**8. Фасад. Facade**

Назначение: Предоставляет для подсистемы укрупненный унифицированный интерфейс вместо множества интерфейсов.

Классификация: паттерн, структурирующий объекты.

Мотивация: Имеется подсистема, в которой несколько классов являются интерфейсными для внешней системы, т.о. наблюдается сильная связанность. Как в данном случае уменьшить степень связанности?



Структура:



Участники:

Class1-3 – классы, получающие сервис у классов подсистемы.

SSClass1-3 – классы, предоставляющие сервисы подсистемы.

SSClass4-5 – внутренние классы подсистемы

Facade – объект, инкапсулирующий все интерфейсы подсистемы и таким образом скрывающий ее содержимое.

Отношения: интерфейсные операции Operation классов SSClass1-3 переносятся в класс Facade (под именами Operation1-3), и классы-клиенты Class1-3 обращаются к этим операциям. Эти классы не имеют прямого доступа к классам подсистемы. Facade, по сути, переадресует Operation1-3 в вызовы соответствующих операций у классов SSClass1-3.

Применимость: используйте Фасад, чтобы предоставить простой интерфейс к сложной подсистеме и чтобы уменьшить количество интерфейсных объектов в подсистеме.

Результаты:

+ уменьшается степень связанности, увеличивается возможность повторного использования подсистем

- уменьшается зацепление самого класса Facade.

**10. Мост. Bridge**

Мотивация: Предоставить возможность определения нескольких реализаций (например, для различных платформ) для каждого класса большой иерархии. Например, для системы



Понадобится реализация классов с использованием другого контекста устройства (в другой ОС) или другой способ вывода на экран (HTML). Тогда придется создавать объекты HTML-Гиперссылка и т.д. Унаследованные от нового класса HTML-Кнопка, который по другому реализует операции DrawText(), DrawLine(), DrawImage(). Для третьей разновидности придется создать третью ветку иерархии и т.д. Решение:



В этом случае под каждую новую реализацию всей иерархии требуется создать всего один объект HTML-Реализация Кнопки.

Структура:



Участники:

Abstraction – абстракция – определяет интерфейс абстракции, хранит ссылку Imp на объект Implementor.

Concretizing – конкретизация – расширяет интерфейс, определенный абстракцией, реализует операции, обращающиеся к Operation().

Implementor – реализатор – определяет интерфейс для классов реализации контекстно-зависимой операции Operation() – OperationImp().

Concrete Implementor 1, 2 – конкретные реализаторы – реализуют операции OperationImp() в соответствии с контекстом.

Мостом называется связь между Abstraction и Implementor

Применимость: используйте паттерн Мост для отделения реализации от интерфейса особенно в случаях большой, постоянно разрастающейся иерархии объектов, имеющих четко выраженный набор базовых операций, а также в случаях, когда реализация может изменяться в будущем (особенно – во время выполнения) или когда одна реализация должна быть разделена между несколькими ветками иерархий классов.

Результаты:

+ дает возможность перенастраивать и динамически переключаться между реализациями

**11. Посредник. Mediator**

Назначение: Уменьшает связанность группы взаимодействующих друг с другом объектов.

Классификация: паттерн поведения объектов.

Мотивация: Как уменьшить связанность группы объектов, активно взаимодействующих друг с другом?



Структура: Паттерн предлагает ввести класс-Посредник, который будет выполнять роль диспетчера сообщений между этими классами-Коллегами.



Mediator – объект, на который возлагается обязанность диспетчеризации сообщений между объектами классов-Коллег.

Классы-Коллеги видят только Посредника, он их также видит все.

Это простейший вариант. Обобщенный подход представлен на следующей диаграмме:



Применимость: Посредник следует применять в следующих случаях:

1. имеется группа взаимодействующих между собой объектов, связи между которыми плохо структурируются;
2. затрудняется повторное использование объекта из-за того, что он связан со многими другими объектами;
3. обязанность, распределенная между несколькими объектами должна подвергаться настройке без порождения множества подклассов.

Очень эффективно использовать Mediator для интерфейсных элементов. Например, для реализации контроллера слоя интерфейса пользователя. В отличие от контроллера слоя предметной области или управления данными, здесь не требуется однонаправленности протокола передачи сообщений (т.к. это – самый верхний слой), а требуется доступ ко всем интерфейсным элементам (окнам).

Результаты:

+ устраняется связанность между классами-коллегами

+ предоставляет способ отделения подсистем, кооперации объектов и объявления протокола их взаимодействия

+ централизует управление группой

- как правило, низкая степень зацепления класса Mediator.

**12. Шаблонный метод. Template Method**

Назначение: Определяет основу алгоритма операции и позволяет подклассам переопределить некоторые шаги, не изменяя структуру алгоритма в целом.

Классификация: паттерн поведения классов.

Другие названия:

Мотивация: Ситуация: поведение классов одной иерархии при выполнении определенной операции в целом похожи между собой, но отличаются в деталях. Как повторить похожее поведение для всех классов иерархии и избежать дублирования кода?

Структура:



AbstractClass – абстрактный класс – определяет абстрактные примитивные операции (PrimitiveOperation), которые являются частью алгоритма операции TemplateMethod и могут быть переопределены в потомках. Все неизменные части алгоритма (в т.ч. – порядок исполнения и вызов примитивных операций) реализуются непосредственно в методе TemplateMethod.

ConcreteClass – конкретный класс – реализует (переопределяет) примитивные операции.

Примитивные операции обязательно являются виртуальными методами, а шаблонный метод – как правило, статический

Применимость: Этот паттерн следует использовать в следующих случаях:

1. чтобы однократно использовать инвариантные части алгоритма;
2. для выделения и локализации в одном классе поведения, общего для всех подклассов;
3. для управления расширением подклассов (примитивные операции – те места, которые разработчик может менять, остальное – нельзя).

Шаблонные методы могут явиться альтернативой вызову метода предка в замещенном методе потомка (здесь всё происходит наоборот).

Результаты:

+ уменьшение дублирования кода;

+ увеличение зацепления (см. п.2 Применимости – за счет обобщения поведения потомков).

**15. Декоратор. Decorator**

Назначение: Динамически добавляет объекту новые обязанности. Является альтернативой порождению классов с целью расширения функциональности.

Классификация: паттерн, структурирующий объекты

Другие названия: Обертка (Wrapper)

Мотивация: Представим ситуацию, что для некоторой иерархии визуальных компонентов требуется, чтобы в некоторых классах присутствовали дополнительные элементы (скроллер, рамка), причем наследование не является характерным признаком такой функциональности (например, она может добавляться и убираться во время выполнения программы). Например, варианты для объекта TextView: с рамкой – BorderedTextView, со скроллером – ScrollableTextView, с рамкой скроллируемый – BorderedScrollabelTextView.

Паттерн Декоратор предлагает ввести специальные объекты-декораторы – один только рисующий рамку, другой – только скроллирующий, унаследованные от верхнего предка иерархии визуальных компонентов. Необходимое сочетание функциональности достигается путем вложения компонентов друг в друга:

BorderDecorator -> ScrollDecorator -> TextView

Структура:



Component – компонент – определяет интерфейс объектов, на которые динамически могут быть возложены дополнительные обязанности.

Concrete Component – конкретный компонент – определяет класс конкретных объектов, на которые могут быть возложены дополнительные обязанности.

Decorator – декоратор – абстрактный класс, определяющий ссылку на вложенный на него объект Component. Реализует операцию Operation() в виде прямого вызова этой же операции у вложенного компонента.

Concrete Decorator - конкретный декоратор – непосредственно выполняет дополнительные обязанности, возложенные на компонент (внутри Operation).

Применимость: используйте паттерн Декоратор:

1) для динамического добавления или снятия обязанностей объектам;

2) когда расширение путем порождения подклассов по каким либо причинам невозможно (например, возможны комбинации сочетания доп. обязанностей).

Результаты:

+ большая гибкость по сравнению со статическим наследованием (динамика, комбинации)

+ облегчает классы верхнего уровня иерархии

- не во всех случаях интерфейсы декоратора и компонента могут совпадать

- может порождаться множество мелких объектов со сложными связями.

**17. Заместитель. Proxy**

Назначение: контролирует доступ к объекту, подменяя его поведение в случае необходимости.

Классификация: паттерн, структурирующий объекты.

Другие названия: Суррогат (Surrogate).

Мотивация:

Если требуется выполнять некоторые действия до того как потребуется «тяжелая» работа объекта. Например, при загрузке изображений в текстовый документ. В этом случае необходимость действительно прогрузить весь растр и применить к нему форматирование (масштаб, яркость/контраст и т.п.) появляется лишь в том случае, когда рисунок становится видимым. В остальных случаях достаточно знать размеры, которые занимает рисунок в тексте. В этом случае имеет смысл использовать заместителя – объект, который по интерфейсу ничем не отличается от рисунка, но откладывает выполнение некоторых операций «на потом».



Здесь отражена только отложенная загрузка (Lazy Load). Если требуется выполнять форматирование, то операции Size и Draw у заместителя необходимо доопределить.

Структура:



Client – клиент – объект, которому требуется «отложенное» выполнение операции Operation() реального субъекта. Имеет ссылку только на заместителя субъекта и вызывает его метод Operation().

AbstractSubject – абстрактный субъект – определяет общий интерфейс для реального субъекта и его заместителя.

RealSubject – реальный субъект, имеющий «тяжелую» операцию Operation().

SubjectProxy – заместитель субъекта – выполняет операцию Operation() в облегченном виде и лишь при необходимости обращается к «тяжёлой» операции реального субъекта.

Особенности реализации.

1. Заместитель может работать с реальным субъектом через абстрактный интерфейс – позволяет образовывать иерархии субъектов, имеющих один класс заместителей.

2. Заместителей удобно использовать для организации «объектов-заглушек».

Применимость: используйте паттерн Заместитель во всех случаях, когда требуется более «изощренное» использование объекта, чем прямой указатель на него. Типичные варианты использования Заместителя:

1. Удаленный заместитель – предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве (COM).

2. Виртуальный заместитель – предназначен для создания «тяжелых» объектов по требованию.

3. Защищающий заместитель – контролирует доступ к субъекту.

4. «Умная ссылка» – позволяет выполнить дополнительные действия к действиям реального субъекта. Действия могут быть необходимы как для клиента (Декоратор), так и для других объектов (системные действия – подсчет ссылок на субъект, блокировка субъекта, помещение субъекта в кэш).

Результаты:

+ оптимизирует работу с «тяжелыми» объектами

+ позволяет защитить реальный субъект, а также выполнить дополнительные действия;

**18. Компоновщик. Composite**

Назначение: компонует объекты в древовидные структуры, позволяет клиентам единообразно трактовать простые и составные объекты.

Классификация: паттерн, структурирующий объекты.

Мотивация: Как спроектировать систему таким образом, чтобы примитивные и составные объекты обрабатывались единым образом?



(здесь операция Draw() – выполняет одинаковые обязанности и у Graphic и у Panel).

Решение – ввести составной объект в иерархию простых объектов:



Структура:



Client – клиент – обращается к операции Operation() простых и составных объектов единым образом.

Component – компонент – объявляет общий интерфейс для простых и составных объектов (операция Operation() ), также объявляет единый интерфейс для управления вложенными объектами (операции Add, Remove, GetChild), причем для простых объектов эти операции остаются нереализованными (ничего не делают, возвращают пустые указатели).

Leaf – простой элемент (лист) – реализует операцию Operation() в соответствии со своим назначением.

Composite – составной объект – может содержать в себе другие экземпляры классов иерархии Component (включая такие же составные объекты), реализует операции работы со списком (Add, Remove, GetChild и др.), реализует операцию Operation() таким образом, что кроме собственных действий композита как объекта в цикле вызывается Operation() для каждого вложенного объекта.

Особенности реализации.

1) Рекомендуется определить в классе Component и активно использовать явные ссылки на владельца. Это упростит выполнение многих операций, в т.ч. – обход древовидной структуры.

2) Интерфейсный класс Component должен включать максимальное количество операций, чтобы избавить клиента от необходимости знать конкретный класс используемого объекта.

Применимость: используйте паттерн Компоновщик, когда необходимо, чтобы клиенты единообразно трактовали простые и составные объекты, а также чтобы в составе одной иерархии классов объединить объекты, связанные связью типа «часть-целое».

Результаты:

+ упрощает код клиента (за счет единообразия простых и составных объектов);

+ облегчает добавление новых подклассов в иерархию;

- не позволяет накладывать прямые ограничения на класс вложенных объектов.

Итератор

Iterator i = A.getIterator();

i.First();

while (!i.isDone())

{

Item x = i.getCurrent();

x.operation();

i.Next();

}