Notes til eksamen i DAT154

By Sondre Lindaas Gjesdal

Pensumliste DAT154 Vår 2020

C# 6 for programmers / Paul Deitel, Harvey Deitel.

Hele boken er i utgangspunktet pensum, MEN:

- Mange konsepter skal være kjent fra tidligere programmeringskurs, slik som klasser, objekter, kontrollstrukturer, metoder, arv, unntakshåndtering. Kapitlene som omhandler dette kan leses vært verfladisk eller hoppes over om dere allerede føler dere behersker dette. Det vil ikke være naturlig å ha utdypende spørsmål om dette på eksamen, MEN dere kan bli bedt om å forlate kode eller krive kode som inneholder disse tingene, så det er viktig dere har forståelse av det.
- Boken bruker en del sider på å utdype spesifikke kontroller i grafiske elementer. Det er ikke spesielt relevant å pugge disse, en oversikt over den overordnende virkemåten til slike kontroller generelt er nok.
- Hold fokus på det vi har snakket om i timene

Windowsprogrammering med Windows SDK/C++

- SDK-delen av boken er pensum (til og med side 72)
- Hold fokus på det vi har snakket om i timene

Microsoft Application Architecture Guide 2nd Edition

- Del 1 (Kapittel 1 t.o.m. 4) er pensum

Forlesningsnotater

Forelesningsnotater/slides og tilhørende kodeksempler som lagt ut på Canvas er pensum

Laboppgaver

Laboppgavene er pensum. Det er her viktig at dere skjønner hva dere har gjort og hvorfor/hvordan det virker. Det forutsettes at dere har implementert alt som oppgavebeskrivelsen ba om.

Stikkord

(Listen inneholder sentrale elementer, men det kan selvsagt dukke opp temaer fra pensum som ikke står på denne kortlisten under eksamen)

C++

What is C

C is a functional programming language form the 70s. Used to make complex systems like drivers and OS

What is C++

- C++ is a high-level programming language developed by Bjarne Stroustrup at Bell Labs. Based on C
- C++ adds OO features to it's predecessor, C. C from 1970s, and C++ from 1980s.

Where is C/C++ used

- OS like Linux, Windows etc
- Drivers for different hardware units
- Advanced logic
- A lot of programs written between 1970 and now.

Differences from Java

1. C++ makes machine code directly
2. Has Pointers
3. Handles memory manually - No garbage collection

Similarities

Object oriented

C++

Typical C/C++ Development Environment usually goes through five phases:

1. Edit
2. preprocess
3. compile
4. link
5. execute.

- Edit: Make a text file
- Preprocess: 1 part compiling - include header files and resolve constant symbols
- Compile: After symbols and include files have been resolved and included - transfers text to machine code
- Link: Bind your machine code modules together with libraries and make a complete program
- Execute: Run the program

Hello World

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World!\n";
    cin.get();
}
```

Header files in C++

The C/C++ Standard Library gives a variety of functions, which signature/prototype is in a header files.

In C normally all the header files begin with the .h extension, but in C++ not always like `<iostream>` A header file in C/C++ contains:

- Function definitions
- Data type definitions
- Macros

Header files offer of a preprocessor directive called `#include`. These preprocessor directives are responsible for instructing the C/C++ compiler that these files need to be processed before compilation.

C++ program normally contains the header file `<istream>` which has input and output functions/objects

- cin for input
- cout for output

The Preprocessor and `#include`

Before the source code is compiled, it gets automatically processed due to the presence of preprocessor directives. All preprocessor instructions typically start with a # sign. The most common is `#include` and `#define`.

File, compiling and linking

In C++ the code files are first made into pure machine code by the COMPILER(cl). These are called object files.

Then the linker binds and links the object files together and makes a real program

Namespace C++

Namespaces allow us to group named entities that otherwise have global scope into narrower scopes, giving them namespace scope.

- Namespace is a feature added in C++ and is not present in C
- A namespace is a declaretive region that provides a scope to the identifiers (names of the types, function, variables etc.) inside it.
- Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

Klasser i C++

Nothing special to note compared to C# and Java. Maybe that one specifies what is public and private with the `private:` and `public:` modifiers(?).

Konstuktører/Destruktører

Constructors are as normal constructors. Can also be defined like this

```
class A
{
  public:
  int x;
  //constructor
  A(); // constructor declared
};

// Constructor definition
A::A()
{
  i = 1;
}
```

Destructor

```
class A
{
  public:
  // defining destructor for class
  ~A()
  {
    // statement
  }
}
```

Example to see how the Constructor and Destructor are calles

```
class A
{
  // Constructor
  A()
  {
    cout << "Constructor called";
  }

  // Destructor

  ~A()
  {
    cout << "destructor called";
  }
};

int main()
{
  A obj1; // constructor called
  int x = 1;
```

```
  if(x)
  {
    A obj2; // constructor called
  }   // destructor called for obj2
}; // constructor called for obj1
```

Minnehåndtering

It's important to delete objects after use, especially in graphics, to prevent memory leaks.

Pekere/addresser

A pointer in C is simply a variable that stores an adress. All tables in C are simply(constant) pointers to a place in memory.

It takes some time to get used to pointers, they are the main difference between C/C++ and Java/C#.

"WIKI: A **Pointer** is a variable that holds a memory address where a value lives. A pointer is declared using the * operator before an identifier. As **C++** is a statically typed language, the type is required to declare a pointer".

A C/C++ pointer normally has a TYPE which indicates which TYPE of variable it points to. Very important to understand this

`char *pch; double *pd; int *pi;`

- Is used to declare a pointer, ex: `int *p;`
- & the address operator is used to extract the address of a variable - typically to initiate a pointer
    - `int x = 10;`
    - `int *p = &x;`

`*` is used to dereference(get the contents) of WHAT a pointer points to. ex: `cout << *p;`

```
int main()
{
  int i = 3;
  double x = 3.14;

  int* pi = &i;
  double* px = &x;

  cout << "i = " << *pi << "x = " << *px;
}
```

Another example:

```
int x = 10, y = 30;

int* p = &x; //10
```

```
cout << "x=" << x << endl;

*p = 20;

cout << "x=" << x << endl; //x= 20

cout << "*p= "<< *p << endl; //*p= 20

p = &y;

cout <<"*p= " << *p << endl;// *p=30
```

- Struktur

SDK

Struktur

- Main method(WinMain)
- Code to register a windows class
- Code to create a main window of that class
- A message that loops **IMPORTANT**
- A window procedure attached to the main window. **IMPORTANT**, Where all the `WM_`commands is located
- An SDK project contains several important files/filetypes:
    - *.h: Regular C++ header files
    - *.rc: Visual Studio resource files. Contains definitions for graphical resources, like dialog boxes and menus.
    - Other resources such as bitmaps.
    - Resource.h: Defines constants for all resources used in the project
    - *.cpp: Regular C++ source files
- Vindusmeldinger
    - Meldingsløkke
    - Meldingshåndtering
- Resurser
    - Editing resource files
        - Can be edited by hand(normally not)
        - Use the Visual Studio Resource Editor
- Dialoger

    - A dialog is used for input of data to the program or for just giving information to the user.
        1. Create the dialog in the resource editor
        2. Write a window procedure for the dialog
        3. Show the dialog:

```
LRESULT DlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam){}
DialogBox(hInst, MAKEINTRESOURCE (IDD_DIALOG1), hWnd, DlgProc);
```

- Dialog Windows Messages
    - `WM_INITDIALOG`
        - Sent when the dialog is created. All dialog initialization code should be called from here(ex: Setting default values in controls)
    - `WM_COMMAND`
        - Sent whenever an event occurs on controls in the dialog, like clicking a button, selecting a menu item, etc.
        - Get the triggering control with `LOWORD(wParam)`

Grafikk

Device Contexts

- A Device Context is a GDI object representing a drawing surface
- It can represent a physical or virtual interface
- Get the device contet for your applications drawing area by calling `BeginPaint` and pass the window handle(received with the `WM_PAINT` message)

GDI-Objekter

The GDI(Graphics Device Interface) contains the API for drawing text and graphics on output devices

Works with the device driver to send the output to a device

- Virtuelle device contexts/kopiering
- Animasjon

Funksjonspekere

- Sometimes we need to pass a function as an argument
- This is used quite heavily in Windows SDK
- To do this, we create a type that can hold a pointer to a function
- Declaring the type:

```
typedef int (*FPTYPE) (int a, int b);
```

- Creating functions that match the created type:

```
int sum(int a, int b) {return a + b;}
int prod(int a, int b) {return a * b;}
```

Example

```
FPTYPE f = 0;
f = sum;
int s = f(3,4);
f = prod;
int p = f(3,4);
```

Function ponters in SDK

- The usage is usually transparent since the type is already declared by the SDK framework
- The most common function pointer is the window procedure

```
typedef LRESULT(CALLBACK* WNDPROC)(HWND, UINT, WPARAM, LPARAM);
```

- Used when we register the window class:

```
WNDCLASSEX wcex;
wcex.IpfnWndProc = (WNDPROC)WndProc;
```

Function pointers in C++

- Function pointers is also used in "regular" C++:
- Example: qsort function in `<stdlib.h>`

```
void qsort (void * base, size_t num, size_t size, int ( * pfn ) ( const void *,
const void * ) );
```

- This function can sort any C++ array
- `pfn` is a function pointer

Example

```
int tab[] = {3,4,5,1,7,2,1,4,2,12};
int cmp(const void * p1, const void * p2)
{
  int t1 = *((int*)p1); // Must cast from void* to int*
  int t2 = *((int*)p2); // Must cast from void* to int*
```

```
  return t1 - t2;
}

WCHAR s[10];
qsort(tab, 10, sizeof(int), Cmp);
for(int i = 0; i<10; i++)
{
  _itow_s(tab[i], s, 10);
  TextOut(hdc, i*50, 100, s, wcslen(s));
}
```

Tråder

.Net

- Runtime platform and development frameworkd for cross-platform applications.
- This means you may mix C#, C++, F#, VB.NET or whatever language
- It is composed of several components

Struktur

CLR

- Common Language Runtime
  - The CLR is a kernel that reside on top of the OS, and can execute code in a language called Common Intermediate Language(CIL)
  - All .NET applications are compiled to CIL instead of native machine code
    - Similar concept as Java bytecode

Managed Code

- Managed Code is code that targets the CLR
- Code that does not target the CLR(old DLL's, API calls, etc) is unmanaged code
- Managed code is handled by the CLR, that provides
  - Security checks/constraints
  - Memory Management/Garbage Collection
  - Access to managed data
- Typer

Assemblies/metadata

- A .NET assembly is an executable (.exe) or library (.netmodule) that requires the CLR to run
- Such a file contains code compiled MSIL in addition to code, it also contains metadata(ildasm.exe)
- Similar to a Java .jar file
- Contains metadata which describes the interface. These data may be read by a browser, for example the class browser in Visual Studio, or more important the compiler during the build process when multiple

assemblies are involved.
- Must be run under a CLR
- Is portable(To any system with a CLR)

Navnerom/referanser

- A class is identified by both its namespace and actual name
- When we use a class, we need to tell the compiler both the namespace and the name of the class

```
System.Console.Writeline("Hello World");
Library.Book book = new Library.Book("Title");
```

- For Commonly used namespaces, we can import the namespace by using the using directive

```
using System;
class ChelloWorld{
  public static void Main(){
    Console.WriteLine("Chello World");
  }
}
```

Some important namespaces

- **System** - Fundamental and base classes, interfaces and exceptions
- **System.Data** - Classes for accessing and managing data (from external sources)
- **System.Drawing** - Basic GDI+ support
- **System.Text** - String handling
- **System.Windows** - GUI components

Creating a namespace

- A namespace is created by wrapping our class(es) in a namespace block.

```
namespace NS
{
  public class X
  {
    public static String HW()
    {
      return "Hello World NS";
    }
  }
}
```

Nesting namespaces

- Namespaces can also be nested

```
namespace NS1
{
  namespace NS2
  {
    public class X
    {
      public static void F()
      {
        System.Console.WriteLine("F's in the chat for guy, bois");
      }

    }
  }
}
```

- The `using` keyword does not provide access to namespaces nested below the specified level

Correct:

```
NS1.NS2.X.F();

using NS1.NS2;
X.F();
```

Incorrect:

```
using NS1;
NS2.X.F();

using NS1;

X.F();
```

Aliases in namespaces

- One can use aliases to provide a simplified name for a class or namespace

```
using s = System; // namespace alias
using c = System.Console; // Class
```

References

- References points to external libraries.
- By default, not everything is referenced from a project
- Might need to set up a reference to access a given namespace
- References can be browsed in the object browser in VS

Reference types

- Reference types are stored on the heap
- A pointer to the memory location occupied by the type is stored on the stack
- Assigning a reference type to a new variable creates a new pointer on the stack pointing to the same object on the heap
- Implicit inheritance from System.Object

Value types

- A value type does not point to the heap, it keeps its value on the stack
- When assigned to a new variable, the content is copied
- Still behaves as an Object, and has methods
- Implicit inheritance from System.ValueType

FCL (Framwork Class Library)

- The FCL is a comprehensive collection of reusable types, including classes, interfaces and data types included in the .NET framework to provide access to system functionality.
- FCL acts as a standard library.
- The reusable types of FCL provide a simple interface to developers due to:
    - Their self-documenting nature
    - Lesser learning curve to understand the framework, which expedites and optimized the development process
    - Seamless integration of third-party components with classes in FCL

Biblioteker

- .Net Library assemblies are almost identical to executable assemblies
- They do not need a Main() method
- To create a library, compile with `/target:library`
- To reference a library in an executable, compile with `/r:libraryname.dll`

Grafiske grensesnitt

- Two Different GUi systems
    - Windows Forms
        - Based on GDI+("Classic" Windows)
        - All design is reresented in language code
    - Windows Presentation foundation'
        - Based on DirectX
        - All design is done in XAML, but can be modified by language code

Windows Forms

- Classic Windows GUI
- All windows and dialogs are defined using a Form
- Created using the "Windows Forms Application"
- Components found in the System.Windows.Forms namespace

Structure

- One file(`program.cs`) containing the main method for the program. Responsible for displaying the graphical form
- One or more forms
    - Design File(C#)
    - Code File(C#)
    - Resource File(RESX)
- Zero or more supporting code files

Form

- A form normally consists of 3 files
    - Design file: This file is maintained by VS and represents your doings in the WYSIWYG editor
    - Code file: This file contains all your code associated with the form
    - Resource file: This file contains resources, such as text strings and images
- Note that this is VS convention. You can easily make a form in a single file or spread it out over hundreds if you do it manually

Components

- Visual Studio provides a toolbox with various components that can be dragged into the designer:
    - Buttons
    - Text fields
    - Combo boxes
    - Etc...
- These can also easily be added and modified from code, each component is defined in a seperate class

WPF - Windows Presentation Foundation

- Windows Presentation Foundation is a new way of designing graphical systems, introduced in .NET 3.0
- Instead of designing the graphical elements using code, they are designed in an XAML(Extensible Application Markup Language) file, which uses XML syntax
- Focuses ona dynamic layout
- WPF is designed to remove the gap between despktop and web applications
    - Can run in browser
    - Can run as standalone
- Use DirectX to remove dependency on aging GDI subsystem
- WPF elements can be edited graphically in Visual Studio, or by editing the XAML file
- WPF elements can also be modified at runtime using code.

- The elements behave similar to regular Windows Forms controls
- GUI components can be skinned using styles
    - Somewhat similar in concept to HTML+CSS
- WPF applications does not have a (visible) Main() method
- Instead, an Application tag in a XAML file(usually App.xaml) defines the startup point
- The Main() method is created behind the scenes by the compiler, as it is required for the assembly to be executable
- Closely related to Silverlight
    - Rich web content
    - Windows Phone
    - Universal Apps
- Universal Apps

Delegater

- A delegate fulfills a similar role to the function pointer known from C++
- However, a delegate can store multiple functions, which will all be called when the delegate is called
- A delegate must be declared with a specific method signature, and can only accept methods conforming to this signature
- Declarin a delegate is a two-step process
    1. Declare a delegate type
    2. Create an instance of the type
- In this regard, a delegate is handled much like a .NET class(type)
- Delegates are declared using the `delegate` keyword

```
delegate returntype name(parameters);
```

- A delegate can be declared at the class level, or the member level
- Examples

```
delegate int calculateDelegate(int a, int b);
public delegate void startComponents();
```

- A delegate instance can be created at member level or local level
- The instance is created like any other variable, using the delegate name as the type
- Example:

```
private calculateDelegate calculate;
public startComponents start;
```

- For a method to be able to be assigned to the delegate, it must have the same signature as defined in the delegate
- Any method that has this signature can be assigned
- Multiple methods can be assigned to the delegate
- To assign a method, use the `+=` syntax on the delegate instance:
- `calculate += plus;`
- A method can also be removed using the `-=` syntax
- If the instance hasn't been instantiated, we can either use assignment (=) or the `new` keyword

```
calculate = plus;
calculate = new calculate(plus);
```

- When calling a delegate, it will call all methods, that has been assigned to it
- Make sure the delegate has at least one assigned method, or a `NullReferenceException` will be thrown
- Call the delegate as one would a method, supplying parameters, if any:

```
calculate(1,3);
start();
```

Delegate simple example

```
delegate void delegateTest();
class A
{
  public delegateTest t;
  public A()
  {
    t += b;
    t();
  }

  public void b()
  {
    System.Console.WriteLine("Delegate Called");
  }
}
```

Return values

- Calling a delegate will return the return value from the called method
- If the delegate has several methods attached, the return value will be from the last method in the list

- A delegate can also be iterated to call each method individually by calling each method reference returned by calling `GetInvocationList()` on the delegate instance

`Func<T,TResult>/Action<T>`

- Predefined generic delegates with up to 16 parameters are available in .NET
- These all have a signature like `Func<T1,T2,..Tn,TResult>` Where T1...Tn is the return type.
- We can instantiate these instead of declaring our own delegate
- The use of Delegates allow us to create a decoupled publisher-subscriber model
    - The service does not need to be aware or depend on the subscribers
    - The subscribers does not need to be dependent on the details of the service
    - One of the basics for creating modular applications

Hendelser (Events)

- An event is a notificaiton of a change in state -> "something" happened
- An object fires an event to notify subscribers about the change in state
- Other objects subscribe to the events of one object to be notified when such changes occur
- Subscribers act when they are notified, instead of being responsible for polling for changes
- Event driven programming has been a known paradigm for a while, but .NET is one of the first to formalize this into the language itself.
- Still, events has been around for a long time. The Windows Messages that drives the message loop in an SDK program is also events, although the handling is complicated
- Events and delegates are closely related. To be able to create an event, we must first create a delegate
    - Example:

```
public delegate void Del(Object o);
```

- We can then create an event using this delegate
    - Example:

```
public event Del Evt;
```

- Subscription to events work just like delegates, use `+=` and `-=`
- Only methods that fit the delegates signature can subscribe to the event
- Firing events is done the same way a delegate is called.
- However, due to the formalized event syntax, an event is normally fired with a single parameter, a reference to the object that fired the event:

```
event(this);
//Or with an additional EventArgs argument
event(this, EventArgs.Empty);
```

System Events

- While custom events can be created and handled, it is far more common to handle system events
    - Ex: Mouse events, keyboard events
- These events are sent to our program as a result of a Window Message
- When an event fires, ALL subscribed methods will be called
- This means that a single event may be handled by multiple methods
- Keep an eye out for conflicts
- One Method can also handle multiple events
- We use the sender object to determine where the event originated

Routed Event

- WPF introduced the concept of Routed Events
- A routed event "passes through" the hierarchy and can be picked up by any handler on the way
- Once a handler has handled an event, it can mark it handled, which will prevent further handling of that event
- Three types:
    - Direct events - Will not travel through the hierarchy
    - Bubbling events - Starts at the source of the event, and works it way outwards towards the root
    - Tunneling events - Starts at the root and travels inwards toward the source. They are prefixed by Preview to seperate them from other event types
- Håndtering
- Typer

Grafikk i .NET

- .NET supports two different GUI options for desktop applications
    - Windows Forms
        - Old system, closer to SDK
        - Preferred if you need custom drawing/animation
        - Uses GDI+
    - Windows Presentation Foundation(WPF)
        - New system
        - Preferred when using a control-based GUI
        - Much more dynamic than Windows Forms, and easier handling of multimedia
        - Uses DirectX

Tegneverktøy

- Almost all drawing tools require a color
- To create a color we need to use static methods from the Color class
- The Color class also has a wide range of static members containing predefined colors
- Use System.Colors to retrieve various system colors
- Pens are used for drawing lines, including outlines of hollow figures
- A Pen needs a color, and optionally, a width
- `Pen p = new Pen(Color.Red);`
- Brushes are used for drawing filled surfaces
- Just as for a pen, a brush needs a color
- There are several types of Brushes:
  - `SolidBrush`: Paints in solid Color
  - `HatchBrush`: Paints a predefined pattern
  - `TextureBrush`: Paints a texture, like an image
  - `LinearGradientBrush`: Paints two colors blended along a gradient
  - `PathGradientBrush`: Paints using a complex gradient of blended colors, based on a unique path defined by the developer
- Examples:

```
SolidBrush myBrush = new Solidbrush(Color.Red);

HatchBrush myBrush = new HatchBrush(HatchStyle.Plaid, Color.Red, Color.Blue);

TextureBrush myBrush = new TextureBrush(new
Bitmap(@"C:\Windows\Web\Wallpaper\Theme1\img3.jpg"));
```

Animasjon

- Just as with SDK, animation is handled by updating required values (like positions), then redraw the scene based on the new information
- Calling `Invalidate()` will force a redraw
- If animation is based upon a timer, you can use the `Timer` class

WPF Drawing

- WPF adds an abstraction level to drawing
- Instead of drawing directly on a drawing surface, you create objects, sets their properties, and WPF will do the low level drawing for you
- These objects can be created either in code or in XAML
- WPF provides the following shapes
  - Ellipse, Line, Path, Polygon, Polyline, and Rectangle
- These shapes allow for varous properties like
  - Line & Fill
  - Position
  - Size

- Animating shapes in WPF is as simple as updating their properties. WPF itself will take care of the rest
- If you wish to update the properties on a timer, use a `System.Windows.Threading.DispatchTimer` for this, as this timer properly supports the WPF application model

Data in WPF

We need entity classes

- Entity classes can be manually created as outlined previously using ADO.NET
- Or they can be automatically generated by VS

Connecting to the DB

- Create a new `DbContext` instance
- Use this to retrieve references to the three tables
    - Student
    - Course
    - Grade

Data binding

- Many WPF controls supports data binding
- This means that they are directly connected to a container class (which may be a database table with LINQ)
- Thus we do not have to write code to keep the controls updated

Create a grid to show data

- For this we will use a ListView control
- The ListView will be bound to the student table (`DbSet<student>`)
- We will then use a GridView as a child control to present the actual data

Bind the data to the control

- This is accomplished somply by setting the DataContext property on the ListView control

```
student.Load();
studentList.DataContext = student.Local;
```

Linq

- Language Integrated Query
- LINQ is a Query Language designed to query in-memory data, databases, XML and other data sources
- Similar to SQL
- Works with all data structures implementing the IEnumerable interface

IEnumerable

- A collection class implementing this interface can be iterated over with a foreach statement, as well as queried by LINQ
- Must implement the GetEnumerator method
- GetEnumerator uses the `yield` keyword to return elements one by one
- Normally done inside a (foreach) loop
- Looks like a return statement, but does not end the method on invocation
- To use LINQ, we must include `System.Linq` namespace
- This adds appropriate extension methods to all classes implementing the IEnumerable interface
- These extension methods provides functionality for running queries on the collection object
- Some important methods
    - Select
    - Where
    - OrderBy
    - GroupBy
    - Join
- The arguments to the LINQ methods are pointers to functions
- Use lambda expressions to create anonymous functions and classes for this purpose
- Databaser
- XML

Web

ASP.NET

- ASP.NET uses .NET technology to create web applications
- Can theoretically be written in any .NET language, but usually C# or VB.NET is used
- Uses HTML for presentation
- Web Forms
- MVC
- Web API

Web Forms

- Uses similar layout to WPF applications:
    - One presentation file(HTML 5) [*.aspx]
    - One designer file (C#, VB.NET, ...)

    - One code-behind file (C#, VB.NET, ...)
- Uses familiar .NET controls

ASP Tags

- The HTML file uses special `asp` tags to represent .NET controls and other special elements
- In code, these behaves just like ordinary .NET controls
- These are processed server-side before the HTML code is sent to the client

Testing and debugging

- Testing and debugging is done using the built-in webserver in VS
- You can use breakpoints and watches just as with a local application
- Note that the browser might time out during a debug session. This doesn't affect debugging, but might affect the result displayed in the browser
- Can debug against any browser, but IE is a bit tighter integrated

Master Pages

- ASP.NET web applications lets you use master pagers to easily give the same look and feel for your entire application.
- Content from child pages is merged with the master page before sending it to the browser.
- Each child page controls which master page to use, allowing you to have multiple such pages
- NOT based on frames/iframes

Database Access

- ASP.NET supports the use of both "classic" database connections, and LINQ/EF connections
- A data component can be bound to a LINQ data source, much like in WPF

Accessing Form Data

- Form data is accessed the same way as in a regular windows forms or WPF application, by accessing the properties of the controls.
- Check the `IsPostBack` property to determine if this is an initial page load, or if the user have submitted form data

Validation

- Data input can be validated using validation controls
- These work both client and server-side
    - `RequiredFieldValidator` check that a required field has been assigned a value
    - `RegularExpressionValidator` checks that the value of a field is acceptable
    - `RangeValidator` ensures that the submitted value is within a given range

AJAX

- ASP.NET has built-in support for asynchronous requests

- This can be used to do partial updates of a web page without reloading the entire page
- Need a `ScriptManager` and an `UpdatePanel` component

Web Services

- ASP.NET supports both SOAP and REST based web services
  - REST supports both XML and JSON
- Uses Windows Communication Foundation(WCF) or Web API

MVC

- Based on ASP.NET
- Uses the familiar Model-View-Controller pattern
  - Separates dataq from logic from presentation
- Uses Razor for createing dynamic html pages
  - Code blocks and inline expressions are indicated by @
  - Code blocks can be C# or VB
- Uses routing to determine which controller and method will handle a request
  - Decoupled, easy to add additional functionality
- Very friendly for code-first deployments

WebAPI

- ADO.NET API for making restful webservices
  - REST (Representational State Transfer) is an Architectural Style
    - Based on stateless operations
    - Decoupled architecture
    - Not limited to a fixed data format, but JSON is often used
    - URI's contain resource identification
    - BASED on HTTP, uses HTTP verbs
- Built on top of the Microsoft MVC web platform
- Serves data in the format requested by the client(XML or JSON, hor HTML use MVC)
  - Marshalled automatically

Universal Windows Platform (UWP)

- Platform built on top of the Windows Runtime (WinRT)
- Designed to run on all modern Windows devices:
  - Windows 10
  - Xbox
  - Surface Hub
  - Hololens
  - Windows Phone/Mobile
- Can be developed in C#/VB/C++/Javascript
- Does not have to be .NET
- Does not support the full .NET ecosystem
- Not intended for non-Windows OS
  - Use Xamarin

- Distributed via Microsoft Store
    - Also sideload support
- UI options similar to WPF
- Secure Execution

Databaser

ADO.NET

- ADO.NET is a .NET library that provides access to datasources such as SQL databases and XML
- Works with LINQ to provide easy "Object-oriented" data access -> DLINQ
- ADO.NET is found in the `System.Data` namespace
- This namespace contains a series of nested namespace that might be useful depending on the features you use:
    - Common, Odbc, OleDB, ProviderBase, Sql, SqlClient, SqlTypes
- To connect to a database, you will need to specify a connection string. This is a semi-colon delimited string with the following fields
    - Data Source: A reference to database server running on the local computer, or the address of a remote computer
    - Initial Catalog: The name of the database to access
    - Integrated Security Set to True to use the windows login credentials of the user running application
    - User ID: The database login(username)
    - Password: The password belonging to the login
- The connection object manages your connection to the database
- Call the `Open()` method to initialize the connection to the databe
- Call the `Close()` method when you are done with the connection
- Database connections are scarce resources. Remember to not keep it open longer than required(hint: Wrap in "using")
- SQL statements are executed through a SqlCommand object contains two very important methods:
    - `ExecuteReader()`: Executes a Select Sql query that returns a resultset
    - `ExecuteNonQuery()`: Executes a query that does not return a resultset(UPDATE, DELETE, INSERT)
- Data are returned from the `ExecuteReader()` call in the form of a SqlDataReader object
- This object has `Read()` method for advancing to the next row in the result set, and various methods(including an indexer) to retrieve the values

ADO.NET and Entity Framework

- When used with EF, we can treat tables as collections and rows as .NET types
- This gives us a more natural object oriented way to manage our data
- Needs a reference to EntityFramework

DbContext

- When using DLINQ we use a DbContext object instead of a SqlConnection object
- Do not attempt to Open/Close this object. This is handled automatically by EF
- The connection string is identical to a SqlConnection object

Entity (Model) Classes

- An entity class is a .net type that defines a database table in the form of a .net class
- This class contains several attributes that identifies it as an entity class
    - Table
    - Column

The `DbSet<T>` class

- The `DbSet<T>` class is a collection of entity classes, and represents a physical table or view in your database
- A table can be loaded from the database simply by calling the set method on the DbContext

Using LINQ

- You can use regular LINQ statements on your `DbSet<T>` objects just as with ordinary collection classes
- Both the method invocation and SQL syntax forms are allowed
- Linq

Parallel LINQ (PLINQ)

- Many LINQ operaitons benefits from being run in parallel
- This allows the computer to put multiple threads at work processing the data
- Just start the LINQ chain with a call to `AsParallel()`
- Not all data/operations can be optimized for parallel processing
- Beware thread safety

Parallel execution

- Parallel for loops
    - Allows us to do multiple loop iterations in parallel instead of one by one sequentially
    - `Parallel.For(int, int, Action<int>)`
    - `Parallel.ForEach(IEnumerable<t>, Action<t>)`
- Parallel delegates:
    - `Parallel.Invoke(Action[])`
    - Note: Operates on an array of delegate, does not parallelize each individual delegate

Asynchronous Processing

- Running a method asynchronously means it will run on it's own thread
- Used to keep the main application responsive while heavy work is processed in the background
    - Or if we need to wait for the results(web requests, etc)
- Keyword `async` marks a method as capable of running asynchronously
- Keyword `await` teels the program to wait for the result of an async request
- Set up a `Task` to run a method asynchronously

Lambda

- While normal expressions return a value, a lambda expression will return a method.
- lambda expressuibs can be used to define on-the-fly functionality that can be injected into methods that accepth them
    - Ex: Sorting algorithms, program control
- This is particularly useful when designing anonymous methods to use with delegates/events
- A lambda expression has these elements
    - A list of parameters
    - The `=>` operator
    - The method body(or a simple expression)
- A lambda expression can refer to a variable that is no longer in scope when the function is called
    - The expression will still have access to this variable
    - The variable will only be garbage collected when the delegate holding the lambda is garbage collected
- Lambda expressions are great for controlling program flow
- They can be created in one part of the application and passed to another
- This allows for mroe dynamic control of program flow at runtime

```
(param) => {body;}
```

Some sample lambda expressions

```
x => x*x;
x => {return x*x;}
() => timer.start();
(x,y) => {x += y; return x/y;}
(int x) => x/2;

// Bigger example

class A
{
  public delegate int delg(int x);
  public delg d;

  public A()
  {
    d += (x) => {return x * x;};
    Console.WriteLine(d(4));
  }
}
```

Arkitektur

Bruk av multiple programmeringsspråk

REP: Common intermediate Language

- Visual Studio does not let us mix different languages in the same VS project
- But it is possible to create different projects inside a solution, where each project uses its own language
- It is also possible to manually assemble the .NET modules into .NET assemblies
- All .NET code no matter the language are compiled to CIL code
- Possible to use several different .NET languages in the same project
- One language can use or subclass classes written in another language
- One language can throw an exception that is caught in another language(module)

Subclasses

- As seen subclasses across languages
- This is written the exact same way as you would extend a native class
- Just remember to import the required **namespaces**(there might be implicit namespaces in the foreign language)

Compiling

- Normally, when code is compiled with a .NET compiler, we get a standalone assembly, which contains
    - Metadata
    - Executable CIL code
- Linking, as known from C++ is normally no longer done manually, but done automatically at runtime

Programvarearkitektur

"Software application architecture is the process of efining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security and manageability" - Microsoft Applicaiton Architecture Guide, 2nd Edition

What is Software Architecture

- Create a design that ensures
    - Usability
        - Business requirements
        - Performance
    - Stability
    - Maintainability
        - Bug fixing
        - Changing requirements
    - Security
- Three participants
    - User
    - Business
    - System
- Conflicts
    - Trade-offs
- Must identify and consider key scenarios and quality attributes

Goal

- Create an architecture that:
    - Fulfills user requirements
    - Reduces business risk
    - Flexible, handles changes in technology and requirements

Considerations

- How are the different parts of the application used?
- How does these parts interact with each other
- What about design requirements

Key Architecture Principles

- Build to change instead of building to las
- Model to analyze and reduce risk
- Use models and visualizations as a communication and collaboration tool
- Identify key engineering decisions
- Seperations of Concerns
- Single Responsibility Principle
- Principle of Least Knowledge
- Don't Repeat Yourself (DRY)
- Minimize Upfront Design

Design Practices

- Keep design patterns consistent within each layer/module
- Do not duplicate functionality within an application
- Prefer composition to inheritance
- Establish a coding style and naming convention
- Maintain system quality using automated QA techniques
- Consider the operation of your application

Application Layers

- Seperate the areas of concern
- Be explicit about layer communication
- Use abstraction between layers
- Do not mix different types of components
- Keep the data format consistent

Components, Modules and Functions

- A component should rely on internal details of others
- Do not overload the functionality
- Understand component communication
- Keep crosscutting code abstracted from the business logic
- Define a clear contract for components

Key Design Considerations

- Application type
  - Mobile apps, desktop clients, web apps don't fit the same patterns
- Deployment Strategy
  - Environment concern, physical seperation, communication
- Technologies
  - May impact possible design choices
- Quality Attributes
  - Security, performance, usability, testability, maintainability, scalability, ... => Seperate from functionality
- Crosscutting concerns
  - Centralized solutions => Logging, security, communication

Patterns

- A pattern is a predefined solution to a common proble,
- Patterns are found in all areas of creativity - Architecture, electronics, crafts, software design, software architecture...
- Documented and published

Architectural Patterns

- Also referred to as Architectural Styles
- Provides and abstract framework
- Improves partitioning
- Promotes design reuse
- Provides a solution to a frequently recurring problem

Client/Server

- Distributed system
  - Both the client and the server holds part of the application
- Clients are typically concerned about UI, while the server takes care of the business logic
- Clients make requests, server fulfills them
- Multiple clients may use one server

Scenarios

- Web Applications
- Mail/messaging systems
- Collaboration software
- Banking systems
- Remote database access
  - Always?

Advantages

- Security
- Centralized data access
- Ease of maintenance

Disadvantages

- Scalability
- Stability

Layered

- Describes a division into layers (and optionally sub-layers)
- Each layer represents a stand-alone unit
- Groups related functionality
- Communication between layers are structured
- Abstractions are used to reduce coupling and promote reuse

Scenarios

- Most application beyond the scope of simple utilities

Advantages

- Clear seperation of concerns
- Reusability
- Replacability
- Opens up deployment options

Message bus

- A message bus architecture facilitates communications between components/services
- Communication can be asynchronous
- 1:1 or 1:n

Scenarios

- When components need to be completely unaware of each others
- When components won't necessarily be online
- When multiple components can services the same request

Advantages

- Asynchronous
- Scalable
- Low complexity
- Loose coupling
- Easy to replace components

Disadvantages

- Structural complexity
    - Everything must learn how to communication with the bus
    - The overall architecture becomes more complicated
- Guaranteed delivery?
- Overhead

Service-oriented architecture(SOA)

- Functionality is provided as a set of independent autonomous services
- Service discovery
    - Redundancy
    - Replacability
- Message-based interaction

Scenario

- Large enterprise ecosystem where most services have multiple consumers

Advantages

- Abstraction
- Discoverability
- Interoperability
- Scalability
- Replacability
- Reusability

Disadvantages

- Complex
- Overhead

Combining styles

- Most styles combine well
    - Message bus handling communictation between layers
    - SOA architecture where services are using a layered style
    - SOA can implement a message bus
- For most larger deployments, this is more a rule than option, as one style doesn't cover the complete picture alone

Key considerations

- Application type
    - Mobile apps, desktop clients, web apps doesn't fit the same patterns
- Deployment Strategy
    - Environmental concerns, physical seperation, communication

- Technologies
    - May impact possible design choices
- Quality Attributes
    - Security, performance, usability, testability, maintainability, scalability,... => Seperate from functionality
- Crosscutting concerns
    - Centralized solutions => Logging, security, communication
- Arkitekturmønstre

Universell utforming og Brukergrensesnitt

Goal

- **Goal is usability**
- The design should be
    - Easy to use by everyone
    - Easy to understand by everyone
        - Text should be understandable
        - Elements should be identifiable
        - Icons/images should be meaningful

Do your research

- When designing a UI, ALWAYS research your target audience and map their needs and requirements based on principles for good UI desing
- Do NOT give your own opinions much weight. An interface suited for a developer is no going to be well suited for the majority of users unless you design an IDE
- DO test the UI on users
    - Mock-ups is often a good start
    - Focus groups
- Consider getting professional help, UX is a whole field of study, not the job of a lone dev. UD is not simple either

Meet expectations

- When someone use a given computer system, they have expectations
    - Windows layout and design
    - Control placements
    - Key bindings
    - Colors
- Comply with the standard look & feel on their platform
- Make sure to use system colors so that the color scheme of your application follows the user settings
- Make sure the application obeys changes to system properties, like font size
- If the platform supports skinning, make sure your application also uses the skins properly

- Make sure your application doesn't break when users change system properties
    - Fixed size dialogs may be problematic if font size is changed
- Remember, system UI customization controls are meant to be used, assume you users do.

Visuals

- Remember that not everyone have 20/20 vision
    - Near/far-sighted, colorblind, blind, injuries, illness
- By following the advice about meeting expectation, your application will support user defined text sizes and colors. This handles a large percentage of the cases
- Use colors with high contrast values
    - Use contrast analyser
- Remember color blindness
    - Not everyone sees all colors the same way
- Try to use a standard window layout, for most programs, this means meny/toolbars at the top, doalog buttons at the bottom etc...
- Don't get fancy. That donut-shaped dialog might be cool, but it's no user-friendly

Content

- Think about Internationalization and Localization
    - This may even impact the design(ex: right-to-left readers)
    - Remember that texts may be longer or shorter in other languages than yours
- Beware the use of symbols
    - Not friendly to screen readers
    - Culturally dependent
    - Some symbols are thaught, other are natural
        - What's natural to you might not be to others

Readability

- Don't try to put too much detail into one page/dialog/window
    - To much detail only means people will overlook more of it
    - Consider what the average user really needs
        - Hide the rest away
- Make sure text is as short and to the point as possible, and try to use simple language

Sensible Navigation

- People have different control preferences, like mouse, keyboard, touch, etc. Others have special needs
    - Let people use their preferred equipment
- Make sure the navigation path trough your application is sensible
    - TEST
- Make navigational elements obvious

Sound, Video and Animation

- Elements such as these can aid the user, but can just as well be a distraction or annoyance.

- How will these work with a screenreader? A braille display? A user who just wants to get done quickly

Robustness

- The design should be able to handle user errors gracefully, without breaking
- User errors should not have major consequences
    - Confirmation
    - Undo

Modulær oppbygning

Partitioning - Modularity

- Partitioning is one of the main ideas behind architectural styles in general
- By partitioning our applications we increase the modularity of the applicatoins
- Modules can be replaced and reused
    - Improves code reusability
    - Easier to maintain the applicaiotn
    - Easier to divide responsibility between teams

Three similar, yet distinctive patterns

- Model - View - Controller (MVC)
    - ASP.NET MVC
- Model - View - Presenter (MVP)
    - Web Forms, Windows Forms
- Model - View - View Model (MVVM)
    - WPF, UWP

Modularity

- Modularity is achieved when components can be easily seperated from each other(loose coupling)
    - Interfaces/Abstract Classes
    - Delegates/Function Pointers
    - Naming Conventions/Routing
- Hard-coded class names prevents seperation(tight coupling)

Classic MVP(Model-View-Controller)

- User interacts with View or Controller
    - Depensd on platform. In any case, View doesn't process data
- View
    - Made from UI components (Html, CSS, Javascript UI, .NET controls, etc)
    - Only responsible for displaying the data from the controller
- Model

    - Business logic and Data

- Controller

    - Processes user input
    - Interacts with the model and passes data to the view
    - Coordinates the view and the model

- View fetches data from model(model notifies view about updates)
- Controller reads data from model. Controller updates model
- User sends commands to controller
- View is viewed by the user. User sends command to the view, to be passed on to the controller.
- Controller manages view(View forwards commands from user)

MVP(Model-View-Presenter)

- User interacts with the View only
- There is one-to-one relationship between View and Presenter
- Presenter does more work than an MVC controller
- View has not reference to Model
- Views often have no logic at all, even relaying on the presenter for formatting
- Provides two-way communication between View and Presenter.
- Ideals in apps where views are quite different
- Example us: Standard Windows Forms app
- Presenter fetches data from model. Presenter updates model
- Presenter sets data in view. View passes on commands from user
- User views View. User sends commands to view

MVVM(Model-View-ViewModel)

- User interacts with the View
- There is many-to-one relationship between View and ViewModel - many Views can be mapped to one ViewModel
- View has a reference to ViewModel, but View Model has no information about the View
- Supports two-way data binding between View and ViewModel.
- ViewModel references model. Model notifies VM on updates

- View binds to properties on VM. VM notifies View on updates
- User views the View. User sends commands to the view.

Extension methods

- Extension methods allow us to add extra methods to an existing class without having access to the source code of the class
- This is heavily used by LINQ, adding methods to all classes implementing IEnumerable
- Only provides access to public properties/methods
- Really just a smart wrapper method, but is used like a real member, preventing all the usual problems with wrappers
- Implemented as a static method in a static class
  - Method signature tells which Class/Interface it applies to (`this` keyword, first parameter)
- Can not
  - Override existing methods
  - Access private/protected fields

LINQ to XML

- LINQ can handle XML files as a data source
- XML files can be bound to GUI elements
- But
  - XML is hierarchical - tree structure
  - XML elements have multiple values
  - Structured, yet unstructured

XML namespaces

- Just as in .NET, namespaces in XML are there to prevent naming conflicts
- Technically, all element names consists of a namespace + a local name
  - Must consider this when filtering on element names
  - Use the LocalName and NamespaceName properties to extract just the element name or the namespace name

XElement

- Core LINQ to XML class
- Represent a single XML element
  - but keep the context
    - Ancestors
    - Children

LINQ to XML methods

- In LINQ to XML we work on lists (IEnumerable) of XElement (and XAttribute) objects
- `Load(...)`: Loads an XML document from file
- `Parse(...)`: Creates XML document from string
- `Save(...)`: Saves the XML document to file

- `Descendants()/DescendantsAndSelf()`: Returns the child elemetns of an XElement
- `Ancestors()/AncestorsAndSelf()`: Returns the parent elemts of an XElement
- `Attributes()`: Returns the attributes of an XElement

More joins

- Joins can be performed across source types
    - EX: XML to EF
- Inner Join
    - The Join operator in LINQ perform an inner join
        - Only returns rows where bot matches
- Left/Right Outer Join
    - Returns all elements from the other
        - No LINQ operator handles outer joins by default
        - Can be accomplished by some clever LINQ statements
        - Or by implementing your own extension method
- Full Outer Joins
    - Returns all rows from both tables, matching where possible
- By default, LINQ checks to see if the two fields are equal
- More complicated checking cna be done by writing your own comparer class implementing `IEqualityComparer<T>`