Code smell series: ***Parallel Inheritance Hierarchies***.

# ***Parallel*** **Inheritance *Hierarchies***

The Code Smells are similar in concept to Development-level Anti-patterns. Sometimes in our code, we introduce code smell unintentionally those makes our design fragile.

**Definition of Code smell**

Code smell, also known as a bad smell, in computer programming code, refers to any symptom in the source code of a program that possibly indicates a deeper problem.

**Martin fowler says:**

"a code smell is a surface indication that usually corresponds to a deeper problem in the system"

Code smell creates lot of problems while introducing new feature or maintains the codebase,

Often developer has to write repeatable code, breaking encapsulation, breaking abstraction etc.

If code smells are not corrected.

So always refactor code smell while developing.

In this article, we discuss on “***Parallel Inheritance Hierarchies***” code smell.

***Parallel Inheritance Hierarchies code smells occurs when an inheritance tree depends on another inheritance tree by composition and they maintain a special relation one subclass of dependent inheritance must dependent one a particular subclass of another Inheritance***

Think about Engineers and their work and target, Computer Engineers work on Computers

And target is deliver projects where as Civil engineer work on Structure and target is to implement structure. So Design perspective there is two parallel hierarchies a. Engineers b. Milestone

Different engineer has different milestone and each engineer has a specified milestone special relation.

The problem is Every time you add a new Engineer in Engineer inheritance you have to introduce a new Milestone in Milestone hierarchy.

**Cause of Parallel Inheritance Hierarchiessmell:**

1. Fail to understand the responsibility. Often due to misunderstanding (Single responsibility principle)
2. Overenthusiasm to break each functionality as separate interface.
3. Fail to introduce proper design pattern.

**Consequences of** **Parallel Inheritance Hierarchies**:

1. Lots of duplicates code.
2. Wrong relationship set by the client
3. Unmaintainable code base.

**Refactor Strategy**

. We can do it by “**Move Method**”, “**Move Field**” techniques.

Now, take an Example Where Parallel Inheritance Hierarchies code smell present. We will implement the Engineer and set Milestone for him.

**Java Code**

**package** com.example.codesmell.parallelinheritence;

**public** **interface** Engineer {

String getType();

**void** setType(String type);

**int** getSalary();

**void** setSalary(**int** salary);

MileStone getMileStone();

**void** setMileStone(MileStone mileStone);

}

**package** com.example.codesmell.parallelinheritence;

**public** **interface** MileStone {

**public** String work();

**public** String target();

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** ComputerEngineer **implements** Engineer{

**private** String type;

**private** **int** salary;

**private** MileStone mileStone;

**public** **void** setType(String type) {

**this**.type = type;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

**public** **void** setMileStone(MileStone mileStone) {

**this**.mileStone = mileStone;

}

@Override

**public** String getType() {

// **TODO** Auto-generated method stub

**return** type;

}

@Override

**public** **int** getSalary() {

// **TODO** Auto-generated method stub

**return** salary;

}

@Override

**public** MileStone getMileStone() {

// **TODO** Auto-generated method stub

**return** mileStone;

}

@Override

**public** String toString() {

**return** "ComputerEngineer [type=" + type + ", salary=" + salary

+ ", mileStone=" + mileStone + "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** ComputerMileStone **implements** MileStone{

@Override

**public** String work() {

**return**"Build a Billing MicroService";

}

@Override

**public** String target() {

**return**"Has to be finshed in 14 PD";

}

@Override

**public** String toString() {

**return** "ComputerMileStone [work()=" + work() + ", target()=" + target()

+ "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** CivilEngineer **implements** Engineer{

**private** String type;

**private** **int** salary;

**private** MileStone mileStone;

**public** **void** setType(String type) {

**this**.type = type;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

**public** **void** setMileStone(MileStone mileStone) {

**this**.mileStone = mileStone;

}

@Override

**public** String getType() {

// **TODO** Auto-generated method stub

**return** type;

}

@Override

**public** **int** getSalary() {

// **TODO** Auto-generated method stub

**return** salary;

}

@Override

**public** MileStone getMileStone() {

// **TODO** Auto-generated method stub

**return** mileStone;

}

@Override

**public** String toString() {

**return** "CivilEngineer [type=" + type + ", salary=" + salary

+ ", mileStone=" + mileStone + "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** CivilMileStone **implements** MileStone{

@Override

**public** String work() {

// **TODO** Auto-generated method stub

**return** "Create Twin Towers";

}

@Override

**public** String target() {

// **TODO** Auto-generated method stub

**return** "Has to be completed in 2 years";

}

@Override

**public** String toString() {

**return** "CivilMileStone [work()=" + work() + ", target()=" + target()

+ "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** Manager {

**public** **static** **void** main(String[] args) {

Engineer comp = **new** ComputerEngineer();

comp.setType("Computer Engineer");

comp.setSalary(50000);

comp.setMileStone(**new** ComputerMileStone());

Engineer civil = **new** ComputerEngineer();

civil.setType("Civil Engineer");

civil.setSalary(60000);

civil.setMileStone(**new** CivilMileStone());

System.***out***.println(comp);

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println(civil);

}

}

**Output :**

ComputerEngineer [type=Computer Engineer, salary=50000, mileStone=ComputerMileStone [work()=Build a Billing MicroService, target()=Has to be finshed in 14 PD]]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CivilEngineer [type=Civil Engineer, salary=60000, mileStone=CivilMileStone [work()=Create Twin Towers, target()=Has to be completed in 2 years]]

We create two interfaces Engineer and Milestone and create Subclass for them but thing to notice is Every engineer has his own special milestone so as we expose the setMileStone method to client it may possible client set a wrong Milestone for an Engineer.

Another thing If we want to add a new Engineer also we need to add a new Milestone for him/her.

It is very difficult to fix this, if we try to fix it may break SRP (Single Responsibility) principal.

So, In my opinion there is no fix solution for dealing with parallel Hierarchies.

There is Three possible way we can deal with it.

**Solution 1:**

1. Keep the parallel hierarchies open and get used to it.

**Pros**:

Better way to maintain SRP.

Code will be flexible.

**Cons**:

To Add a new feature, we have to create two classes every time.

May change in one need to change in Other hierarchies as they are coupled.

Harder to maintain.

**Solution 2:**

Make it Partial Hierarchies so we can open provision for parallel hierarchies.

**Pros**:

Only maintain One hierarchy

When you are not sure about responsibility try to adopt it.

Provide flexibility

**Cons**:

May break Single Responsibility principle

**Technique**:

Make a concrete class and implements both interfaces. Client got instance of this class by static factory method.

Let see the solution:

**package** com.example.codesmell.parallelinheritence;

**public** **class** PartialComputerEngineer **implements** Engineer,MileStone{

**private** String type;

**private** **int** salary;

@Override

**public** String work() {

**return**"Build a Billing MicroService";

}

@Override

**public** String target() {

**return**"Has to be finshed in 14 PD";

}

@Override

**public** String getType() {

// **TODO** Auto-generated method stub

**return** type;

}

@Override

**public** **void** setType(String type) {

**this**.type=type;

}

@Override

**public** **int** getSalary() {

// **TODO** Auto-generated method stub

**return** salary;

}

@Override

**public** **void** setSalary(**int** salary) {

**this**.salary=salary;

}

@Override

**public** MileStone getMileStone() {

// **TODO** Auto-generated method stub

**return** **this**;

}

@Override

**public** **void** setMileStone(MileStone mileStone) {

**throw** **new** UnsupportedOperationException("Not Supported");

}

@Override

**public** String toString() {

**return** "PartialComputerEngineer [type=" + type + ", salary=" + salary

+ ", work()=" + work() + ", target()=" + target()

+ ", getType()=" + getType() + ", getSalary()=" + getSalary()

+ "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** PartialCivilEngineer **implements** Engineer,MileStone{

**private** String type;

**private** **int** salary;

@Override

**public** String work() {

**return** "Create Twin Towers";

}

@Override

**public** String target() {

**return** "Has to be completed in 2 years";

}

@Override

**public** String getType() {

**return** type;

}

@Override

**public** **void** setType(String type) {

**this**.type=type;

}

@Override

**public** **int** getSalary() {

// **TODO** Auto-generated method stub

**return** salary;

}

@Override

**public** **void** setSalary(**int** salary) {

**this**.salary=salary;

}

@Override

**public** MileStone getMileStone() {

// **TODO** Auto-generated method stub

**return** **this**;

}

@Override

**public** **void** setMileStone(MileStone mileStone) {

**throw** **new** UnsupportedOperationException("Not Supported");

}

@Override

**public** String toString() {

**return** "PartialCivilEngineer [type=" + type + ", salary=" + salary

+ ", work()=" + work() + ", target()=" + target()

+ ", getType()=" + getType() + ", getSalary()=" + getSalary()

+ "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** EngineerFactory {

**public** **static** Engineer getEngineer(Class clazz) **throws** InstantiationException, IllegalAccessException

{

**return** (Engineer) clazz.newInstance();

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** Manager {

**public** **static** **void** main(String[] args) **throws** InstantiationException, IllegalAccessException {

Engineer comp = EngineerFactory.*getEngineer*(PartialComputerEngineer.**class**);

comp.setType("Computer Engineer");

comp.setSalary(50000);

Engineer civil = EngineerFactory.*getEngineer*(PartialCivilEngineer.**class**);

civil.setType("Computer Engineer");

civil.setSalary(50000);

System.***out***.println(comp);

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println(civil);

}

}

**Output :**

PartialComputerEngineer [type=Computer Engineer, salary=50000, work()=Build a Billing MicroService, target()=Has to be finshed in 14 PD, getType()=Computer Engineer, getSalary()=50000]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PartialCivilEngineer [type=Computer Engineer, salary=50000, work()=Create Twin Towers, target()=Has to be completed in 2 years, getType()=Computer Engineer, getSalary()=50000]

**Solution 3:**

Collapse a hierarchy.

**Pros**:

Only maintain One hierarchy

Easy to maintain

**Cons**:

Break Single Responsibility principle often

**Technique**:

Make a common interface and move methods from another interface.

Let see the solution:

**package** com.example.codesmell.parallelinheritence;

**public** **interface** EngineerMileStone {

String getType();

**void** setType(String type);

**int** getSalary();

**void** setSalary(**int** salary);

**public** String work();

**public** String target();

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** RefactorComputerEngineer **implements** EngineerMileStone{

**private** String type;

**private** **int** salary;

@Override

**public** String getType() {

**return** type;

}

@Override

**public** **void** setType(String type) {

**this**.type=type;

}

@Override

**public** **int** getSalary() {

**return** salary;

}

@Override

**public** **void** setSalary(**int** salary) {

**this**.salary=salary;

}

@Override

**public** String work() {

**return**"Build a Billing MicroService";

}

@Override

**public** String target() {

**return**"Has to be finshed in 14 PD";

}

@Override

**public** String toString() {

**return** "RefactorComputerEngineer [type=" + type + ", salary=" + salary

+ ", getType()=" + getType() + ", getSalary()=" + getSalary()

+ ", work()=" + work() + ", target()=" + target() + "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** ReFactorCivilEngineer **implements** EngineerMileStone{

**private** String type;

**private** **int** salary;

@Override

**public** String getType() {

**return** type;

}

@Override

**public** **void** setType(String type) {

**this**.type=type;

}

@Override

**public** **int** getSalary() {

**return** salary;

}

@Override

**public** **void** setSalary(**int** salary) {

**this**.salary=salary;

}

@Override

**public** String work() {

**return** "Create Twin Towers";

}

@Override

**public** String target() {

**return** "Has to be completed in 2 years";

}

@Override

**public** String toString() {

**return** "ReFactorCivilEngineer [type=" + type + ", salary=" + salary

+ ", getType()=" + getType() + ", getSalary()=" + getSalary()

+ ", work()=" + work() + ", target()=" + target() + "]";

}

}

**package** com.example.codesmell.parallelinheritence;

**public** **class** Manager {

**public** **static** **void** main(String[] args) **throws** InstantiationException, IllegalAccessException {

EngineerMileStone comp = **new** RefactorComputerEngineer();

comp.setType("Computer Engineer");

comp.setSalary(50000);

EngineerMileStone civil = **new** ReFactorCivilEngineer();

civil.setType("Civil Engineer");

civil.setSalary(60000);

System.***out***.println(comp);

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println(civil);

}

}

**Output**

RefactorComputerEngineer [type=Computer Engineer, salary=50000, getType()=Computer Engineer, getSalary()=50000, work()=Build a Billing MicroService, target()=Has to be finshed in 14 PD]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ReFactorCivilEngineer [type=Civil Engineer, salary=60000, getType()=Civil Engineer, getSalary()=60000, work()=Create Twin Towers, target()=Has to be completed in 2 years]

.

.

.