# Использование паттерна Стратегия(Алгоритм).

В тот самый момент, когда вы, известнейший специалист по паттернам проектирования, заходите в зал заседаний корпорации «МегаГигаКо», генеральный директор и члены совета директоров празднуют свой контракт на проектирование новой серии автомобилей в самом спокойном виде, каком только можно было бы себе это представить – высокими ударами ладоней, беготнёй по залу и криками на всю катушку.

«Этот контракт сулит огромный доход для нашей компании!», выкрикивает директор, слегка расплескав шампанское на стол из-за избытка эмоций. «Всё что нам нужно сделать, это убедиться, что мы понимаем процесс проектирования ВЕРНО.» Он поворачивается к проектору, и когда на стене появляются несколько больших диаграмм, директор продолжает: «Итак, вот моя идея…»

«Неверно понимаете», громко говорите вы.

Директор «МегаГигаКо» выглядит удивлённым и пытается продолжить: «Но если мы…»

«Нет», говорите вы, качая головой.

«Что…»

«Извините», киваете вы директору и его прекрасным диаграммам: «это очевидно, что вы рискуете всем своим контрактом, потому что делаете всё неправильно. Я вижу у вас дюжину проблем просто посмотрев на вот эту диаграмму.»

Совет директоров недовольно шумит, и директор переходит в наступление: «Что насчёт вашей нескромной персоны?»

«Я и есть профессионал паттернов проектирования, который решит все ваши большие проблемы проектирования», сделав эффектную паузу, вы продолжаете: «за большую плату, разумеется».

Директор поспешно записывает предварительную цифру оплаты ваших трудов, которая, несмотря на свою величину, всё же не кажется вам достаточно большой.

«И опять неверно понимаете», произносите вы.

Директор смотрит на вас, приподняв брови.

«Паттерны проектирования», терпеливо объясняете вы, «представляют собой решения известных проблем программирования. Но не только это, они также представляют собой хорошую программистскую практику, которая позволяет гораздо легче поддерживать и дополнять ваш код. Так что, как вы можете заметить, привлечь такого эксперта как я имеет большой смысл – ибо когда я вижу проблему, которая уже решена с помощью паттерна проектирования, я могу рассказать о ней всё.

«Ну», осторожно говорят программисты компании «МегаГигаКо», «идея паттернов выглядит хорошо, но мы уже пользуемся методами объектно-ориентированного программирования в наших разработках. Разве это не покрывает наши проблемы?»

«Нет», говорите вы в очередной раз. На самом деле, это один из моментов, лежащих в основе паттернов проектирования – они расширяют объектно-ориентированное программирование.

# Композиция(агрегация) против наследования: первая попытка разработки новых машин.

Пресловутые программисты корпорации «МегаГигаКо» знают о наследовании, и они начали разработку новых машин, наплевав на ваши предупреждения подождать встречи с вами. Предполагалось, что они разработают серию машин, так что они стартовали с создания базового класса Vehicle и метода с названием go, который отображает текст «Сейчас я еду.».

public abstract class Vehicle

{

public Vehicle()

{

}

public void go()

{

System.out.println(“Сейчас я еду.”);

}

}

Затем они создали ещё классы, такие как StreetRacer, используя класс Vehicle как базовый класс и применив наследование:

public class StreetRacer extends Vehicle

{

public StreetRacer()

{

}

}

Всё идёт по плану. Если создать класс StreetRacer и запустить его вот так:

public static void main(String[] args)

{

StreetRacer streetRacer = new StreetRacer();

streetRacer.go();

.

.

.

}

То мы увидим на экране:

Сейчас я еду.

Неплохо выглядит. Так здорово, что корпорация «МегаГигаКо» решила следовать этим курсом и сконструировала болид «Формулы-1», который также наследует класс Vehicle как вы можете увидеть ниже:

public class FormulaOne extends Vehicle

{

public FormulaOne()

{

}

}

И вы можете запустить и Стритрейсера и болид Формулы-1 одновременно вот так:

public static void main(String[] args)

{

StreetRacer streetRacer = new StreetRacer();

FormulaOne formulaOne = new FormulaOne();

streetRacer.go();

formulaOne.go();

.

.

.

}

И мы получаем:

Сейчас я еду.

Сейчас я еду.

«Неплохо», говорит директор совместно с советом директоров. «Кому нужны паттерны проектирования?», спрашивают они, бросая на вас невежливые взгляды. Но затем они получают контракт на производство вертолётов. Вертолёты, замечают они, это просто ещё одна разновидность машин. Так что они создают вертолёты, используя класс Helicopter, расширяя класс Vehicle:

public class Helicopter extends Vehicle

{

public Helicopter()

{

}

}

Но тут появляется проблема. Если запустить вертолёт наряду со стритрейсером и болидом формулы-1 вот так:

public static void main(String[] args)

{

StreetRacer streetRacer = new StreetRacer();

FormulaOne formulaOne = new FormulaOne();

Helicopter helicopter = new Helicopter();

streetRacer.go();

formulaOne.go();

helicopter.go();

.

.

.

}

То мы получим по сообщению от каждого аппарата.

Сейчас я еду.

Сейчас я еду.

Сейчас я еду.

Это выглядит как-то неправильно, с сомнением в голосе произносит директор. Почему вертолёт должен «ехать»? Разве он не должен «лететь»? И проблема становится ещё больше, когда «МегаГигаКо» получает контракт на производство самолётов, которые опять расширяют базовый класс Vehicle.

public class Jet extends Vehicle

{

public Jet()

{

}

}

Соответственно, при запуске всех аппаратов, мы получаем:

Сейчас я еду.

Сейчас я еду.

Сейчас я еду.

Сейчас я еду.

«Вот это точно полная ерунда», произносит директор, «самолёты не едут, они в воздухе. Они летают – и причём весьма быстро».

«Нет проблем», говорят программисты корпорации. «Мы можем переопределить метод go в классах Helicopter и Jet чтобы дать им соответствующее поведение. И они создают что-то вроде вот такого, что заставит вертолёт летать, а не ехать:

public class Helicopter extends Vehicle

{

public Helicopter()

{

}

public void go()

{

System.out.println(“Сейчас я лечу.”);

}

}

«Окей», говорит директор, «но совет директоров уже проголосовал за изменение сообщения на следующую неделю с «Сейчас я лечу.» на «Сейчас я лечу со скоростью 200 миль в час.». И, насколько я знаю этих ребят, изменений ещё будет много.»

Однако с этим проблема, так как ваши программисты распространяют этот способ для одного класса — ехать на машине, лететь на вертолёте — на все остальные поколения классов. Это не то чтобы большая проблема, но, если вы хотите справиться с этой загвоздкой, которая будет происходить довольно часто, как и в данном случае, в дальнейшем приведёт к очевидным проблемам с поддержкой кодовой базы.

Вы говорите, что возможно наследование — это не решение в данном случае, когда вам приходится распространять изменения одной задачи в течении нескольких поколений классов. Вам придётся поддерживать много специфического кода в поколениях классов при изменении поставленной задачи. И поскольку все производные классы будут становиться все более объёмными и сложными, поддерживать изменения в них будет очень сложно. Вам придётся обновлять метод go бесконечно. Проблема, которую вы пытаетесь решить, заключается в том, как избежать распространения обработки конкретной изменчивой задачи над несколькими поколениями классов. И, поскольку вам не избежать этого, вам придётся редактировать прорву файлов кода.

Возможно есть куда лучший способ справиться с методом передвижения транспортных средств, чем использование наследования в этом случае. «Эй», встревает в разговор главный программист, «как насчёт использовать интерфейсы вместо наследования? Вы можете создать интерфейс IFly с методом go так, чтобы, например, класс «Helicopter» реализовывал метод go, посмотрим ниже»:

public class Helicopter implements IFly

{

public Helicopter()

{

}

public void go()

{

System.out.println(“Now I’m flying.”);

}

}

«Ничего хорошего», говорите вы программисту, «вы не решили проблему вообще — каждый класс и каждый подкласс всё ещё должен реализовывать свою собственную версию метода go, что также плохо, как и было с наследованием. И, поскольку интерфейсы не содержат реализаций, вам всё равно придётся писать собственный код в каждом классе, что заставляет «повторное использование кода» выйти в окошко.

# Обеспечивайте изменения с помощью композиции (глагол «иметь»), нежели с помощью наследования (глагол «являться»).

Всё вокруг меняется. В коммерческой разработке вещи меняются быстро, поэтому стоит планировать изменения. Если перед вами стоит небольшая проблема, требующая небольшого решения, вам, вероятно, не придётся планировать значительных изменений. Но если вы работаете над серьёзным проектом с крупным размером кодовой базы и продолжительное время, вы должны начать думать с точки зрения изменений. Требования, которым ваш код должен соответствовать, будут варьироваться от раза к разу, и вам придётся модифицировать код так, чтобы он был готов к будущим изменениям. Многие разработчики не осознают потенциальную возможность кода меняться, и они неизменно пожалеют об этом позже. Насколько большим должен быть проект, чтобы вам следовало бы изящно закладывать изменения? Это на самом деле субъективная оценка, часть искусства программирования. Когда начнёшь понимать, как справляться с постоянными изменениями, тогда и начнёшь понимать, когда их нужно допускать в коде. Это и есть та самая проницательность в проектировании, которую вы уже, возможно, подмечали: делим код на те части, которые подвержены изменениям и на остальной код, и стараемся делать их независимыми друг от друга для удобства дальнейшей поддержки. Также следовало бы всегда пытаться делать эти части повторно используемыми.

Вот почему так важно планирование изменений, и, также, вот почему наследование частенько не может справиться с ними в полной мере. Наследование предполагает отношения между базовыми и наследными классами в духе «являться». (Т.е. это можно прочитать таким образом, что наследный класс как бы является базовым.) Вертолёт(Helicopter) «является» Машиной(Vehicle), что автоматически означает, что класс Helicopter наследует поведение класса Vehicle. Базовый класс обрабатывает определённую задачу одним образом, но затем производный класс меняет это поведение, а затем следующий производный класс ещё добавит изменений. Так что в итоге вам потребуется справиться с изменением во всех последующих генерациях наследных классов.

С другой стороны, вы можете выделить наиболее изменчивые части кода и инкапсулировать их как объекты, по мере необходимости. В итоге вся задача будет сводиться к манипулированию этими выделенными объектами, а не цепочками поколений классов. Используя композицию, легко выбрать конкретный объект, который сейчас требуется, вместо того чтобы иметь жёсткую структуру со скрытой внутри всей наследной цепочки реализацией. Это даёт более удобное отношение «имеет» (а не «является). Стритрейсер(StreetRacer) имеет определённый способ передвижения, инкапсулированный в объект; Вертолёт(Helicopter) имеет другой способ передвижения, и это опять же отражается отдельной способностью, инкапсулированной в объект. И каждый выполняет свою задачу.

Как идея разделения изменяющегося кода работала бы в случае с Vehicle/StreetRacer/Helicopter в этой главе? По словам директора «МегаГигаКо», часть, которая будет меняться больше всего, - это метод go, так что эту часть и стоит отделить от остального кода. В терминах паттернов проектирования каждая реализация метода go называется «алгоритм» (это и есть немного другое именование паттерна «стратегия»). Так что стоило бы создать множество алгоритмов(стратегий), которые бы могли использоваться различными объектами, как например: StreetRacer, FormulaOne, Helicopter и Jet. Каждый алгоритм работает с одной полной задачей, поэтому не необходимости распространять обработку этой задачи на цепочку поколений классов.

# Создаём алгоритмы

Чтобы быть уверенными в том, что все алгоритмы реализуют заданные методы (в данном случае один метод – go), нужно создать интерфейс, назовём его GoAlgorithm, который будут реализовывать все алгоритмы.

public interface GoAlgorithm

{

public void go();

}

Интерфейс GoAlgorithm имеет один метод: go. Чтобы убедиться, что любой алгоритм может быть использован любой машиной (Vehicle), все алгоритмы должны реализовывать этот интерфейс, что означает – они все должны определять метод go. Первый алгоритм, GoByDrivingAlgorithm, отображает «Сейчас я еду.». Вот как этот алгоритм выглядит:

public class GoByDrivingAlgorithm implements GoAlgorithm

{

public void go()

{

System.out.println(“Сейчас я еду.”);

}

}

В то же самое время алгоритм GoByFlying будет отображать «Сейчас я лечу.».

public class GoByFlying implements GoAlgorithm

{

public void go() {

System.out.println(“Сейчас я лечу.”);

}

}

И, в свою очередь, алгоритм GoByFlyingFast, используемый самолётами, отображает «Сейчас я лечу быстро.»

public class GoByFlyingFast implements GoAlgorithm

{

public void go()

{

System.out.println(“Сейчас я лечу быстро.”);

}

}

Отлично! Мы только что отделили алгоритмы от кода. Мы начали использовать технику «имеет» вместо техники «является». А теперь нам осталось заставить всё это работать.

# Использование алгоритмов.

Теперь у нас есть кое-какие алгоритмы и нам нужно создать объекты с отношениями «имеет» вместо «является». После создания объекта из алгоритма, нам нужно бы как-то его хранить, так что добавим новый метод в класс машины(Vehicle) под названием setGoAlgorithm. Этот метод сохраняет алгоритм, который мы хотим использовать, во внутренней приватной переменной goAlgorithm:

public abstract class Vehicle

{

private GoAlgorithm goAlgorithm;

public Vehicle()

{

}

public void setGoAlgorithm (GoAlgorithm algorithm)

{

goAlgorithm = algorithm;

}

.

.

.

}

Таким образом, если в классе-наследнике нам потребуется использовать какой-либо конкретный алгоритм, всё что нам понадобится сделать – это вызвать метод setGoAlgorithm с параметром в виде конкретного объекта алгоритма, вот так:

setGoAlgorithm(new GoByDrivingAlgorithm());

Метод go класса Vehicle также должен будет поменяться. Раньше он отображал сообщение «Сейчас я еду.» вот так:

public void go()

{

System.out.println(“Сейчас я еду.”);

}

Однако сейчас он должен вызывать метод go определённый алгоритмом, так что вот как всё это выглядит сейчас:

public abstract class Vehicle

{

private GoAlgorithm goAlgorithm;

public Vehicle()

{

}

public void setGoAlgorithm (GoAlgorithm algorithm)

{

goAlgorithm = algorithm;

}

public void go() {

goAlgorithm.go();

}

}

Теперь, всё что нам осталось сделать – выбрать нужный алгоритм для использования соответствующим классом машины.

Например, стритрейсер будет использовать GoByDrivingAlgorithm:

public class StreetRacer extends Vehicle

{

public StreetRacer()

{

setGoAlgorithm(new GoByDrivingAlgorithm());

}

}

Болид формулы-1 также будет использовать этот алгоритм:

public class FormulaOne extends Vehicle

{

public FormulaOne()

{

setGoAlgorithm(new GoByDrivingAlgorithm());

}

}

Но вертолёт будет использовать алгоритм GoByFlyingAlgorithm:

public class Helicopter extends Vehicle

{

public Helicopter()

{

setGoAlgorithm(new GoByFlyingAlgorithm());

}

}

Пришло время посмотреть на весь код в действии!

Класс StartTheRace создаст все нужный объекты машин и вызовет все методы go.

public class StartTheRace

{

public static void main(String[] args)

{

StreetRacer streetRacer = new StreetRacer();

FormulaOne formulaOne = new FormulaOne();

Helicopter helicopter = new Helicopter();

Jet jet = new Jet();

streetRacer.go();

formulaOne.go();

helicopter.go();

jet.go();

}

}

И вот что мы получаем в итоге:

Сейчас я еду.

Сейчас я еду.

Сейчас я лечу.

Сейчас я лечу быстро.

Прямо то что нужно! Мы использовали «имеет» вместо «является». Теперь любой код может использовать эти алгоритмы, потому что код этих алгоритмов не «похоронен» внутри кода StreetRacer, Helicopter и остальных классов.

Эта техника даёт нам отличную альтернативу использованию наследования.

Если бы мы использовали наследование, то нам пришлось бы обрабатывать поведение по всей цепочке – от базового до всех наследных классов, как в том примере от программистов «МегаГигаКо», где им пришлось переопределять метод go в классе каждой машины.

Используя технику «имеет, а не является», мы может создать прекрасную коллекцию алгоритмов – так много как нам надо – и затем выбирать их по необходимости.

Совет директоров постановил, что теперь сообщение должно поменяться с «Сейчас я лечу.» на «Сейчас я лечу со скоростью 321км/ч.». Это вообще не проблема, просто изменим GoByFlying алгоритм:

public class GoByFlying implements GoAlgorithm

{

public void go() {

System.out.println(“Сейчас я лечу со скоростью 321км/ч.”);

}

}

Весь код, использующий этот алгоритм поменяется автоматически; нет необходимости шерстить все классы. В данном случае, когда вы концентрируетесь на обработке конкретного поведения класса, вы лучше контролируете ситуацию, особенно когда нужно делать много изменений.

# Выбор алгоритмов в режиме выполнения.

«Минутку!», восклицает директор «МегаГигаКо».

«Что-то мне подсказывает, что самолёты не только летают. Перед тем как взлететь, они проезжают какой-то путь. И когда садятся, тоже немного едут по посадочной полосе. Так разве не должно ли их поведение меняться, не должны ли они сначала ехать, потом лететь, а потом ехать снова?»

«Теоретически, да.», вздыхают программисты «МегаГигаКо», «Но это потребует дописать кучу дополнительного кода .»

«Не совсем», встреваете вы, «Особая привлекательность использования алгоритмов как внешних объектов как раз состоит в том, что вы можете менять поведение по ходу выполнения.»

Если бы вы захардкодили поведение машин внутри их классов, то менять действия по ходу выполнения было бы невозможно. Но когда вы используете внешние объекты алгоритмов, которые «имеют» классы машин, всё становится очевидно лёгкой задачей.

Продемонстрируем как алгоритмы способны работать в динамике. Для этого создадим класс RealJet. Чтобы изобразить из самолёта такси, сначала зададим ему алгоритм GoByDrivingAlgorithm как показано ниже:

public class RealJet

{

public static void main(String[] args)

{

Jet jet = new Jet();

jet.setGoAlgorithm(new GoByDrivingAlgorithm());

.

.

.

}

Чтобы заставить самолёт весело гнать по взлётной полосе, используем метод go:

public class RealJet

{

public static void main(String[] args)

{

Jet jet = new Jet();

jet.setGoAlgorithm(new GoByDrivingAlgorithm());

jet.go();

.

.

.

}

Мы может на лету менять алгоритмы, а затем вызывать метод go:

public class RealJet

{

public static void main(String[] args)

{

Jet jet = new Jet();

jet.setGoAlgorithm(new GoByDrivingAlgorithm());

jet.go();

jet.setGoAlgorithm(new GoByFlyingFastAlgorithm());

jet.go();

jet.setGoAlgorithm(new GoByDrivingAlgorithm());

jet.go();

}

}

Вот и всё! Наше самолётное такси отлично едет, летит, затем проезжает при приземлении.

Вообще не проблема:

Сейчас я еду.

Сейчас я лечу быстро.

Сейчас я еду.

Как вы видите, переключать алгоритмы на лету не составляет никаких трудностей. Другими словами, вы можете использовать любую стратегию поведения прямо во время выполнения. Всё это даёт нам паттерн «Стратегия(Алгоритм)».

# Глава 3. Использование паттерна «Декоратор».

По пути на свою новую работу, консультантом по паттернам проектирования(которому платят неприлично большую зарплату) корпорации «ГигантоКомпьютеры», вы решаете заскочить в кафешку.

«Что будете заказывать?», бурчит угрюмый повар, стоящий за гриль-печью.

«Бургер», говорите вы, продолжая бороться с вёртким подносом.

Повар набирает сумму стоимости на кассе, как вдруг вспоминает: «Жареной картошечки добавить?».

«Конечно», быстро отвечаете вы.

Повар очищает набранные цифры на кассе и вводит новую сумму, проговаривая «Бургер с картошечкой».

«А давайте-ка лучше чизбургер!», выпаливаете вы, повинуясь внезапному импульсу.

Повар бросает на вас уничтожающий взгляд и начинает набирать заново, с усилием тыкая пальцами в клавиши и приговаривая: «Бургер с сыром и картошкой. Окей?».

«Мм…», произносите вы, рассматривая вывешенное меню, «Может быть сделаете вот этот вот чизбургер с беконом?».

Повар довольно недружелюбно в упор смотрит на вас, и, кажется, уже на грани высказать вам всё что о вас думает, но повторяет вводить заказ на кассу, начиная весь процесс заново.

«Эй», говорите вы, «Да вы могли бы облегчить себе жизнь, если бы воспользовались паттерном Декоратор!»

«Да, да», машинально отвечает повар, пытаясь догадаться о чём вы говорите: «Тысячу раз говорил им тоже самое…».

Вы радостно хватаете свой беконный чизбургер с картошкой и спрашиваете: «Как насчёт добавить немного нарезанных кружочками помидоров сверху?».