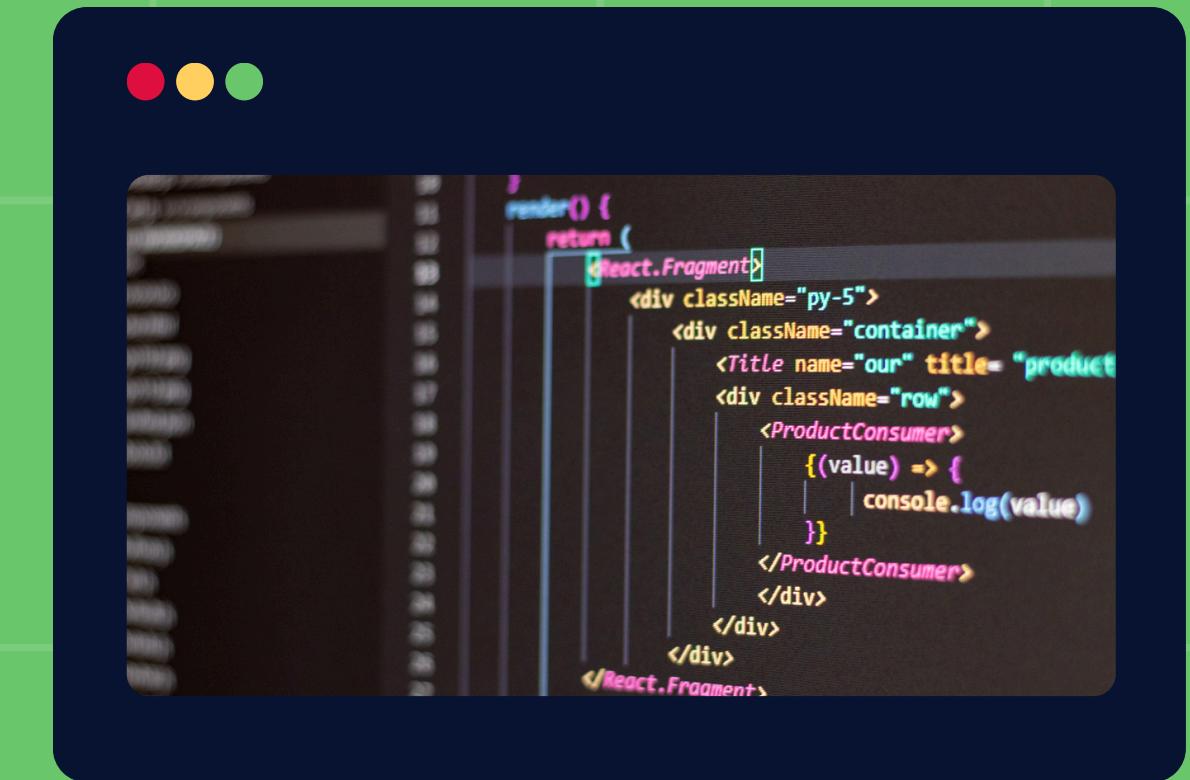


Adapter w Javie

“Wzorce choinkowe”



```
render() {
  return (
    <React.Fragment>
      <div className="py-5">
        <div className="container">
          <Title name="our" title= "product" />
          <div className="row">
            <ProductConsumer>
              {(value) => {
                console.log(value)
              }}
            </ProductConsumer>
          </div>
        </div>
      <React.Fragment>
    
```

A screenshot of a code editor showing a React component. The component uses a Fragment to group several elements: a container div containing a title and a row of ProductConsumer components, followed by another Fragment. The code is written in a syntax highlighting style.

Prezentacja autorstwa
Katarzyna Bęben

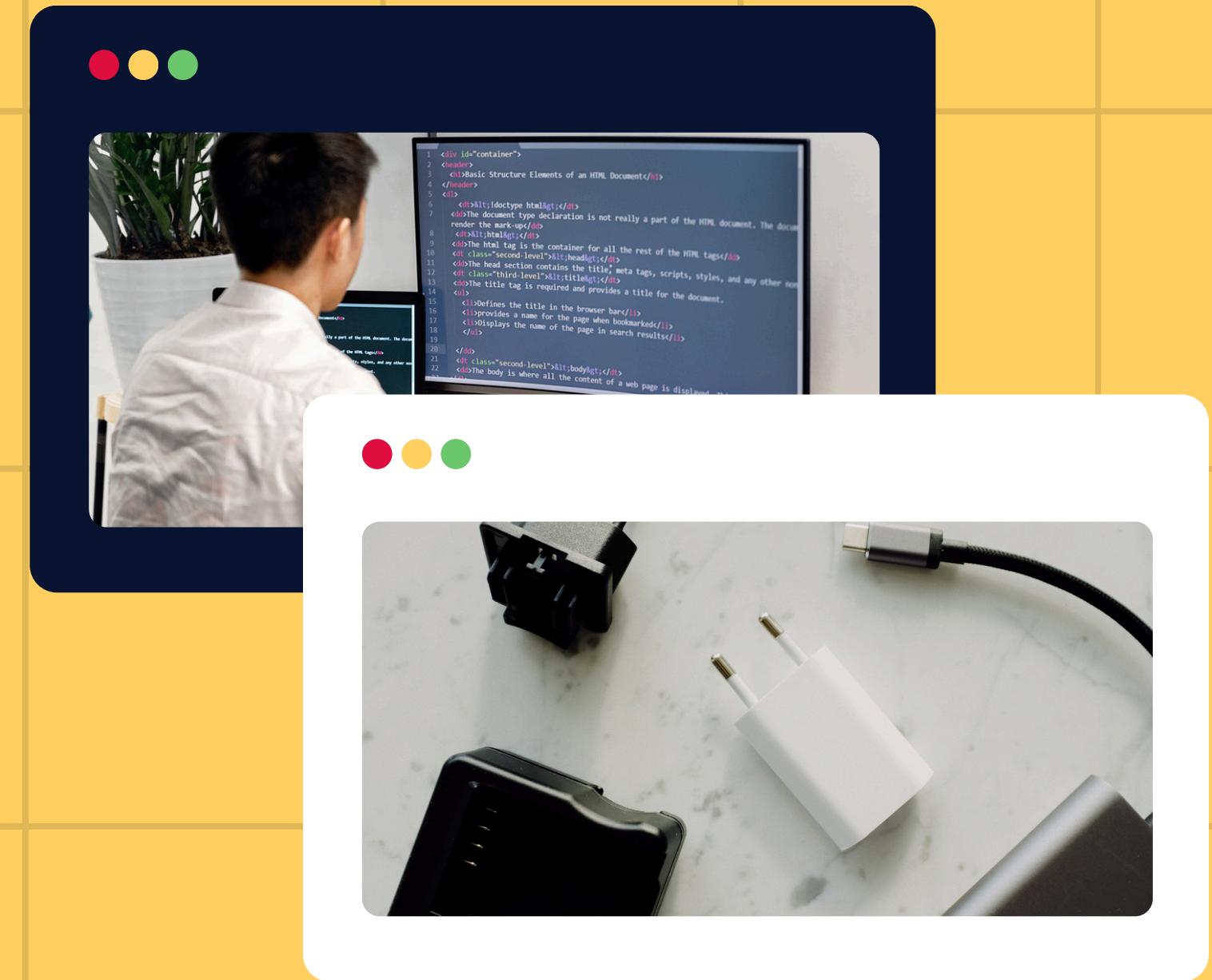
Czym jest adapter?



Adapter jest strukturalnym wzorcem projektowym umożliwiającym współpracę niekompatybilnych obiektów.

Wzorce strukturalne koncentrują się na łączeniu obiektów i klas w większe struktury poprzez definiowanie relacji między nimi zachowując ich elastyczność i wydajność.

Inne nazwy: Opakowanie, Nakładka, Wrapper



Kiedy używamy tego wzorca?

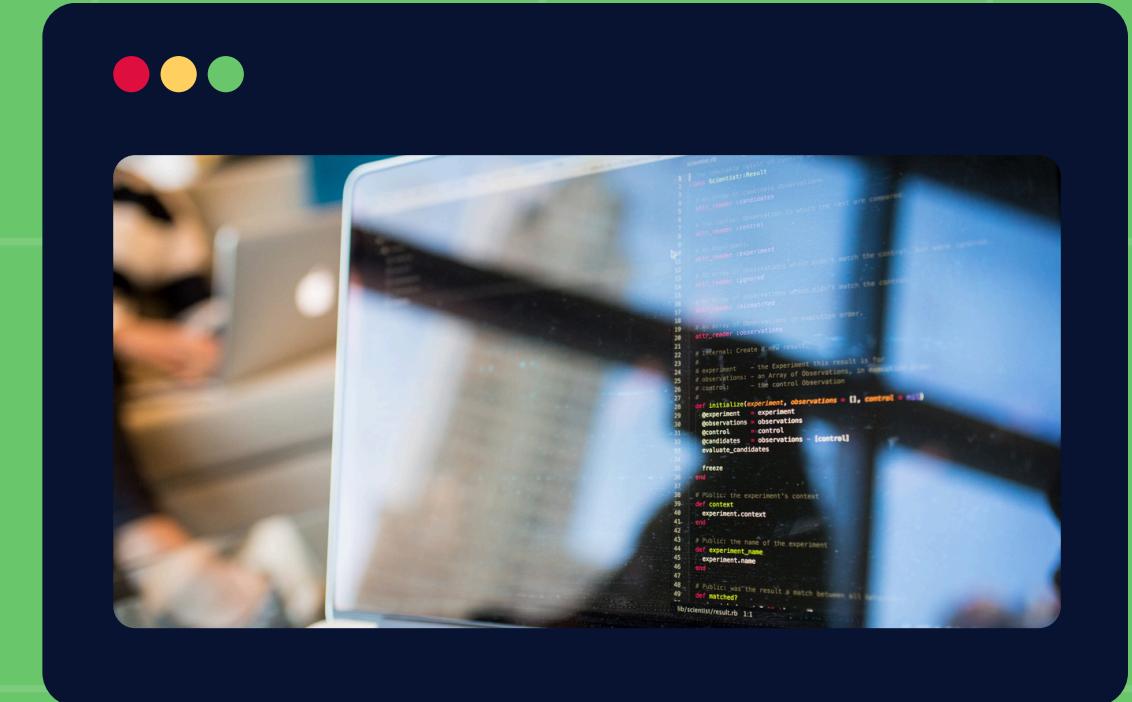
Kiedy musimy korzystać z klasy z niezgodnym interfejsem



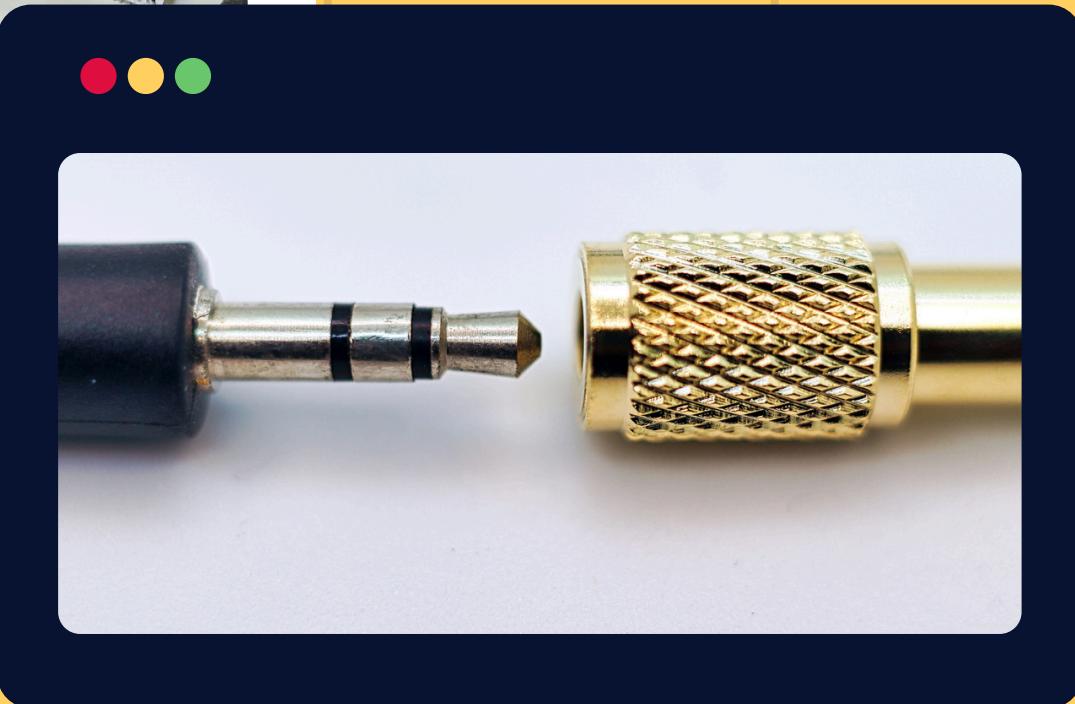
Kiedy chcemy użyć istniejący kod bez jego modyfikacji



Kiedy integrujemy systemy lub biblioteki o różnych interfejsach



Działanie



Client ---> Target <--- Adapter ---> Adaptee

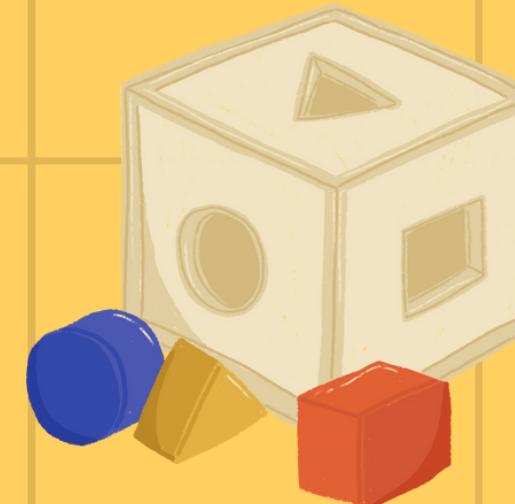
- Client: Kod, który ma używać adaptera
- Target: Wymagany interfejs
- Adapter: Klasa dostosowująca Adaptee do interfejsu Target
- Adaptee: Klasa z niekompatybilnym interfejsem, którą chcemy dostosować



Przepakowanie jednego żądania w drugie, tak aby strona wysyłająca żądanie pracowała bez zmian i aby nie zmieniać istniejących struktur a dostosować ich działanie wewnątrz adaptera.

Cała logika zostaje w adapterze i jest oddzielona od logiki pozostałych klas.

Adapter udaje że to on jest punktem docelowym, a tak naprawdę tylko kryje inne obiekty.



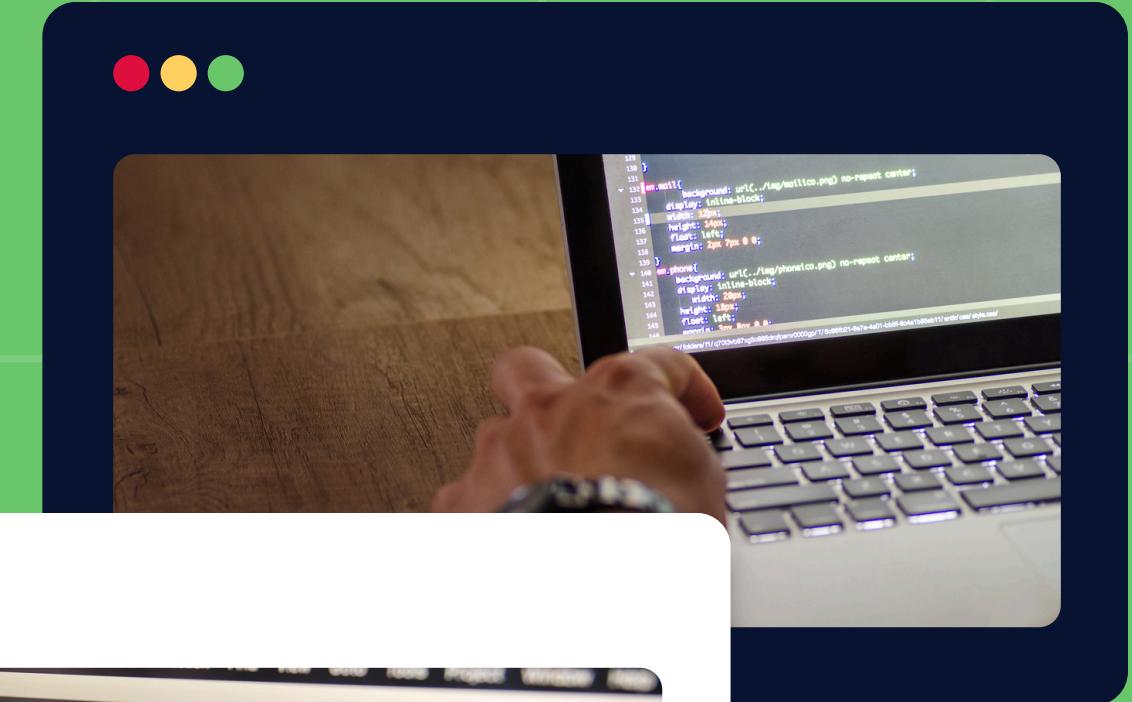
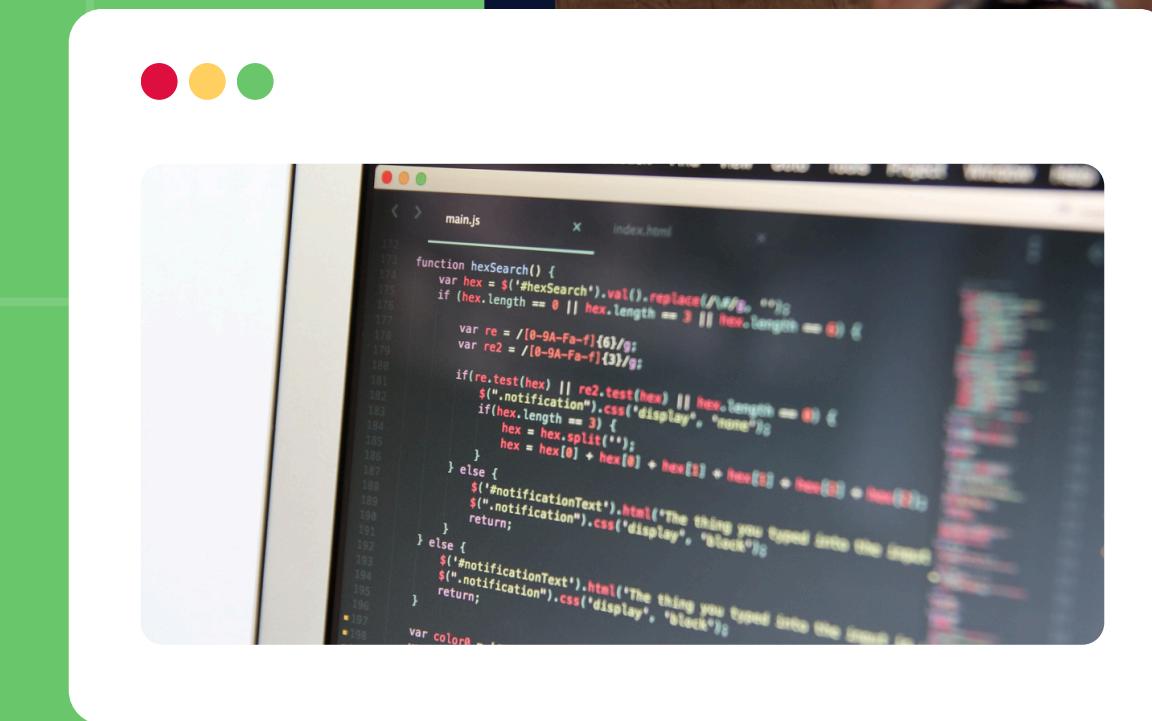
Przykład

```
// Interfejs oldCar
public interface OldCar {
    void driveManual();
}

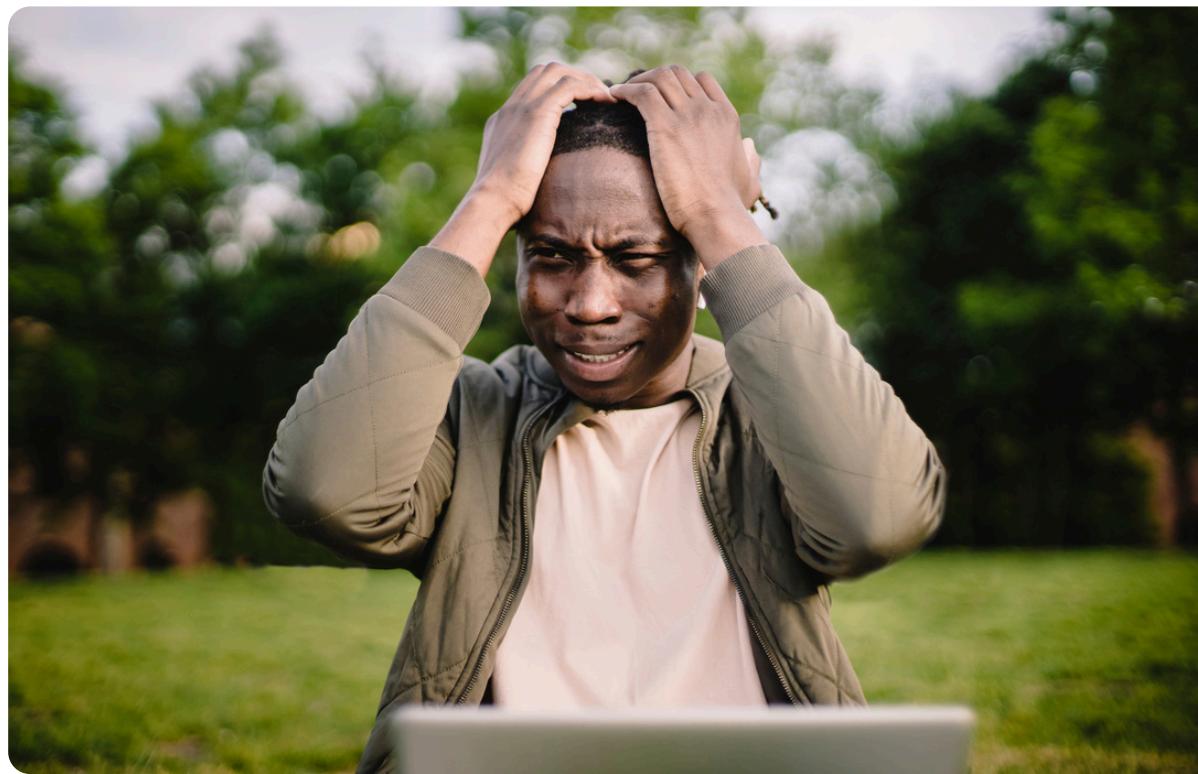
// Klasa realizująca interfejs OldCar
public class ManualCar implements OldCar {
    @Override
    public void driveManual() {
        System.out.println("ManualCar: Jazda
manuelna");
    }
}

// Interfejs newCar
public interface NewCar {
    void drive();
}

// Klasa Driver korzystająca z newCar
public class Driver {
    public void drive(NewCar car) {
        car.drive();
    }
}
```

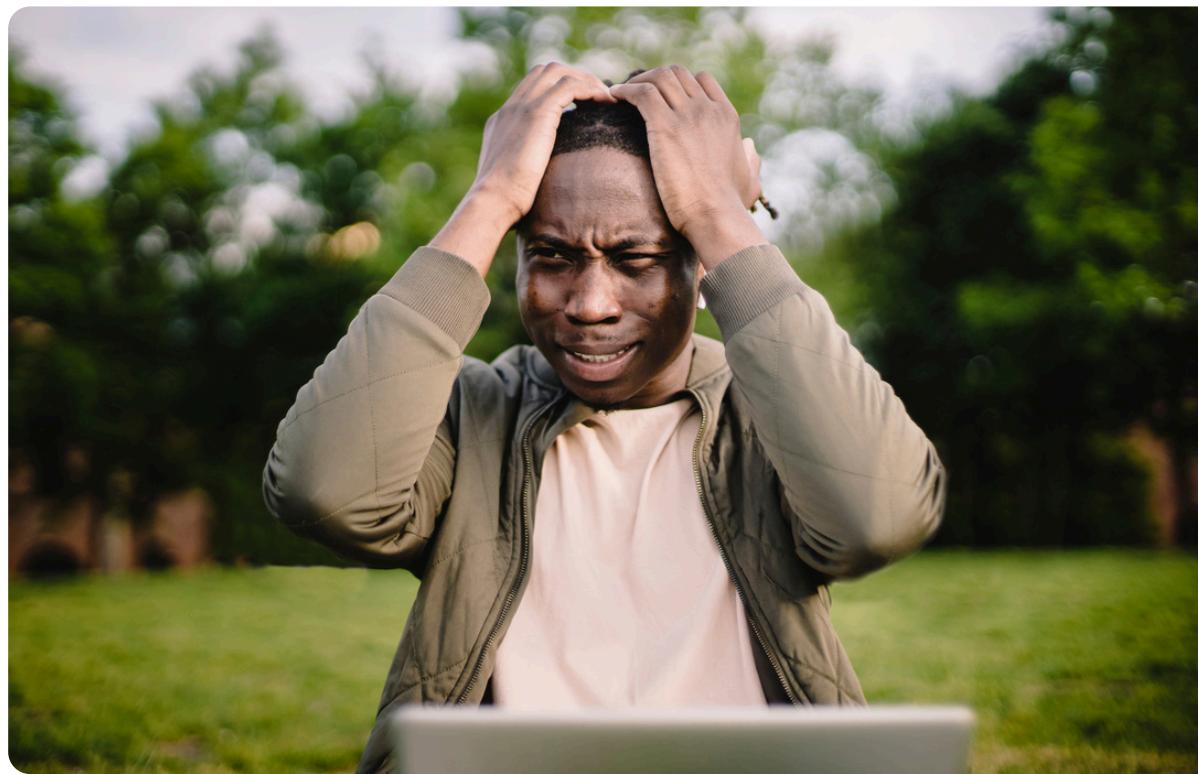


Problem . . .



```
public class ChceJechac {  
    public static void main(String[] args) {  
        OldCar manualCar = new ManualCar();  
  
        Driver driver = new Driver();  
        System.out.println("Driver nie umie  
jeszcze jeździć manualem!");  
        driver.drive(manualCar);  
    }  
}
```

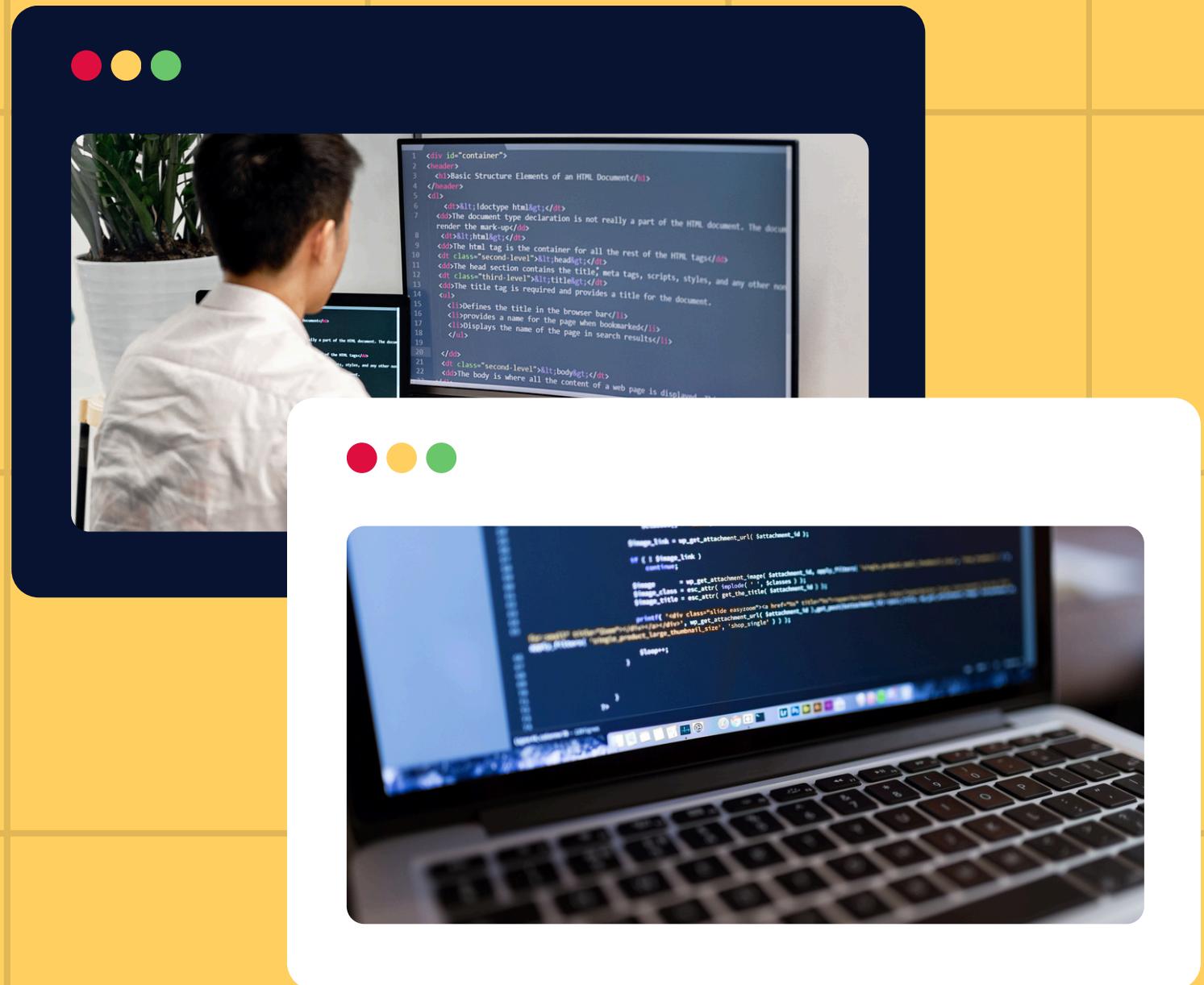
Problem . . .



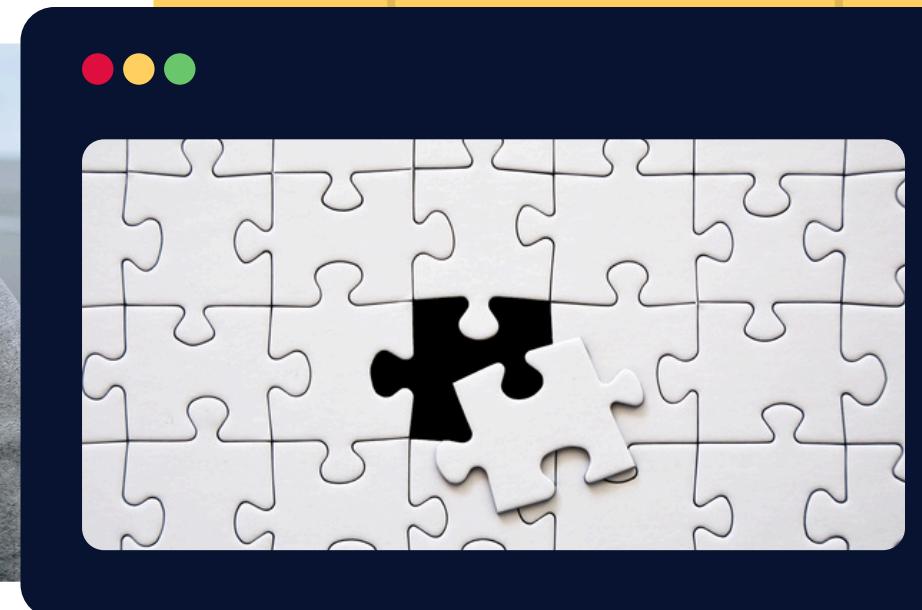
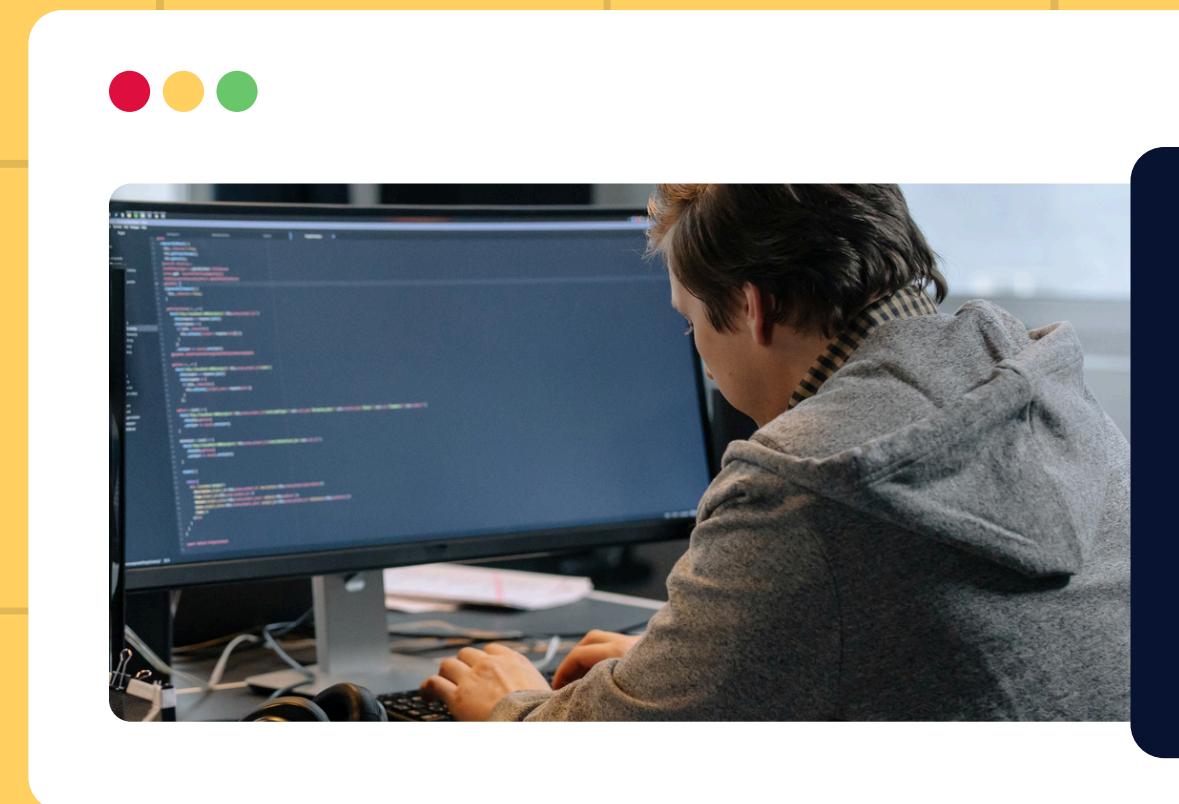
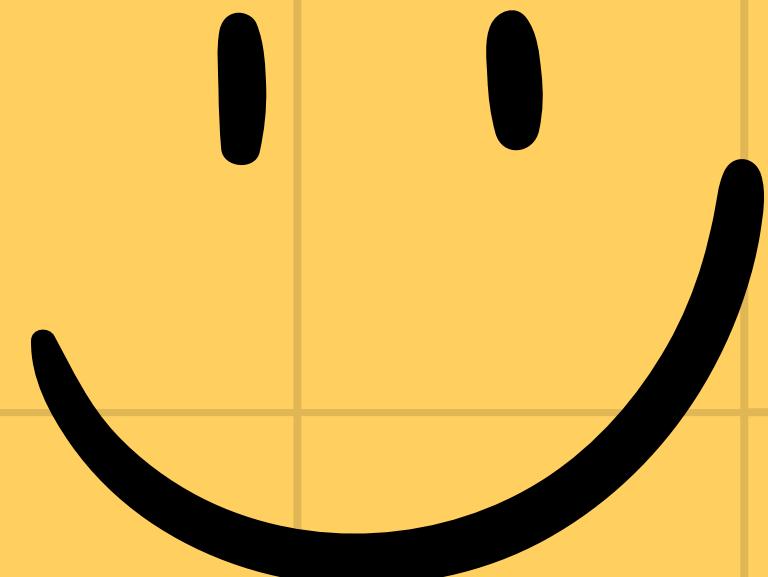
```
public class ChceJechac {  
    public static void main(String[] args) {  
        OldCar manualCar = new ManualCar();  
  
        Driver driver = new Driver();  
        System.out.println("Driver nie umie  
jeszcze jeździć manualem!");  
        driver.drive(manualCar);  
    }  
}
```

Rozwiążanie

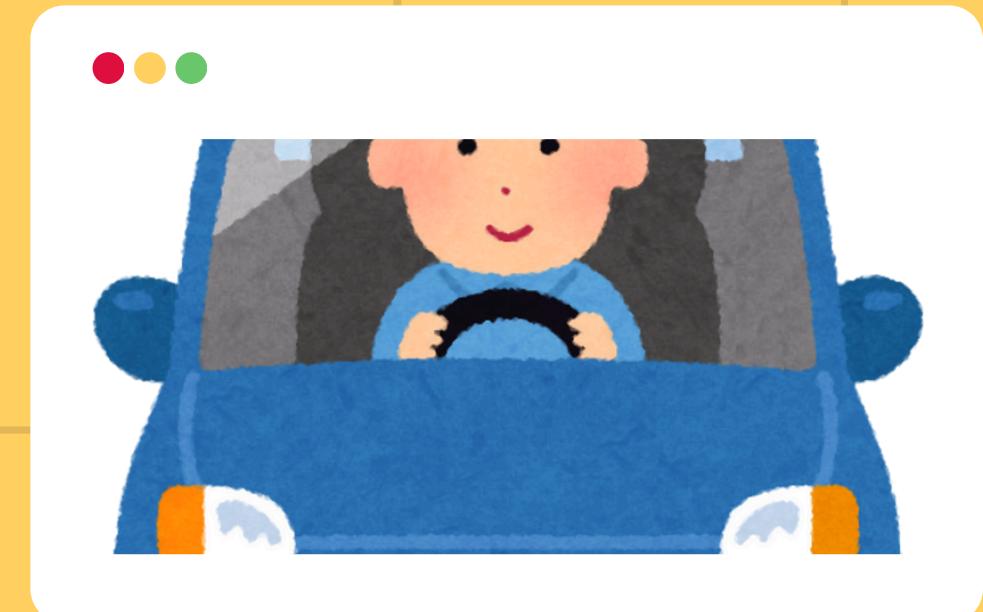
```
● ● ●  
// Adapter dostosowujący OldCar do NewCar  
public class CarAdapter implements NewCar {  
    private OldCar oldCar;  
  
    public CarAdapter(OldCar oldCar) {  
        this.oldCar = oldCar;  
    }  
  
    @Override  
    public void drive() {  
        oldCar.driveManual();  
    }  
}
```

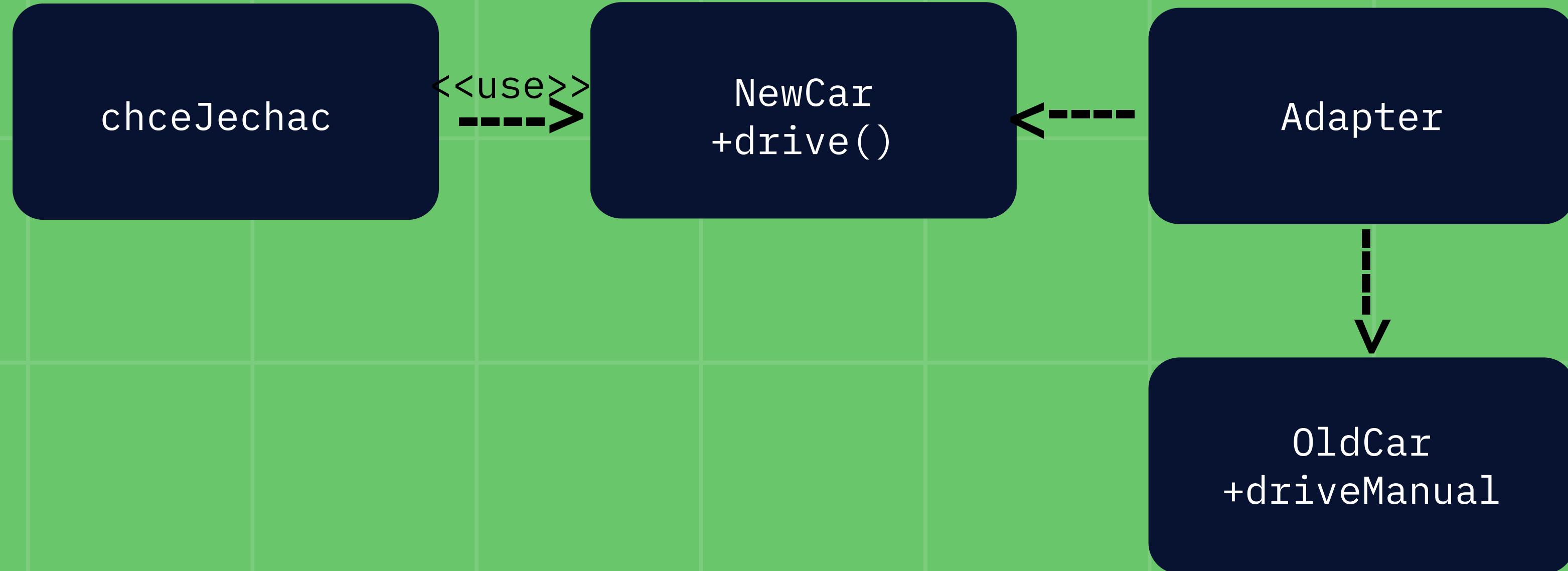


Działa !



```
public class ChceJechac {  
    public static void main(String[] args) {  
        OldCar manualCar = new ManualCar();  
        NewCar adaptedCar = new  
        CarAdapter(manualCar);  
  
        Driver driver = new Driver();  
        System.out.println("Driver: Jazda z  
zaadaptowanym samochodem");  
        driver.drive(adaptedCar);  
    }  
}
```



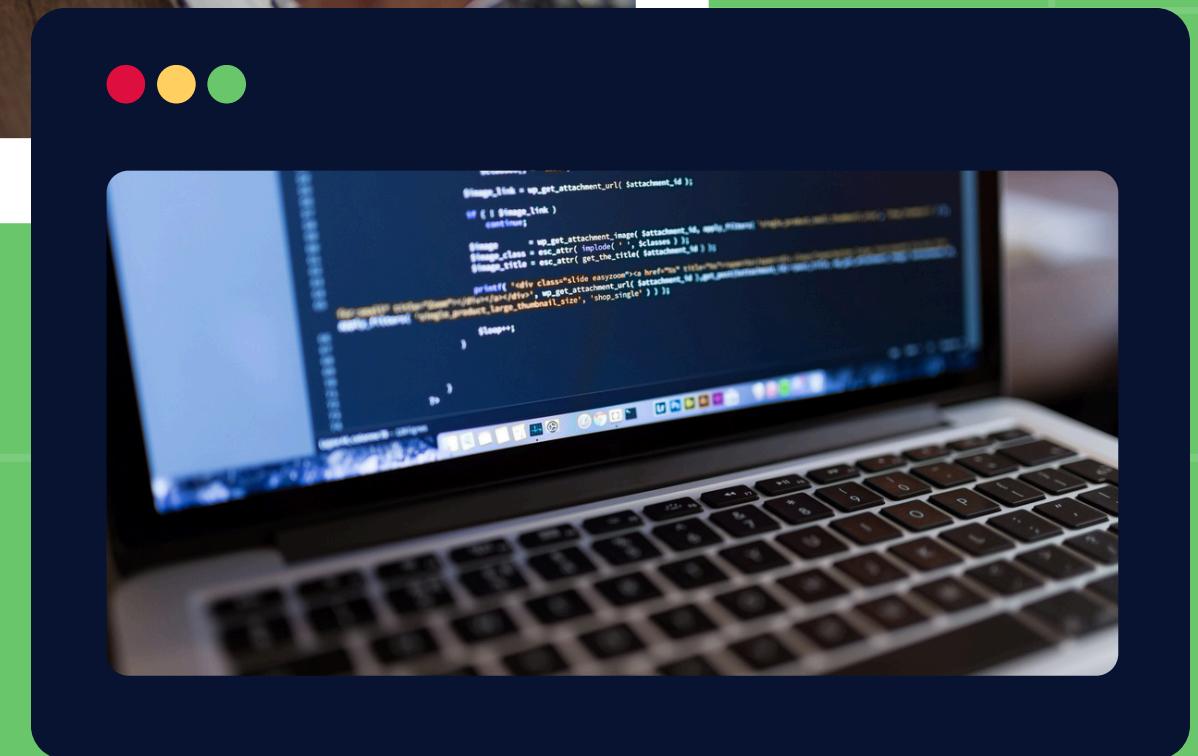
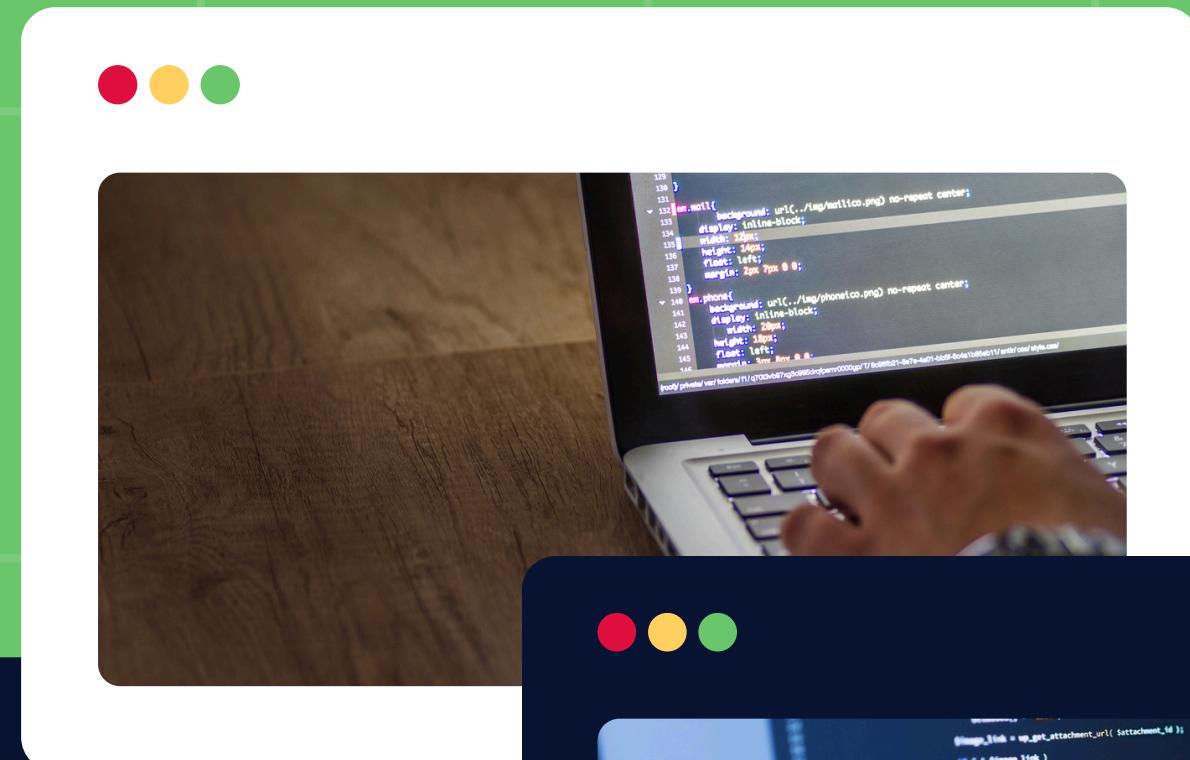


Podsumowanie



- # Adapter umożliwia współpracę niekompatybilnych interfejsów.
- # Pozwala na rozszerzenie funkcjonalności bez modyfikacji istniejącego kodu.
- # Jest kluczowy w integracji systemów i bibliotek.
- # Spełnia regułę pojedynczej odpowiedzialności (oddzielona logika) oraz otwarte/zamknięte (łatwa podmiana adaptera).





Dziękuję za
uwagę