Laboration 2: Othello
Grafik, Händelser, Databindning och Trådar

Objektorienterad Programutveckling (TOCK12) hösten 2022

Innehåll

1 Inledning

1.1 Syfte

Laborationen behandlar skapandet av grafiska gränssnitt, händelsehantering, databinding, trådar och objektorientering, inklusive objektorienterad analys och design med UML.

1.2 Lärandemål

Denna laborationen examinerar *delar av* nedanstående lärandemål (med grön färg) i kursplanen.

Kunskap och förståelse

1. Visa kunskap om grundläggande tekniker och terminologi som används inom objektorienterad programmering.
2. Visa kunskap om designprinciperna inom SOLID och GRASP och hur dessa relaterar till varandra.
3. Visa kunskap om designmönster inom GoF och hur SOLID och GRASP relaterar till dessa.
4. Visa förståelse för grundläggande modelleringstekniker och principer inom objektorienterad analys och design.

Färdighet och förmåga

1. Visa förmåga att kunna tillämpa SOLID och GRASP designprinciper samt GoF designmönster vid utveckling av en objektorienterad applikation.
2. Visa förmåga att kunna konstruera objektorienterade applikationer enligt goda objektorienterade principer.
3. Visa grundläggande förmåga att kommunicera en programdesign med ett modelleringsspråk.

Värderingsförmåga och förhållningssätt

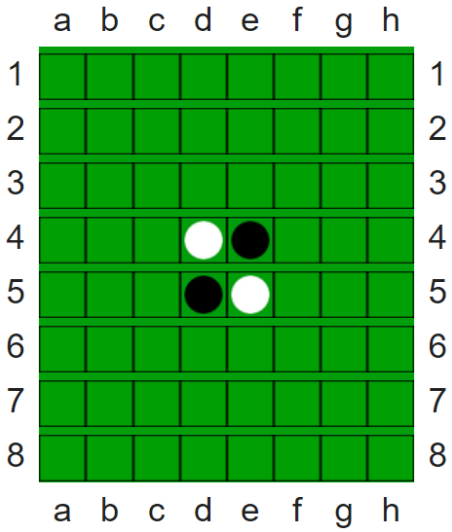
1. Visa förmåga att bedöma lämplighet av objektorienterade programkonstruktioner utifrån ett givet problem.

2 Uppgift

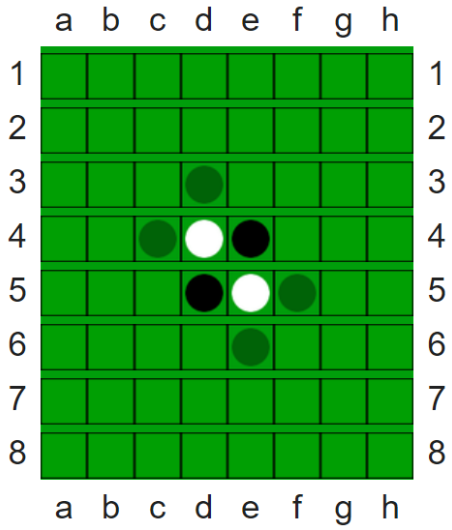
Ni har fått i uppdrag av en kund att skapa ett simpelt *Othello* spel, i form av en WPF applikation. https://en.wikipedia.org/wiki/Reversi

Programmet skall fungera på tre olika sätt: antingen skall en ensam spelare kunna spela mot datorn, eller så skall två spelare kunna spela mot varandra. Dessutom skall två datorspelare kunna spela mot varandra.

Othello är ett tvåmannaspel där den ena spelaren har svarta brickor och den andre spelaren har vita brickor. De brickor som används är svarta på den ena sidan och vita på den andra. Spelplanen är 8×8 rutor och utgångsläget är att två svarta och två vita brickor ligger *diagonalt* på de fyra mittenrutorna enligt figuren nedan. Svart har alltid första draget.



Ett drag innebär att en spelare placerar en bricka så att **minst** en av motståndarens brickor *fångas* av den just placerade brickan, dvs. ligger mellan denna och någon annan av den aktive spelarens brickor (utan några mellanliggande tomma rutor). Exempelvis, när svart skall utföra sitt första drag, är något av de fyra dragen (mörkgröna) som visas i figuren nedan möjliga (3d, 4c, 5f eller 6e).



Samtliga brickor, som på detta sätt *fångas*, vänds. Detta gäller även hela rader av brickor, men naturligtvis bara de som *fångas* av den brickan som placerades i draget (utan några mellanliggande tomma rutor). När draget är genomfört är det nästa spelares tur (se https://en.wikipedia.org/wiki/Reversi för fler exempel).

Om en spelare inte kan genomföra ett drag, går turen över till motståndaren. Den spelare som har flest brickor i sin färg då brädet är fullt (eller då igen av spelarna kan genomföra ett drag) vinner.

Programmet skall åtminstone innehålla följande klasser:

- `GameWindow` (namnet på huvudfönstret av typ `Window`, dvs döp om denna från `MainWindow`).
- `SetupGameDialog` (av typ `Window`, som används för att inhämta spelarnamn och spelartyp).
- `WinnerDialog` (av typ `Window`, som visar vilken spelare som vann spelpartiet).
- `DrawnDialog` (av typ `Window`, som visas då ett spelparti är oavgjort).
- `Player` (som representerar en *generell* spelare).
- `HumanPlayer` (som representerar en mänsklig spelare).
- `ComputerPlayer` (som representerar en *artificiell* datorspelare).
- `GameBoard` (som utgör en modell för spelbrädet, dvs är **ej** en *vy* klass).
- `GameManager` (som utgör en *kontroller* klass och representerar spelet som helhet).

Samtliga klasser ovan som slutar på `Dialog` är *dialoger*. Se denna länken för hur man skapar egendefinierade `Dialoger`.

Ett tips är att skapa en klass `GameGrid` av typ `UserControl`, som utgör det grafiska spelbrädet i spelet. En `UserControl` är en egendefinierad grafisk kontroll som kan användas som ett XAML element i ett fönster (`Window`). Det är frivilligt att skapa en egen `UserControl`, och denna länken visar hur man gör. Lägg märke till att `GameGrid` klassen är en vy (*view*) klass medan `GameBoard` klassen (i punktlistan ovan) är en modell (*model*) klass.

Det är naturligtvis möjligt att skapa fler klasser för att få en bra design.

Det är också frivilligt att använda en Model-View-ViewModel (MVVM) arkitektur eller inte.

3 Konstruktion av spelare

En spelare kan utformas på olika sätt. Här skall vi tillämpa en objektorienterad design med arv och polymorfism.

Givet att en spelare får reda på alla möjliga **giltiga drag** när det är spelarens tur, skall spelaren kunna tala om vilket det valda draget är (om en spelare överhuvudtaget har några giltiga drag).

Eftersom spelet skall utformas så att man antingen kan spela mot en annan mänsklig spelare eller mot en datorspelare, finns det två huvudtyper av spelare: `HumanPlayer` och `ComputerPlayer`. De två typerna har samma operationer, men operationerna utförs på olika sätt. För en mänsklig spelare erhålls nästa drag via input från det grafiska användargränssnittet, medan en datorspelare använder en algoritm för att beräkna nästa drag. Utgå därför från en abstrakt basklass `Player` och skapa sedan de två subklasserna `HumanPlayer` och `ComputerPlayer`. Antagligen blir det nödvändigt att använda **trådar** då en spelare beräknar eller väntar på nästa drag.

3.1 `Player`

Klassen `Player` är basklass för `HumanPlayer` och `ComputerPlayer`. Klassen innehåller t.ex. attributen `Name` och `Disk` samt en abstrakt metod (t.ex. `RequestMove()`) som `GameManager` kan anropa för att begära beräkning av nästa drag (här är det lämpligt att skicka med en kopia på `GameBoard` samt en lista med giltiga drag som parametrar). En spelares namn inhämtas från dialogen `SetUpGameDialog` och används i `WinnerDialog`. `Disk` anger om spelaren har svarta eller vita brickor (kan förslagsvis vara en `enum`).

3.2 `HumanPlayer`

Då nästa drag begärs från `GameManager`, väntar man tills den mänskliga användaren har utfört draget.

3.3 `ComputerPlayer`

Denna klassen beräknar nästa drag utifrån den information som finns i klassen `GameBoard` och listan med giltiga drag.

Då nästa drag begärs från `GameManager`, beräknas nästa drag utifrån den information som finns i klassen `GameBoard` och listan med giltiga drag.

Att skapa en intelligent datorspelare kan göras på flera olika sätt. Dock är inte detta det viktigaste i laborationen, utan det räcker att **slumpa ut ett av de giltiga dragen och returnera detta**. Dock måste eran design kunna hantera att datorspelaren *tänker* ett tag, utan att det grafiska användargränssnittet *hänger sig*. Alltså, om ni t.ex. använder en `Thread.Sleep(2000);` innan ni returnerar datorspelarens nästa drag, så skall det grafiska användargränssnittet inte *frysa*.

Som en frivillig uppgift (gör endast detta om ni hinner med), kan ni prova på att implementera *minimax algoritmen med alpha-beta klippning (pruning)* för att skapa en intelligent datorspelare. Hur intelligent datorspelaren är (svårighetsnivån) bestäms av sökdjupet i spelträdet och den *heuristiska evalueringsfunktionen* som används i minimax algoritmen. Datorspelaren använder då en **kopia** på `GameBoard` för att *söka framåt* i spelträdet. En beskrivning av minimax algoritmen finns här och här, samt några exempel på bra evalueringsfunktioner för othello finns här, här och här.

4 Konstruktion av GameManager och GameBoard

4.1 GameManager

GameManager är kontroller-klassen som administrerar pågående parti. GameManagers uppgift är enkel. Den skall gång på gång, omväxlande, be de två spelarna (subklasser till *Player*) att ange sitt nästa drag. Den skall också hålla koll på om någon av spelarna har vunnit, om spelet är oavgjort, eller om någon spelare har utfört ett ogiltigt drag, osv. Eftersom GameManagern inte skall veta vilken typ av spelare den kommunicerar med, skall polymorfism användas vid anrop av spelarnas operationer. GameManager kommunicerar givetvis också med *GameBoard*.

4.2 GameBoard

GameBoard är klassen som representerar modellen av spelbrädet (en *modell* klass, **ej** en *vy* klass). Den innehåller exempelvis en 8x8 matris med det pågående partiets tillstånd. När en spelare utför ett drag, kommer detta att registreras i GameBoarden. Då GameGriden (en *vy* klass, dvs en del av del grafiska användargränssnittet) ritar om spelplanen, hämtas informationen om partiets tillstånd från GameBoarden. Dessutom innehåller GameBoarden funktionaliteten (algoritmerna) för att manipulera 8x8 matrisen, avgöra om ett drag är giltigt, vända brickor, mm.

Ett spel avslutas när hela spelplanen är fylld med brickor eller när ingen av spelarna kan utföra ett drag. I båda fallen är det spelaren med flest brickor i sin egen färg som har vunnit.

5 Konstruktion av gränssnittsobjekt

En av huvudprinciperna vid objektorienterad design är att man skall bygga upp sitt program i form av separata, väl avgränsade, klasser. Detta underlättar förändringar i programmet. Därför är det viktigt att bibehålla inkapsling och tänka noggrant på att separera logik från det grafiska gränssnittet, det så kallade *Graphical User Interface (GUI)*. De klasser som innehåller logiken (*modell* klasser) skall inte känna till eller på något sätt vara beroende av gränssnittsobjekt (*vy* klasser). Detta medför att ett byte av det grafiska gränssnittet inte påverkar några av de övriga delarna i programmet.

Tänk på att GUI objekt skall modifieras på applikationens GUI tråd (*primärtråden*), dvs om ni har skapat en egen tråd (en *sekundärtråd*) i erat program som ni utför något jobb på och vill presentera resultatet i ett GUI objekt, så måste ni först växla över till applikationens GUI tråd (*primärtråden*) innan ni ändrar på GUI objektet. Ni kan växla *trådkontext* till GUI tråden (*primärtråden*) via koden nedan:

```
App.Current.Dispatcher.Invoke(() =>
{
    // Skriv kod som manipulerar gränssnittobjekt här.
});
```

Tyngre beräkningar, t.ex. då `GameManager` ber `Player` (framförallt `ComputerPlayer`) att ange sitt nästa drag, skall köras på en egen tråd (en *sekundärtråd*) så att det grafiska gränssnittet inte **`fryser`** medan beräkningen utförs.

5.1 `SetUpGameDialog`

I denna dialogen namnges de två spelarna samt om de skall vara `HumanPlayer` eller `ComputerPlayer`. Då dialogen stängs skapar (instantierar) man de två spelarna och skickar referenser till `GameManager`. Det exakta utseendet på dialogen får väljas fritt.

5.2 `GameWindow`

`GameWindow` är av typ `Window` (WPF applikationens huvudfönster) och finns uppe så länge programmet körs. Den innehåller, bland annat, en instans av `GameGrid` (kan vara av typ *UserControl*) samt knapparna och/eller menyerna **`New Game`** (som visar `SetUpGameDialog`) och **`Exit Game`** (som avslutar applikationen). Dessutom innehåller `GameWindow` andra XAML element som presenterar den aktuella ställningen i spelet (hur många brickor respektive spelare (svart/vit) har för närvarande).

5.3 `GameGrid`

`GameGrid` utgör den grafiska representationen av spelplanen, och kan i simplaste fallet vara ett XAML element av typ *Grid*, *Canvas* eller någon annan layout manager. Alternativt kan denna klassen vara av typ `UserControl` (som i sin tur innehåller ett element av typ *Grid*, *Canvas* eller någon annan layout manager). `GameGrid` inhämtar en mänsklig spelares (`HumanPlayer`) val av ruta då ett drag utförs.

Observera, trots att gränssnittsobjekten (*vy* klasser) ritar spelplanen, brickorna, menyer, knappar, osv. samt presenterar spelets nuvarande ställning och inhämtar val från användaren, så är det inte gränssnittsobjektens uppgift att hålla reda på spelets nuvarande tillstånd, utan när spelplanen skall ritas om hämtas all information från `GameBoard`en (en *model* klass).

5.4 `WinnerDialog` och `DrawnDialog`

Dessa klasserna utgör dialoger som informerar användaren om att en spelare har vunnit, där spelarnas namn skall visas i dialogen (`WinnerDialog`), samt att spelet är oavgjort (`DrawnDialog`). Dialogerna innehåller också en `OK` knapp som stänger dialogen.

6 Konstruktion av spelets huvudklass

6.1 `App`

`App` klassen skulle kunna döpas om till t.ex. `Othello`, men ni behöver inte göra detta. Denna klassen ärver från `Application` eftersom spelet är en WPF applikation.

7 OOAD och två-skiktad logisk arkitektur

Välj ett användningsfall och skapa ett Use Case, Systemsekvensdiagram (SSD), domänmodell (DM), Designsekvensdiagram (DSD) och Designklassdiagram (DCD) för detta.

För att ni inte skall blanda ihop klasser och objekt som skall ingå i UML-diagrammen, skall ni använda en två-skiktad logisk arkitektur med ett presentationslager (*PresentationLayer*) och ett affärslager (*BusinessLayer*). Detta innebär att ni skall skapa två projekt i eran lösning (*Solution*):

- *OthelloPresentation*, vilket utgör presentationslagret och är av projekttyp **WPF Application**.
- *OthelloBusiness*, vilket utgör affärslagret och är av projekttyp **Class Library**.

Lägg alla modell (*model*) klasser i *OthelloBusiness*. Klasserna i detta lagret kommer att finnas med i era UML diagram (och innehåller inga vy (*view*) klasser). Eran *GRASP Controller* klass (*GameManager*) kommer också att finnas i detta lagret.

Lägg övriga klasser, dvs. vy (*view*) klasser i *OthelloPresentation*. Inga klasser i detta lagret skall finnas med i era UML diagram. Om ni använder en Model-View-ViewModel (MVVM) design, så finns era `View` och `ViewModel` klasser här (lägg alla era `Model`) klasser (som skall finnas med i era UML diagram) i *OthelloBusiness*, så blir det lättare att hålla isär dem i denna laborationen.

Glöm inte att skapa en referens till projektet *OthelloBusiness* från projektet *OthelloPresentation* (högerklicka på *Dependencies* noden i *OthelloPresentation* projektet, välj *Add Project Reference* och kryssa i checkboxen för *OthelloBusiness* projektet).

Jag skulle föreslå att ni väljer användningsfallet för när ni skapar ett nytt spel (hällre än när en spelare utför ett drag, eftersom ni slipper modellera asynkrona anrop och trådar/aktiva objekt då).

8 Tips

När man använder Model-View-ViewModel (MVVM) designen, vill man oftast ha en ViewModel för varje View (`Window`). Dock föreslår jag (om ni använder MVVM) att ni använder en och samma ViewModel för alla era Views (`Window`) i denna laborationen. Annars blir det onödigt komplext för er.

Därför, om ni använder MVVM, och ni har lagt eran enda ViewModel klass i namespace `OthelloPresentation.ViewModels`, så föreslår jag att ni skapar följande `xmlns` i `App.xaml` ...

```
xmlns:viewmodels="clr-namespace:OthelloPresentation.ViewModels"
```

... och lägger till följande i eran `App.xaml`s *resource dictionary*:

```
<Application.Resources>
    <viewmodels:MainViewModel x:Key="MainViewModel"/>
</Application.Resources>
```

Då blir en och samma instans av eran ViewModel (`MainViewModel`) tillgänglig i samtliga era `Window` klasser. I XAML filen för era `Window` klasser kan ni helt enkelt sätta eran ViewModel som `DataContext` på följande sätt (detta sätts alltså i start-taggen för respektive `Window` element i XAML filen):

```
<Window
   ...
   DataContext="{StaticResource MainViewModel}"
>
```

9 Övrig information

Arbeta i grupper om 2. Ni väljer själva grupp på Canvas.

9.1 Handledning

Handledning sker vid tre tillfällen enligt de tider som finns i schemat. Tänk på att komma till handledningen med väl förberedda frågor. Bokningslistor för handledningen finns i Canvas (klicka på ett handledningstillfälle längst ner på startsidan för kursen), då kan ni välja ett handledningspass för det tillfället).

För varje handledningstillfälle (Handledning 1, 2 samt 3), kan ni boka ett handledningspass per grupp. Med reservation för ändringar (kolla alltid schemat), så gäller för närvarande nedanstående datum för varje handledningstillfälle. Ni ser också detta i TimeEdit och Canvas (klicka på en laboration längst ner på startsidan för kursen, då kommer ni till Canvas sidan för laborationen):

- Handledning 1: Tisdag, 4 Okt, 9:00-11:45, 13:00-14:45.
- Handledning 2: Fredag, 7 Okt, 13:00-17:30.
- Handledning 3: Tisdag, 11 Okt, 9:00-11:45 samt Onsdag, 12 Okt 10:00-11:45.

På Canvas sidan för laborationen hittar ni även Zoom-länkar för varje handledningstillfälle. Samma Zoom-länk används av alla grupper för ett handledningstillfälle (när det är nästa grupps tid, lämnar föregående grupp mötet).

9.2 Redovisning

Laborationen redovisas via Canvas (på laborationens Canvas sida), där samtliga filer lämnas in som en arkivfil (zip-fil eller rar-fil). Följande filer skall finnas med i arkivfilen:

- **Designdokumentation**: Användningsfall samt samtliga UML diagram för användningsfallet ni har valt.
- **Källkod**: Zippa ihop hela **Solution** foldern med subfoldrar (ni kan ta bort subfoldrarna **bin** och **obj** under varje **Projekt** folder om vill). Dokumentera relevanta typer och medlemmar m.h.a. kommentarer i koden (se Appendix A). Använd C#s namngivnings- och kodningskonventioner (se Appendix B).

Inlämning och examination av laborationen sker vid fyra tillfällen enligt schemat (informationen finns också i TimeEdit och på Canvas sidan för laborationen). Nedanstående datum gäller för varje examinationstillfälle:

- Deadline 1: Fredag, 21 Okt 2022, 23:59 (Vecka 42)
- Deadline 2: Fredag, 11 Nov 2022, 23:59 (Vecka 45)
- Deadline 3: Fredag, 13 Jan 2023, 23:59 (Vecka 2)
- Deadline 4: Fredag, 18 Aug 2023, 23:59 (Vecka 33)

9.3 Betyg

Endast betygen Underkänd (U) eller Godkänd (G) förekommer på laborationen.

A Appendix: C# kodkommentarer

https://docs.microsoft.com/en-us/dotnet/csharp/codedoc

För att dokumentera typer (t.ex. klasser) och metoder, så är de vanligaste dokumentationstaggarna `<summary>`, `<param>` och `<returns>`.

För att dokumentera en klass, skriver man sin dokumentationstext ovanför klassnamnet, enligt:

```
/// <summary>
/// Beskriv klassen här
/// </summary>
```

För att dokumentera en metod, skriver man sin dokumentationstext ovanför metodnamnet, enligt:

```
/// <summary>
/// Beskriv metoden här
/// </summary>
/// <param name="paramname"> Beskriv parameter här </param>
/// <returns> Beskriv returvärdet här </returns>
```

För att dokumentera attribut och properties kan samma syntax användas som för en klass, men är inte lika vanligt.

B Appendix: C#s namngivnings- och kodningskonventioner

Microsoft har definierat att antal namngivningskonventioner för C# med syftet att erhålla ett enhetligt utseende för all C# kod. https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

Microsoft har även definierat att antal kodningskonventioner för C# med syftet att erhålla välstrukturerad kod. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

Vi följer konventionerna med start i första laborationen så att ni lär er skriva C# kod på samma sätt som alla andra C#/.NET utvecklare gör.

C Appendix: Code Snippets

https://docs.microsoft.com/en-us/visualstudio/ide/visual-csharp-code-snippets?view=vs-2022

Programmeringsidiom

I Visual Studio finns ett snabbt sätt att skapa kodskelett för vanliga programmeringsidiom via **code snippets**. Exempelvis, om man skriver in `cw` efterföljt av ett `tab`-tecken, dvs `cw <tab>`, så ersätter Visual Studio detta med `Console.WriteLine()`. På ett likande sätt finns code snippets för, bl.a:

```
cw → Console.WriteLine()
prop → public int MyProperty { get; set: }
ctor → public <classnam> { }
```

Override

Om man implementerar en ärvd klass med virtuella eller abstrakta metoder kan man skriva `override <mellansalg>` så får man upp en lista med överridbara metoder, som generar ett metodskelett när man väljer en metod.

Implementera interface

Om man skall implementera ett interface i en typ, sätt cursorn direkt efter interfacenamnet och tryck `<ctrl> + <.>` så dyker ett val upp om att generera metodskelett för alla metoder i interfacet.

Using

Om man inte har importerat (using) det namespace som en typ är definierad i, sätt cursorn någonstans i typnamnet och tryck `<ctrl> + <.>` så dyker ett val upp att generera using direktivet automatiskt (så länge en referens till assemblyn finns i projeket).

Dokumentationskommentarer

För att generera ett skelett för att dokumentera t.ex. en klass eller metod, sätt cursorn precis ovanför namnet och skriv tre slash tecken `///`.

Region

För att dölja/visa kod i editorn kan man klicka på `+/-` symbolerna i vänstermarginalen. Man kan skapa egna namngivna **regioner** som kan döljas/visas genom att skriva koden mellan #region och #endregion direktiv, t.ex:

```
#region Attributes
// skriv attribut här
#endregion

#region Properties
// skriv properties här
#endregion

#region Constructors
// skriv konstruktorer här
#endregion
```

```
#region Methods
// skriv metoder här
#endregion
```