

# Interfaces

OOP 2020

Thomas Bøgholm  
boegholm@cs.aau.dk

# Indtil videre

- Alt foregår i klasser
  - Data+opførsel
  - Detaljer kan gemmes med access modifiers
    - Færre dependencies=>(færre rettelser | færre fejl)
- Nedarvningsrelation mellem klasser
  - Specialisering
  - Abstraktion
  - Polymorfi

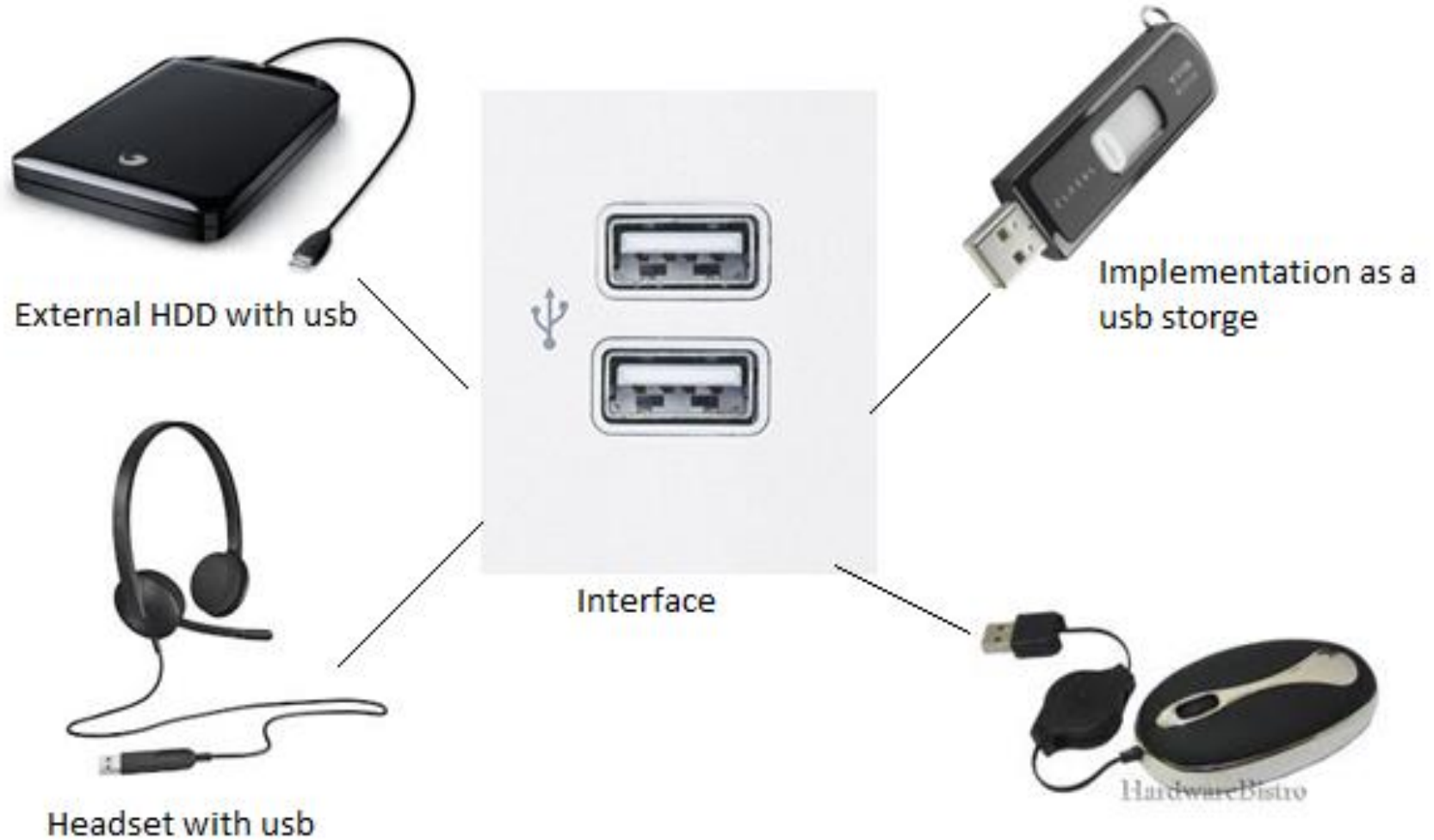
# Overblik

- Interfaces:
  - Polymorfi i flere hierarkier.
  - Kodegenbrug uden nedarvning (aggregering/komposition)
  - Abstrakte klasser vs. interfaces

# Interfaces

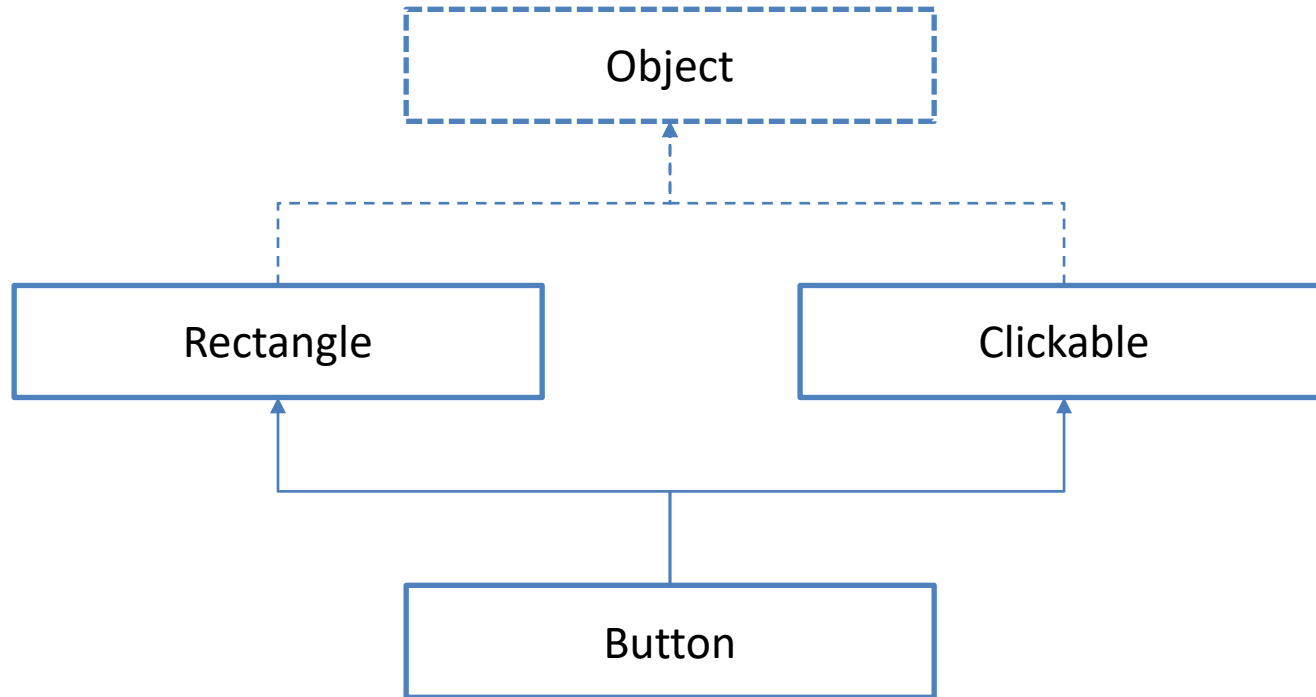
- **interface** keyword der definerer et antal (logisk relaterede) medlems-*signaturer*.
- Interface  $\approx$  abstrakt klasse med udelukkende (public) abstrakte medlemmer
  - Ingen implementation, ingen data, og kan ikke instantieres.
  - Definerer et interface som alle subtyper overholder.
    - ALLE medlemmer skal implementeres i subtyper

# Interfaces vi kender



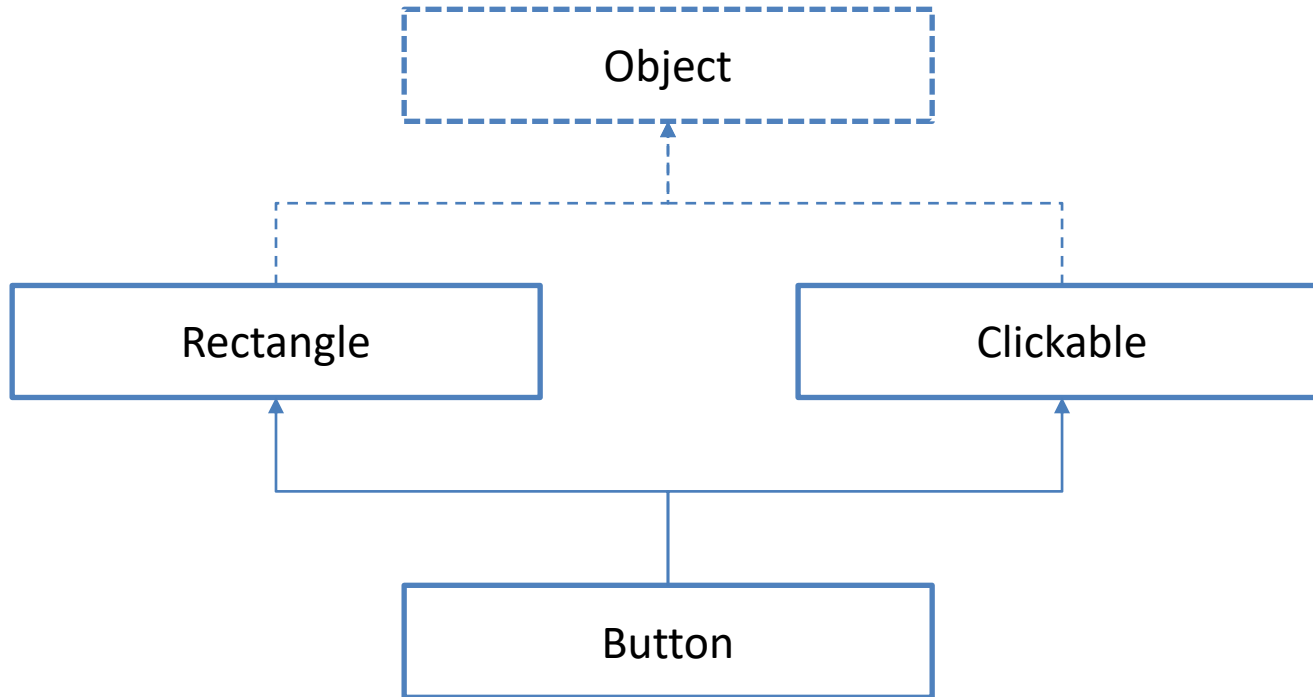
- Resten foregår i VS

# Multipel nedarvning



- Smart!

# Multipel nedarvning

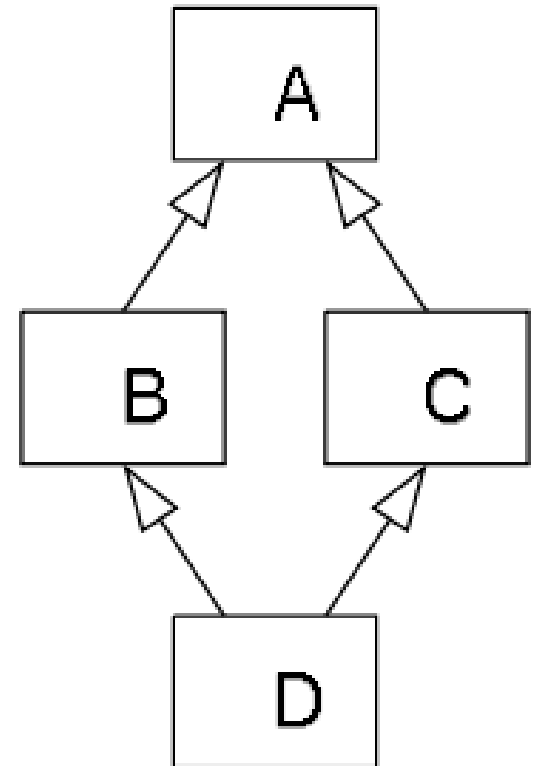


- Det kan vi ikke i C#... `button.Equals()`?



# Multipel nedarvning

- Diamantproblemet:
- A definerer en virtuel metode *Foo()* som B og C - men ikke D - overrider.
  - Hvad giver d.Foo()?
  - Python: b.Foo(). Resolution: D->B->C->A
  - Scala: c.Foo(). Resolution: D->C->A->B->A
- Derudover giver multipel nedarvning overflødigt fedt på subklasser.



# Single nedarvning

- Ingen multipel nedarvning i C#
- En person kan ikke være bade worker og player

```
class Worker { public string WorkHard() { return "Working hard"; } }  
class Player { public string PlayHarder() { return "Playing harder"; }}  
  
public class Person: Worker //Arver fra Worker  
{  
}  
  
public class Person: Player //Arver fra Player  
{  
}
```

# Single nedarvning

- Ingen multipel nedarvning i C#
- Vi kan dog genbruge kode via objektreference (aggregering/komposition).
- Begrænsning: Vi har kun polymorfi ifht. én af forældrene (se næste side)

```
class Worker { public string WorkHard() { return "Working hard"; } }  
class Player { public string PlayHarder() { return "Playing harder"; }}  
  
public class Person: Worker //Arver fra Worker  
{  
    private Player _player; //Reference til Player (aggregering)  
  
    public string PlayHarder()  
    {  
        return _player.PlayHarder();  
    }  
}
```

# Fordelen ved polymorfi er væk for Person:

```
static void Main(string[] args)
{
    Player per = new Player();
    Player lis = new Player();
    Person ole = new Person();

    //ole kan ikke bruges, da han ikke er en gyldig Player
    List<Player> players1 = new List<Player> { per, lis, /* ole */ };

    //Vi kan kun gruppere players og personer sammen via object:
    List<object> players2 = new List<object> { per, lis, ole };

    //Men så skal vi lave cast for at tilgå PlayHarder()-metoden.
    foreach (object player in players2)
        if (player is Player)
            ((Player)player).PlayHarder();
        else if (player is Person)
            ((Person)player).PlayHarder();
}
```

```
//definition af interface
```

```
interface IFileCompressor
```

```
{
```

```
    void Compress(string targetFile, string[] sourceFiles);
```

```
    void Uncompress(string targetFile, string expandedDir);
```

```
}
```

```
// implementation af interface
```

```
class ZipCompressor : IFileCompressor
```

```
{
```

```
    public void Compress(string targetFile, string[] sourceFiles) { /*...*/ }
```

```
    public void Uncompress(string targetFile, string expandDirectoryName) {
```

```
    /*...*/ }
```

```
}
```

```
class RarCompressor : IFileCompressor
```

```
{
```

```
    public void Compress(string targetFile, string[] sourceFiles) { /*...*/ }
```

```
    public void Uncompress(string targetFile, string expandDirectoryName) {
```

```
    /*...*/ }
```

```
}
```

```
... 7zCompressor, GzCompressor, ...
```

Kan kun være public

# InterfaceImplementation

- En klasse kan kun arve fra én klasse
- Men kan implementere flere interfaces
- Herved får vi fordelene ved polymorfi



# Polymorfi med interfaces

```
interface IWorker {    string Work ();    }
interface IPlayer {    string Play () ;    }

class SoftwareEngineer : IWorker {
    public string Work () { return "developers, developers,.."; }
}

class Gamer : IPlayer {
    public string Play () { return "Dota, dota, dota,..."; }
}

class TaxiDriver : IWorker, IPlayer {
    public string Work () { return "Driving along"; }
    public string Play () { return "Playing bingo"; }
}
```

```
private static void TestInterfacePolymorfi() {
    SoftwareEngineer s = new SoftwareEngineer();
    Gamer m = new Gamer();
    TaxiDriver t = new TaxiDriver();

    List<IWorker> workers = new List<IWorker>() { s, t };
    List<IPlayer> players = new List<IPlayer>() { m, t };
}
```

Foreach:

Work()

Play()

# Interfaces og genbrug

- Interfaces giver *kun* **polymorfi** - ingen direkte genbrug
- Vi kan dog bruge **aggregering/komposition** til genbrug
- Ved aggregering/komposition delegeres ansvaret for udførelse af metode videre til et andet objekt.

```
class SoftwareMusician : IWorker, IPlayer
{
    private IWorker _workerSlave = new SoftwareEngineer();
    private IPlayer _playerSlave = new Musician();

    public string Play()
    {
        return _playerSlave.Play();
    }

    public string Work()
    {
        return _workerSlave.Work();
    }
}
```



# Eksempler på library interfaces

- Comparable
  - Sammenligning med compareTo
- Comparer
  - Afkobling af sammenligning

# IComparable

- Bruges til at erklære typer for sammenlignelige

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

# IComparable

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
}
```

```
class Person : IComparable
{
    public Person(string navn, int alder)
    {
        Navn = navn;
        Alder = alder;
    }
    public string Navn { get; set; }
    public int Alder { get; set; }
    public int CompareTo(object obj)
    {
        ..?
    }
}
```

# IComparable

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
}
```

```
class Person : IComparable
{
    public Person(string navn, int alder)
    {
        Navn = navn;
        Alder = alder;
    }
    public string Navn { get; set; }
    public int Alder { get; set; }
    public int CompareTo(object obj)
    {
        return Alder.CompareTo(((Person)obj).Alder);
    }
}
```

# IComparer

```
class ComparePersonByName : IComparer
{
    public int Compare(object x, object y)
    {
        return ((Person) x).Navn.CompareTo(((Person) y).Navn);
    }
}
```

```
Person[] ps = new Person[]
{
    new Person("Thomas", 10),
    new Person("Anders", 15)
};
Array.Sort(ps, new ComparePersonByName());
```

# IComparer<T>

- Comparer til personer

```
class ComparePersonByName : IComparer<Person>
{
    public int Compare(Person x, Person y)
    {
        return x.Navn.CompareTo(y.Navn);
    }
}
```

```
List<Person> ps = new List<Person>();
ps.Add(new Person("Thomas", 10));
ps.Add(new Person("Anders", 15));

ps.Sort(new ComparePersonByName());
```

# Dependency Injection

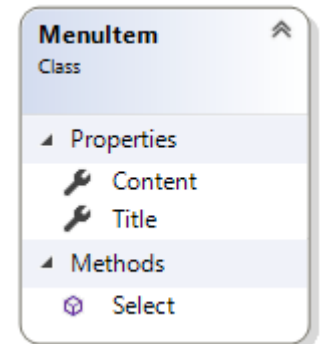
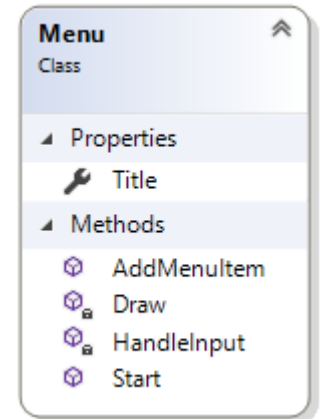
- I stedet for at hardcode hvilke objekter metodekald skal delegeres videre til, kan vi benytte **dependency Injection** til at så vi kan udskifte objekter dynamisk.
- Dependency Injection:
  - Dependency: Referencer til andre klasser kaldes også for *afhængigheder* (eng. "dependencies").
  - Injection: I stedet for at en klasse selv skaber nyt objekt, så leveres eller "indsprøjtes" objektet udefra.
- **Constructor injection**: Objekt leveres på instantierings-tidspunktet.
- **Property injection**: Objekt kan udskiftes løbende via setter.

# OO med interfaces

## Menu-eksempel

```
class Menu
{
    public string Title { get; }
    public void Start() { ... }
    public void AddMenuItem(MenuItem item) { ... }
    private void Draw() { ... }
    private void HandleInput() { ... }
}
```

```
class MenuItem
{
    public string Content { get; }
    public string Title { get; }
    public void Select() { }
}
```



Vi vil gerne have submenus!



# OO med interfaces

## Menu-eksempel

```
class Menu : MenuItem
```

```
{
```

```
    public string Title { get; } ← Kan fjernes
```

```
    public void Start() { ... }
```

```
    public void AddMenuItem(MenuItem item) { ... }
```

```
    private void Draw() { ... }
```

```
    private void HandleInput() { ... }
```

```
}
```

```
class MenuItem
```

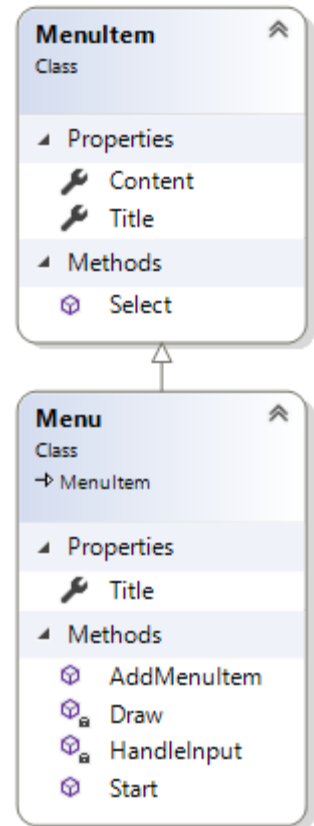
```
{
```

```
    public string Content { get; } ← Overflødigt fedt
```

```
    public string Title { get; }
```

```
    public void Select() { }
```

```
}
```



Vi vil gerne have submenus!

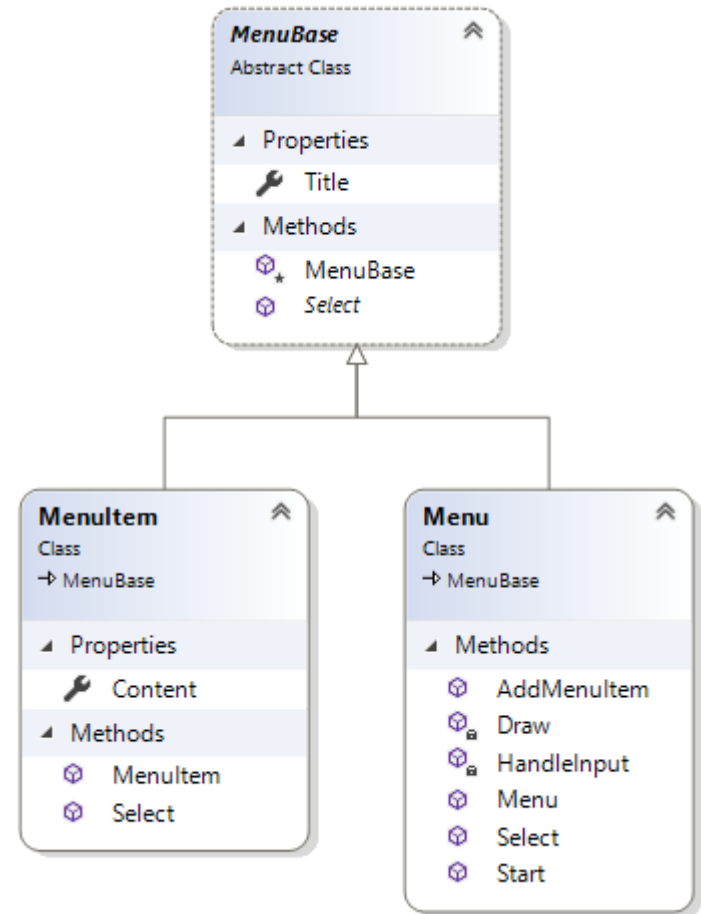
# OO med interfaces

## Menu-eksempel

```
abstract class MenuBase
{
    protected MenuBase(string title)
    {
        Title = title;
    }
    public abstract void Select();
    public string Title { get; }
}

class Menu : MenuBase
{
    public Menu(string title) : base(title) { }
    public override void Select() { }
    public void Start() { }
    public void AddMenuItem(MenuBase item) { }
    private void Draw() { }
    private void HandleInput() { }
}

class MenuItem : MenuBase
{
    public MenuItem(string title, string content) : base(title)
    {
        Content = content;
    }
    public string Content { get; }
    public override void Select() { }
}
```



Mindre fedt?

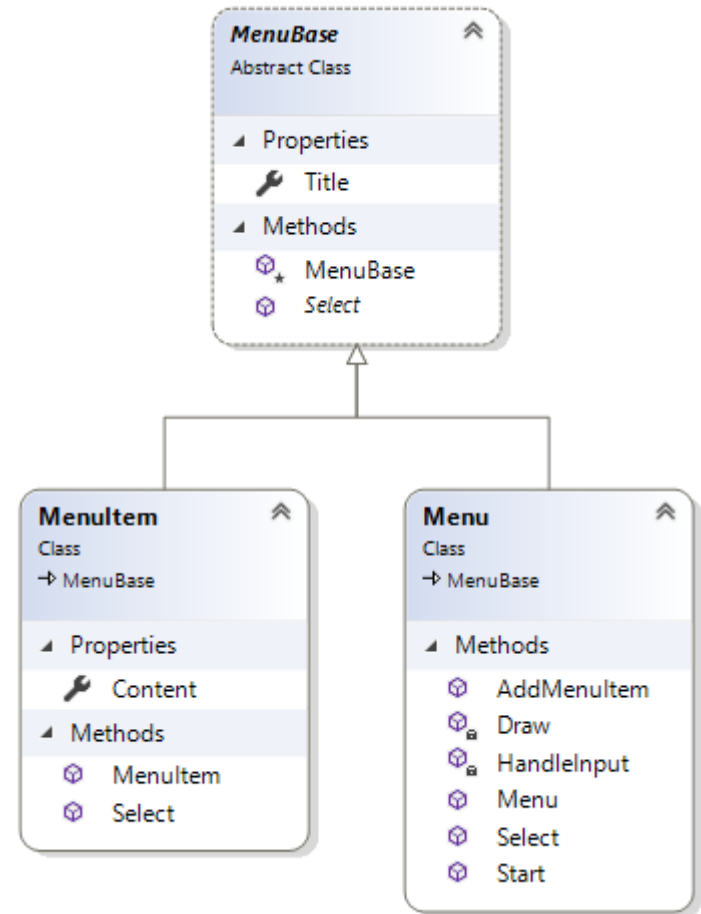
# OO med interfaces

## Menu-eksempel

```
abstract class MenuBase
{
    protected MenuBase(string title)
    {
        Title = title;
    }
    public abstract void Select();
    public string Title { get; }
}

class Menu : MenuBase
{
    public Menu(string title) : base(title) { }
    public override void Select() { }
    public void Start() { }
    public void AddMenuItem(MenuBase item) { }
    private void Draw() { }
    private void HandleInput() { }
}

class MenuItem : MenuBase
{
    public MenuItem(string title, string content) : base(title)
    {
        Content = content;
    }
    public string Content { get; }
    public override void Select() { }
}
```



Kan jeg lave en winforms-menu?  
- uden at lave det hele om?

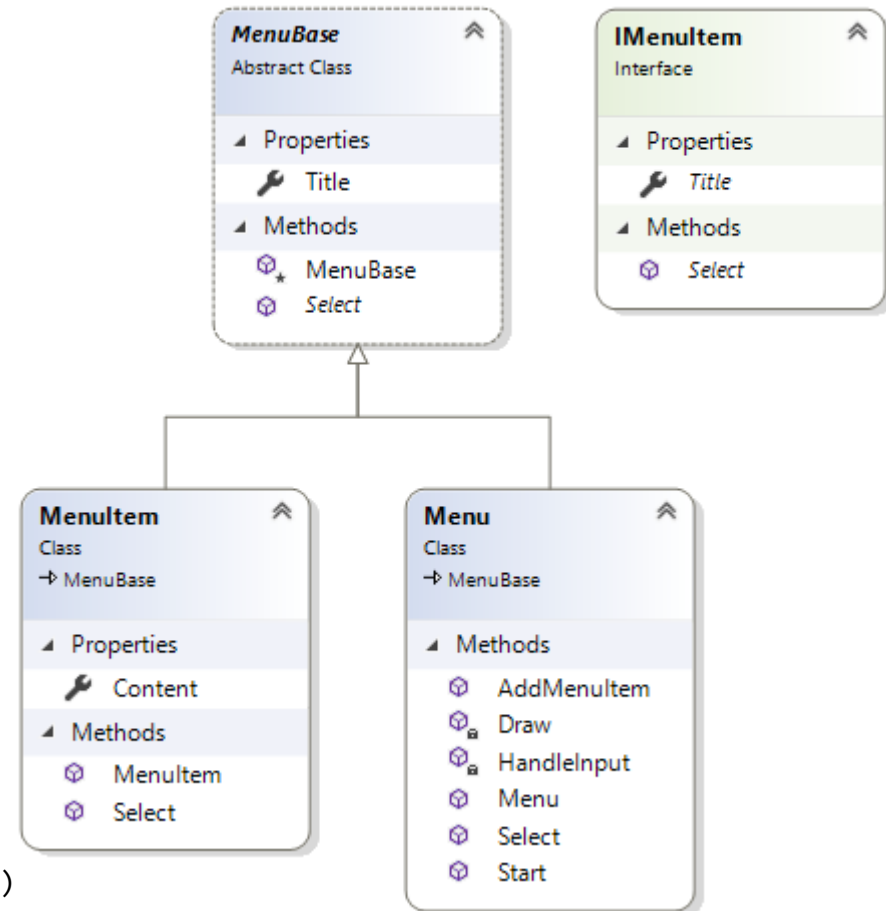
# OO med interfaces

## Menu-eksempel

```
abstract class MenuBase : IMenuItem
{
    protected MenuBase(string title)
    {
        Title = title;
    }
    public abstract void Select();
    public string Title { get; }
}

class Menu : MenuBase
{
    public Menu(string title) : base(title) { }
    public override void Select() { }
    public void Start() { }
    public void AddMenuItem(IMenuItem item) { }
    private void Draw() { }
    private void HandleInput() { }
}

class MenuItem : MenuBase
{
    public MenuItem(string title, string content) : base(title)
    {
        Content = content;
    }
    public string Content { get; }
    public override void Select() { }
}
```



Kan jeg lave en winforms-menu?  
- uden at lave det hele om?

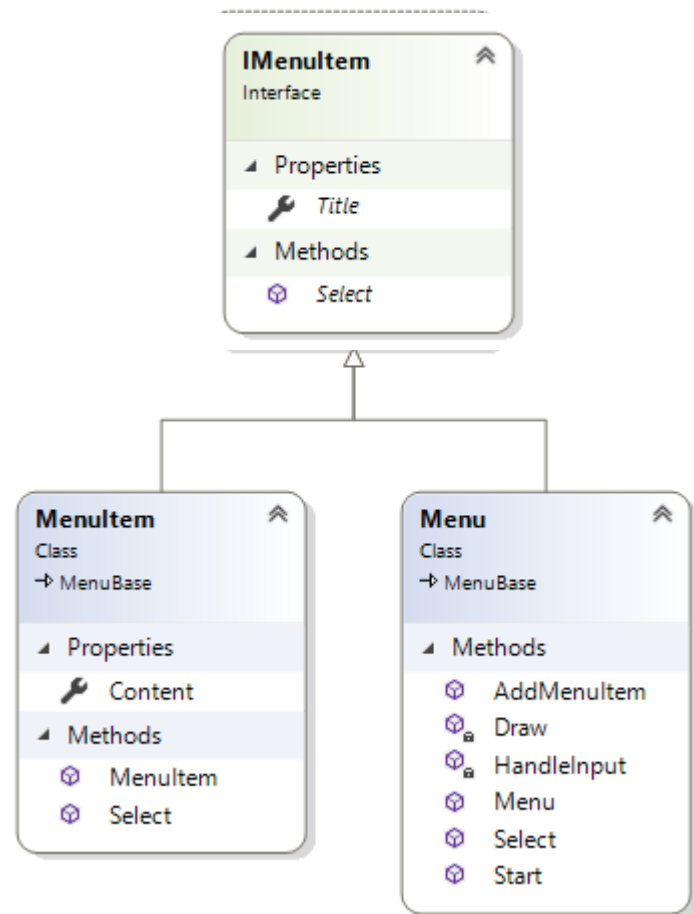
# OO med interfaces

## Menu-eksempel

```
class Menu : IMenuItem
{
    public Menu(string title) { }
    public void Title { get; }
    public void Select() { }
    public void Start() { }
    public void AddMenuItem(IMenuItem item) { }
    private void Draw() { }
    private void HandleInput() { }
}
```

```
class MenuItem : IMenuItem
{
    public string Content { get; }
    public override void Select() { }
}
```

```
class FormMenu : Form, IMenuItem
{
    public Menu(string title) { }
    public void Title { get; }
    public void Select() { }
    public void Start() { }
    public void AddMenuItem(IMenuItem item) { }
    private void Draw() { }
    private void HandleInput() { }
}
```



Kan jeg lave en winforms-menu?  
- uden at lave det hele om?

```

class Menu : IMenuItem
{
    public IMenuUserInterface Ui { get; set; }
    public Menu(IMenuUserInterface userInterface)
    {
        this.Ui = userInterface;
    }
    public void Select()
    {
        //...
        Ui.Display(this);
    }
    //...
}

class ConsoleUserInterface : IMenuUserInterface{ /*...*/ }
class WindowsInterface: Form, IMenuUserInterface { /*...*/ }
class WebInterface : Web, IMenuUserInterface { /*...*/ }

```

```

interface IMenuUserInterface
{
    void DisplayTitle(string title);
    void Display(IMenuItem item);
    void HighLight(IMenuItem item);
}

```

```

class Menu : IMenuItem
{
    public IMenuUserInterface Ui { get; set; }
    public Menu(IMenuUserInterface userInterface)
    {
        this.Ui = userInterface;
    }
    public void Select()
    {
        //...
        Ui.Display(this);
    }
    //...
}

```

```

interface IMenuUserInterface
{
    void DisplayTitle(string title);
    void Display(IMenuItem item);
    void HighLight(IMenuItem item);
}

```

Hvis vi altid bruger interfaces kan vi skifte alt ud!

```

class ConsoleUserInterface : IMenuUserInterface { /*...*/ }
class WindowsInterface: IMenuUserInterface { /*...*/ }
class WebInterface : IMenuUserInterface { /*...*/ }

```

```

// start med console-interface
Menu menu = new Menu(new ConsoleInterface());
// ...
// Skift mening senere:
menu.UserInterface = new WebInterface();
// Jeg vil hellere have vinduer
menu.UserInterface = new WindowsInterface();

```

Constructor-injection


Property-injection

# Interface nedarvning

Interfaces can arve fra hinanden.

```
interface I1 { void Foo(); }
interface I2 { void Bar(); void Baz(); }
interface I3 : I1, I2 { } // indeholder Foo, Bar og Baz

class A : I3
{
    public void Foo() { }           //implicit sealed
    public virtual void Bar() { }   //åben for redefinition
    public void Baz() { }
}
```





# Ændringer i interfaces

- Hvis nye medlemmer tilføjes til et eksisterende interface har det konsekvenser for implementerende klasser
- Bedre at lave nyt interface der arver fra det gamle
  - Bagudkompatibilitet med libraries etc.
  - En ændring kræver implementation i alle implementerende klasser

# Opsummering

- Interfaces:
  - Polymorfi i multiple hierarkier.
  - Kodegenbrug via komposition/aggregering
  - Abstrakte klasser vs. interfaces