```
public static class IDCreator
{
    private static int UniqueID = 0;
    public static int GetUniqueID()
    {
        UniqueID++;
        return UniqueID;
    }
}
```

Rysunek 1. Klasa IDCreator

Instytucja banku przechowuje konta bankowe krasnali,sklepu oraz gildii krasnali za pomocą słownika Dictionary<ID,BankAccount>. Założeniem było aby każde konto bankowe posiadało swój unikalny klucz ID. W myśl zasady KISS (Keep it simple) utworzyłem statyczną klasę IDCreator, która zawiera metodę GetUniqeID() zwracającą unikalne ID. Metoda wywoływana jest przy tworzeniu kont bankowych i przy każdym wywołaniu zwraca wartość ID o jeden większą. Założeniem tej klasy była prostota, zrozumienie jak i również łatwość użycia.

```
public class Shop
{
    private BankAccount _bankAccount;
    public Shop()
    {
        _bankAccount = AccountCreator.CreateNewAccount(IDCreator.GetUniqueID());
    }
    public void PerformShopping(Dwarf dwarf)
    {
        dwarf.Buy(_bankAccount);
    }

    public void PerformShoppingForAllDwarves(List<Dwarf> dwarves)
    {
        foreach(Dwarf dwarf in dwarves)
        {
            PerformShopping(dwarf);
        }
    }
    public BankAccount GetBankAccount() => _bankAccount;

}
```

Rysunek 1. Klasa Shop

Klasa Shop umożliwia krasnalom dokonywanie zakupów. Posiada ona metodę PerformShoppingForAllDwarves, która przyjmuje na wejściu listę krasnali. Dla każdego krasnala wykonuje ona metodę PerformAction, która uruchamia odpowiednią strategię zakupu krasnala (rys. 3). Krasnal Father posiada strategię zakupu alkoholu za połowę jego pieniędzy (rys 4). Krasnal Single posiada strategię zakupu jedzenia za połowę jego wszystkich pieniędzy (rys 5). Krasnal Lazy nie kupuje nic w sklepie. W myśl zasady SOLID, a dokładniej zasady Open - close principle mówiącej o tym by kod był zamknięty na modyfikacje a otwarty na rozszerzenia, zastosowałem wzorzec strategii. Strategia umożliwia wykonywanie jednej czynności na wiele sposobów (polimorfizm). Dzięki temu dodanie nowej strategii zakupów dla nowego typu krasnali nie spowoduje modyfikacji w już istniejącym kodzie a rozszerzy jego funkcjonalność.

```
public interface IShoppingStrategy
{
    void Buy(BankAccount dwarfBankAccount,BankAccount bankAccountToTopUp);
}
```

Rysunek 3. Interfejs strategii zakupu

```
public class FatherShoppingStrategy : IShoppingStrategy
{
    public void Buy(BankAccount dwarfBankAccount, BankAccount bankAccountToTopUp)
    {
        decimal moneyForTransaction = dwarfBankAccount.CheckMoneyOnAccount() / 2;
        dwarfBankAccount.MakeTransaction(bankAccountToTopUp, moneyForTransaction);
        Logger.GetInstance().AddLog("Dwarf father bought alcohol");
    }
}
```

Rysunek 4. Strategia zakupu dla krasnala typu Father

```
public class LazyShoppingStrategy : IShoppingStrategy
{
    public void Buy(BankAccount dwarfBankAccount, BankAccount bankAccountToTopUp)
    {
        // Do nothing , Too lazy to buy
        Logger.GetInstance().AddLog("Dwarf lazy did not buy nothing");
    }
}
```

Rysunek 5. Strategia zakupu dla krasnala typu Lazy

```
public class SingleShoppingStrategy : IShoppingStrategy
{
    public void Buy(BankAccount dwarfBankAccount, BankAccount bankAccountToTopUp)
    {
        decimal moneyForTransaction = dwarfBankAccount.CheckMoneyOnAccount() / 2;
        dwarfBankAccount.MakeTransaction(bankAccountToTopUp, moneyForTransaction);
        Logger.GetInstance().AddLog("Dwarf single bought food");
    }
}
```

Rysunek 6. Strategia zakupu dla krasnala typu Single

Klasa sklep została przetestowana za pomocą testów jednostkowych przy użyciu narzędzia NUnit. Na rysunkach 7 oraz 8 zostały przedstawione wybrane testy. Sprawdzają one prawidłowe działanie sklepu dla klientów typu Father oraz Lazy. Testy umożliwiły mi sprawdzenie działania odpowiednich funkcji niezależnie od innych klas oraz niezależnie od poziomu ukończenia projektu jako całości. Zastosowanie wzorca projektowego strategia dla zakupów umożliwia testowanie poszczególnych strategii osobno. Zmniejsza się zależność w kodzie co ułatwia nie tylko rozszerzanie kodu ale jak i również jego testowanie.

```
[Test]
public void WhenDwarfFatherVisitsShopThenShopAccountIncreasesForHalfDwarfMoney()
{
    // Given
    Shop shop = new Shop();
    shop.GetBankAccount().TopUp(100);
    Dwarf dwarf = DwarfFactory.CreateDwarf(DwarfType.Father);
    dwarf.GetBankAccount().TopUp(200);
    decimal expectedDwarfMoney = 100;
    decimal expectedShopMoney = 200;

    // when
    shop.PerformShopping(dwarf);

    //then
    decimal actualDwarfMoney = dwarf.GetBankAccount().CheckMoneyOnAccount();
    decimal actualShopMoney = shop.GetBankAccount().CheckMoneyOnAccount();
    Assert.AreEqual(expectedDwarfMoney, actualDwarfMoney);
    Assert.AreEqual(expectedShopMoney, actualShopMoney);
}
```

Rysunek 7. Test jednostkowy klasy Shop sprawdzający prawidłowe działanie dla krasnali typu Father.

```
[Test]
public void WhenDwarfLazyVisitsShopThenShopAccountDoesntChange()
{
    // Given
    Shop shop = new Shop();
    shop.GetBankAccount().TopUp(100);
    Dwarf dwarf = DwarfFactory.CreateDwarf(DwarfType.Lazy);
    dwarf.GetBankAccount().TopUp(200);
    decimal expectedDwarfMoney = 200;
    decimal expectedShopMoney = 100;

    // when
    shop.PerformShopping(dwarf);

    //then
    decimal actualDwarfMoney = dwarf.GetBankAccount().CheckMoneyOnAccount();
    decimal actualShopMoney = shop.GetBankAccount().CheckMoneyOnAccount();
    Assert.AreEqual(expectedDwarfMoney, actualDwarfMoney);
    Assert.AreEqual(expectedShopMoney, actualShopMoney);
}
```

Rysunek 8. Test jednostkowy klasy Shop sprawdzający prawidłowe działanie dla krasnali typu Lazy.

Zaimplementowałem podstawową wersję logującą w klasie Logger (rys.9). Użyłem do tego celu wzorca projektowego Singleton. Wzorzec ten daje sposobność posiadania dokładnie jednej instancji danej klasy w tym przypadku loggera. Loger ma możliwość logowania danych z każdego miejsca w kodzie co bardzo usprawniło nam pracę przy tworzeniu raportu końcowego symulacji krasnali. Singleton często postrzegany jest jako antywzorzec - jest nie rozszerzalny oraz trudno testowalny ze względu na to iż jest klasą statyczną. Zdecydowałem sie na jego użycie świadomie oraz dobrowolnie. Uważam, że w tej sytuacji spełnia swoje zadanie. Aby poradzić sobie z przetestowaniem loggera musiałem utworzyć metodę ClearData, którą uruchamiam przed każdym testem związanym z logowaniem.

```
public sealed class Logger
{
    static private Logger logger = new Logger();
    private List<string> reports = new List<string>();

    private Logger() { }
    static public Logger GetInstance() => logger;
    public void AddLog(string log)
    {
        reports.Add(log);
    }
    public List<string> GetLogs() => reports;

    public void ClearData()
    {
        reports.Clear();
    }
}
```

Rysunek 9. Klasa Logger