1.

Controller: (na przykładzie jednego z moich projektów dostępnych na githubie)

klasa pojazdu:
https://github.com/vinterdo/ProjectDrive/blob/master/ProjectDrive/Assets/Scripts/Vehicle.cs

klasa obsługująca input gracza:
https://github.com/vinterdo/ProjectDrive/blob/master/ProjectDrive/Assets/Scripts/VehiclePlayerController.cs

klasa obsługująca sztuczną inteligencję:
https://github.com/vinterdo/ProjectDrive/blob/master/ProjectDrive/Assets/Scripts/VehicleAIController.cs

Creator:
https://github.com/vinterdo/ArtifactsRider/blob/master/ArtifactsRider/VAPI/Physics/Factories/JointFactory.cs
Fabryka, odpowiadająca za tworzenie różnego rodzajów Jointów (w silniku fizycznym Joint jest to siła, łącząca ze sobą 2 różne ciała - w zależności od typu może być mniej lub bardziej sprężysta, działać tylko jeżeli ciała są od siebie odpowiednio daleko, nie pozwalać znajdować się tym ciałom pod pewnym kontem, itd).

High cohesion:
https://github.com/vinterdo/CBorgEngine/blob/master/oglBase/oglBase/meshRenderer.h
https://github.com/vinterdo/CBorgEngine/blob/master/oglBase/oglBase/meshRenderer.cpp
klasa nie przechowuje danych, odpowada jedynie za rysowanie modelu, który musi być przekazany osobno.

Polymorphism:
https://github.com/vinterdo/DeusExMachina/tree/master/src/main/java/com/vinterdo/deusexmachina/tileentity

klasy w pakiecie do którego jest link wyżej dziedziczą z klas z pakiety base, które są bardziej generyczne, np. TEDEM.java to klasa która jest najwyżej (najbardziej generyczna) a TEI.java to już klasa bardziej wyspecjalizowana, którą nadal w wielu miejsach traktujemy jako TEDEM.java.
Skróty oznaczają:
TileEntityDeusExMachina
TileEntityInventory

Indirection:
https://github.com/vinterdo/DeusExMachina/blob/master/src/main/java/com/vinterdo/deusexmachina/tileentity/base/TEMultiblockMaster.java

https://github.com/vinterdo/DeusExMachina/blob/master/src/main/java/com/vinterdo/deusexmachina/tileentity/base/TEMultiblock.java
Master wszystkim zarządza, zwykły multiblok zawsze zwraca się do niego jeżeli potrzebuje coś zrobić. Zatem prowadzi także “mediację” między obiektami TEMultiblock

4. Ten kod łamie zasadę LSP - kod kliencki, musiałby wiedzieć jakiego typu jest dana figura żęby poprawnie wyświetlić jej pole. Klasa prostokąt może mieć pola width i height, ale klasa kwadrat już nie powinna (bo to to samo, nie dość że zmusza nas to do złamania LSP to jeszcze łamie redundancję). Dlatego hierarchia powinna wyglądać tak:

```
abstract class Figure
{
	public float getArea();
}

class Square : Figure
{
	float width;

	public override float getArea()
	{
		return width * width;
	}
}

class Rectangle : Figure
{
	float width;
	float height;

	public override float getArea()
	{
		return width * height;
	}
}

…

void printArea(Figure f)
{
	print(“figura ma pole “ + f.getArea());
}
```

5. Biblioteka standardowa javy: interfejs Collection, zawiera między innymi metody get(index) i set(index, element). Przykładem klasy implementującej ten interfejs jest ImmutableList z biblioteki guava - ponieważ ta klasa nie pozwala na zmienianie wartośći już

zbudowanej listy, funkcja set(index, element) zawsze rzuca wyjątek UnsupportedOperationException.

https://google-collections.googlecode.com/svn/trunk/javadoc/com/google/common/collect/ImmutableList.html

6.
SRP mówi o tym, że klasa powinna mieć “single responsibility” (nie potrafię znaleźć dobrego tłmaczenia dla tego na polski). Nie powinno być więcej niż 1 powodu do modyfikacji tej klasy - z tego wynika że klasa powinna zajmować się jedną rzeczą, np. klasa odpowiadająca za model 3d nie powinna trzymać danych modelu i rysować go - powinny to robić 2 osobne klasy. Powodem do modyfikacji klasy mogłaby być zmiana sposobu trzymania danych modelu albo zmiana API graficznego w programie.

ISP zmusza nas do implementowania minimum w interfejsach - osoba która chce użyć interfejsu HasColor powinna móc tylko nadpisać metody getColor, setColor, a nie musieć implementować metod typu invertColor, mixColor(Color) itd - może to zrobić osobna klasa używając 2 pierwszych metod z interfejsu.

7.