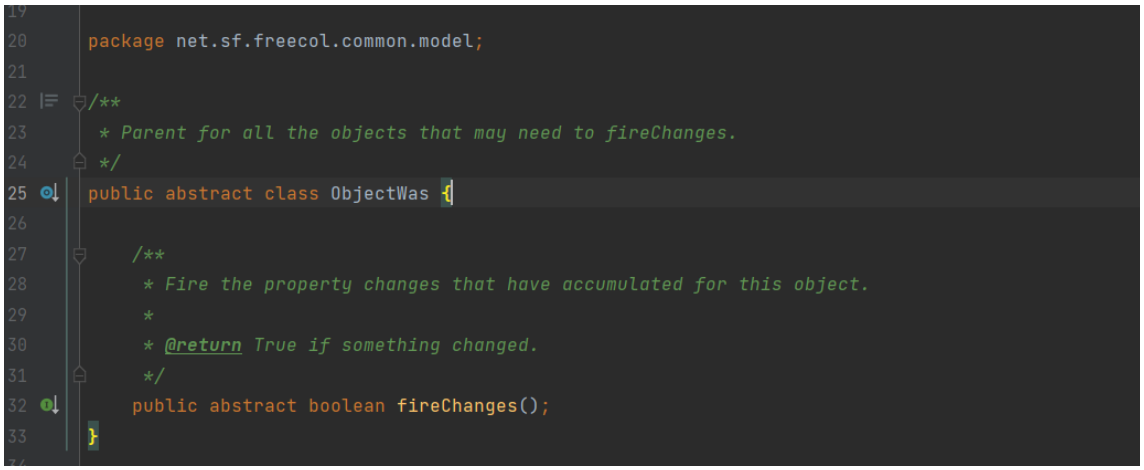
DESIGN PATTERNS

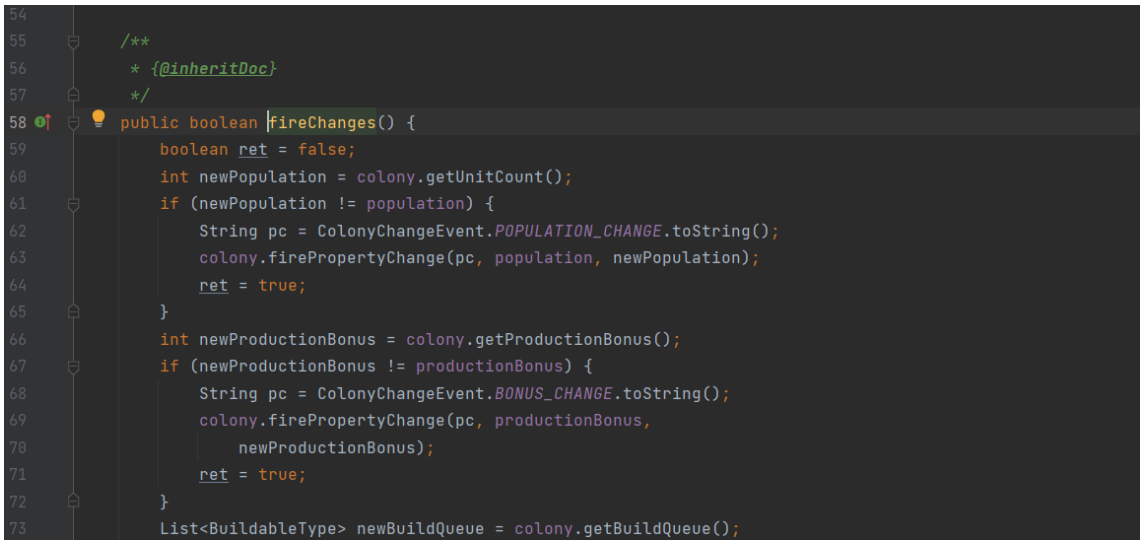
Template Method

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 destes passos para as subclasses. Estes passos podem ser representados como métodos, e o padrão permite que as subclasses forneçam as suas próprias implementações para estes métodos, mantendo de forma geral a estrutura do algoritmo consistente.



```
package net.sf.freecol.common.model;

/**
 * Parent for all the objects that may need to fireChanges.
 */
public abstract class ObjectWas {

    /**
     * Fire the property changes that have accumulated for this object.
     *
     * @return True if something changed.
     */
    public abstract boolean fireChanges();
}
```

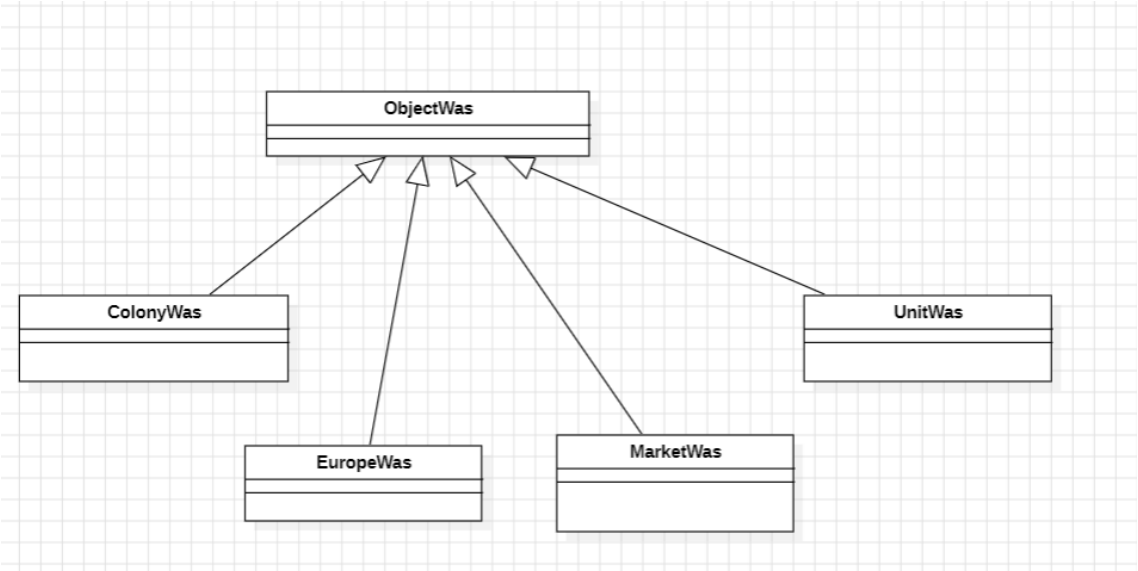
Path: net/sf/freecol/common/model/ObjectWas.java



```
    /**
     * {@inheritDoc}
     */
    public boolean fireChanges() {
        boolean ret = false;
        int newPopulation = colony.getUnitCount();
        if (newPopulation != population) {
            String pc = ColonyChangeEvent.POPULATION_CHANGE.toString();
            colony.firePropertyChange(pc, population, newPopulation);
            ret = true;
        }
        int newProductionBonus = colony.getProductionBonus();
        if (newProductionBonus != productionBonus) {
            String pc = ColonyChangeEvent.BONUS_CHANGE.toString();
            colony.firePropertyChange(pc, productionBonus,
                newProductionBonus);
            ret = true;
        }
        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.

```java
/**
 * A factory class for creating {@code Resource} instances.
 * @see Resource
 */
public class ResourceFactory {

    private static final Logger logger = Logger.getLogger(ResourceFactory.class.getName());


    /**
     * Ensures that only one {@code Resource} is created given the same {@code URI}.
     */
    private final Map<URI, Resource> resources = new HashMap<>();


    /**
     * Returns an instance of {@code Resource} with the
     * given {@code URI} as the parameter.
     *
     * @param key The key part of the resource mapping.
     * @param cachingKey The caching key.
     * @param uri The {@code URI} used when creating the instance.
     * @return The <code>Resource</code> if created.
     */
    public Resource createResource(String key, String cachingKey, URI uri) {
        final Resource r = resources.get(uri);
        if (r != null) {
            return r;
        }
```

Path: net/sf/freecol/common/resources/ResourceFactory.java

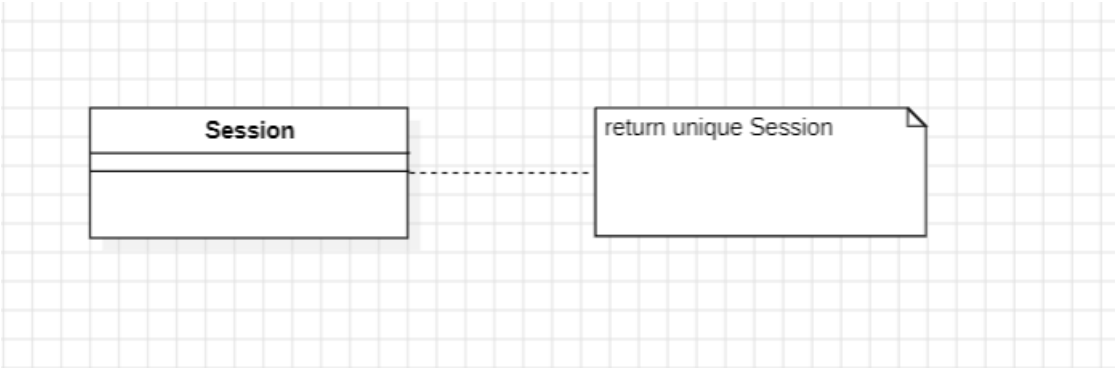
Exemplo deste padrão é a classe “ResourceFactory” que tem como objetivo criar instacias do tipo “Resource”.

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 aceder a esta instância única.

```java
public abstract class Session {

    private static final Logger logger = Logger.getLogger(Session.class.getName());

    /** A map of all active sessions. */
    private static final Map<String, Session> allSessions = new HashMap<>();

    /** The key to this session. */
    private String key;

    /** Has this session been completed? */
    private boolean completed = false;


    /**
     * Protected constructor, we only really instantiate specific types
     * of transactions.
     *
     * @param key A unique key to lookup this transaction with.
     */
    protected Session(String key) {
        Session s = getSession(key);
        if (s != null) {
            throw new IllegalArgumentException("Duplicate session: " + key
                                               + " -> " + s);
        }
        this.key = key;
        this.completed = false;
        logger.finest( msg: "Created session: " + key);
    }
```

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



Conseguimos 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. O acesso é feito de maneira estática através da classe, o que é típico de uma implementação do tipo “Singleton”

.

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