

Легковес(Приспособленец)

Легковес — это структурный паттерн проектирования, который позволяет вместить большее количество объектов в отведённую оперативной память за счёт экономного разделения общего состояния объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Содержание

- Введение
- Проблема
- Решение
- Структура
- Пример

Введение

- **Название и классификация паттерна**

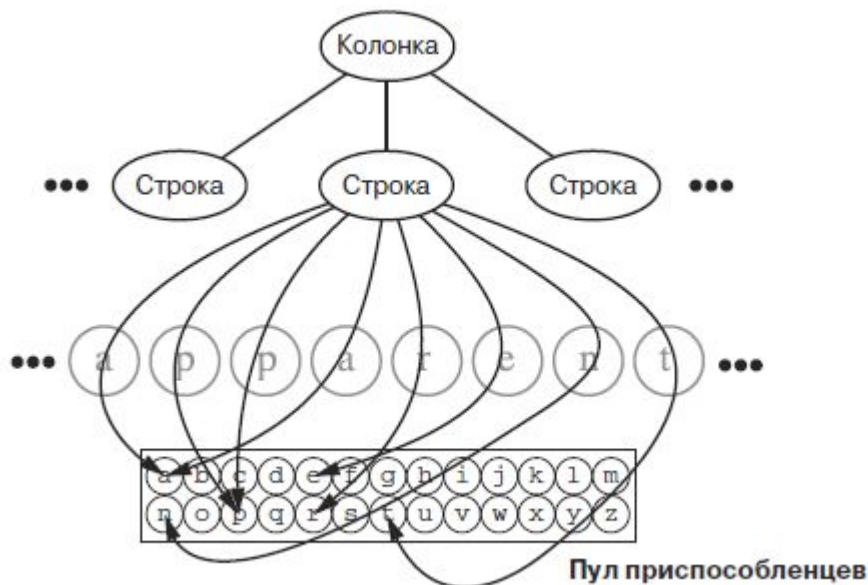
Приспособленец – паттерн, структурирующий объекты.

- **Назначение**

Использует разделение для эффективной поддержки множества мелких объектов

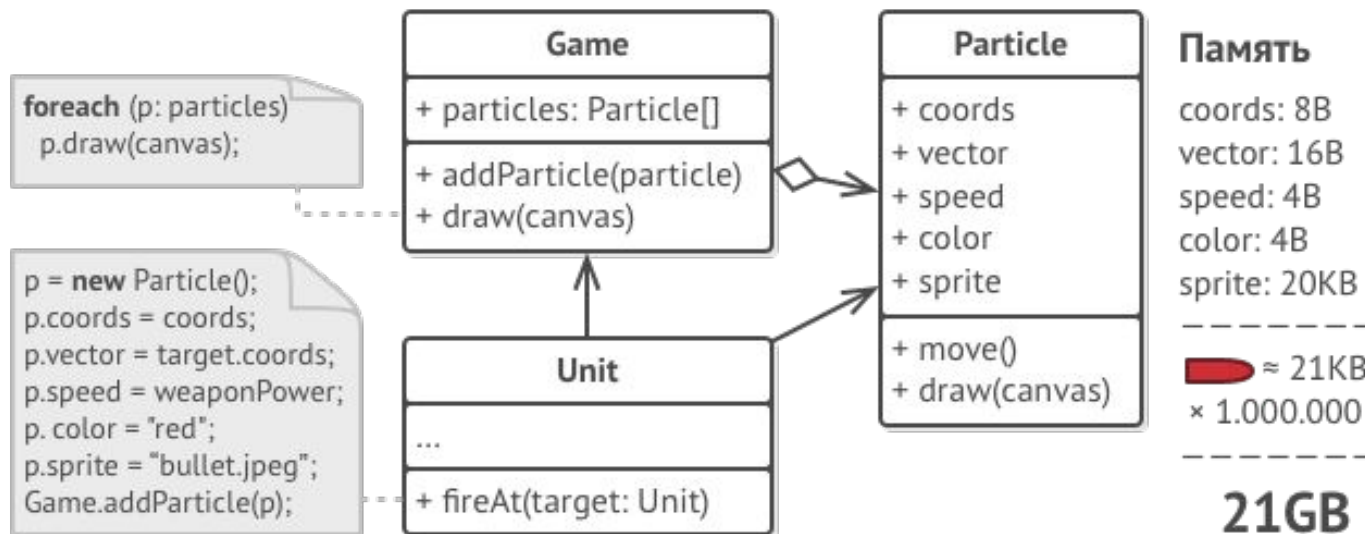
Приспособленцы моделируют концепции или сущности, число которых слишком велико для представления объектами.

Например, редактор документов мог бы создать по одному приспособленцу для каждой буквы алфавита



Проблема

Нехватка ресурсов

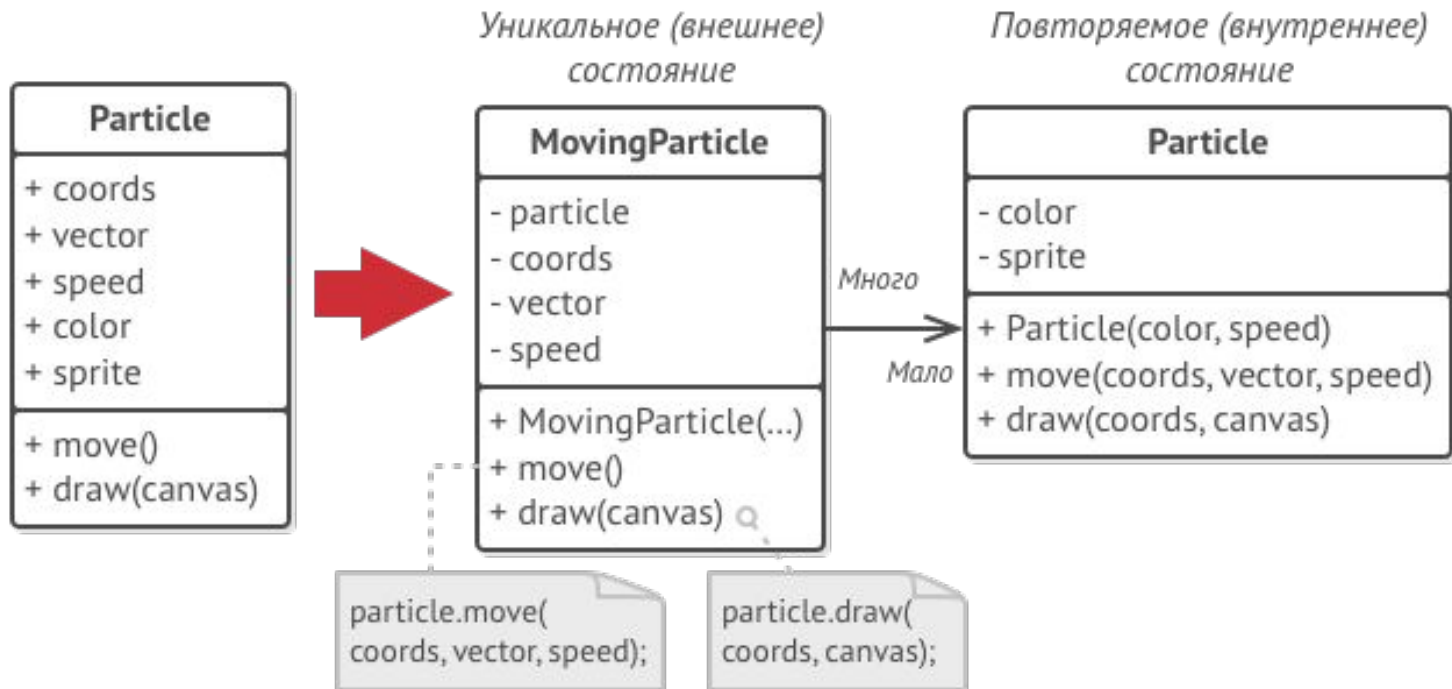


Решение

Если внимательно посмотреть на класс частиц, то можно заметить, что цвет и спрайт занимают больше всего памяти.

Более того, они хранятся в каждом объекте, хотя фактически их значения одинаковые для большинства частиц.

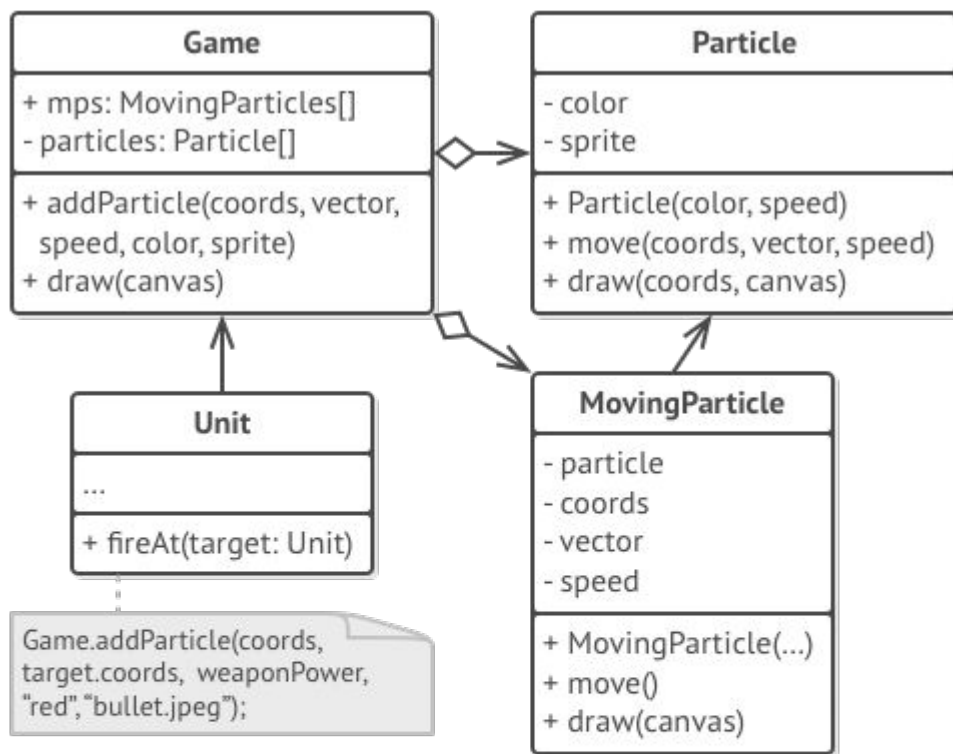
Неизменяемые данные объекта принято называть «внутренним состоянием». Все остальные данные — это «внешнее состояние».







Легковесы

- Паттерн Легковес предлагает не хранить в классе внутреннее состояние, а передавать его в те или иные методы через параметры.
- Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах.
- Но главное, понадобится гораздо меньше объектов, ведь они теперь будут отличаться только внешним состоянием, а оно имеет не так много вариаций.

Эффект легковеса

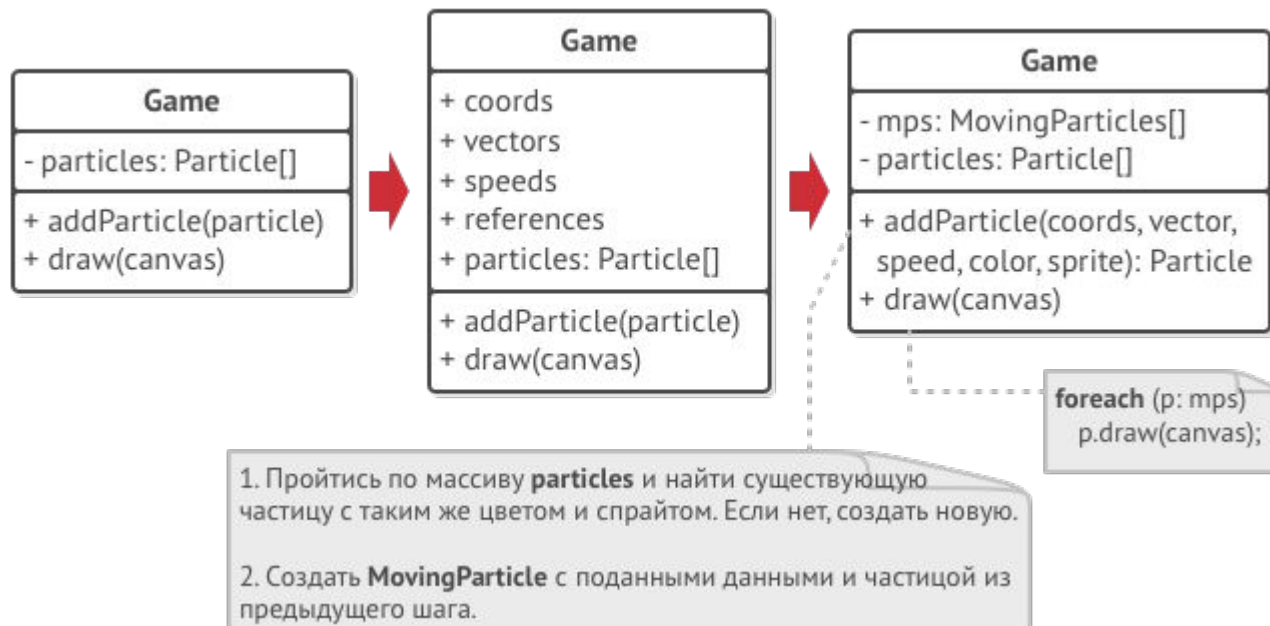


Память	coords: 8B	
	vector: 16B	 × 1
color: 4B	speed: 4B	 × 1.000.000
sprite: 20KB	particle: 4B	
-----	-----	
 ≈ 21KB	 ≈ 32B	32MB

Объекты - контексты

Эlegantным решением было бы создать дополнительный класс-контекст, который связывал внешнее состояние с тем или иным легковесом. Это позволит обойтись только одним полем массивом в классе контейнера.

Объекты-контексты занимают намного меньше места, чем первоначальные. Ведь самые тяжёлые поля остались в легковесах, и сейчас мы будем ссылаться на эти объекты из контекстов вместо того, чтобы хранить дублирующее состояние.



Требования к Легковесам

Неизменяемость Легковесов

Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменять после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора. Он не должен иметь сеттеров и публичных полей.

Фабрика Легковесов

Для удобства работы с легковесами и контекстами можно создать фабричный метод, принимающий в параметрах всё внутреннее (а иногда и внешнее) состояние желаемого объекта.

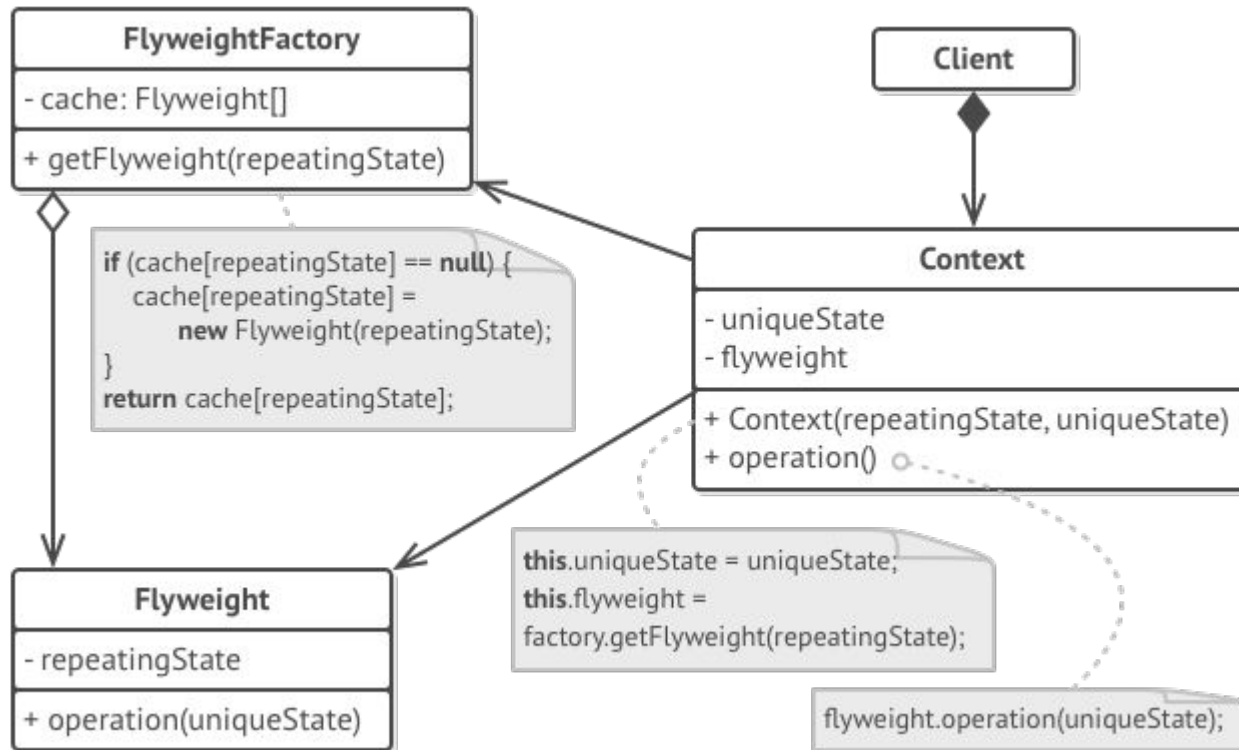
Главная польза от этого метода в том, чтобы искать уже созданные легковесы с таким же внутренним состоянием, что и требуемое. Если легковес находится, его можно повторно использовать. Если нет — просто создаём новый.

Обычно этот метод добавляют в контейнер легковесов либо создают отдельный класс-фабрику. Его даже можно сделать статическим и поместить в класс легковесов.

Структура

1. Легковес применяется в программе, имеющей **громадное** количество одинаковых объектов. Этих объектов было так много, что они не помещались в доступную оперативную память без ухищрений. Паттерн разделил данные этих объектов на две части — контексты и легковесы.
2. **Легковес** содержит состояние, которое повторялось во множестве первоначальных объектов. Один и тот же легковес можно использовать в связке со множеством контекстов. Состояние, которое хранится здесь называется *внутренним*, а то, что он получает извне — *внешним*.
3. **Контекст** содержит "внешнюю" часть состояния, уникальную для каждого объекта. Контекст связан с одним из объектов-легковесов, хранящих оставшееся состояние.
4. Поведение оригинального объекта чаще всего оставляют в Легковесе, передавая значения контекста через параметры методов. Тем не менее поведение можно поместить и в *Контекст*, используя *Легковес* как объект данных.
5. **Клиент** вычисляет или хранит контекст, то есть внешнее состояние *легковесов*. Для клиента *Легковесы* выглядят как шаблонные объекты, которые можно настроить во время использования, передав контекст через параметры.
6. **Фабрика легковесов** управляет созданием и повторным использованием легковесов. Фабрика получает запросы, в которых указано желаемое состояние легковеса. Если легковес с таким состоянием уже создан, фабрика сразу его возвращает, а если нет — создаёт новый объект.

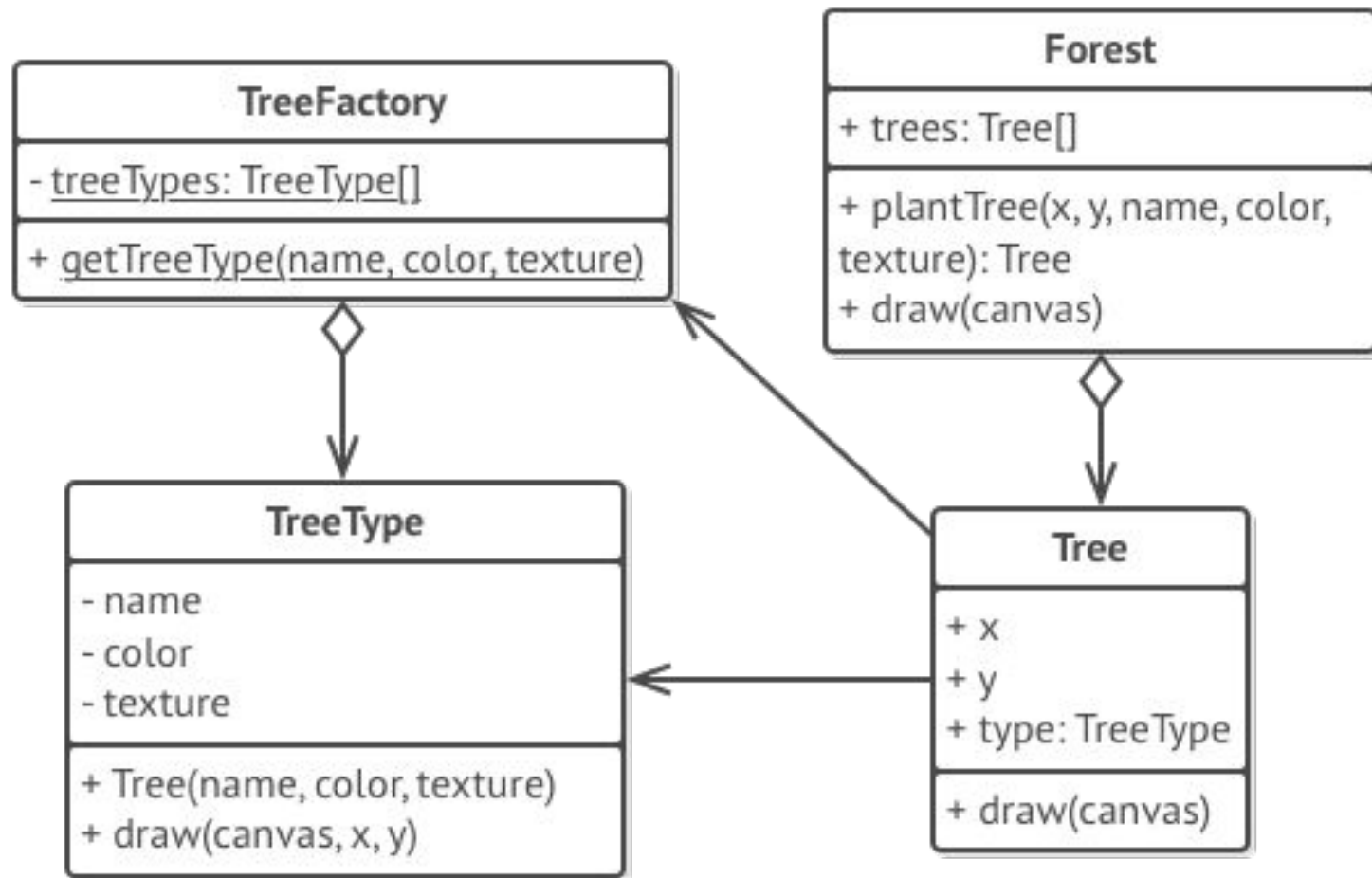
Cxema



Пример

- В этом примере Легковес помогает сэкономить оперативную память при отрисовке на холсте миллионов объектов-деревьев.
- Легковес выделяет повторяющуюся часть состояния из основного класса Tree и помещает его в дополнительный класс TreeType.
- Теперь, вместо хранения повторяющихся данных во всех объектах, отдельные деревья будут ссылаться на несколько общих объектов, хранящих эти данные. Клиент работает с деревьями через фабрику деревьев, которая скрывает от него сложность кеширования общих данных деревьев.
- Таким образом, паттерн Легковес позволяет экономить память в случаях, когда вы имеете дело с миллионами тяжёлых, но слегка похожих друг на друга объектов.

Схема приложения



Класс легковес

// Этот класс-легковес содержит часть полей, которые описывают деревья.
Эти поля
// не уникальные для каждого дерева в отличие, например, от координат —
// несколько деревьев могут иметь ту же текстуру. Поэтому мы переносим
// повторяющиеся данные в один единственный объект и ссылаемся на него
из
// конкретных деревьев.

class TreeType is

field name

field color

field texture

constructor TreeType(name, color, texture) { ... }

method draw(canvas, x, y) **is**

Create a bitmap from type, color **and** texture.

Draw bitmap on canvas at X **and** Y.

Фабрика и контекст

```
// Фабрика легковесов решает когда нужно создать новый легковес, а когда  
    МОЖНО
```

```
// обойтись существующим.
```

```
class TreeFactory is
```

```
    static field treeTypes: collection of tree types
```

```
    static method getTreeType(name, color, texture) is
```

```
        type = treeTypes.find(name, color, texture)
```

```
        if (type == null)
```

```
            type = new TreeType(name, color, texture)
```

```
            treeTypes.add(type)
```

```
        return type
```

```
/ Контекстный объект, из которого мы выделили легковес TreeType. В программе
```

```
// могут быть тысячи объектов Tree, так как накладные расходы на их хранение
```

```
// совсем небольшие — порядка трёх целых чисел (две координаты и ссылка).
```

```
class Tree is
```

```
    field x,y
```

```
    field type: TreeType
```

```
constructor Tree(x, y, type) { ... }
```

```
    method draw(canvas) is
```

```
        type.draw(canvas, this.x, this.y)
```

Класс Forest

// Классы Tree и Forest являются клиентами Легковеса. При желании
ИХ МОЖНО СЛИТЬ

// в один класс, если класс Tree нет нужды расширять далее.

class Forest is

field trees: collection of Trees

method plantTree(x, y, name, color, texture) is

type = TreeFactory.getTreeType(name, color, texture)

tree = new Tree(x, y, type);

trees.add(tree)

method draw(canvas) is

tree.draw(canvas)

foreach tree in trees

Применимость

Когда не хватает оперативной памяти для поддержки всех нужных объектов.

Эффективность паттерна Легковес во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все перечисленные условия:

- в приложении используется большое число объектов;
- из-за этого высоки расходы оперативной памяти;
- большую часть состояния объектов можно вынести за пределы их классов;
- многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено.

Шаги реализации

- Разделите поля класса, который станет легковесом, на две части:
 - внутреннее состояние — значение этих полей одинаковое для большого числа объектов.
 - внешнее состояние (контекст) — значения полей уникальны для каждого объекта.
- Оставьте поля внутреннего состояния в классе, но убедитесь, что их значения неизменяемы. Эти поля должны инициализироваться только через конструктор.
- Превратите поля внешнего состояния в аргументы методов, где эти поля использовались. Затем, удалите поля из класса.
- Создайте фабрику, которая будет кэшировать и повторно отдавать уже созданные объекты. Клиент должен запрашивать легковеса с определённым внутренним состоянием из этой фабрики, а не создавать его напрямую.
- Клиент должен хранить или вычислять значения внешнего состояния (контекст) и передавать его в методы объекта легковеса.

Lightweight_demo

```
import java.awt.*;

public class Lightweight_demo {
    static int CANVAS_SIZE = 500;
    static int TREES_TO_DRAW = 1000000;
    static int TREE_TYPES = 2;
    public static void main(String[] args) {
        Forest forest = new Forest();
        for (int i = 0; i < Math.floor(TREES_TO_DRAW / TREE_TYPES); i++) {
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                "Summer Oak", Color.GREEN, "Oak texture stub");
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");    }
        forest.setSize(CANVAS_SIZE, CANVAS_SIZE);
        forest.setVisible(true);
        System.out.println(TREES_TO_DRAW + " trees drawn");
        System.out.println("-----");
        System.out.println("Memory usage:");
        System.out.println("Tree size (8 bytes) * " + TREES_TO_DRAW);
        System.out.println("+ TreeTypes size (~30 bytes) * " + TREE_TYPES + "");
        System.out.println("-----");
        System.out.println("Total: " + ((TREES_TO_DRAW * 8 + TREE_TYPES * 30) / 1024 / 1024) +
            "MB (instead of " + ((TREES_TO_DRAW * 38) / 1024 / 1024) + "MB)");    }
    private static int random(int min, int max) {
        return min + (int) (Math.random() * ((max - min) + 1));    }}
}
```

TreeType

```
import java.awt.*;

public class TreeType {
    private String name;
    private Color color;
    private String otherTreeData;

    public TreeType(String name, Color color, String otherTreeData) {
        this.name = name;
        this.color = color;
        this.otherTreeData = otherTreeData;  }

    public void draw(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x - 1, y, 3, 5);
        g.setColor(color);
        g.fillOval(x - 5, y - 10, 10, 10);  }}
```

TreeFactory

```
import java.awt.*;
import java.util.HashMap;
import java.util.Map;

public class TreeFactory {
    static Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, Color color, String otherTreeData) {
        TreeType result = treeTypes.get(name);
        if (result == null) {
            result = new TreeType(name, color, otherTreeData);
            treeTypes.put(name, result);
        }
        return result;
    }
}
```

Tree

```
import java.awt.*;

public class Tree {
    private int x;
    private int y;
    private TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }

    public void draw(Graphics g) {
        type.draw(g, x, y);
    }
}
```

Forest

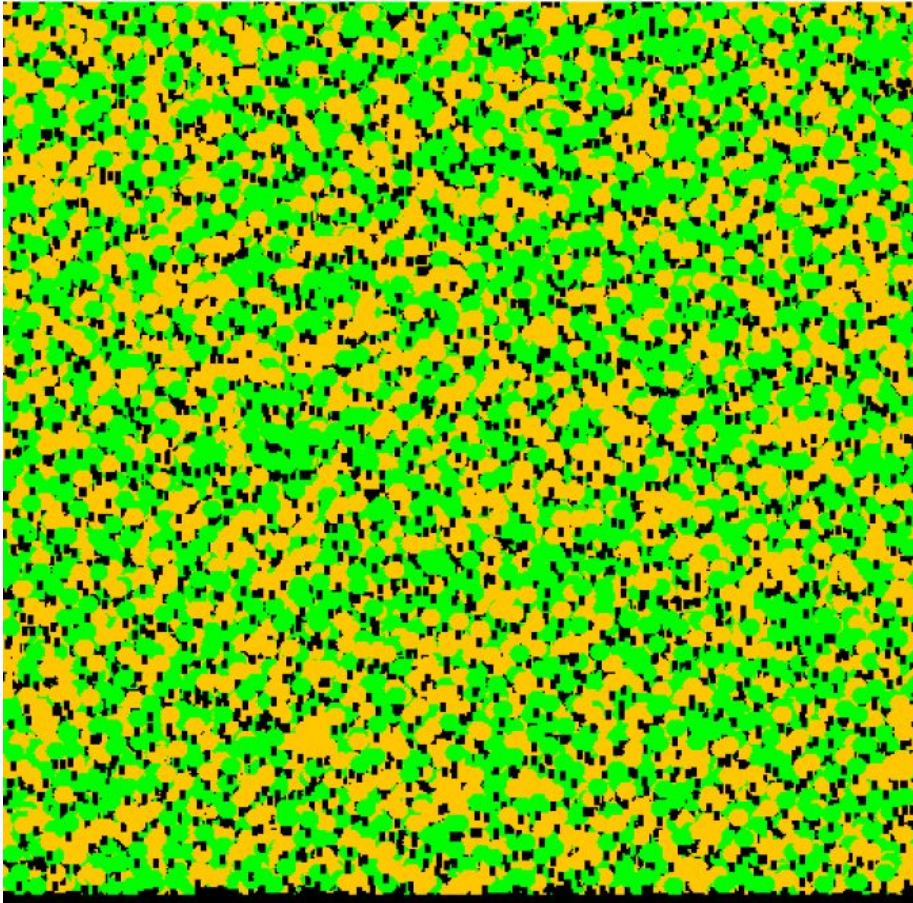
```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

public class Forest extends JFrame {
    private List<Tree> trees = new ArrayList<>();

    public void plantTree(int x, int y, String name, Color color, String otherTreeData) {
        TreeType type = TreeFactory.getTreeType(name, color, otherTreeData);
        Tree tree = new Tree(x, y, type);
        trees.add(tree);
    }

    @Override
    public void paint(Graphics graphics) {
        for (Tree tree : trees) {
            tree.draw(graphics);
        }
    }
}
```

Результат



1000000 trees drawn

Memory usage:

Tree size (8 bytes) * 1000000
+ TreeTypes size (~30 bytes) * 2

Total: 7MB (instead of 36MB)

Преимущества и недостатки

- Экономит оперативную память.(+)
- Расходует процессорное время на поиск/вычисление контекста.(-)
- Усложняет код программы за счёт множества дополнительных классов.(-)

Отношения с другими паттернами

- Компоновщик часто совмещают с Легковесом, чтобы реализовать общие ветки дерева и сэкономить при этом память.
- Легковес показывает как создавать много мелких объектов, а Фасад показывает как создать один объект, который отображает целую подсистему.
- Паттерн Легковес может напоминать Одиночку, если для конкретной задачи у вас получилось уменьшить количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:
 - Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.
 - Вы можете иметь множество объектов легковесов одного класса, в отличие от одиночки, который требует наличия только одного объекта.

Литература

- <https://refactoring.guru/ru/design-patterns/flyweight>

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