

DESIGN PATTERNS

Template Method

A ideia principal por trás deste padrão Template Method é criar uma estrutura que define a sequência de passos de um algoritmo, mas deixa a implementação detalhada de alguns desses passos para subclasses. Esses passos podem ser representados como métodos, e o padrão permite que as subclasses forneçam suas próprias implementações para esses métodos, mantendo a estrutura do algoritmo consistente.

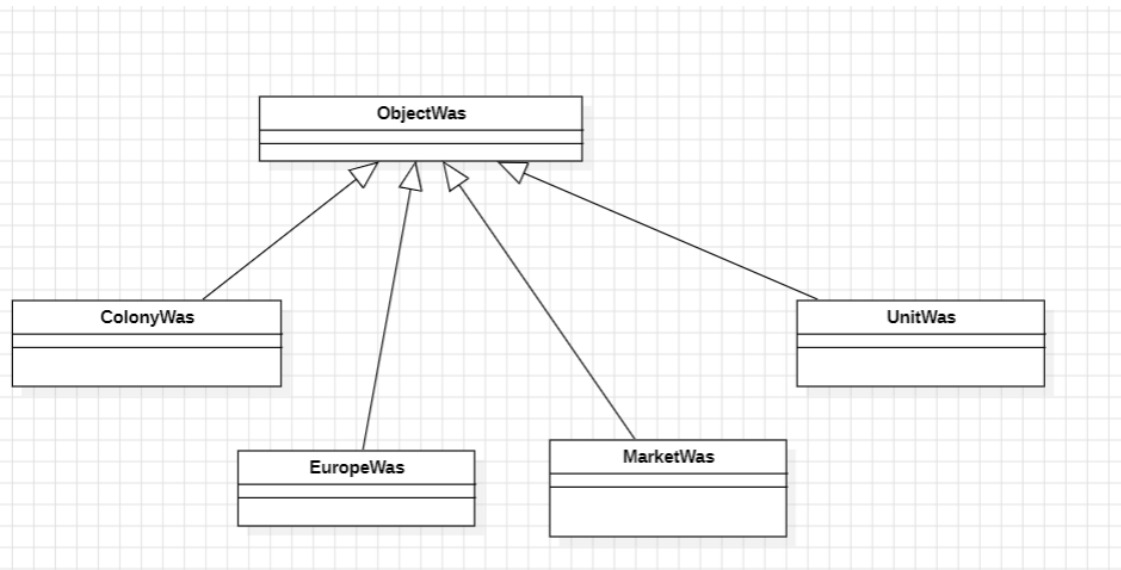
```
19
20 package net.sf.freecol.common.model;
21
22 /**
23  * Parent for all the objects that may need to fireChanges.
24  */
25 public abstract class ObjectWas {
26
27     /**
28      * Fire the property changes that have accumulated for this object.
29      *
30      * @return True if something changed.
31      */
32     public abstract boolean fireChanges();
33 }
```

Path: net/sf/freecol/common/model/ObjectWas.java

```
54
55 /**
56  * {@inheritDoc}
57  */
58 public boolean fireChanges() {
59     boolean ret = false;
60     int newPopulation = colony.getUnitCount();
61     if (newPopulation != population) {
62         String pc = ColonyChangeEvent.POPULATION_CHANGE.toString();
63         colony.firePropertyChange(pc, population, newPopulation);
64         ret = true;
65     }
66     int newProductionBonus = colony.getProductionBonus();
67     if (newProductionBonus != productionBonus) {
68         String pc = ColonyChangeEvent.BONUS_CHANGE.toString();
69         colony.firePropertyChange(pc, productionBonus,
70             newProductionBonus);
71         ret = true;
72     }
73     List<BuildableType> newBuildQueue = colony.getBuildQueue();
```

Path: net/sf/freecol/common/model/ColonyWas.java

Como Podemos ver acima o método fireChanges() é redefinido na subclasse ColonyWas implementando de forma diferente comparando com outras subclasses.



Factory method

O Factory Method é um padrão de criação que define uma interface para a criação de objetos, mas permite que as subclasses decidam qual classe concreta instanciar. Pode ser identificado quando uma classe contém um método de criação que retorna objetos de subclasses.

```
30 /**
31  * A factory class for creating {@code Resource} instances.
32  * @see Resource
33  */
34 public class ResourceFactory {
35
36     private static final Logger logger = Logger.getLogger(ResourceFactory.class.getName());
37
38
39     /**
40      * Ensures that only one {@code Resource} is created given the same {@code URI}.
41      */
42     private final Map<URI, Resource> resources = new HashMap<>();
43
44
45     /**
46      * Returns an instance of {@code Resource} with the
47      * given {@code URI} as the parameter.
48      *
49      * @param key The key part of the resource mapping.
50      * @param cachingKey The caching key.
51      * @param uri The {@code URI} used when creating the instance.
52      * @return The <code>Resource</code> if created.
53      */
54     public Resource createResource(String key, String cachingKey, URI uri) {
55         final Resource r = resources.get(uri);
56         if (r != null) {
57             return r;
58         }
59     }
```

Path: [net/sf/freecol/common/resources/ResourceFactory.java](https://net.sf.freecol/common/resources/ResourceFactory.java)

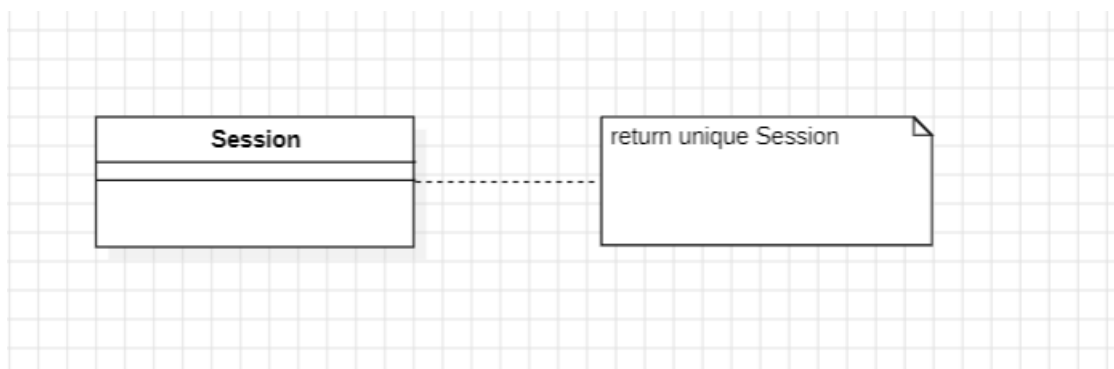
Exemplo deste padrão é a classe ResourceFactory que tem como objetivo criar instancias do tipo Resource.

Singleton Method

Este padrão garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a essa instância. É frequentemente identificado pela presença de um método estático para acessar a instância única.

```
38 public abstract class Session {
39
40     private static final Logger logger = Logger.getLogger(Session.class.getName());
41
42     /** A map of all active sessions. */
43     private static final Map<String, Session> allSessions = new HashMap<>();
44
45     /** The key to this session. */
46     private String key;
47
48     /** Has this session been completed? */
49     private boolean completed = false;
50
51
52     /**
53      * Protected constructor, we only really instantiate specific types
54      * of transactions.
55      *
56      * @param key A unique key to lookup this transaction with.
57      */
58     protected Session(String key) {
59         Session s = getSession(key);
60         if (s != null) {
61             throw new IllegalArgumentException("Duplicate session: " + key
62                 + " -> " + s);
63         }
64         this.key = key;
65         this.completed = false;
66         logger.finest(msg: "Created session: " + key);
67     }
```

Path: net/sf/freecol/server/model/Session.java



Consequimos perceber que realmente se trata de um Singleton Method através destes dois parâmetros característicos deste design pattern:

Variável Estática Privada: A variável `private static final Map<String, Session> allSessions` é uma instância privada e estática que mantém todas as instâncias de `Session`. Ela é acessada de maneira estática através da classe, o que é típico de uma implementação Singleton.

Construtor Privado: O construtor da classe Session é definido como protected, o que significa que não pode ser acessado diretamente por código externo. Isso impede que novas instâncias da classe sejam criadas fora da classe.