

Благодаря сети безопасности вашей тестовой оболочки, вы будете более уверенно вносить изменения, когда обнаружится такая необходимость.

Помните, что величайшие улучшения в языках произошли именно из встроенных тестов, предоставляемых контролем типов, системой исключений и пр., но только до сих пор.

Остальной путь вы должны проделать сами, создавая надежную систему исполняющую тесты, которые проверят основные специфические свойства вашего класса или программы.

7 Пишите тестовый код первым(прежде чем написать класс) для того, чтобы проверить, что разработка класса завершена.

Если вы не способны написать тестовый код, вы не знаете, на что похож ваш класс.

Вдобавок, написание тестового кода часто позволяет выявить все дополнительные свойства и ограничения, которые вы могли бы заложить в класс.

Эти свойства и ограничения не всегда появляются на стадии анализа и разработки.

Кроме того, тесты представляют собой пример кода, показывающий, как следует использовать ваш класс.

8 Все проблемы разработки ПО могут быть упрощены добавлением дополнительного уровня концептуального обобщения.

Это фундаментальное правило разработки ПО является основой абстракции, -**наиважнейшей** особенности объектно-ориентированного программирования.

9 Обобщение должно иметь значение

Это значение может быть столь простым как «совмещение часто используемого кода в одном методе».

Если вы добавляете уровни обобщения (абстракции, инкапсуляции и пр.), которые не имеют значения, это столь же плохо, как не иметь их вообще.

+……..

Литература

- **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. —СПб.: Питер, 2015. — 368 с.: ил.ISBN 978-5-496-00389-6
- **Роберт Пёрсиг** Дзен и искусство ухода за мотоциклом
- Брюс Эккель **«Thinking in Java»**. Издательство – Питер – 2015
- **https://refactoring.guru/ru/design-patterns**