Arquitetura de Software

Padrões de Projetos – Padrões Estruturais

Nairon Neri Silva

Sumário

Padrões de Projeto Estruturais

1. *Adapter*
2. *Bridge*
3. *Composite*
4. *Decorator*
5. ***Façade***
6. ***Flyweight***
7. *Proxy*

Façade

Intenção

- Fornecer um interface unificada para um conjunto de interfaces de um subsistema

- Define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado

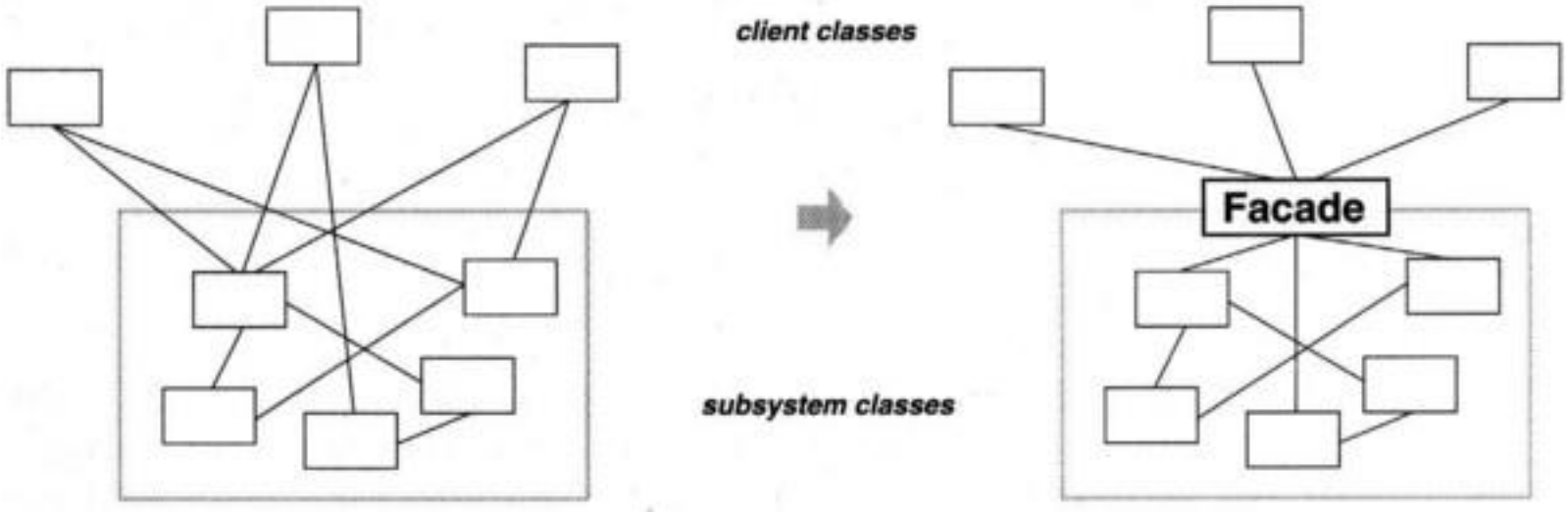
Motivação

- Estruturar um sistema em subsistemas ajuda a reduzir a complexidade

- Um objetivo comum de todos os projetos é minimizar a comunicação e as dependências entre subsistemas

Motivação

- Uma maneira de atingir este objetivo é introduzir um objeto Façade (fachada), o qual fornece uma interface única e simplificada para os recursos e facilidades mais gerais de um subsistema



Motivação

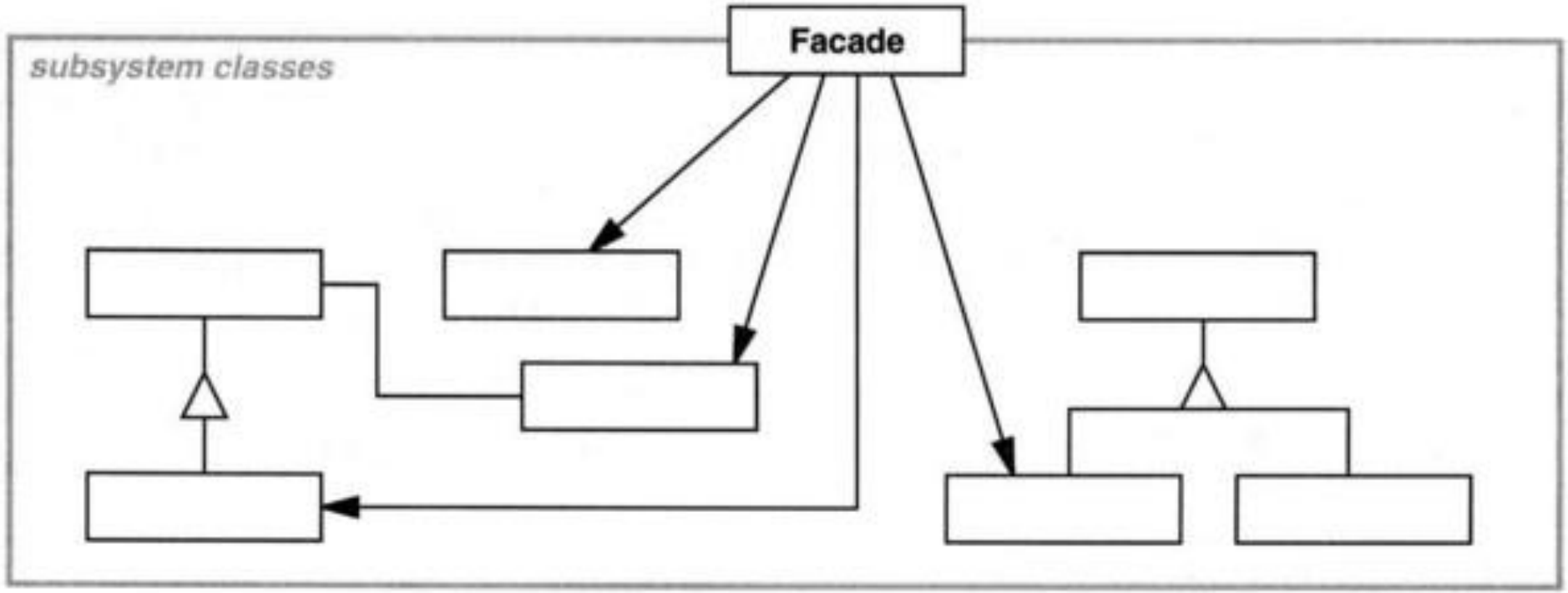
- Exemplo: considere um ambiente de programação que fornece acesso às aplicações para o seu (subsistema) compilador
  - Várias classes internas (*Scanner*, *Parser*, *ProgramNode*, *BytecodeStream*, *ProgramNodeBuilder*, etc.)
  - A maioria dos clientes de um compilador não se preocupa com detalhes do processo de compilação, pois estão mais interessados em obter o resultado
  - Para eles, as interfaces poderosas, porém de baixo nível, só complicam sua tarefa

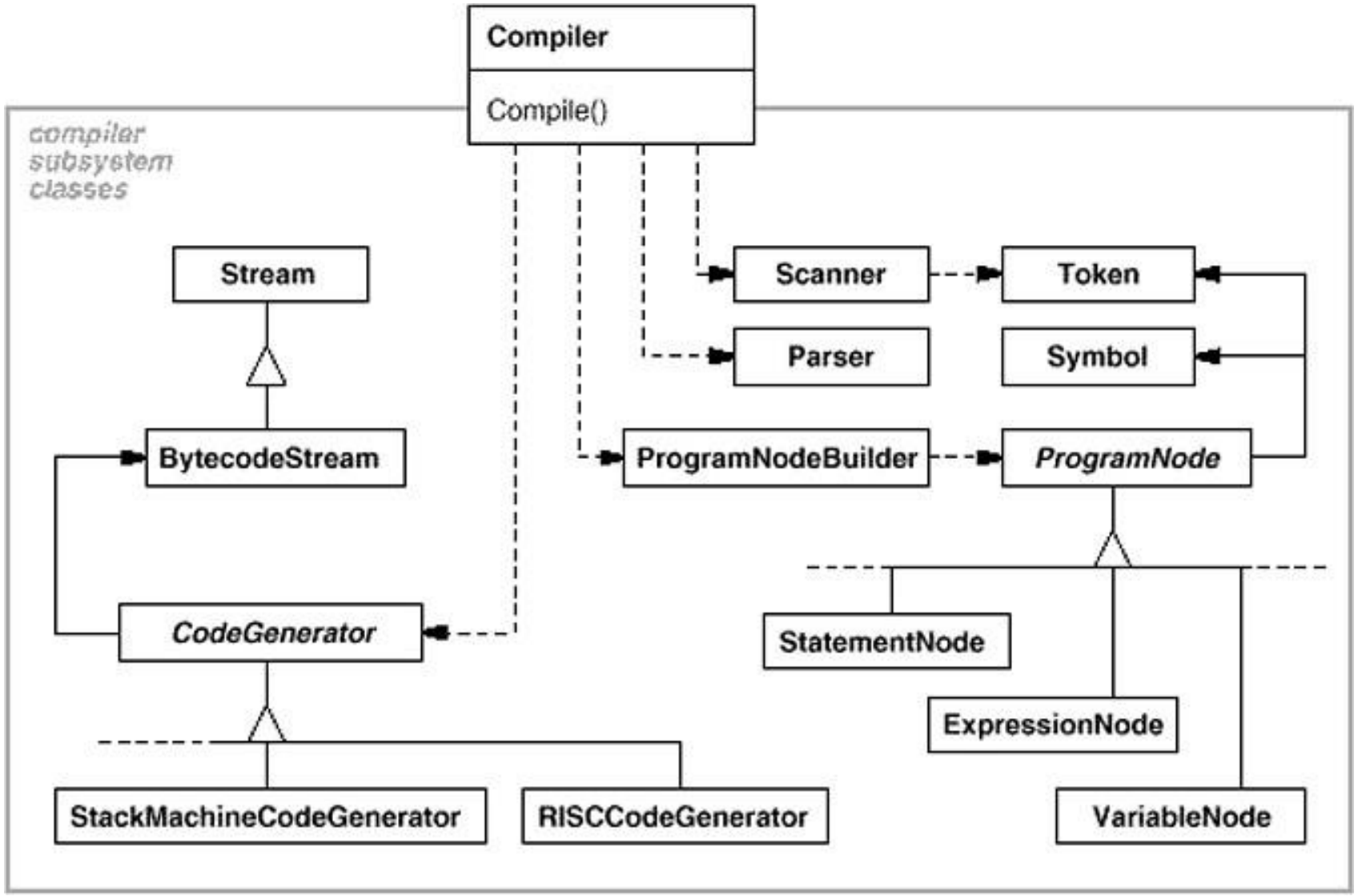
Aplicabilidade

1. Quando desejamos fornecer uma interface simples para um subsistema complexo

2. Quando existirem muitas dependências entre os clientes e as classes de implementação de uma abstração

3. Quando desejamos estruturar sistemas em camadas – criar ponto de entrada para cada nível


Facade
subsystem classes


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

Exemplo/Participantes

1. ***Façade (Compiler)***
   - Conhece quais as classes do subsistema são responsáveis pelo atendimento de uma solicitação – delega as mesmas

2. ***Classes de subsistema (Scanner, Parser, etc.)***
   - Implementam a funcionalidadedo subsistema
   - Encarregam-se do trabalho atribuído a elas
   - Não tem conhecimento da fachada

Consequências

1. Isola os clientes dos componentes do subsistema, tornando o subsistema mais fácil de usar

2. Promove um acoplamento fraco entre o subsistema e seus clientes

3. Não impede as aplicações de utilizarem as classes do subsistema caso necessite fazê-lo

Exemplo prático:

- Imagine que precisamos ligar um carro e para isso é preciso realizar diversas tarefas para que tudo funcione:
  - ✓ Ligar a central de injeção eletrônica;
  - ✓ Permitir a entrada de ar no coletor;
  - ✓ Injetar combustível;
  - ✓ Acionar o motor de partida;
  - ✓ Monitorar a temperatura atual do motor;
- Da mesma forma, desligar o motor também requer um conjunto de etapas.

Exemplo prático:

- Para solucionar esse problema (de ligar e desligar o carro), o que precisamos é criar uma fachada para que o cliente acesse uma única classe que possua dois métodos: um para ligar e outro para desligar o carro.

Exemplo retirado do site: https://www.baeldung.com/java-facade-pattern

Saiba mais...

- https://www.youtube.com/watch?v=A7mNiaBACYs&t=16s
- https://refactoring.guru/pt-br/design-patterns/facade
- https://www.baeldung.com/java-facade-pattern

Acesse os endereços e veja mais detalhes sobre o padrão Façade

Flyweight

Intenção

- Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina

Motivação

- Algumas aplicações poderiam se beneficiar da sua estruturação em objetos em todo o seu projeto, porém uma implementação ingênua seria proibitivamente cara

- Exemplo: a maioria das implementações de editores de documentos tem recursos para a formatação e edição de texto que são, até certo ponto, modularizados

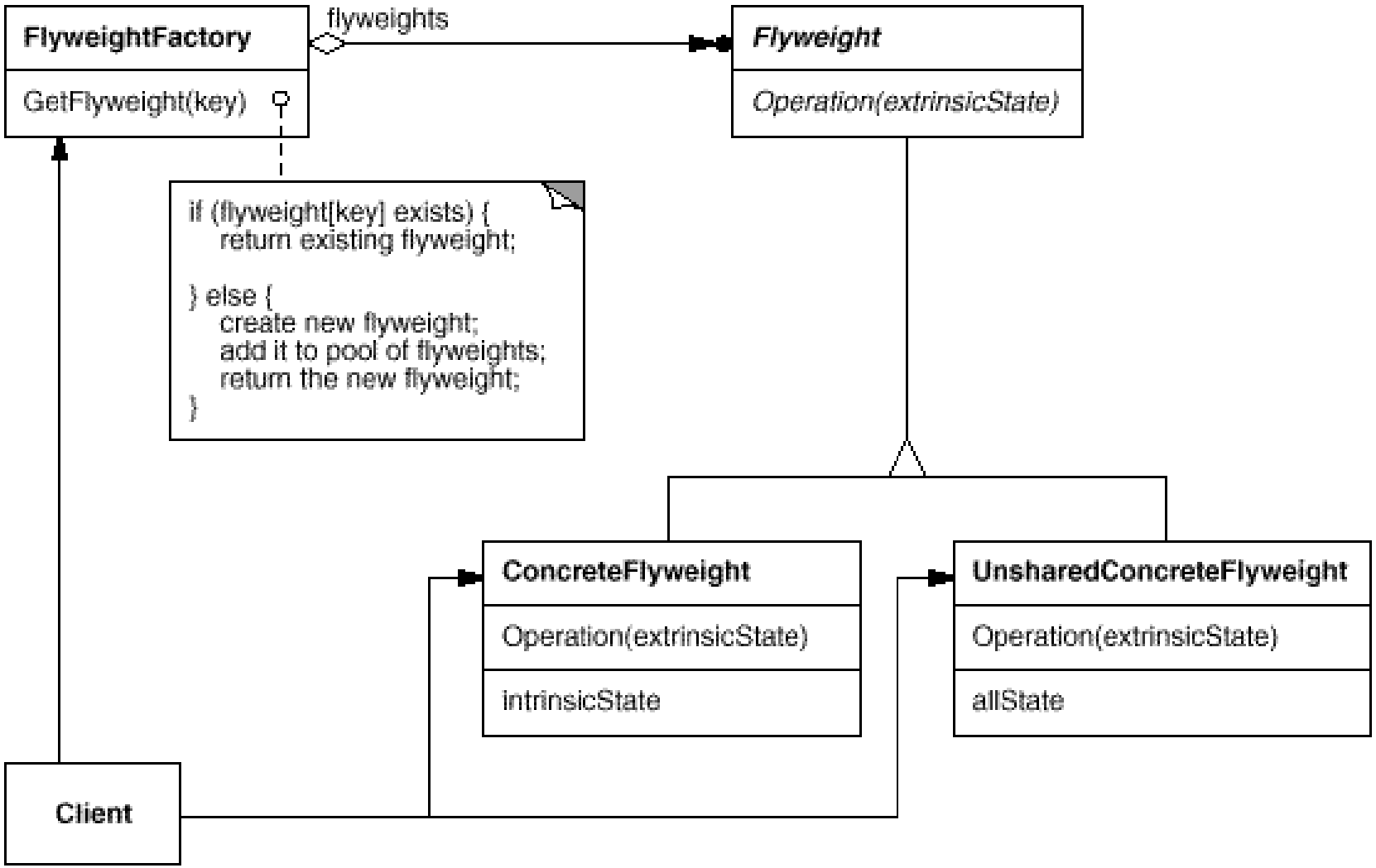
Motivação

- Continuando...
  - Esses editores utilizam objetos para representar elementos embutidos (tabelas, figuras, etc.)
  - No entanto, geralmente não chegam a usar objetos para cada caractere do documento – mesmo que dessa forma pudessem atingir o máximo da flexibilidade na aplicação
  - Pois esbarram na questão do ”custo”

- O padrão ***Flyweigth*** descreve como compartilhar objetos para permitir o seu uso em granularidades finas sem incorrer num custo proibitivo

Aplicabilidade

- Quando as seguintes condições forem verdadeiras
  1. Uma aplicação utiliza um grade número de objetos
  2. Os custos de armazenamento são altos, por causa da grande quantidade de objetos
  3. A maioria dos estados de objetos pode ser tornada extrínseca
  4. Muitos grupos de objetos podem ser substituídos por relativamente poucos objetos compartilhados, uma vez que estados extrínsecos são removidos
  5. A aplicação não depende da identidade dos objetos – uma vez que estes podem ser compartilhados

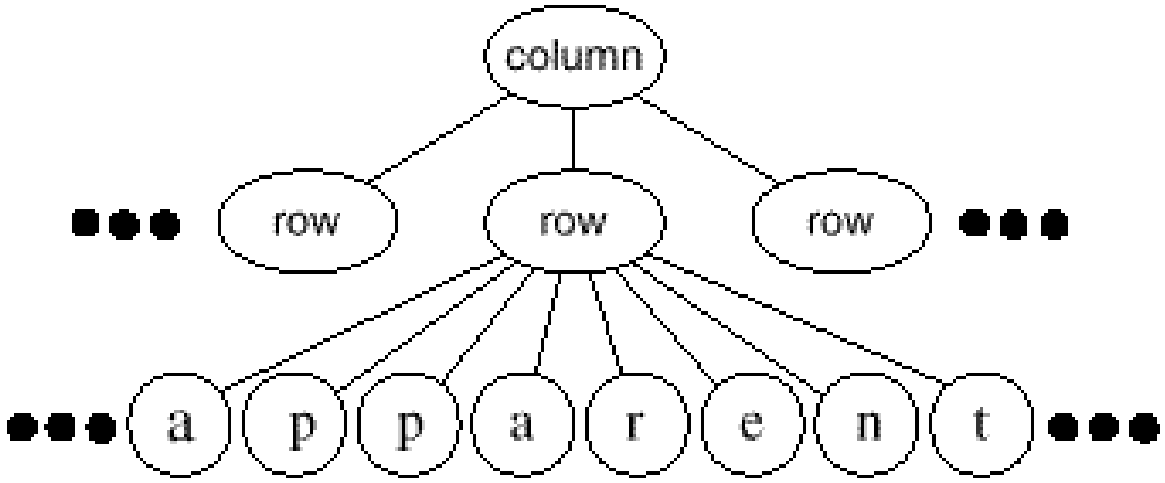

FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

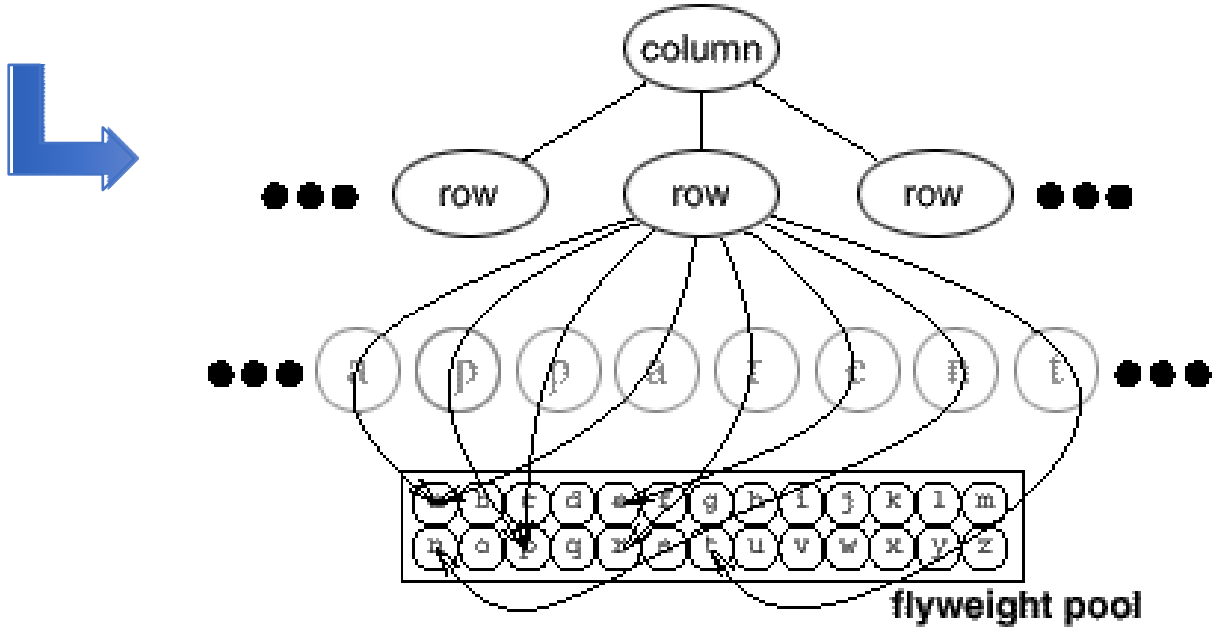
Exemplo/Participantes

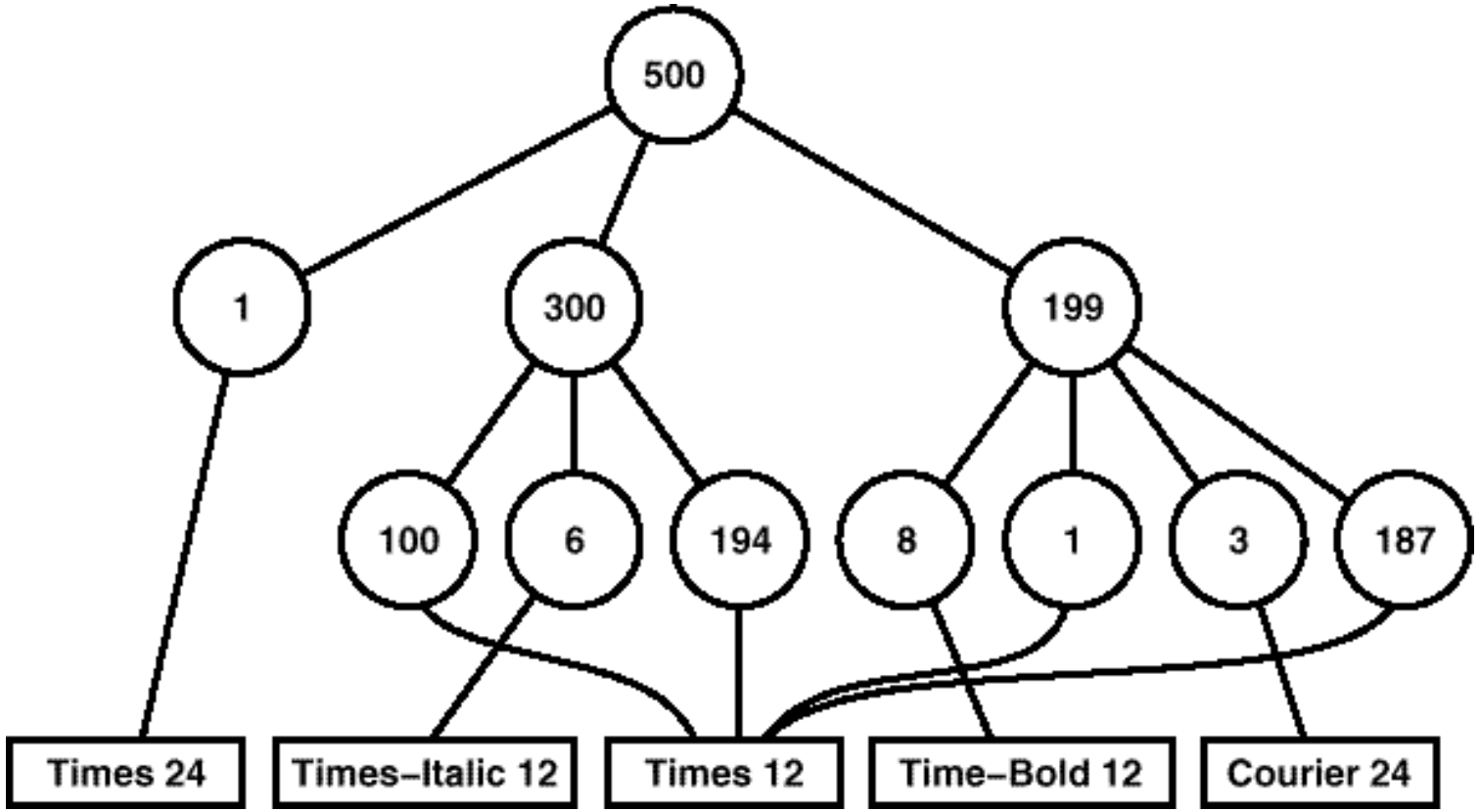
1. ***Flyweight (Glyph)***
   - Declara uma interface através da qual *flyweights* pode receber e atuar sobre estados extrínsecos
2. ***ConcreteFlyweight (Character)***
   - Implementa a interface ***Flyweight*** e acrescenta armazenamento para estados intrínsecos (independentes do contexto)
3. ***UnsharedConcreteFlyweight (Row,Column)***
   - Nem todas as subclasses de ***Flyweight*** necessitam ser compartilhadas

Exemplo/Participantes

4. ***FlyweightFactory***
   - Cria e gerencia objetos ***Flyweight***
   - Garante que os ***Flyweights*** sejam compartilhados apropriadamente – quando solicitado fornece uma instância existente ou cria uma, se nenhuma existir
5. ***Client***
   - Mantém uma referência para ***Flyweights***
   - Computa ou armazena o estado extrínseco dos ***Flyweights***


column
row
row
row
a
p
p
a
r
e
n
t


column
row
row
row
flyweight pool


500
1
300
199
100
6
194
8
1
3
187
Times 24
Times–Italic 12
Times 12
Time–Bold 12
Courier 24

Consequências

1. Podem introduzir custos de tempo de execução associados com a transferência, procura e/ou computação de estados extrínsecos

2. Contudo, tais custos são compensados pela economia de espaço
   - Redução do número total de instâncias
   - A quantidade de estados intrínsecos por objeto
   - Se o estado extrínseco é computado (e não armazenado)

Exemplo prático:

Exemplo presente no site: https://refactoring.guru/pt-br/design-patterns/flyweight/java/example

1. Neste exemplo, vamos renderizar uma floresta (1.000.000 árvores)! Cada árvore será representada por seu próprio objeto que possui algum estado (coordenadas, textura e assim por diante). Embora o programa faça seu trabalho principal, naturalmente consome muita RAM.
2. O motivo é simples: muitos objetos árvore contêm dados duplicados (nome, textura, cor). É por isso que podemos aplicar o padrão *Flyweight* e armazenar esses valores em objetos separados de *flyweight* (a classe TreeType). Agora, em vez de armazenar os mesmos dados em milhares de objetos Tree, vamos fazer referência a um dos objetos *flyweight* com um conjunto específico de valores.
3. O código cliente não notará nada, pois a complexidade da reutilização de objetos flyweight estará dentro de uma fábrica de flyweight.

Saiba mais...

- https://www.youtube.com/watch?v=WPQa64bdQbk
- https://refactoring.guru/pt-br/design-patterns/flyweight
- https://www.baeldung.com/java-flyweight

Acesse os endereços e veja mais detalhes sobre o padrão Flyweight