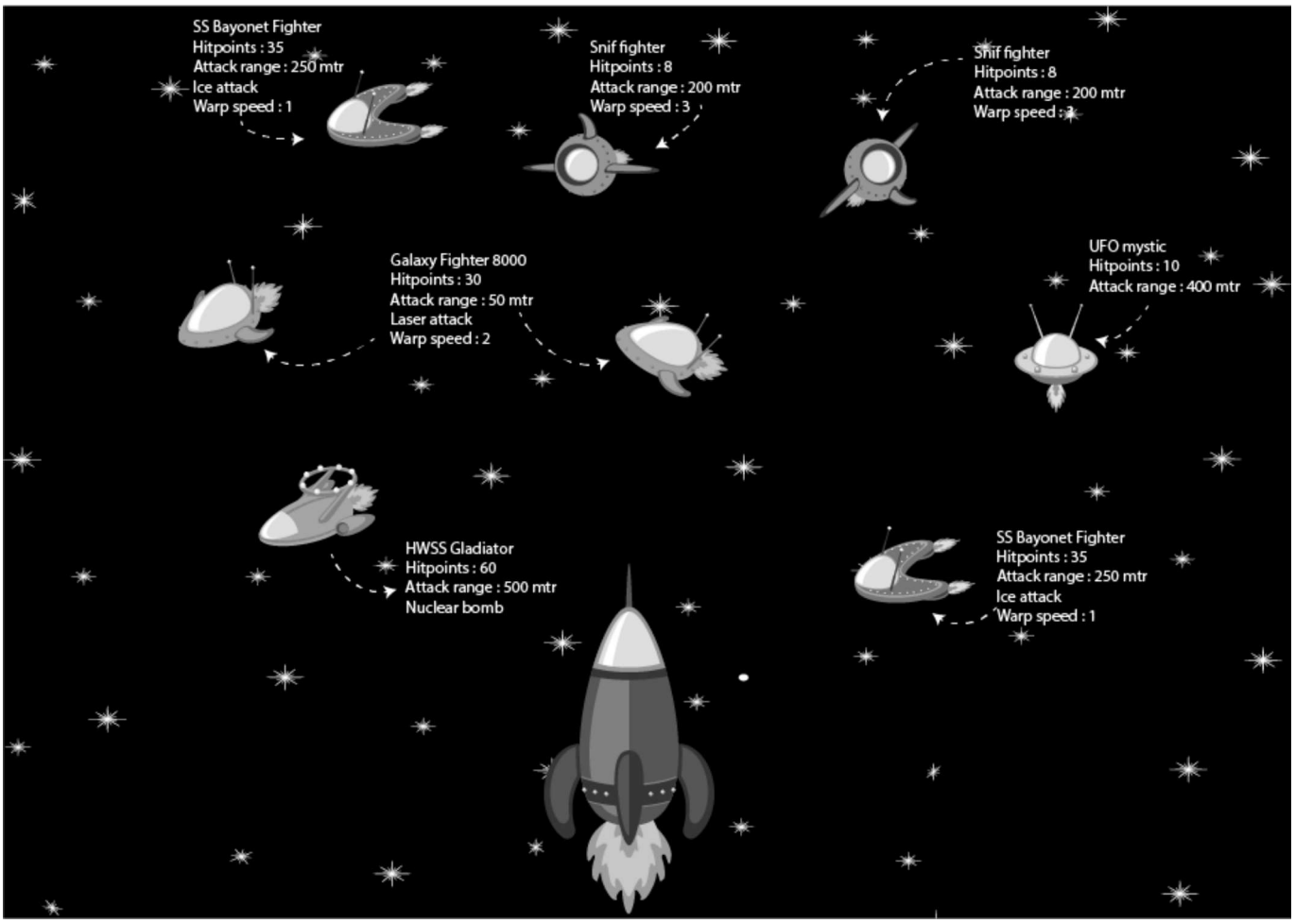
Marker som gennemført

Et factory patterns fineste opgave er at indkapsle objektoprettelse og være ansvarlig for at instantiere objekter. Det smarte ved at benytte en factory er at du mindsker koblingen ved ikke at lade dine klasser om objektoprettelsen. Et factory pattern er et rigtigt godt eksempel på høj samhørighed - altså gør det du er bedst til. En factory pattern bruges kun til objektoprettelser, dog skal man være opmærksom på at en factory først giver mening når man har en nogle subklasser, der deler en fælles superklasse.

Du kan med fordel benytte et factory pattern når :

- Du ikke ved, før under run-time hvilke objekter du vil instantiere
- Alle klasser har en fælles superklasse
- Du vil indkapsle objektoprettelsen

Lad os forestille os at vi skal lave et mindre skydespil, hvor der dukker "random" fjender på forskellige lokationer op hele tiden, som skal skydes. Alle fjender skal dele en fælles superklasse som vi kan kalde enemy.



Lad os først få oprettet superklassen EnemyShip som alle fjender skal arve fra

?

```
public abstract class EnemyShip {

  string name;
  byte hitpoints;
  int attackRange;

  protected EnemyShip(string name, byte hitpoints, int attackRange)
  {
      Name = name;
      Hitpoints = hitpoints;
      AttackRange = attackRange;
  }

  public string Name { get => name; set => name = value; }
  public byte Hitpoints { get => hitpoints; set => hitpoints = value; }
  public int AttackRange { get => attackRange; set => attackRange = value;}

  public override string ToString()
  {
     return Name;
  }
}
```

Efterfølgende skal vi have defineret de forskellige subklasser.

```
public class SpaceFighter : EnemyShip
{
   byte warpSpeed;

   public SpaceFighter(string name, byte hitpoints, int attackRange, byte warpSpeed)
    : base(name, hitpoints, attackRange)
   {
       WarpSpeed = warpSpeed;
   }

   public byte WarpSpeed { get => warpSpeed; set => warpSpeed = value; }
}
```

```
  public class SpaceBomber : EnemyShip
  {
      int  bombSize;

      public SpaceBomber(string name, byte hitpoints, int attackRange, byte bombSize)
         : base(name, hitpoints, attackRange)
      {
         BombSize = bombSize;
      }

      public int BombSize { get => bombSize; set => bombSize = value; }
}
```

```
public class Ufo : EnemyShip
{

   public Ufo(string name, byte hitpoints, int attackRange)
          : base(name, hitpoints, attackRange)
   {

   }
}
```

Nu er vi klar til det vi egentlig skulle, nemlig at oprette og definere en factory klasse, som skal tage sig af at instantiere forskellige typer af fjender.

```
public class EnemyFactory
{
    Random r = new Random();
    public EnemyShip CreateEnemyShip()
    {
       //Herinde laver vi en random som sørger for at vi får et random fjendeskib
       int t = r.Next(5) + 1;
       switch (t)
       {
          case 1: return new SpaceFighter("SS Bayonet Fighter", 35, 250, 1); ;
          case 2: return new SpaceFighter("Snif fighter", 8, 200, 3);
                    case 3: return new SpaceFighter("Galaxy Fighter 8000", 30, 60, 2);
          case 4: return new Ufo("Galaxy Fighter 8000", 10, 400);
          case 5: return new SpaceBomber("HWSS Gladiator", 60, 500, 254);
          default:
             return null;

        }
    }
}
```

Bemærk, når man benytter et factory pattern at oprettelsesmetoden altid starter med Create - og at klassen altid navngives med Factory.

Nu er der ikke andre end selve factoryklassen der ved noget om hvordan man opretter disse fjender og det betyder at instantieringen er indkapslet og derved er koblingen mellem klasserne lav.

Eksempel

```
for (int i = 0; i < 100; i++)
{
   Console.WriteLine(new EnemyFactory().CreateEnemyShip().ToString());
}
```

Senest ændret: onsdag den 10. januar 2018, 13:02

◄ Opgave singleton : Chew and phew

Spring til...

Opgave factory : Vito's pizza ►