TEKNOLOGISK
INSTITUT

Module 6

Introducing Object-oriented Programming

TEKNOLOGISK
INSTITUT

The Concept of clases and objects

- Classes are “blueprints” for objects
  - An object is an instance of a class
- One template….
  - Many objects
    - Every objects contains the same members but different data
- Members can be
  - Fields (data)
  - Properties (wraps fields)
  - Methods (behavier)
  - Events
- Members visibility
  - Public
  - Private
  - Protected
  - Internal

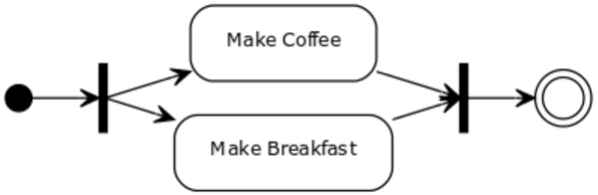
TEKNOLOGISK
INSTITUT

The Three Pillars of OOP

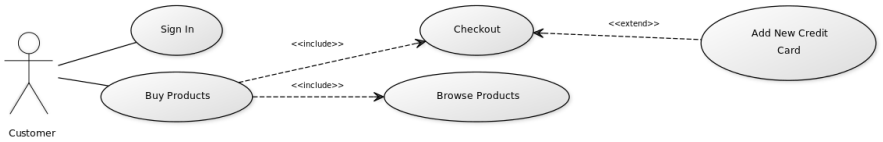
- Encapsulation
  - The ability to keep state and operations in an object accessible or modifiable only through the objects' interface
- Inheritance
  - The ability to create a family of objects with the possibility of sharing state and operations
- Polymorphism
  - The ability of a single object to take many forms making it possible to treat a collection of derived types as a group with the same interface

Architecture

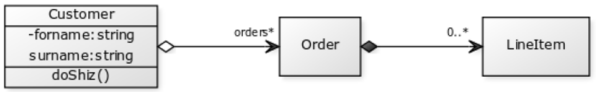
- Class design
  - Attempt to realistically reflect (part of) the rea
  - Use abstraction
  - Different types of objects
    - Physical objects
    - Workflows
    - Calculations
  - Objects containing other objects
- UML
  - Some use UML
- Design Patterns
  - Gang of Four
    - http://www.dofactory.com/net/design-patterns
  - Fowler



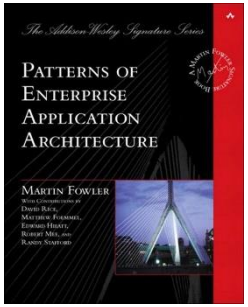
Activity diagram (yuml.me)

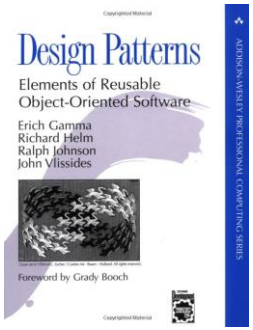


Use case diagram (yuml.me)



Class diagram (yuml.me)





TEKNOLOGISK
INSTITUT

Defining Classes

- Classes are defined using the class keyword

```
class Person
{
    // Members
}
```

- Naming classes

- Access modifiers
  - internal (default – only visible in same assembly)
  - public
  - private (nested only)
  - protected (nested only – only visible from derived classes)

```
public class Person
{
    // Members
}
```

Allocating objects

- Objects are instantiated by the new keyword
  - Reference variable on the stack
  - Objects on the heap

```
Person p1;
p1 = new Person();

Person p2 = new Person();
```

- Object are reference types!!!

```
Person p1;
p1 = new Person();

Person p2 = new Person();
p2 = null;

p1 = p2;
```

Fields

TEKNOLOGISK INSTITUT

- Fields are the only members in a class that represents state
- They make objects differ from other objects
- Represent the objects data

```
public class Person
{
    string name;
    int age;
    Gender gender;
}
```

- Access modifiers
  - **internal** (only visible in same assembly)
  - **public**
  - **private** (default)
  - **protected** (only visible from derived classes)

```
public class Person
{
    // data (fields)
    public string name;
    private int age;
    private Gender gender;
}
```

```
Person p1 = new Person();
p1.name = "Mathias";
p1.age = 11;
```

'Person.age' is inaccessible due to its protection level

TEKNOLOGISK INSTITUT

Default Constructor

- Every class has a *default constructor* method supplied out-of-the-box
  - Takes no arguments and has no return type
  - Sets all field data to a default value
- The constructor is invoked when an object is allocated with new

- The default constructor can be redefined

- Use ctor as snippet

- Use the this-keyword
  - reference to the current object
  - resolve naming conflicts
  - useful with IntelliSense

```
public class Person
{
    public string name;
    private int age;
    private Gender gender;

    public Person()
    {
        this.name = "";
        this.age = 0;
        this.gender = Gender.Male;
    }
}
```

TEKNOLOGISK INSTITUT

Custom Constructors

- Any set of overloaded custom constructors can be defined
- When you define a custom constructor, the compiler silently removes the built-in default constructor!
  - You can add the default constructor again

```
public class Person
{
    fields

    public Person()...

    public Person(string name, int age, Gender gender)...

    public Person(System.Random ran)...
}
```

```
public class Person
{
    public string name;
    private int age;
    private Gender gender;

    public Person()
    {
        this.name = "";
        this.age = 0;
        this.gender = Gender.Male;
    }

    public Person(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
```

TEKNOLOGISK INSTITUT

Chaining Constructors

- Constructors can be chained using `this`
- In this way the core construction code can be kept non-duplicated

```
public class Person
{
    fields

    public Person() : this("", 0, Gender.Male)
    {

    }

    public Person(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }


}
```

TEKNOLOGISK
INSTITUT

Partial Classes

- The implementation of a class can be divided into multiple `.cs`-files

```
public partial class Person {
    public string name = "";
}

public partial class Person
{
    public int age;
}
```

Simple data encapsulation

- Use access modifiers, constructors and/or the readonly-keyword to protect data

```
Person p = new Person("Mikkel", 14, Gender.Male);
```

```
public class Person {

    // not protected
    public string name;
    // readonly
    public readonly int age;
    // protected
    private Gender gender;

    // custom constructor (and no default constructor)
    public Person(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

}
```

TEKNOLOGISK INSTITUT

Simple data encapsulation

- Use private fields and public methods to control access to data
  - read only? write only?

```
public class Person
{
    private string name;
    public string getName()
    {
        return this.name.ToUpper();
    }
    public void setName(string name)
    {
        if (name == null)
            name = "";
        this.name = name;
        //this.name = name ?? "";
    }
}
```

```
Person p = new Person();
p.setName("Mathias");
string n = p.getName();
Console.WriteLine(n);   // MATHIAS
```

```
Person p = new Person();
p.setName(null);
string n = p.getName(); // ok - is ""
```

Structs vs. classes

TEKNOLOGISK INSTITUT

- Structs are “blueprints” for values
  - Cannot support inheritance
  - Are value types
  - Are passed by value (like integers)
  - Cannot have a null reference (unless Nullable is used)
- Classes are “blueprints” for objects
  - Can support inheritance
  - Are reference (pointer) types
  - The reference can be null
- Both Classes and Structs:
  - Are compound data types typically used to contain a few variables that have some logical relationship
  - Can contain methods and events
  - Can support interfaces