Datavetenskap

Erik Hermansson och Thomas Karlsson

Distribuerade grafiska användargränssnitt i .NET

Examensarbete, C-nivå

2003:15

Distribuerade grafiska användargränssnitt i .NET

Erik Hermansson och Thomas Karlsson

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Erik Hermansson

Thomas Karlsson

Godkänd, 2003-06-03

Handledare: Hannes Persson

Examinator: Stefan Alfredsson

Sammanfattning

Det här arbetet är centrerat runt en laboration för att styra en rymdfärja med hjälp av ett grafiskt användargränssnitt. Arbetet har gått ut på att skapa en distribuerad simulator och en klient som använder den. Laborationen bygger på en gammal version och skillnaden mellan den nya mot den gamla är att den nya är skriven för ramverket .NET från Microsoft. I .NET ingår en utvecklingsmiljö, Microsoft Visual Studio .NET, tillsammans med ett nytt programmeringsspråk. Detta nya språk, C#, har använts för att konstruera lösningen.

I denna uppsats kommer viktiga delar av .NET att sammanfattas tillsammans med C# och tekniker såsom Model View Controller och remoting. Allt detta för att kunna skapa en grafisk och distribuerad klient till rymdfärjan.

Avslutningsvis förklaras hur laborationens simulator är uppbyggd, tillsammans med skärmbilder från en tillhörande klient. Klienten är den del av laborationen som studenterna ska skapa och därför tillhandahålls endast ögonblicksbilder från en möjlig lösning. Dessutom tillkommer diskussioner om hur vissa delar bör implementeras.

Distributed Graphical User Interfaces in .NET

Abstract

This work is centered on a laboration for controlling a space shuttle via a graphical user interface. The work has been to create a distributed simulator with a client that controls the simulator. The laboration is based on an older version and the difference between the new laboration and the old one, is that the new is written for the Microsoft .NET framework. A part of .NET is the integrated development environment Microsoft Visual Studio .NET together with a new programming language. The new language, C#, has been used in order to construct the solution.

In this paper important parts of .NET will be summarized together with C# and techniques like the Model View Controller and remoting. All of this will be included in order to create a graphical and distributed client to the space shuttle.

In conclusion the laboration simulator will be explained in detail together with its client. Because of the fact that the simulator client is what the students are meant to create in the laboration, only a few screenshots of a possible solution will be shown. In addition a few discussions about certain implementation details will be held.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Introduktion

Avdelningen för datavetenskap vid institutionen för informationsteknologi vid Karlstads universitet har planer på att i kursen Grafiska användargränssnitt använda sig av den nya .NET-tekniken (läses dotnet) i en laboration. Denna uppsats är tänkt att ge ett underlag för laborationen och exempel på hur den kan genomföras. Samtidigt som uppsatsen skapar underlag för en ny laboration ska den också vara användbar i framtagandet av nya kurser eller andra laborationer med .NET och C# (läses c-sharp) som inriktning.

1.1 Bakgrund

Bakgrunden till detta arbete är att personalen som har undervisat i kursen Grafiska användargränssnitt vill ha en ny laboration. Laborationen bygger på en äldre version som tidigare implementerades i C++ och Qt. Det är tänkt att den nya versionen ska använda .NET och C# som utvecklingsverktyg. I och med att de idéerna som tas upp i arbetet är relativt nya företeelser måste en solid grund byggas som förklarar de viktigaste delarna av både .NET, Microsofts nya utvecklingsverktyg, och hur språket C# är konstruerat.

1.2 Problem

Det uppsatsen kommer att ta upp är hur .NET är uppbyggt och vad det kan göra. Även om arbetet belyser, för uppsatsen, viktiga delar av .NET är ändå tyngdpunkten på C# och vad man kan göra med det. Om C# kommer syntax, kontrollstrukturer, klasser, händelsehantering, gränssnittsprogrammering, Model View Controller (MVC) och andra väsentliga delar att diskuteras.

När alla teoretiska punkter har belysts kommer flera tekniker att sammanfogas i en implementation. Implementationen är en laboration som går ut på att ifrån en distribuerad modell skapa ett grafiskt gränssnitt, för att styra en simulerad rymdfärja från uppskjutning till rymd.

För att kunna vara så flexibel och användbar som möjligt för lärare och andra kommer arbetet att diskutera fler delar av .NET och C# än vad som behövs för att genomföra konstruktionslösningen.

1.3 Syfte

Examensarbetets syfte är att ge en bra grund för lärare vid Karlstads universitet att förstå och eventuellt kunna använda .NET och C# i framtida kurser och laborationer. Vidare ska en laboration implementeras för att användas i kursen Grafiska användargränssnitt.

1.4 Förutsättningar och mål

Uppsatsens mål är att illustrera följande:

- .NET
    - Illustrera hur arkitekturen är uppbyggd och fungerar
    - Användningsområden
    - Fördelar och nackdelar med .NET
- C#
    - Syntax
    - Klasser, objekt och variabler
    - Arv, gränssnitt och polymorfism
    - Händelsehantering
    - Hur det används
    - Fördelar och nackdelar gentemot andra stora programspråk, t.ex. C++ och Java

Utöver detta skall en implementationsuppgift utföras som skall innefatta följande:

- En rymdfärjemodell
    - Liknas vid en ”vanlig” rymdfärja
    - Signalerar ut händelser och larm
    - Kan påverkas av funktionsanrop
    - Distribuerad
- Ett grafiskt användargränssnitt
    - Ska visa rymdfärjemodellens data
    - Ska kunna påverka modellen
    - Ska använda MVC-tekniken
    - Flera samtidiga gränssnitt mot en modell

De förutsättningar som krävts för att realisera denna uppsats har varit MS (Microsoft) Visual Studio .NET, ett utvecklingsverktyg för att programmera i C#. Dessutom har en viss förståelse för objektorientering krävts.

1.5 Disposition

Arbetet är uppdelat i tre huvudsakliga delar. Kapitel 1 förklarar hur .NET fungerar och fortsätter med att ge en syntaktisk bild av C#. Detta för att ge en grund för att förstå teknikerna i resten av arbetet, där många av teorierna från den första delen implementeras. Kapitel 3 förklarar konstruktionslösningen och de tekniker som använts i konstruktionen, och kapitel 4 visar upp och förklarar implementeringen. Uppsatsen avslutas med två kapitel som sammanfattar arbetet och ger författarnas personliga syn på resultatet.

2 .NET & C#

.NET är Microsofts nya miljö för exekvering av program, som kan liknas med Javas virtuella maskin (JVM). .NET möjliggör plattformsoberoende applikationer, förutsatt att .NET finns utvecklad för plattformen i fråga. Det finns projekt som i skrivande stund implementerar denna exekveringsmiljö för Unix och Linux, vilket bäddar för större genomslagskraft i framtiden. Till exempel kommer program skrivna i Windows att kunna köras på nästan vilka operativsystem som helst. Till skillnad från Java är .NET även språkoberoende, något som kommer vara till stor hjälp för programmerare som då slipper lära sig nya språk. Det uppsatsen kommer att behandla om .NET är den huvudsakliga arkitekturen, tillämpningsområden och fördelar/nackdelar.

Även om Microsoft möjliggjort användningen av många olika språk i .NET, utvecklades ändå ett nytt språk, som likt Java är tänkt att vara intuitivt, lätt att lära sig och vara väl lämpat för Internetbaserade applikationer. Språket fick namnet C#. När programmeringsspråket skapades lånade utvecklarna det bästa från flera världar, då främst från Visual Basic, C++ och Java. Det är, likt sina föregångare, objektorienterat och de flesta källorna till vanliga programmeringsmisstag har eliminerats. Däribland pekare.

Orsaken att uppsatsen tar upp dessa specifika delar av .NET och C# är för att ge en tillräcklig grund att förstå konstruktionslösningen i kapitel 3 och implementationen i kapitel 4 och för att ge kunskapen att förstå de delar av .NET och C# som författarna ansett viktigast.

2.1 .NET

.NET är en nytt koncept från Microsoft Corporation, framtaget för att ge utvecklare en framtidssäker grund att stå på, utan att tvinga fram omskrivning av gammal kod. .NET gör det möjligt att inkludera gammal teknik i nya applikationer på ett enkelt sätt, samtidigt som den gamla koden säkras i den gemensamma kontrollstrukturen som tillhandahålls. .NET kan innebära många olika begrepp, men när det i den här uppsatsen talas om .NET, är det exekveringsmiljön, .NET-ramverket (framework), som avses. Ramverket sköter exekvering av alla .NET-utvecklade applikationer. .NET innefattar bland annat också produkter och Internetbaserade tjänster skrivna för .NET-ramverket. Se mer i [5].

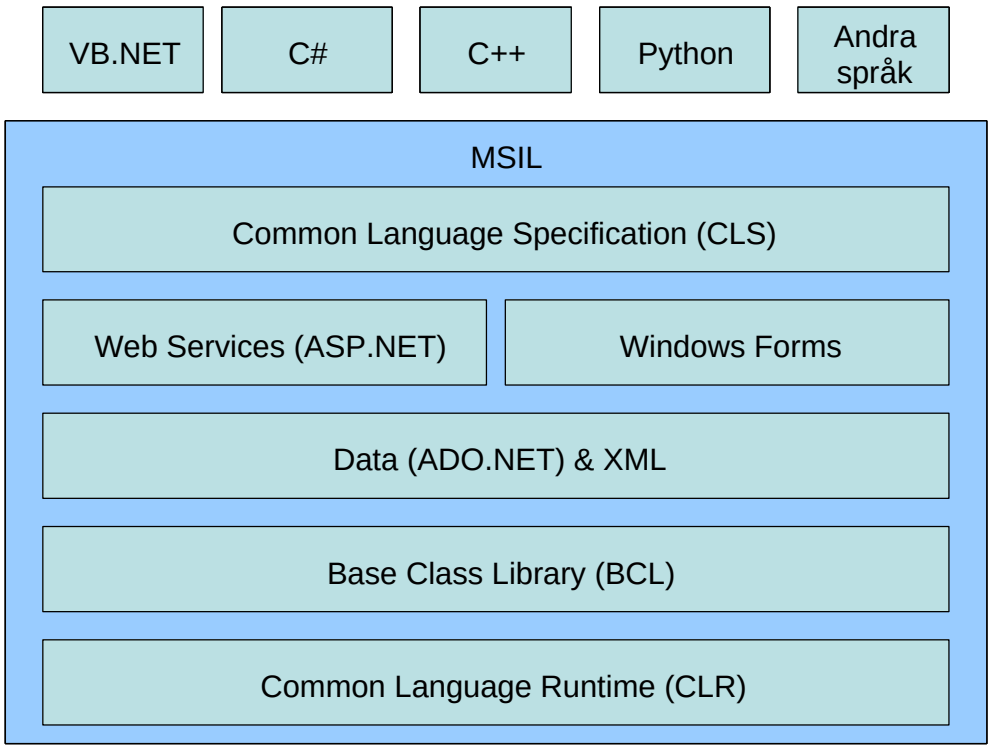
I kapitel 2.1.1 ges en sammanfattning av .NET-ramverket, och i kapitel 2.1.2 diskuteras några fördelar och nackdelar med denna nya teknik.

2.1.1 Redogörelse för .NET

.NET är alltför omfattande för att beskrivas i sin helhet, och därför kommer denna uppsats framför allt att behandla de delar som kan anses som viktigast utifrån implementationsuppgiften. Det här avsnittet börjar med att ge en introduktion till ramverkets arkitektur. Därefter ges en sammanfattning av vad Microsoft Intermediate Language (MSIL) är, följt av en förklaring av vad Just In Time-kompilering (JIT-kompilering) innebär.

2.1.1.1 Arkitekturen

Med .NET introducerar Microsoft några nyheter för windowsprogrammerare, bland annat intermediär kod och JIT-kompilering. Dessa nyheter möjliggörs med Microsofts nya runtime-system, Common Language Runtime (CLR). Förutom nämnda begrepp använder den sig också av automatisk minneshantering och garbage collection (automatisk avallokering av minne), något som tidigare mest förknippats med Javaprogrammering.



Figur 1 – .NET-ramverkets arkitektur

Figur 1 visar hur de olika delarna i .NET hänger ihop. På det understa lagret finns CLR, .NETs virtuella maskin, som förutom att ha hand om minneshantering, även sköter språkintegrering, process-, tråd- och undantagshantering samt dynamisk bindning under programkörning. Base Class Library (BCL) sköter standardfunktionalitet, som till exempel in-, utmatning och strängmanipulation. Även funktionalitet för kommunikation och trådning

finns i detta lager. Nästa lager tillhandahåller funktionalitet för ADO.NET och XML (eXtensible Markup Language). ADO.NET (ActiveX Data Objects) är en samling klasser för databashantering med inbyggt stöd för bland annat SQL-transaktioner (Structured Query Language). XML är ett metamärkspråk som används för att lagra och skicka data. För mer information om XML, se [10]. De två delarna i nästa lager är web services (webbtjänster), där ASP.NET (Active Server Pages) ingår, och Windows Forms. I hela lagret återfinns klasser för representation av grafiska användargränssnitt, där ASP.NET är utformat för skapande av Internetapplikationer, medan Windows Forms ger gränssnitt för Windowsapplikationer. Common Language Specification (CLS) är en samling regler som ska garantera att program skrivna i olika språk kan fungera tillsammans. De flesta klasser i ramverket följer CLS, men det finns vissa undantag i de språk som vill stödja en viss funktionalitet specifikt för det språket. Mer information om CLS finns att hitta på [2]. I det översta lagret finns de olika programspråken som .NET stödjer.

När nu en helhetsbild av .NET-ramverket har givits, kommer de viktigaste delarna som rör uppsatsen att gås igenom mer ingående.

2.1.1.2 Intermediär kod

Här följer två exempel på hur strängen ”Hello World!” skrivs ut på skärmen. Det första exemplet är skrivet i C# och nästa exempel visar samma program i MSIL, ett assemblerliknande språk som MS har skapat för att abstrahera bort plattformsberoende.

Hello World! i C#:

```
using System;
class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine(”Hello World!”);
    }
}
```

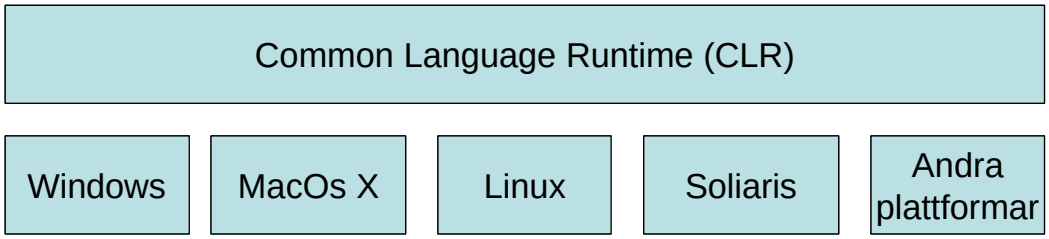
Kod som utför samma sak men i MSIL:

```
  .method private hidebysig static void Main(string[] args) cil↵
managed
  {
     .entrypoint
     .custom instance void↵
[mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
     //Code size     11(0xb)
     .maxstack 1
     IL_0000: ldstr  ”Hello World!”
     IL_0005: call   void↵
[mscorlib]System.Console::WriteLine(string)
     IL_000a: ret
  }
```

Vad är då nyttan med MSIL? Dagens program är skrivna i något programmeringsspråk. Sedan kompileras källkoden en gång för varje plattform som programmet ska stödja. Problemet med detta är just att kompilera, och eventuellt att skriva om vissa delar innan, för varje system. Detta tar tid och kostar i slutändan företag stora pengar. Orsaken är att olika system inte har samma maskinkod, dvs. det språk maskinen förstår och kan exekvera. Olika maskinkod betyder att program i grund och botten ser olika ut på olika plattformar.

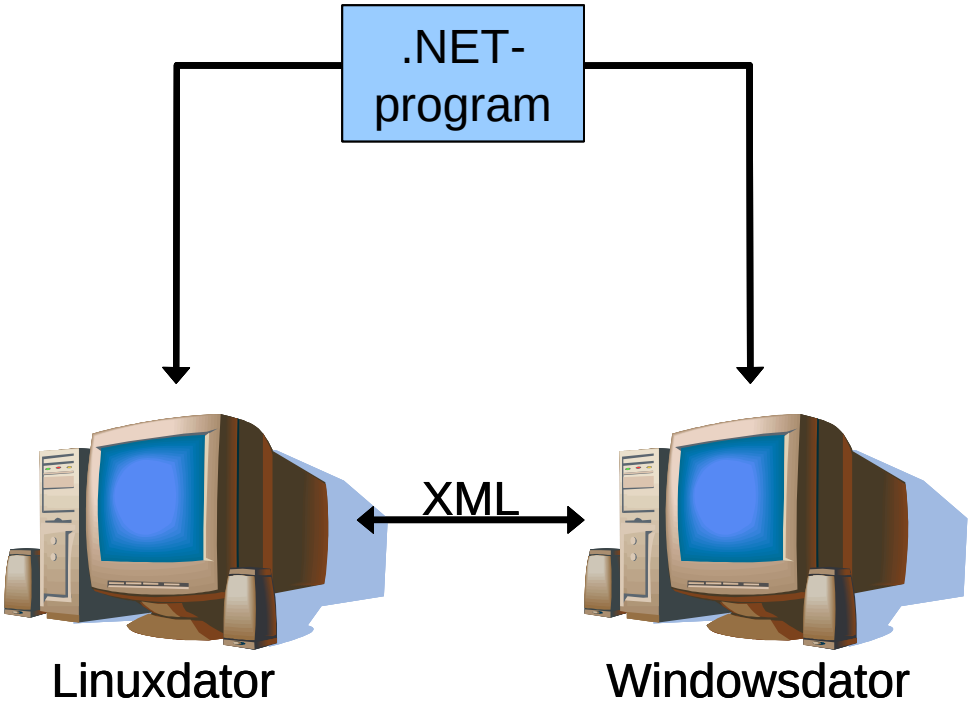
Microsoft med .NET har, likt SUN med JVM, utvecklat ett sätt att kringgå detta problem. De kompilerar källkod till en intermediär kod som är ett mellanting av maskin- och källkod. Intermediär betyder mellanhand eller förmedlande, något som passar bra in på MSIL. Vad .NET och JVM gör är att abstrahera bort olikheterna i hårdvaran. Detta görs genom att förmedla kommunikationen mellan applikationer och hårdvaran på ett sånt sätt att all hårdvara tolkas lika, oberoende av plattform. Som Figur 1 visar är all kod för alla standardklasser i .NET skrivna i MSIL.

MSIL är ett språk, likt andra programspråk, i den mening att det är möjligt att programmera i det, men då det liknar assembler är detta svårt att använda i praktiken.



Figur 2 – .NET abstraherar bort plattformen

Den intermediära koden är helt plattformsoberoende, men inte miljön den körs i, se Figur 2. Microsoft håller på att implementera .NET-miljön för FreeBSD och MacOS X, och ett projekt under namnet Mono[1] gör detsamma fast med Linux som målplattform. Detta gör det möjligt att köra .NET-program på vilken plattform som helst, vilket illustreras av Figur 3. Som bilden visar används XML som informationsbärare mellan datorer för att ge ett universellt gränssnitt att kommunicera mot.



Figur 3 – .NET-program är plattformsoberoende

.NET och JVM är alltså ganska lika, men skiljer sig starkt på en viktig punkt. All kod i .NET blir kompilerad till MSIL oavsett vilket programmeringsspråk som har använts, medan Java bara kan använda programmeringsspråket Java. Detta betyder rent praktiskt, i .NET:s fall, att det inte spelar någon roll om programmerare vill programmera i C++, C# eller Cobol. Alla språk i .NET använder dessutom samma klasser, det så kallade basklassbiblioteket, BCL, och det går att blanda klasser i olika språk inom samma program. Det går till exempel att skapa ett objekt av en Visual Basic-klass i C#.

2.1.1.3 Just In Time-kompilering

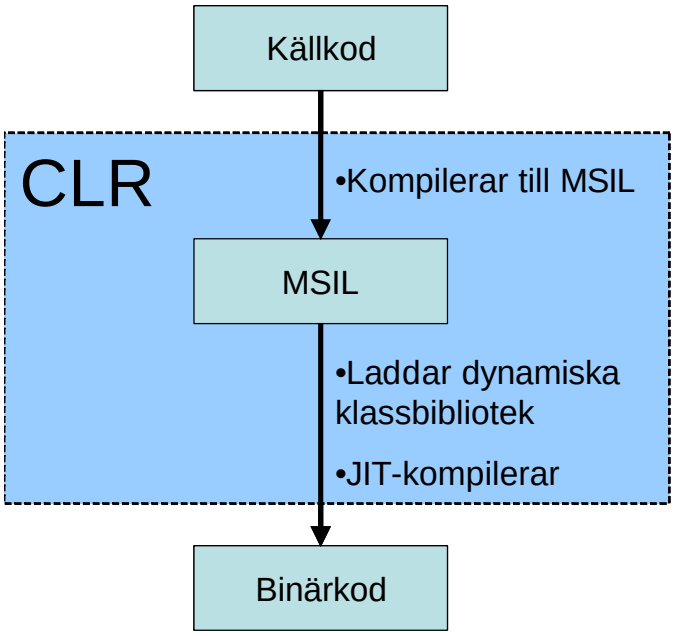
Som nämndes ovan är MSIL inte ett maskinspråk, det vill säga datorer kan inte förstå det, utan en kompilering krävs. Det är här Just In Time(JIT)-kompilatorn kommer in i bilden, då

[1] http://www.go-mono.com

den kompilerar den mellanliggande koden till binärkod. I .NET sker detta under körning allt eftersom koden behövs. Kod som inte behövs kompileras alltså inte.

Det är JIT-kompilatorn som gör om den plattformsoberoende MSIL-koden till plattformsberoende maskinkod. JIT-kompilatorn måste alltså implementeras på de plattformar man vill köra MSIL-kod på.

Nedanstående bild, se Figur 4, förklarar hur källkoden blir till körbar kod med hjälp av JIT-kompilatorn via MSIL.



Figur 4 – Förenklad kodcykel

Det finns möjlighet att, vid kompilering av källkod, kompilera direkt till maskinkod. Detta gör programmet snabbare vid uppstart, men programmet blir då plattformsberoende. Se mer i [6].

2.1.2 Tillämpningsområden

.NET och C# skapades främst för att snabbt och enkelt kunna skapa webbaserade tjänster, som interaktiva hemsidor och webbtjänster (web services). Det går förstås att skapa andra applikationer också, till exempel databasapplikationer.

Microsoft har utvecklat många tilläggsklasser för olika områden inom programmering. Det är enkelt att skapa sockets, komprimera filer och göra mycket annat med ganska lite jobb för att det redan finns färdiga klasser för programmeraren att använda.

På det stora hela är .NET skapat för att kunna använda till många olika områden, främst webbutveckling, men även för mycket annat. Förekomsten av många verktyg, tilläggsklasser och exempel i Microsoft Studio .NET-dokumentationen och på Internet gör utveckling så

mycket enklare. .NET är en bra investering för många företag och privatpersoner som vill producera applikationer snabbt och enkelt.

2.1.3 Fördelar och nackdelar

Här nedan följer några viktiga fördelar och nackdelar som ramverket .NET har gentemot andra utvecklingsmiljöer. En av de viktigaste egenskaperna och fördelarna är att .NET är plattforms- och språkoberoende. Detta kan jämföras med SUN:s JVM, som förvisso är plattformsoberoende, men som endast fungerar med programspråket Java. Ännu en skillnad består i hur JIT-kompileringen fungerar. I Java kompileras allt klassvis, dvs. en klass i taget, när det behövs, medan .NETs JIT-kompilering sker metodvis, dvs. en metod i taget. Kontentan av detta blir att i Java kompileras hela klassen även om bara en metod ska användas, medan .NET endast kompilerar den metod som skall användas. Om detta är en fördel eller nackdel för respektive modell kan diskuteras. Båda metoderna har fördelar och nackdelar. För mer information om SUN och Java, se [7].

En nackdel som delas av både .NET och Java är att det i nuläget krävs en nedladdning av exekveringsmiljön. I de senast utgivna Windows-versionerna har .NET inkluderats redan från början så detta problem kommer inte att bli långvarigt för .NET.

En annan nackdel i .NET är effektivitetsaspekten. Eftersom koden kompileras under körning blir programmen långsammare. Dessutom går det inte att komma ifrån att kompilerade program i C/C++ är mycket effektivare än en .NET-applikation, se [8][14][15]. Dock finns då annat som talar för .NET jämfört med andra effektivare programspråk; automatiskt minneshantering, typsäkerhet och snabbare programutveckling. Tiden det tar att utveckla en applikation i C eller C++ ofta är många gånger större än motsvarande i .NET. Allt detta blir en avvägning som företag och privatpersoner får göra på egen hand.

2.2 C#

C# är ett helt nytt objektorienterat språk, utvecklat av Anders Hejlsberg och Scott Wiltamuth. När det skapades tänkte utvecklarna på att skapa en bra blandning av enkelhet, uttrycksfullhet och effektivitet. C# är MS egen version av Java och kan ses som en blandning av C++ och Java. Från Java lånas vissa viktiga aspekter, såsom gränssnitt och många tilläggsklasser. Dessa används för att utföra de flesta vanliga operationerna som programmerare önskar.

Detta kapitel börjar med att förklara de basala delarna av C#; variabler, datatyper, matriser, kontrollstrukurer osv. Därefter fortsätter det med att förklara klass, värdeklass och skillnaderna dem emellan. Andra viktiga delar behandlas också, såsom undantagshantering, händelsehantering, grafiska användargränssnitt och dokumentation av kod. Kapitlet avslutas med att ge en enkel jämförelse mellan C#, C++ och Java.

Orsakerna att så många olika delar av C# ges i detta kapitel är att ge läsaren en god förståelse för språket, visa hur kraftfullt och enkelt det är att programmera i och för att förstå konstruktionslösningen i kapitel 4.

2.2.1 Datatyper

C# kan sägas vara helt objektorienterat. Med detta menas att språket uppför sig på ett objektorienterat sätt, men vissa undantag har gjorts för att snabba upp exekveringen. De vanligaste datatyperna såsom heltal och decimaltal, representeras inte av klasser utan av strukturer (eng. structs). Dessa typer kallas värdetyper till skillnad från klasser som kallas referenstyper. För att få dessa värdetyper att bete sig som objekt, inför C# två nya begrepp, boxing och unboxing. För att vara så effektiv som möjligt, försöker .NET att använda variabler som värdetyper i så stor utsträckning som möjligt, men om kraften hos ett objekt behövs, konverteras värdetypen till en referenstyp. Denna konvertering kallas boxing. Unboxing är motsatsen, det vill säga när ett objekt konverteras till ett värde. Båda sker automatiskt i exekveringsmiljön.

Vilka är då de vanligaste datatyperna i C#? Nedan följer en tabell, se Tabell 1, över de mest använda typerna och några av deras egenskaper.

Datatyp (C#-alias)	Storlek (bitar)	Klassnamn	Intervall		
char	16	System.Char	0	till	65535
sbyte	8	System.SByte	-128	till	127
byte	8	System.Byte	0	till	255
short	16	System.Int16	-32768	till	32767
ushort	16	System.UInt16	0	till	65535
int	32	System.Int32	-2147483648	till	2147483647
uint	32	System.UInt32	0	till	4294967295
long	64	System.Int64	-9,22E+18	till	9,22E+18
ulong	64	System.UInt64	0	till	1,84E+19
float	32	System.Single	1,50E-45	till	3,40E+38
double	64	System.Double	5,0E-324	till	1,7E+308
decimal	128	System.Decimal	1,80E-28	till	7,90E+28
bool	8	System.Boolean	True	och	False

Tabell 1 – Vanliga datatyper i C#

Som tabellen visar finns det något som kallas alias. Detta är egentligen bara en förkortning för att slippa behöva skriva ut det riktiga klassnamnet när man skapar en variabel av de vanliga datatyperna. Detta innebär att man till exempel skriver char för System.Char-klassen.

Det bör observeras att typen char inte lagrar ASCII-värden som C/C++ gör, utan använder unicode för representation av tecken precis som Java gör. Med unicode kan alla språks tecken representeras.

2.2.1.1 Variabler

Variabeldeklarationer i C# liknar dem i C/C++ och Java. För att deklarera en värdevariabel skrivs:

```
<typ> <variabelnamn> [= <initialvärde>];
```

Ex.
```
int temp;
float var = 3,14f;
```

I exemplet ovan visas användningen av suffix. Dessa används för att förtydliga vilken typ en variabel är av, men också för att försäkra att beräkning görs med rätt typ. Till exempel kan float- och decimalberäkningar utföras som double om suffix inte anges. Nedan följer en tabell, se Tabell 2, över suffixen för de vanliga datatyperna.

Datatyp	Suffix
uint	u
long	l
ulong	ul
float	f
double	d
decimal	m

Tabell 2 – Typsuffix

2.2.1.2 Matriser

Att använda matrisvariabler i C# är likt C++ och Java, men vissa ändringar i syntaxen har gjorts. För att skapa en matrisvariabel skrivs:

```
<typ>[] <variabelnamn> [initialisering];
```

Ex.
```
// Oinitialiserad deklaration
MinKlass[] minArray;
int[] integerArray = new int[128];

// Initialiserad deklaration. Matris av tre strängar
string[] landLista = { ”Sverige”, ”Norge”, ”Danmark” };

// Ännu ett exempel på initialiserad matris
int[] åldrar = new int[5] { 10, 33, 28, 45, 16 };
```

För att skapa matriser med fler dimensioner än en, skrivs kommatecken mellan hakparenteserna. Ett kommatecken för varje dimension utöver den första.

Ex.
```
int[ , ] pos = new int[2, 3] {{1, 4, 0}, {8, 7, 15}};
short[ , , ] koord = new short[10, 11, 12];
```

För att skapa matriser utan att varje dimension måste ha lika storlek, används en annan syntax. Dessa matriser kallas ”jagged”. Nedan följer ett exempel där varje månad har lika många heltalsvärden som dagar i den aktuella månaden.

Ex.
```
int[][] månader = new int[12][];
månader[0] = new int[31];
månader[1] = new int[28];
     .
     .
månader[11] = new int[31];
```

När en matris har givits en storlek kan inte denna ändras. Om en dynamisk matris behövs kan någon klass från System.Collections med fördel användas.

2.2.2 Kontrollstrukturer

Kontrollstrukturerna i C# liknar dem i C/C++, med vissa små undantag. Alla kommer att nämnas, men endast de nya språkkonstruktionerna kommer att förklaras i detalj. Först kommer selektioner att tas upp, följt av iterationer.

2.2.2.1 Selektioner

Programsatsen **if-else** ser ut och fungerar exakt som i C/C++ och Java, och används för att exekvera en programsats när ett villkor är uppfyllt. Den enda skillnaden mot nämnda språks motsvarighet är att villkoret måste vara ett booleskt värde, dvs. noll och icke-noll resulterar inte i falskt respektive sant.

switch-satsen kan vara ett bra val när det finns flera fasta alternativ. Nyckelordet **switch** följs av en parentes med variabeln som ska undersökas. Denna variabel måste vara av strängtyp eller heltal, alternativt att det är ett uttryck som returnerar en av de två typerna.

Notera att alla **case**-block, inklusive det sista, måste avslutas med nyckelordet **break**. Detta för att förhindra att flera utfall exekveras efter varandra. Däremot kan flera fall exekvera efter varandra om man grupperar dem. Detta sker genom att fallet lämnas tomt.

Ex.
```
switch(text)
{
    case "ja":
    case "Ja":
    case "JA": Console.WriteLine("Du accepterade"); break;
    case "nej":
    case "Nej":
    case "NEJ": Console.WriteLine("Du accepterade inte");break;
    default: goto case "NEJ"; break;
}
```

I exemplet ovan visas användningen av **default**, som används för att ta hand om de fall som inte matchar något av de andra. Dessutom visas även hur **goto** kan användas för att hoppa i en switchsats. **goto** kan även användas för att hoppa ur ett eller flera programblock till en i koden markerad del. Det går inte att hoppa in i ett programblock med **goto**.

2.2.2.2 Iterationer

for används för att iterera genom en kodblock ett visst antal gånger. Syntaxen för **for** ser likadan ut i C# som i C/C++ och Java, och fungerar på samma sätt. Detsamma gäller för **while** och **do-while**. Som tidigare nämnts om **if**-satser så måste villkoret vara ett booleskt uttryck.

Vid iterering över matriser eller andra datamängder används med fördel **foreach**. **foreach** stegar igenom alla element i en mängd och returnerar ut ett värde i en specificerad nyckel. **foreach**-satsen påbörjas med ordet **foreach** med en parentes efter. I denna parentes deklareras en nyckel av en viss typ, följt av nyckelordet **in** och en mängd. Typen på nyckeln ska vara av samma typ som i mängden man vill iterera genom. Därefter skrivs det programblock man vill ska exekvera för varje element i mängden.

Ex.
```
foreach(string namn in stringArray)
{
    Console.WriteLine(namn);
}
```

Nyckeln i **foreach**-satsen är lokal för blocket den deklarerats i.

2.2.3 Klasser

Klassen är den grundläggande byggstenen i C#. Alla program som skrivs, använder minst en klass, men oftast betydligt fler. Syntaxen för att deklarera en klass följer nedan:

```
<åtkomst> class <klassnamn> [: basklass [, gränsnitt]*]
{<klasskropp>}
```

Ex.
```
public class MinKlass
{}
```

Efter klassdefinitionen följer klassens kropp. Den innehåller alla klassens medlemmar. För mer information om klasser, se [3][4][5].

2.2.3.1 Åtkomstmodifierare

Varje **enum** (se kapitel 2.2.3.3), klassmedlem, **struct** (se kapitel 2.2.4) och gränsnitt (se 2.2.3.8) kan ha en åtkomst, specificerad av programmeraren. Detta innebär att det mesta som deklareras kan vara av olika åtkomst för andra klasser och program. Inom en klass har alla medlemmar **private** som standardåtkomst. Se Tabell 3 för alla standardåtkomster i C#.

Medlemmar till	Defaultåtkomst	Tillåtna åtkomster
enum	public	None
class	private	public
		protected
		internal
		private
		protected internal
interface	public	None
struct	private	public
		internal
		private

Tabell 3 – Standardåtkomster i C#

Tabell 4 visar de 5 olika åtkomstmodifierarna som finns att tillgå i C#. Programmerare som har programmerat i Java och C++ känner igen **public**, **protected** och **private**. De fungerar precis som i de båda språken, men **internal** och **protected internal** fungerar lite annorlunda. **internal** har åtkomst inom samma assembly, som i de vanligaste fallen innebär inom samma program. En assembly är en kompileringsenhet, men dessa är inte så viktigt för denna uppsats och kommer därför inte att beskrivas närmare. För mer information om åtkomstmodifierare och assemblies, se [3][4][5][11].

Åtkomstmodifierare	Åtkomliga av
private	Medlemmar av samma klass
protected	Medlemmar av samma klass och ärvda klasser
internal	Medlemmar av samma program
protected internal	Medlemmar av samma program och ärvda klasser
public	Alla - publik åtkomst

Tabell 4 – Åtkomstmodifierare i C#

2.2.3.2 Namnrymder

För att få en bra struktur på programkoden i .NET, använder man namnrymder (namespaces) och organiserar med dessa klassbiblioteken hierarkiskt. Med .NET:s SDK följer en mängd namnrymder, såsom System, System.Windows.Forms osv. Namnen på dessa är valda så att man lätt ska kunna identifiera vilka klasser de innehåller. En klass fullständiga namn är klassnamnet med dess namnrymd som prefix. För att avgränsa namnrymder från de som är nästlade inom dem används en punkt. Samma används för att avgränsa namnrymderna från klasserna. Ta till exempel namnrymden System.Windows.Forms, som innehåller en klass som heter Button. För att skapa ett object av klassen Button, skrivs:

```
System.Windows.Forms.Button but = new System.Windows.Forms.Button();
```

Som synes kan det bli väldigt långa deklarationer om klassens fullständiga namn skall användas varje gång. För att slippa detta, skrivs ordet using för att tala om att en viss namnrymd skall användas, och det medför att endast namnet på klassen behövs vid deklarationen. Exemplet ovan skulle då se ut så här:

```
using System.Windows.Forms;
      ...
Button button = new Button();
```

using-kommandon skrivs överst i filen, dvs. innan några klassdefinitioner gjorts. För att skapa en namnrymd skrivs nyckelordet **namespace** följt av ett namn och klamrar som omsluter namnrymden.

Ex.
```
namepace MinNamnRymd
{
    //Klassdefinitioner och liknande
}
```

För mer information om namnrymder, se [3][4][5].

2.2.3.3 Klassmedlemmar

En klass har många attribut och dessa kallas i objektorientering för objektets egenskaper. En klass har också flera metoder som kallas för objektets operationer. Många av de medlemmar som kommer att nämnas här förklaras i senare kapitel. De nämns ändå här för att samla alla klassmedlemmar på en plats. Nedan följer ett skelett av en klass och allt som en klass kan innehålla med kapitelhänvisningar till var informationen står att finna i uppsatsen:

```
public class MinKlass
{
    // Kapitel 2.2.3.4 - Metoder
        // Konstruktorer
        // Destruktorer
        // Metoder
    // Detta kapitel
        // Fält (variabler)
        // Konstanter
        // Egenskaper (properties)
        // Enumeratorer
        // Indexerare
    // Kapitel 2.2.5 - Händelsehantering
        // Händelser
        // Delegater
    // Kapitel 2.2.3 - Klasser
        // Inre klasser
}
```

Alla klassmedlemmar kan vara statiska eller inte. För att ange att en medlem är statisk används nyckelordet **static**. För mer information om **static**, se 2.2.3.4. Att en medlem är statisk innebär samma sak som i C++ och Java.

De attribut en klass har, eller klassens data, indelas i fält, egenskaper och konstanter. Klassens fält utgörs av variabler och består av olika typer av objekt. Det finns två typer av objekt en klass kan ha som fält; referensobjekt och värdeobjekt. Referensobjekt är vanliga objekt instantierade från en helt vanlig klass, medan värdeobjekt är variabler instantierade från en värdeklass, se kapitel 2.2.4. Värdeobjekt kallas i andra programmeringsspråk för primitiva variabler.

Nedan följer ett exempel på hur konstanter används. Nyckelordet **const** används för att ange att en variabel är en konstant. PI kommer att vara en konstant som har värdet 3,14 och om programmeraren försöker ändra värdet under exekvering kommer detta att generera ett kompileringsfel. Konstanter måste alltid sättas vid definitionen och de är alltid statiska. Detta av optimeringsskäl.

Ex.
```
public class MinKlass
{
    public int tal = 12;
    private string namn;
    private static antInstanser;
    public const float PI = 3.14;
}
```

Det finns en sorts konstanter som sätts i runtime, dvs. under körning, och dessa kallas **readonly**. De kommer inte att användas i konstruktionslösningen.

Egenskaper (properties) är en programspråkskonstruktion som gör så kallade ”getters” och ”setters” onödiga. De möjliggör en säker åtkomst till en klass fält utan att behöva använda två metoder som kontrollerar användningen. Egenskaper har inbyggda getters och setters som gör att programmeraren kan sätta villkor på åtkomst och skrivning. Egenskaper används som ett publikt klassfält. Nedan följer ett exempel:

Ex.
```
using System;
public class MinKlass
{
    private int gradeScore;
    public int GradeScore
    {
        get
        {
            return gradeScore;
        }
        set
        {
            if (value < 0 || value > 100)
                gradeScore = -1;
            else
                gradeScore = value;
        }
    }

    public static void Main()
    {
        MinKlass obj = new MinKlass();
        obj.GradeScore = 101;
        Console.WriteLine(“GradeScore är {0}!”,obj.GradeScore);
        Console.WriteLine(”Tryck Enter för att avsluta.”);
        Console.ReadLine();
    }
}
```

Följande skrivs ut på skärmen:

```
GradeScore är -1!
Tryck Enter för att avsluta.
```

Egenskapen GradeScore i exemplet ovan kan bara sättas till ett värde mellan 0 och 100. Om någon istället skulle försöka att sätta den till ett felaktigt värde sätts den automatiskt till -1. För att komma åt värdet som GradeScore sätts med används nyckelordet **value** i set-

metoden. Som exemplet visar är GradeScore bara ett skal ovan gradeScore som håller det egentliga värdet. Enligt namnkonvention ska egenskapen ha samma namn som den privata variabeln, men börja med en versal.

Enumeratorer liknar sina föregångare i C++ och Java, men med vissa förbättringar. Enumeratorer kommer att användas i konstruktionslösningen, men på precis samma sätt som i ovan nämnda språk. En annan konstruktion som är ny för C# är indexerare. Indexerare kommer inte att användas i konstruktionslösningen, men kan beskrivas som en blandning av en matris, en egenskap och en metod.

Som tidigare nämnts, se kapitel 2.2.3.1, kan alla medlemmar inom en klass ha olika åtkomst. För mer information om de olika klassmedlemmarna, se [3][4][5].

2.2.3.4 Metoder

En metod är en operation som ett objekt kan utföra. Den kan jämföras med en funktion i ett imperativt språk, men har ett annat gränssnitt. En funktion kan man alltid kalla på, men en metods åtkomst styrs av dess åtkomstmodifierare. För att deklarera en metod, skrivs först den åtkomstmodifierare som metoden ska ha. Om denna utelämnas kommer ett standardvärde att sättas, se kapitel 2.2.3.1. Härnäst skrivs metodens returtyp följt av namnet på metoden och en parentes. Efter parentesen deklareras de parametrar metoden ska ta emot och listan avslutas sedan med ännu en parentes. Om metoden inte tar emot några parametrar så är parameterlistan tom. En metod kan, likt en funktion, ha flera inparametrar men bara en utparameter. Utparametern skickas tillbaka med nyckelordet **return** följt av den primitiva variabel, objekt eller värde som ska skickas tillbaka.

```
<åtkomst> <returtyp> <metodnamn>(<parameterlista>)
{
    //Metodskropp
    [return [variabel];]
}
```

I ett vanligt metodanrop skickas parametrarna som referenser, om de är objekt, och som värde om de är värdetyper, men det går även att skicka värdetyperna som referenser. Detta sker genom att man skriver nyckelordet **ref** framför den variabel som man vill skicka som referens, både i metoddefinitionen och i anropet till den.

Ex.
```
void makeDouble(ref int a)
{
    a *= 2;
}

...
makeDouble(ref x);
...
```

I exemplet ovan skickas en variabel in i en metod, som dubblerar dess värde, och x som skickades in i makeDouble() ändras automatiskt när variabeln a gör det.

Ibland kan antalet inparametrar till en metod variera, och för att det inte ska behöva skrivas en version för varje fall, finns det i C# ett enkelt sätt att komma runt detta problem. Sist i parameterlistan i metoddeklarationen skrivs nyckelordet **params** följt av en matristyp och ett variabelnamn. De inkommande parametrar som blir över, hamnar i denna matris, förutsatt att de är av rätt typ.

Ex.
```
//Metoddefinitionen
void Pappa(string namn, params string[] barn)
{
    string text = namn + " är pappa till:";
    foreach(string barnNamn in barn)
        text += " " + barnNamn;
    Console.WriteLine(text);
}

...
//Metodanrop
Pappa("Jan-Ove", "Lisa", "Kalle", "Anna", "Anton");
...
```

Om parametern efter **params** är av object-typ, kan vad som helst skickas in i metoden.

Två speciella metoder för en klass är konstruktorn och destruktorn. Konstruktorn är den metod som skapar objektet, medan destruktorn är den som förstör det. Destruktorn nämns för att den finns, men den bör oftast inte användas. Orsaken till detta är den automatiska minneshanteringen. Det går helt enkelt inte att veta när destruktorn kommer att exekveras, pga. att skräpsamlaren (garbage collector) kan vänta en obestämd tid innan den tar bort objekten i fråga (se [3] för mer information om skräpinsamling). Konstruktorn, däremot, fungerar på samma sätt som i C++ och Java.

Om man skriver nyckelordet **static** i en metoddeklaration, blir metoden statisk. Med statisk menas att metoden blir en klassmetod, till skillnad från en vanlig metod som är en objektmetod. Klassmetoder finns bara i en upplaga, och alla objekt anropar samma metod,

men objektmetoder finns i en upplaga för varje objekt. Det samma gäller för variabler. Statiska metoder kan komma åt statiska variabler, men ickestatiska metoder kan komma åt både ickestatiska och statiska variabler. För mer information om metoder, se [3][4][5].

2.2.3.5 Main-metoden

Main-metoden är en metod som alltid används för att starta ett program. Den är statisk, ty när ett program startas finns inga objekt så metoden kan inte vara annat än statisk.

Ex.
```
public class MinKlass
{
    public static Main(void)
    {
        System.Console.WriteLine(”Hello World!”);
    }
}
```

2.2.3.6 Arv

Arv är ett av objektorienteringens grundstenar; att kunna ärva egenskaper och metoder från en basklass. Syntaxen är lik den i C++, dvs.:

```
<åtkomst> [sealed] class <namn> : <basklass>
{ <klasskropp> }
```

Den enda skillnaden är att klasser i C++ inte har accessmodifierare eller nyckelordet **sealed**. Detta kommando anger att det inte går att ärva från klassen. Orsaken till varför det går att förhindra arv är på grund av säkerhets- och optimeringsskäl. Om kompilatorn vet att det inte går att ärva ytterligare en nivå, kan vissa optimeringar göras som annars är omöjliga.

En annan stor skillnad mot C++ är att det endast går att ärva från en klass. Det är alltså ett syntaktiskt fel att ärva från fler än en, detta för att förhindra t.ex. diamantarv och liknande problem. För att ändå behålla de bra sidorna med multipelt arv har gränssnitt anammats. Gränssnitt tas upp i ett senare kapitel, se 2.2.3.8. Nedan följer ett exempel på arv:

```
public abstract class Figure
{
    protected int x;
    protected int y;
    public abstract void draw();
}
```

```
public class Circle : Figure //Här ärver Circle från Figur
{
    //Klass som ärver alla egenskaper och metoder från Figur.
    //Denna klass innehåller variablerna x och y.
    protected int radie;
    public override void draw()
    {
        //Implementation för att rita cirklar
    }
}
```

En annan vanligt förekommande detalj som detta exempel tog upp är abstrakta klasser. Klassen Figure är abstrakt för att klassdeklarationen innehöll nyckelordet **abstract**. Abstrakta klasser går inte att instantiera, men de kan användas som basklasser att ärva ifrån. Detta för att det går att kräva att de klasser som ärver från basklassen måste implementera de metoder som angivits vara abstrakta. I exemplet ovan är metoden draw abstrakt. Om Circle, som ärver från Figure, inte hade deklarerat metoden draw med nyckelordet **override,** eller inte deklarerat den överhuvudtaget, hade ett kompileringsfel genererats. **override** är ett nyckelord som anger omdefiniering av en metod från en basklass. Den används också tillsammans med **virtual** för att genomföra polymorfism, se 2.2.3.7.

new är ett annat vanligt nyckelord som kan ses med metoder. Det anger att en basklassmetod ska, i klassen, gömmas bakom en metod med samma namn. För att komma åt en gömd metod måste **base** användas. Nedan följer ett kort exempel på hur **new** används.

```
public class BasKlass
{
    public void Draw()
    {
        ...
    }
}

public class Test : BasKlass
{
    new public void Draw()
    {
        // Ny metod som gömmer basklassens Draw()-metod
        base.Draw(); // Anropar Draw() från basklassen
    }
}
```

Se [3][4][5] för mer information om arv, omdefiniering, gömning och abstrakta klasser.

2.2.3.7 Polymorfism

Polymorfism är en vanlig teknik för att dynamiskt kunna välja vilken metod som ska exekveras under programkörning istället för under kompilering. Detta gäller i största fall för ärvda klasser som kan ha flera uppsättningar av metoder med samma namn. De metoder som ska stödja polymorfism måste i sin basklass ange detta med nyckelordet **virtual**. Nedan följer ett lite större exempel på hur polymorfism används. Märk väl hur man använder **override** för att omdefiniera den metod som i basklassen angavs som virtual.

Ex.
```
using System;
public class Figure
{
    public virtual void WriteType()
    {
        Console.WriteLine(“Jag är en figur.”);
    }
}

public class Circle : Figure
{
    public override void WriteType()
    {
        Console.WriteLine(”Jag är en cirkel.”);
    }
}

public class Square : Figure
{
    public override void WriteType()
    {
        Console.WriteLine(”Jag är en kvadrat.”);
    }
}

public class Test
{
    public static void Main()
    {
        Figure[] matris = new Figure[3];
        matris[0] = new Figure();
        matris[1] = new Circle();
        matris[2] = new Square();
        foreach(Figure figur in matris)
            figur.WriteType();
        Console.WriteLine(“\r\nTryck Enter för att avsluta.”);
        Console.ReadLine();
    }
}
```

Exemplet ovan har följande utskrift:

```
Jag är en figur.
Jag är en cirkel.
Jag är en kvadrat.

Tryck Enter för att avsluta.
```

Exemplet ovan visar ett enkelt exempel på hur man kan använda polymorfism. Både Circle och Square ärver från Figure, som har en virtuell metod. Denna omdefinieras i respektive klass till ett mer passande meddelande. Observera att det bara är virtuella metoder som kan vara polymorfa. Även om matris-objekten, i Main-metoden i exemplet ovan, är av Figure eller ärvda från densamma så anropas alltid rätt metod i **foreach**-satsen som utskriften visar. Om något om polymorfism är oklart finns det mer att läsa i [3][4][5].

2.2.3.8 Gränssnitt – interface

Som redan påpekats är multipelt arv inte tillåtet i C#. Det är inte tillåtet i Java heller. Så hur går det till om en klass vill ärva mer än från bara en klass? Det är här gränssnitt kommer in. Även om det inte går att ärva från mer än en klass går det att implementera hur många gränssnitt som helst. Ett gränssnitt tvingar den klass som ska använda gränssnittet att implementera de metoder som har anges. Man kan även ange annat, såsom t.ex. events, men det gås inte igenom här. Ett gränssnitt kan ärva från flera andra gränssnitt och konventionellt startar gränssnittsnamn med ett versalt i (i för interface). Ett gränssnitt har denna syntax:

```
<åtkomst> interface <namn> [: <gränssnitt> [, <gränssnitt>]*]
{
    <returtyp> <metodsnamn>(<parameterlista>);
}
```

När ett gränssnitt ska användas skrivs som när en klass ärvs. Efter basklassen, om någon finns, skrivs en lista av gränssnitt åtskiljda av kommatecken. Nedan följer ett exempel på hur gränssnitt och gränssnittsvariabler används.

```
using System;
namespace InterfaceTesting
{
    public interface IDrawable
    {
        void Draw();
    }
```

```
    public class Circle : IDrawable
    {
        public void Draw()
        {
            Console.WriteLine("O");
        }
    }

    public class LeftArrow : IDrawable
    {
        public void Draw()
        {
            Console.WriteLine("<-");
        }
    }

    public class Test
    {
        public static void Main()
        {
            IDrawable[] figurMatris = new IDrawable[2];
            figurMatris[0] = new Circle();
            figurMatris[1] = new LeftArrow();
            foreach(IDrawable figur in figurMatris)
                figur.Draw();
            Console.WriteLine("Tryck Enter för att avsluta.");
            Console.ReadLine();
        }
    }
}
```

Programmet skriver ut:

```
O
<-
Tryck Enter för att av avsluta
```

För mer information om gränssnitt, se [3][4][5].

2.2.4 Värdeklasser – structs

Värdeklasser är MS sätt att kunna använda primitiva datatyper, men ändå kunna kalla dem klasser. Detta för att få hela språket att bestå av klasser, men inte behöva offra effektivitet. Syntaxen liknar klassens, men en **struct** kan inte ärva från annat än gränssnitt och de kan inte agera basklass för en annan klass. Dock kommer alla datatyper automatiskt att ärva från klassen Object.

En **struct** kan inte ha en standardkonstruktor definierad, dvs. en konstruktor utan parametrar. Detta pga. att .NET implicit definierar den. Syntaxen följer nedan med en lista på vad en struct kan innehålla:

```
<åtkomst> struct <namn> [: <gränssnitt>[, <gränssnitt>]*]
{
    // Icke standardkonstruktorer
    // Fält
    // Egenskaper
    // Indexerare
    // Metoder
    // Händelser
    // Delegater
    // Enumeratorer
    // Inre värdeklasser
}
```

Det som får en värdeklass att kunna fungera som både en klass och som en vanlig datatyp är två egenskaper som kallas boxing och unboxing. Boxing konverterar en primitiv typ till en referenstyp. Unboxing gör motsatsen, det vill säga att konvertera från en referenstyp till en primitiv typ. All konvertering sker implicit så programmerare behöver inte tänka på hur det går till.

För mer information om värdeklasser, se [3][4][5].

2.2.5 Händelsehantering

Händelsehantering är ett kraftfullt begrepp som är en vanligt förekommande del av dagens grafiska användargränssnitt. Händelsehantering är dock inte alltid så lätt att realisera. När C# skapades tänkte utvecklarna på att göra det så effektivt och enkelt som möjligt. Nedan visas hur det fungerar och sedan kommer uppsatsen ytterligare att ta upp händelsehantering i samband med MVC i kapitel 3.

2.2.5.1 Delegater – delegates

Delegater är typsäkra funktionspekare. De kan inte som i C++ peka på vilken metod som helst, utan bara på metoder som har samma struktur som delegaten, dvs. samma returtyp och parameterlista. Delegater skapas med nyckelordet **delegate** följt av en returtyp, namnet på delegaten och en parentes med en parameterlista.

Ex. `delegate string del(string x, string y);`

Delegater deklareras på samma sätt som vanliga klassobjekt. Nedan följer ett exempel på hur ovanstående delegat används tillsammans med en metod.

Ex.
```
using System;
//Delegaten som kan peka på metoder som tar in två strängar och
//returnerar en sträng.
delegate string del(string x, string y);

public class DelegatExempel
{
    public del methodPointer; //Deklaration av delegatobjektet.

    //En metod som adderar två strängar.
    public string addString(string a, string b)
    {
        return a + b;
    }

    public static void Main()
    {
        DelegatExempel ex = new DelegatExempel();

        //Skapa delegatobjektet och ange en metod att peka på.
        ex.methodPointer = new del(ex.addString);

        //Använda delegatobjektet för att peka på addString().
        string str = ex.methodPointer("Hello", "World!");
        Console.WriteLine(str);
    }
}
```

I exemplet skapas en delegat som sätts att peka på en metod. När objektet av delegaten används, används den metod man pekar på, och i detta fall är det metoden addString().

2.2.5.2 Händelser – events

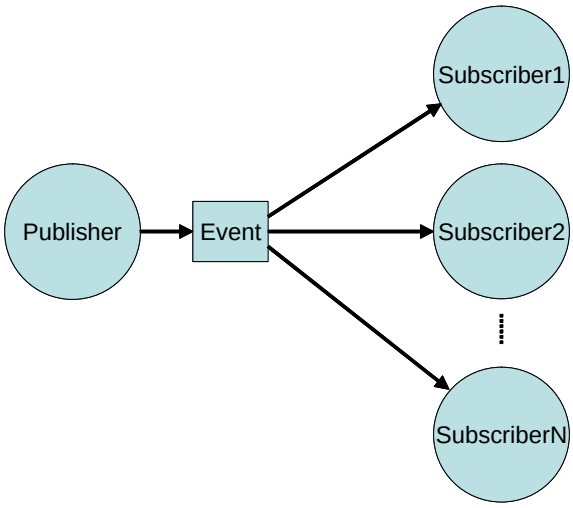
Största meningen med delegater är att agera som gränssnitt till händelser (events). Händelser är helt enkelt speciella delegater, och används för att låta objekt veta att något har hänt. Objekten i en applikation låter en eller flera av sina metoder prenumerera på en händelse, och dessa metoder anropas när händelsen utlöses. Rent tekniskt är varje händelse en lista med metoder som ska anropas när händelsen anropas. Skapandet av händelser har följande syntax:

```
[åtkomst] event <typ> <namn>;
```

Alla händelser baseras på delegater, och därför måste typen vara en delegattyp.

Ex.
```
using System;

//Först skapas en delegat...
delegate void del(string x);

class HändelseExempel
{
    //...därefter skapas en händelse av delegattypen
    public event del ev1;

    //Skapa en metod för att ta emot händelsen
    public void callback(string str)
    {
        Console.WriteLine(str);
    }

    //I Mainmetoden anges att metoden "callback" ska prenumerera
    //på händelsen, och händelsen anropas med en sträng.
    public static void Main()
    {
        HändelseExempel he = new HändelseExempel();
        //callback-metoden anges som prenumerant.
        he.ev1 += new del(he.callback);
        he.ev1("Hello World!"); //här utlöses händelsen.
    }
}
```

I exemplet ovan skapas en delegat och en händelse som baseras på den skapade delegaten. Som visas i exemplet används ”+=” för att prenumerera på en händelse. Det som händer är att en delegat läggs till i en prenumerantlista som används när händelsen utlöses. Logiskt nog används ”-=” för att avboka en prenumeration. I och med att en lista används, kan flera metoder prenumerera på en händelse. Samma delegattyp som händelsen är av måste användas när man lägger till en prenumeration.



Figur 5 – Publisher/Subscriber-modellen

Figur 5 visar att flera prenumeranter kan ta del av samma händelse i C#. Prenumeranterna kan vara flera olika objekt, det behöver inte vara samma. För mer information om delegater och händelser, se [3][5].

2.2.6 Grafiska användargränssnitt

Grafiska användargränssnitt byggs i C# upp av komponenter från namnrymden System.Windows.Forms. Huvudkomponenten är formuläret som utgör själva fönstret. För att skapa ett tomt fönster skapas först en klass som ärver från System.Windows.Forms.Form. Sedan öppnas fönstret med hjälp av kommandot Application.Run() som körs i Main-metoden. Argumentet till Application.Run() är ett objekt av den klass som skapats.

Ex.
```
using System;
using System.Windows.Forms;
class MyFirstWindow : Form
{
    public static void Main()
    {
        MyFirstWindow window = new MyFirstWindow();
        window.Height = 100;
        window.Width = 300;
        Application.Run(window);
    }
}
```

Andra komponenter läggs till som klassvariabler.

Ex.
```
using System;
using System.Windows.Forms;
class MyFirstWindow : Form
{
    Button button;

    MyFirstWindow()
    {
        Height = 100;
        Width = 300;
        button = new Button();
        Controls.AddRange(new Control[] {this.button});
    }

    public static void Main()
    {
        MyFirstWindow window = new MyFirstWindow();
        Application.Run(window);
    }
}
```

I exemplet ovan har även en konstruktor lagts till, och koden i denna lägger till den nya komponenten till formuläret. Detta måste göras för att komponenten ska visas grafiskt. När detta görs fungerar formuläret som en behållare för grafiska komponenter, och faktum är att alla komponenter kan användas som sådana. Det gör det möjligt att bland annat lägga en knapp i en knapp.

Figur 6 – Ett första fönster

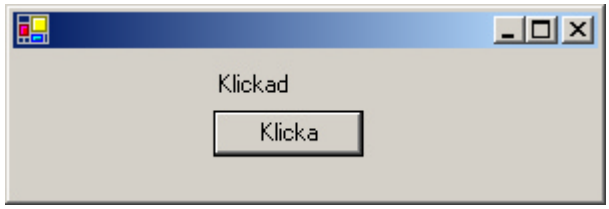
Koden i exemplet ovan ritar upp ett fönster med en knapp som Figur 6 visar, men knappen har ingen text och den är placerad uppe i vänster hörn i fönstret. Knappens text ändras enkelt med egenskapen Text för knappen, men för att ändra positionen på måste knappens Location-egenskap sättas med ett System.Drawing.Point-objekt med de önskade koordinaterna.

Ex.
```
using System;
using System.Windows.Forms;
using System.Drawing;

class MyFirstWindow : Form
{
    Button button;

    MyFirstWindow()
    {
        Height = 100;
        Width = 300;
        button = new Button();
        button.Text = "Klicka";
        button.Location = new Point(100, 30);
        this.Controls.AddRange(new Control[]{this.button});
    }

    public static void Main()
    {
        Application.Run(new MyFirstWindow());
    }
}
```

Figur 7 – Ett förbättrat fönster

Koden i exemplet ovan resulterar i ett fönster med en knapp i, se Figur 7. På knappen står texten "Klicka", men inget händer när den blir klickad på. För att få något att hända används händelsehantering, som förklarades i föregående avsnitt. Alla grafiska objekt har färdiga händelser, allt ifrån klickning till att känna av när musen är över objektet. Det finns även färdiga delegater som talar om vilken typ av metod som ska ta emot händelsen. Allt som behövs är en metod som tar emot och behandlar händelsen, och en länk från händelsen till den metoden. I exemplet nedan kopplas Click-händelsen för knappen till en metod som skriver ut texten "Klickad" i en etikett (eng. label), se Figur 8.

Ex.
```
using System;
using System.Windows.Forms;
using System.Drawing;
class MyFirstWindow : Form
{
    Button button;
    Label label;
    MyFirstWindow()
    {
        Height = 100;
        Width = 300;
        button = new Button();
        button.Text = "Klicka";
        button.Location = new Point(100, 30);
        button.Click += new EventHandler(button_clicked);
        label = new Label();
        label.Location = new Point(100, 10);
        this.Controls.AddRange(new Control[]{this.button,
                                             this.label});
    }

    public void button_clicked(object sender, EventArgs e)
    {
        label.Text = "Klickad";
    }

    public static void Main()
    {
        Application.Run(new MyFirstWindow());
    }
}
```



Figur 8 – Fönster med en etikett

I exemplet ovan kopplas knappens Click-händelse till metoden button_clicked(), som har den struktur som EventHandler-delegaten kräver. EventHandler är en delegat som är definierad i namnrymden System. Den är en standarddelegat som används för händelser som inte skickar någon speciell data. Parametrarna till metoden är ett objekt, som representerar sändaren av händelsen, och ett händelseargumentobjekt. Just EventArgs har ingen data, och det behövs inte eftersom Click-händelsen från knappen inte skickar någon data. Vid de tillfällen data skickas finns det händelseargument som tar emot den. Sändarobjektet kan konverteras från typen object till den typ sändaren är, i detta fall Button, för att få tillgång till information om objektet.

Placering av objekt och koppling mellan händelser och metoder sker automatiskt om Visual Studio .NET används. Då räcker det att dra ut komponenter på formuläret, sätta deras egenskaper i ett fönster och dubbelklicka på den händelse som ska kopplas.

2.2.6.1 Komponenter för laborationen

Det finns många olika komponenter i C# som kan användas direkt ifrån gränssnittet i MS Visual Studio .NET. Exempel på vanligt använda komponenter är knappar, etiketter, menyer och mycket annat.

Att skapa egna gränssnittskomponenter i C# kräver väldigt lite programmering. I huvudsak krävs att komponentklassen ärver från System.Windows.Forms.UserControl, och att OnPaint-metoden omdefinieras till att rita upp det visuella i komponenten. Metoden tar in en parameter i form av ett System.Windows.Forms.PaintEventArgs-objekt. Ur detta hämtas själva ritytan som ett System.Drawing.Graphics-objekt. Detta objekt har i sin tur metoder för att rita.

I bilaga A återfinns kod för en komponent som fungerar som en hastighetsmätare på en bil och klassen heter Dial. Den är tänkt att visa gaspådrag i konstruktionslösningen. Det enda som behövs för att styra den är att sätta egenskapen Value till ett värde mellan 0 och 100 för att uppdatera armen på mätaren.

2.2.7 XML-dokumentation

I och med C# har Microsoft givit programmerare ett enkelt sätt att dokumentera sin kod och publicera dokumentationen. Med XML får olika programmerare samma struktur på dokumentationen, och den följer samma standard, vem som än skrivit den. Detta kan liknas med SUN:s Javadoc, som följer med Javas Software Development Kit (SDK).

För att skapa en XML-fil kompileras C#-filen med tilläggsparametern /doc: följt av namnet på den XML-fil som ska skapas från kommentarerna i koden.

Ex. `csc KodFil.cs /doc:XMLFil.xml`

Kompilatorn klarar alla taggar som är korrekta enligt XML-standarden, men ett antal fördefinierade taggar med inbyggd funktionalitet har skapats. Dessa har ett namn som lätt kan kopplas till deras betydelse. För en summering av ett stycke kod kan

```
<summary> ”summering” </summary>
```

användas, och för att beskriva en parameter till en metod skrivs

```
<param name=”namnet på parametern”>”beskrivande text”</param>
```

För att kompilatorn ska veta vad som är XML-kommentarer och vad som är vanliga kommentarer, inleds XML-kommentarerna med tre snedstreck istället för de vanliga två. Kompilatorn vet då vilka kommentarer som är XML och dessa kan med en stilmall (eng. style sheet) transformeras till HTML-sidor om så önskas. Det är bara att klicka fram den information man söker om programmet, förutsatt att programmeraren skrivit den.

För en lista över de fördefinierade XML-taggarna med exempel på hur de fungerar, se bilaga B.

2.2.8 En jämförelse mellan C#, C++ och Java

Det är svårt att göra en rättvis jämförelse mellan tre av dagens vanligaste förekommande språk. De är snarlika. Alla tre är objektorienterade, imperativa språk, som härstammar från C.

Den största strukturella likheten är att språken är objektorienterade, även om C# utger sig för att vara helt objektorienterat, till skillnad från C++ och Java som använder primitiva datatyper. Dock är den mest synbara likheten att de tre språken har nästan identisk syntax och konstruktioner, även om C# har lagt till **foreach** och, precis som Java, eliminerat den största

orsaken till programmeringsfel; pekare. C# har också lagt till indexerare och egenskaper (eng. properties) vilket är helt nya begrepp. Anders Hejlsberg och Scott Wiltamuth har försökt att ta de bästa språkkonstruktionerna från C/C++ och Java. Resultatet är ett en lättläst språk med intuitiva konstruktioner som känns igen från andra språk.

En viktig skillnad mellan språken är dock att C# är starkare typat än både C++ och Java och programmeraren behöver mer explicit uttrycka vad som skall göras. I Java går det att omdefiniera eller gömma metoder utan att vara medveten om det, medan samma försök i C# genererar kompileringsfel. Om ett mer explicit språk är bra eller dåligt beror mest på vad programmeraren anser. Vissa anser att det bara blir mer att skriva, medan andra tycker om att det blir uttrycklig kod som är svårare att misstolka av läsare till koden. Explicita konstruktioner är självdokumenterande, vilket kan vara till hjälp för andra.

Det är mycket arbete med att skapa ett program i C++ för att det är på en så låg abstraktionsnivå. Just det ville MS komma ifrån när C# skapades. Ett av designmålen med C# var RAD (Rapid Application Development). Det ska gå att skapa Internetapplikationer på kort tid och de ska vara lätta att söka igenom efter fel. Just detta har MS lyckats ganska bra med, ty MS Visual Studio .NET har många funktioner som snabbar upp utveckling och ett lättsamt grafiskt IDE (Integrated Development Environment).

Effektivitetsdiskussionen är ett propagandakrig på Internet. Den ena sidan, med MS i spetsen, anser att C# är effektivare än Java. Den andra sidan, bestående av vissa C++- och Java-programmerare och Sun anser att C# är mycket ineffektivt jämfört med C++ och Java. Vilken sida som har rätt är inte av betydelse här. Istället får läsaren komma till egna slutsatser.

För mer information om språken och effektivitetsaspekterna, sök information på Internet eller se [12][13][14][15][16].

3 Beskrivning av konstruktionslösning

Nedan kommer olika tekniker att beskrivas som kan användas för skapa konstruktionslösningen i kapitel 4. Uppsatsen kommer att belysa fördelar och nackdelar med dessa tekniker och förklara varför vissa har förkastats till förmån för andra. Nedan följer även en beskrivning av problemet.

Den första tekniken är MVC (Model View Controller), vilket är ett sätt att programmera gränssnitt. Denna teknik ger programmeraren möjlighet att separera programlogik, gränssnittskomponenter och kontrollstruktur till separata delar i en applikation. MVC kommer att användas för att separera modellen (programlogiken) till en distribuerad dator.

För att genomföra kommunikation mellan två eller flera datorer behövs en mellanvara (eng. middleware) som sköter paketerandet av datan (eng. marshalling) och skickar den. De typer av mellanvara som finns tillgängliga är webbtjänster (eng. web services) och remoting och beskrivs nedan.

3.1 Problembeskrivning

Problemet består i att skapa en modell av en rymdfärja, som ska styras via ett grafiskt gränssnitt. Modellen ska simulera en uppskjutning av rymdfärjan. Lösningen ska dessutom vara distribuerad, dvs. modellen ska kunna befinna sig på en dator, medan det grafiska användargränssnittet finns på en annan. Det ska även vara möjligt att ansluta flera gränssnitt till samma modell.

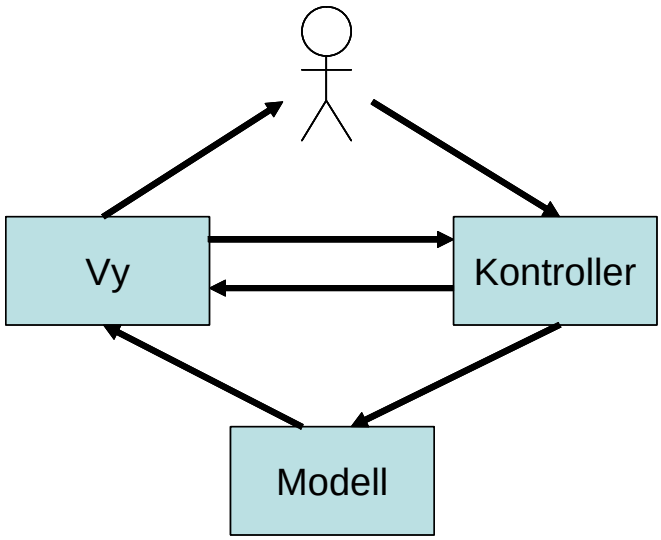
Uppskjutningen är indelad i sex faser där den första fasen är IDLE, vilket innebär att färjan står på marken och dess motorer inte är påslagna. När motorerna slås på, men färjan ännu inte är lossad från sin hållare, går den i in fasen LAUNCHING. När hållaren lossas flyger färjan iväg och den går in i nästa fas som är LAUNCHED. Därefter passerar färjan ett antal avstånd till jorden då nya faser tar över. Nästa fas på tur är ROLL_LIKE_A_CROCODILE, i vilken färjan ska rulla runt sin egen längsgående axel, och efter den kommer RELEASE_BOOSTERS, då färjan ska släppa de hjälpraketer som använts vid starten. Till sist kommer rymdfärjan in i fasen IN_SPACE, vilket indikerar att färjan är ute i rymden. Då ska motorn stängas av och färjans rymdraketer sättas på istället.

Om operatören missar att utföra dessa handlingar utlöses ett larm för varje händelse. Sammanlagt finns det nio larm som kan utlösas. Ett av dessa talar till exempel om att mer kraft måste ges till motorerna, medan ett annat talar om att de är överhettade.

För att testa modellen ingår det även i uppgiften att skapa ett grafiskt gränssnitt som återspeglar modellens tillstånd. Via detta gränssnitt ska det även vara möjligt att manipulera modellens tillstånd.

3.2 Redogörelse av MVC

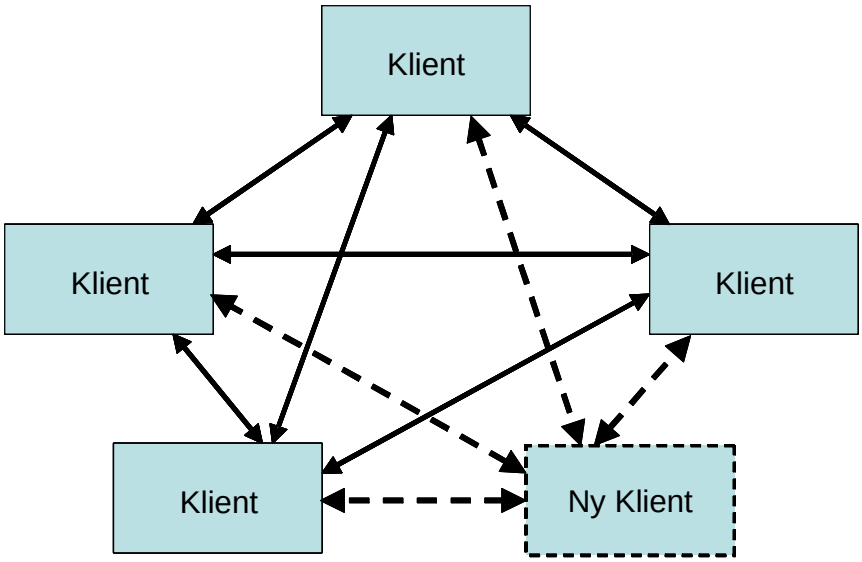
MVC är en förkortning av Model View Controller, och innebär att man delar upp en applikation i tre huvuddelar. Modellen, som innehåller logiken i programmet, kontrollern, som styr modellen och vyn, som visar informationen.



Figur 9 – MVC-konceptet

Figuren ovan visar en fjärde del, användaren, som normalt sett inte räknas in i MVC, men som lagts till för att tydligare visa hur konceptet fungerar.

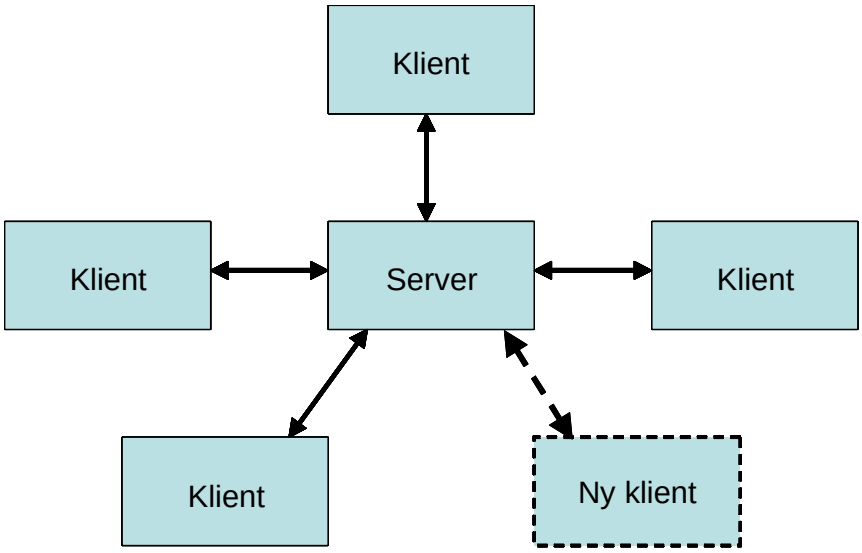
Denna uppdelning görs för att få ett mer modulärt program, och förenkla ändringar. Logiken kan ändras utan att någon annan del påverkas, GUI:t (Graphical User Interface) kan ändras och göras mer användarvänligt och så vidare. Ta ett program för att skicka textmeddelanden mellan flera datorer, ett s.k. ”chatprogram” som exempel. Utan MVC skulle varje deltagande klient behöva känna till alla andra klienter, och om en ny klient tillkommer måste den läggas till på alla klienter.



Figur 10 – Vanlig lösning utan MVC

Figur 10 visar att varje klient måste ha funktionalitet motsvarande kontroller, vy och modell i sig, och det är tydligt hur komplicerat det blir utan MVC. Skulle ännu en klient vilja ansluta sig måste varje annan klient behöva uppdateras om den nya noden för att kommunikation ska vara möjlig.

Med MVC räcker det att varje nod känner till modellen, och den i sin tur håller koll på vilka noder som deltar. Nya noder registreras hos modellen, och de andra noderna behöver inte uppdatera någonting.



Figur 11 – MVC-lösning

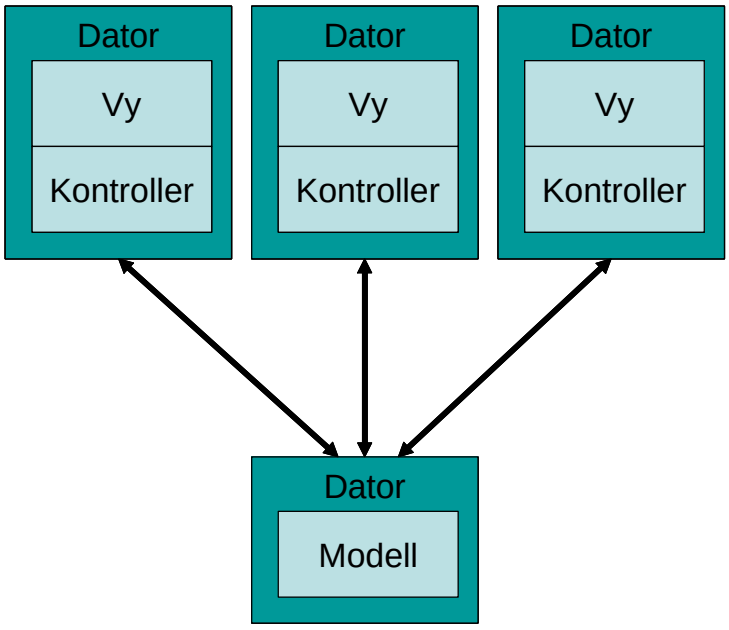
I Figur 11 ovan agerar klienterna både kontrollers och vyer, och servern är modellen. När en ny klient ansluter behöver endast servern uppdatera sig, för att den nya deltagaren ska kunna vara med i kommunikationen.

Den som är spindeln i nätet i systemet är modellen. Den tar emot signaler i form av händelser både utifrån och från sig själv, exempelvis från en timer, och behandlar dem. Till exempel kan en modell få i uppdrag att hämta data från en databas och returnera den. Resultatet skickas till intresserade vyer, och modellen måste hålla reda på vilka vyer som ska få informationen. Logiken arbetar oberoende av vyerna, och själva kärnan i modellen känner inte till dem. Signalerna utifrån kan komma både från kontrollers och andra modeller.

Vyerna är de som hanterar utdatan från modellen. En vy känner till modellen men den känner inte till andra vyer.

En kontroller är den del som hanterar indata från användaren. Kontrollern anropar sedan modellen som reagerar därefter. Tanken bakom MVC är att kontrollerna och vyerna ska vara åtskilda och att det inte ska ske någon kommunikation mellan dem, men i vissa fall kan det krävas att en kontroller styr en vy direkt, till exempel för att göra en GUI-inställning. Det är dessutom vanligt att kontrollern och vyn är kombinerad. En radioknapp till exempel, visar ju information samtidigt som den möjliggör ett val.

En fördel med MVC är att det stödjer distribuerade applikationer, där modellen ligger på en dator, och kontrollern och vyn ligger på var sin dator. Det kan till och med finnas flera datorer med kontroller och vyer, som Figur 12 visar.



Figur 12 – MVC är distribuerbart

För mer information om MVC, se [20].

3.3 Kommunikation med distribuerade objekt

Kommunikation med distribuerade objekt realiseras i huvudsak på två olika sätt i .NET, med webbtjänster (eng. web services) och med s.k. remoting. Det som i huvudsak skiljer de båda metoder åt är komplexiteten. Webbtjänster är den lite enklare varianten att kommunicera mellan datorer, där stora delar av koden genereras automatiskt och programmeraren behöver inte tänka så mycket hur kommunikationen fungerar. Remoting jobbar på en lägre abstraktionsnivå, vilket både medför större kontroll, men även mer komplexitet. Nedan förklaras de olika teknikerna och varför remoting valdes framför webbtjänster i konstruktionslösningen. Vidare sker en redogörelse för hur remoting används i konstruktionslösningen.

3.3.1 Webbtjänster – Web Services

Webbtjänster (eng. web services) är ett sätt att återanvända logik redan skriven, istället för att behöva skriva ny. Den använda logiken kan vara skriven av något företag, som MS, eller andra programmerare. Det positiva är att denna logik kan återanvändas i nya program. Det fungerar genom att programmeraren skickar ett meddelande till webbtjänsten och får svar tillbaka, ungefär som ett RPC (Remote Procedure Call). Detta sker utan att användaren till programmet behöver vara medveten om det.

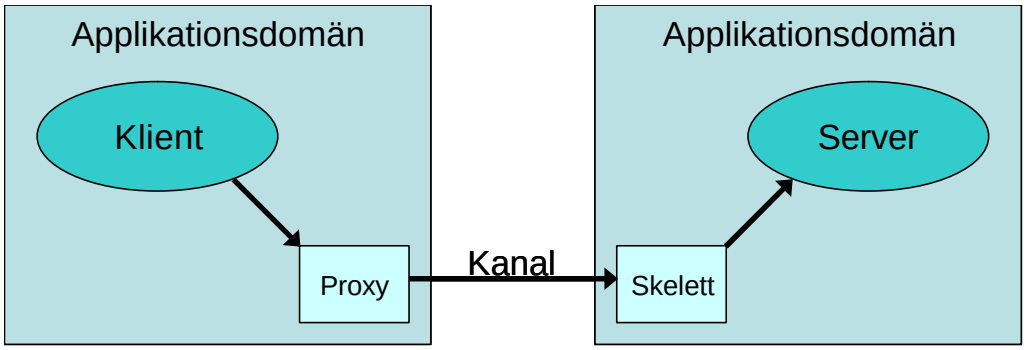
Ett möjligt scenario skulle kunna vara att skriva en texteditor. Ett möjligt problem skulle då kunna vara att implementera en stavningskontroll. Ponera då att en programmerare någonstans i världen redan skapat en stavningskontroll som en webbtjänst. Denna tjänst kan användas direkt i texteditorn utan att användaren är medveten om det och så länge en aktiv Internet-uppkoppling finns att tillgå. Detta pga. att ett SOAP (Simple Object Access Protocol)-meddelande måste skickas till tjänsten för att använda den.

Eftersom en webbtjänst i grund och botten fungerar likt ett RPC, kunde inte denna teknik användas i konstruktionslösningen. Lösningen är en MVC-konstruktion där logiken finns i servern. Servern måste, oberoende av klienten, kunna skicka ut händelser när viktiga förändringar sker. Detta är inte möjligt med endast RPC, ty med RPC kan servern endast returnera svar på klienternas anrop.

För mer information om webbtjänster, se [3][5][17].

3.3.2 Remoting

Precis som tidigare nämnts används remoting för att kommunicera mellan två eller flera datorer. Den största skillnaden mellan att använda web services och remoting är att i det senare fallet måste mer information om uppkopplingen specificeras. Figur 13 visar hur remoting kan användas. Detta kan liknas vid ett RPC-anrop. Remoting kan dock konfigureras så att kommunikation kan gå i båda riktningarna.



Figur 13 – Vanlig typ av remoting

Den kommunikation som skickas mellan de två, eller flera, applikationerna skickas över en kanal (eng. channel). Kanalen specificerar vilken typ av protokoll som informationsutbytet använder. Oftast används TCP (Transmission Control Protocol) istället för HTTP (HyperText Tranfer Protocol) som transportprotokoll, pga. snabbare kommunikation. Detta för att med TCP skickas datan binärt, medan det med HTTP skickas i klartext. HTTP kan alltså vara en fördel, om servern i fråga ligger bakom en brandvägg. Brandväggar är vanligtvis konfigurerade att släppa igenom HTTP-paket. Oavsett vilket protokoll som används är meddelandena är alltid skrivna i SOAP. Någon närmare beskrivning av hur SOAP-protokollet fungerar kommer inte att ges i denna uppsats. Orsaken till detta är att hanteringen av SOAP-meddelandena sker automatiskt av remotingen.

Proxy och skelett, se figuren ovan, är de delar av mellanvaran som sköter sändning och mottagning av meddelanden mellan kommunicerande parter. Dessa delar brukar skapas automatiskt i C#, men kan dock skapas manuellt genom att ärva klassen RealProxy. För mer information om remoting än vad som följer, se [3][5][18][19] eller besök MSDN[2] (MS Developer Network).

[2] http://msdn.microsoft.com

3.3.2.1 Konfiguration av remoting

Det finns två sätt att konfigurera remoting. Det första varianten använder en fil med konfigurationsinformation. Den andra skriver in informationen direkt i kod. Problemet med den senare är att om informationen ändras, måste programmet kompileras om. I konstruktionslösningen används konfigurationsfiler.

Konfigurationsfiler är skrivna i XML och är inte så svåra att förstå. Servrar använder sig av service-taggen för att ange sin URI (Uniform Resource Identifier) och av vilken typ anropen ska vara. Det finns två typer av distribuerade objekt; SingleCall och Singleton. Om SingleCall används skapas ett nytt objekt för varje nytt anrop, medan Singleton endast skapar ett objekt. I konstruktionslösningen används Singleton-typen, ty annars går data mellan anrop förlorade. Klienter behöver ange URL:n (Uniform Resource Locator) till servern, samt vilken typ av kanal som ska användas. För exempel på hur konfigurationsfiler kan se ut, se nästföljande kapitel.

3.3.2.2 Remotingexempel

I bilaga C följer ett exempel på hur ett ”Hello World!”-program skulle kunna se ut i C# med hjälp av remoting. Koden i bilagan är kommenterad för att beskriva vad som händer. Nedan följer en detaljerad beskrivning av hur klasserna fungerar.

HostedServer är den klass som tar emot distribuerade anrop. I detta fall endast getServerResponse(), som returnerar strängen ”Hello World!”. Alla klasser som ska kunna refereras via ett nätverk måste ärva från klassen MarshalByRefObject. Om en programmerare försöker nå ett objekt som inte har ärvt från MarshalByRefObject så kommer en kopia att användas istället. MarshalByRefObject skapar alltså automatiskt ett proxy från klienten för att nå det distribuerade objektet istället för att ladda ner en kopia av objektet.

I konfigurationsfilen för HostedServer, web.config, anges URI:n för applikationen. Denna används av klienten för att hitta rätt process på serverns adress. HostedServer kompileras som en DLL (Dynamic Link Library), och då behövs inte en Main-metod anges. Konfigurationsfilen kan ha namnet web.config och laddas in automatiskt när den behövs. För att kompilera HostedServer.cs skrivs:

```
csc /t:library HostedServer.cs
```

Välj att skapa ett DLL-projekt om MS Visual .NET används.

Klienten, Client-objektet, börjar med att konfigurera remoting med hjälp av den statiska metoden Configure från RemotingConfiguration. För att få en referens till HostedServer-objektet används den statiska metoden GetObject hos klassen Activator. GetObject tar två parametrar, ett Typeobjekt av HostedServer (returneras från **typeof**-operatorn) och URL:n till datorn där servern finns. Det är GetObjekt-metoden som skapar proxyt till det distribuerade objektet. När nu en referens till HostedServer-objektet har skapats kan metoden getServerResponse() användas. I Client skrivs den returnerade strängen ut, dvs. ”Hello World!”. I konfigurationsfilen anges URL:n till servern och i taggen wellknown anges även URI:n. I taggen channels anges att trafiken ska gå över en HTTP-förbindelse. Skriv

```
csc /r:<sökväg till HostedServer>\HostedServer.dll Client.cs
```

för att kompilera Client.cs.

Det enda som RemotingHost gör är att fungera som hjälp för klienten att hitta servern. Detta gör att programmeraren slipper låta remotingen gå via en IIS (Internet Information Server) som annars är ett krav. RemotingHost konfigurerar sin remoting och väntar sedan på en användarinmatning. Detta för att RemotingHost måste vara igång när klienten försöker få en referens till HostedServer. Konfigurationsfilen ser likadan ut som för HostedServer, med ett litet undantag; det finns en channels-tagg som meddelar att trafiken ska gå via HTTP. För att kompilera RemotingHost-klassen, skriv:

```
csc RemotingHost.cs
```

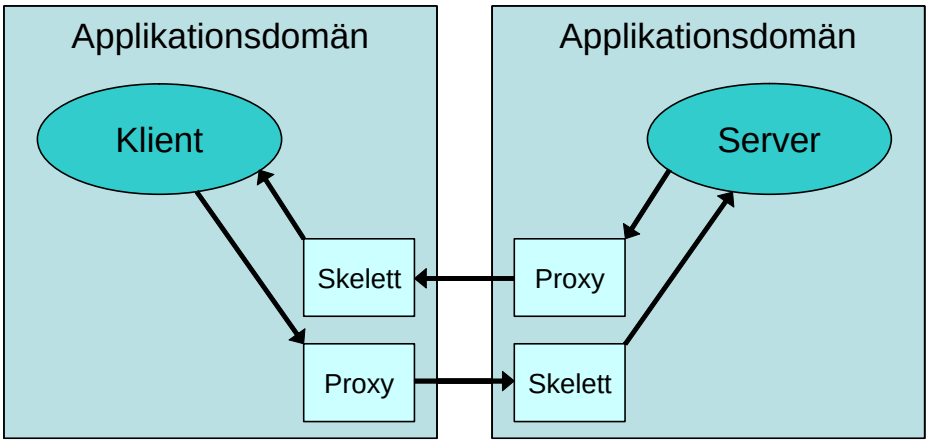
3.3.2.3 Hantering av livstid för distribuerade objekt

När distribuerade objekt används är livstidshanteringen lite speciell. För att förhindra att distribuerade objekt tas bort av skräpsamlaren inom en angiven tidsrymd, måste ett hyreskontrakt (eng. lease) upprättas. Detta görs med en lease manager. I konstruktionslösningen används en variant av livstidshanteringen som ser till att det distribuerade objektet inte tar bort sig själv. För att göra det omdefinieras metoden InitializeLifetimeService(), som nedärvts från klassen MarshalByRefObject, till att endast returnera NULL. Detta förhindrar skräpinsamlaren från att automatiskt ta bort objektet i fråga.

3.3.3 Remoting i konstruktionslösningen

Remoting i konstruktionslösningen är lite annorlunda mot exemplet i kapitel 3.3.2.2. Orsaken till detta är att både server- och klientobjektet ska kunna skicka meddelanden oberoende av

varandra. Klienten måste skicka signaler för att ändra på modellen, och den måste i sin tur kunna meddela viktiga förändringar till klienten. Detta innebär att remoting i konstruktionslösningen måste se ut som Figur 14 visar.



Figur 14 – Remoting i konstruktionslösningen

För att få kommunikation att fungera, som bilden ovan visar, måste både klient- och serverklassen ärva från MarshalByRefObject. Klienten i raketlaborationen använder ett grafiskt användargränssnitt och ärver därför från System.Windows.Forms.Form. Alla formulär (eng. forms) i .NET ärver från MarshalByRefObject och därför ärver klienten automatiskt det som ligger till grund för att få kommunikationen mellan klient och server att fungera korrekt.

Klienten i raketlaborationen tar kontakt med servermodellen på liknande sätt som i remotingexemplet ovan, men hur modellen får reda vilka klienter som är uppkopplade sker på ett helt annorlunda sätt; med hjälp av händelser. När klienterna meddelar prenumeration på serverns händelser med += operatorn (se kapitel 2.2.5.2) genereras ett proxy på serversidan för att kunna skicka dessa händelser. När sedan -= används för att ta bort prenumerationen, kommer förbindelsen att brytas.

Precis som i tidigare nämnt remotingexempel måste en så kallad host utility finnas tillgänglig för att få klienten att kunna hitta servern. Annars måste en IIS användas vilket kan försvåra raketlaborationen för personal och studenter. Laborationen är gjord för en kurs i grafiska användargränssnitt, och därför ska programmeringen vara så enkel som möjligt. Remoting kan implementeras med hjälp av abstrakta klasser eller gränssnitt. Dessa används för att de kommunicerande parterna skall kunna känna till varandras publika gränssnitt. Detta är dock mycket komplicerat att implementera och för att studenterna ska slippa detta valdes

en annan lösning. Den kräver att en version av klienten existerar hos servern och vice versa. Detta gör att de respektive delarna automatiskt skapar proxy och skelett på vardera sida.

4 Implementation och test

Nedan följer en redogörelse för implementationslösningen för klient och serverdel. Implementationen är baserad på tekniken remoting och konstruktionsmönstret MVC enligt tidigare konstruktionsbeskrivning.

Efter separata beskrivningar undersöks modell och gränssnitt tillsammans för att visa interaktion dem emellan. Efter detta kommer en kort förklaring hur MVC är implementerat tillsammans med en kort diskussion om hur MVC bäst kan användas. Sist beskrivs den typ av testning som har gjorts.

4.1 Serverlösning

Modellen består av ett antal mekanismer. Vissa av dem reagerar på interna händelser medan andra är interaktiva. Först kommer de interna mekanismerna att förklaras och sedan de interaktiva. Eftersom många av de interaktiva mekanismerna fungerar på samma sätt, kommer dessa att sammanfattas i ett gemensamt stycke.

4.1.1 Interna mekanismer

Det mesta av uppskjutningen av rymdfärjan kretsar kring dess avstånd till jorden. Detta avstånd lagras internt som en heltalsvariabel. Avståndet styrs av en timer, och varje gång timern utlöser ökas avståndet med ett fixt värde. För att öka avståndet snabbare minskas således uppdateringsintervallet på timern, och detta görs genom att öka kraften på raketerna.

Vid varje uppdatering av avståndet skickas dess värde till de klienter som är uppkopplade mot modellen med hjälp av en händelse. Denna händelse är en av många i modellen, se bilaga D.2. All kommunikation som går från modellen till klienterna sker via händelser. Händelserna baseras på tre delegater där en har ett phase-objekt, som beskriver aktuell fas enligt Figur 15, som parameter. Den andra har ett heltal och den tredje saknar parametrar. Alla saknar returvärde. Delegaterna nedan återfinns även i bilaga D.1.

```
public delegate void shuttlePhaseDelegate(phase shuttlePhase)
public delegate void shuttleIntDelegate(int val)
public delegate void shuttleNoParamDelegate()
```

Uppskjutningen är indelad i ett antal faser, se Figur 15, och beroende på vad avståndet till jorden är, skickas olika faser ut med en händelse. Vid varje fasbyte kräver modellen att något görs i klienterna, och om detta inte görs skickas larmhändelser. Det finns en händelse för varje larm. För varje rutin modellen kräver att klienten uträttar finns även en boolesk variabel, som håller reda på om det är gjort eller inte. Larmhändelserna skickas tills motsvarande booleska variabel är sann.

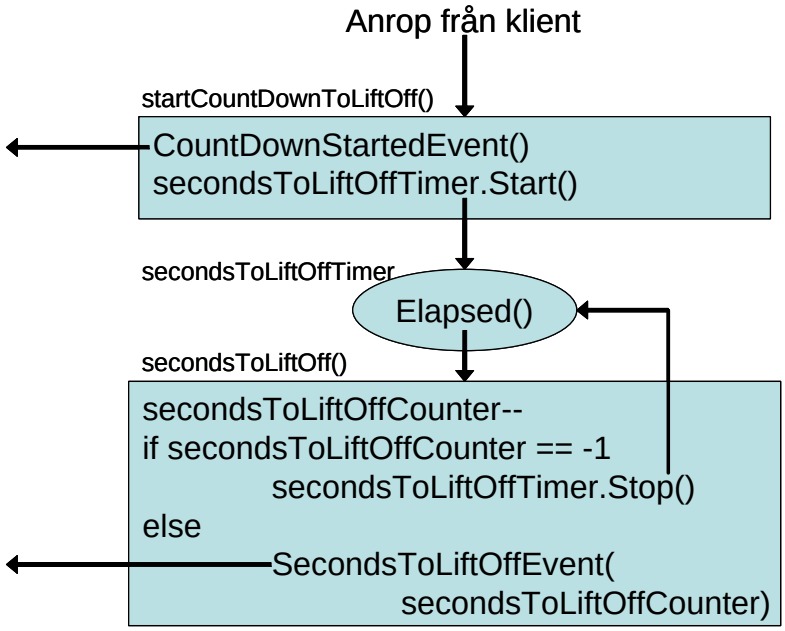
- IDLE – Färjan står på marken utan kraft i rakterna.
- LAUNCHING – Färjan står på marken men med kraft i raketerna.
- LAUNCHED – Färjan har nu lämnat marken.
- ROLLING – Färjan ska nu börja rulla runt sin längsgående axel.
- BOOSTER – Färjan ska nu släppa hjälpraketerna.
- SPACE – Nu är färjan ute i rymden och ska byta från huvudraket till rymdraket.

Figur 15 – Uppskjutningens olika faser

Två larm som inte baseras på färjans avstånd till jorden är det som talar om att färjan behöver mer kraft till raketerna och det som talar om att raketerna har för mycket kraft och är överhettade. Till dessa larm finns inga booleska variabler, utan villkoren för dem kontrolleras kontinuerligt.

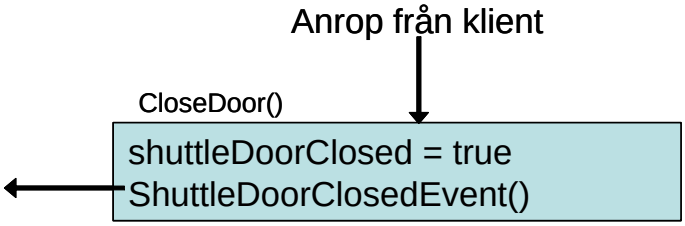
4.1.2 Interaktiva händelser

En uppskjutning av en rymdfärja startar vanligtvis med en nedräkning, så även i detta fall. Som Figur 16 visar gör klienten gör ett anrop till startCountToLiftoff(), och denna metod skickar ut en händelse som talar om att nedräkningen har startat. Därefter startas en timer som sköter själva nedräkningen. Varje gång timern löser ut minskas värdet på en nedräkningsvariabel med ett, tills nedräkningen gått ner till noll. Då stoppas timern. Förutom att räkna ner variabeln skickar metoden även ut en händelse med det nya värdet på nämnda variabel.



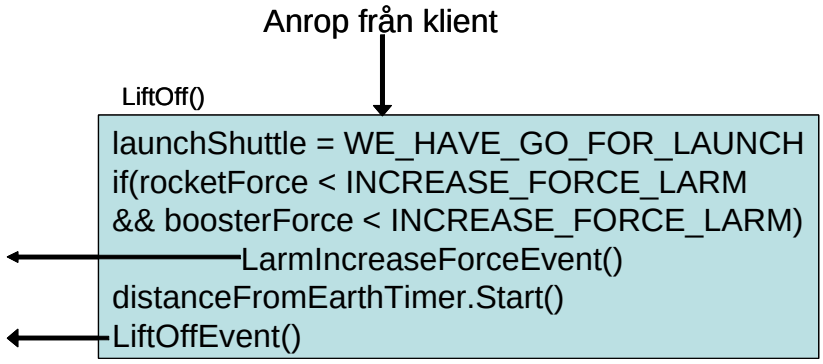
Figur 16 – Nedräkningen i modellen

Nästa sak som bör göras är att stänga dörren, vilket visas i Figur 17. Principen för att stänga dörren är densamma som för att lossa hållaren som håller rymdfärjan, börja rullningen, släppa hjälpraketerna, stänga av huvudraketen, och starta rymdraketen. Klienten anropar en metod i modellen, som sätter den tillhörande booleska variabeln till **false**, och skickar ut en signal som talar om att operationen lyckades. När hjälpraketen släpps och huvudraketen stängs av, sätts dessutom värdet på dessa raketer till noll.



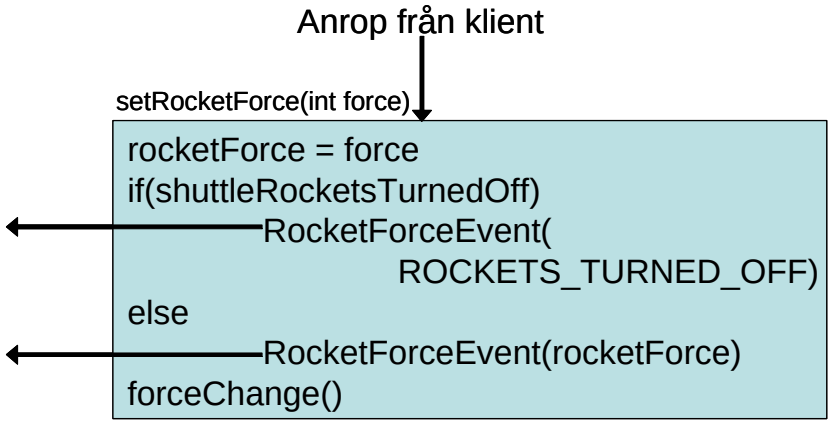
Figur 17 – Dörrstängningen i modellen

När färjan är lossad och dörren är stängd, är det dags att flyga iväg. Det görs genom att klienten anropar iväg-flygnings-metoden. Som Figur 18 visar, sätts först en variabel som talar om att färjan får flyga iväg. Därefter kontrolleras ifall tillräckligt med kraft finns fördelad till raketerna, och skulle detta inte uppfyllas, skickas en larmhändelse till klienten. Efter detta startas timern som sköter avståndshanteringen, och en händelse som talar om att färjan är på väg skickas ut.



Figur 18 – Ivägflygningen i modellen

För att ha tillräckligt med kraft på raketerna måste motsvarande variabel ändras, och det görs genom att klienten anropar en sätt-metod för både huvudraketen och hjälpraketen. Dessa metoder fungerar på samma sätt. Den interna raketkraften sätts lika med den kraft som skickas in i metoden, och därefter kontrolleras så att raketen inte är avstängd eller bortkopplad. Skulle så vara fallet skickas en händelse ut som talar om att ingen kraft finns i raketen, men i annat fall skickas den nuvarande kraften ut med samma händelse. Sist anropas den metod som sätter intervallet på avståndstimern, som ju beror på kraften på raketerna. Serverns exekveringsförlopp för hur kraften sätts är visualiserad i Figur 19.

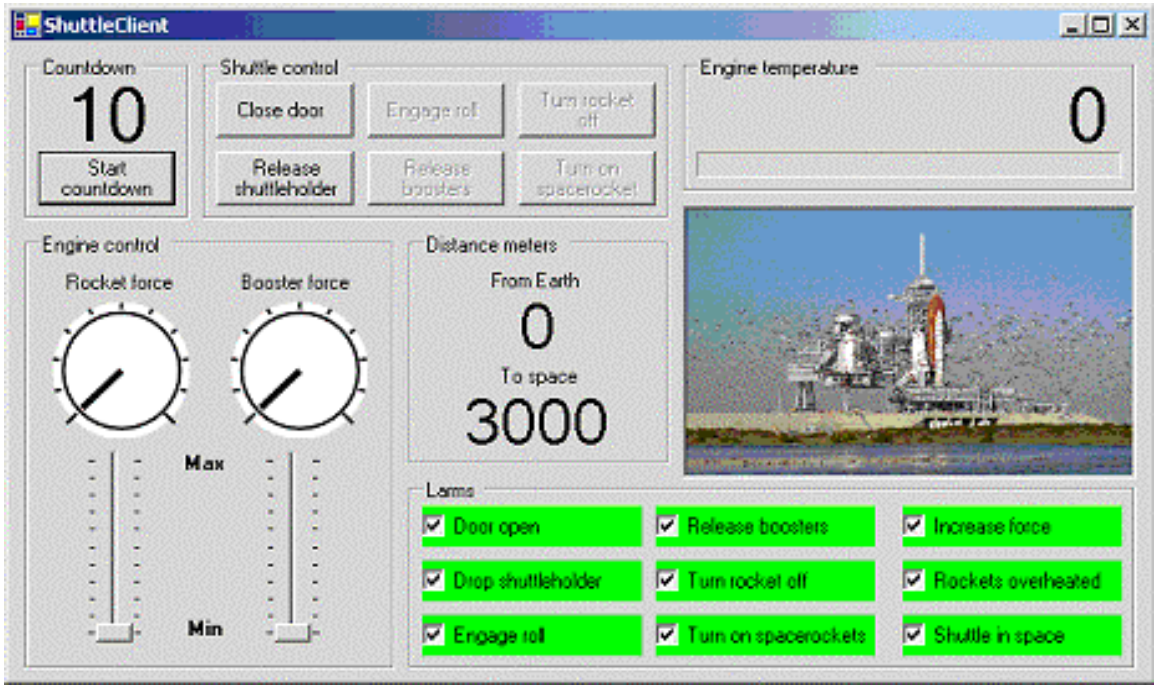


Figur 19 – Hur kraften på raketen sätts i modellen

En komplett lista över serverns metoder, delegater och händelser finns i bilaga D. Koden finns att erhålla hos Hannes Persson (hannes.persson@kau.se), Karlstads universitet.

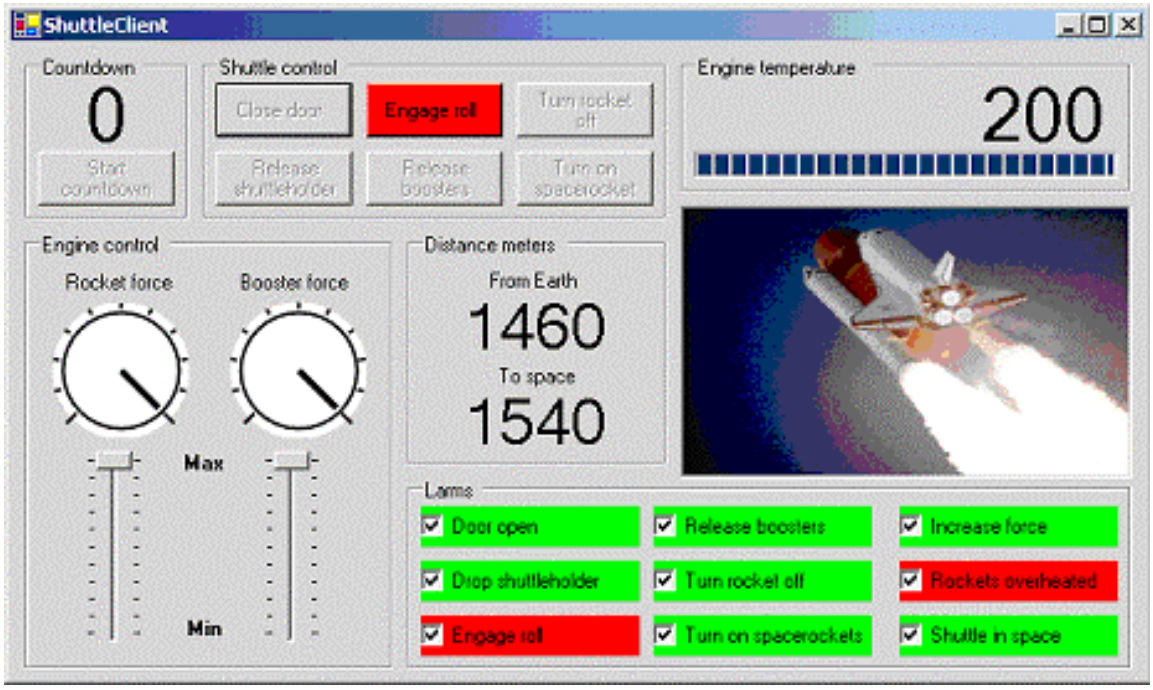
4.2 Klientlösning

Användargränssnittet ska både styra modellen och återspegla de ändringar som sker i den. För att kunna göra detta skapas ett proxy-objekt mot modellen som ger tillgång till dess publika medlemmar. När ett proxy finns, kan modellen styras med vanliga metodanrop. Hur detta går till förklaras i kapitel 3.3.2. För att återspegla ändringarna i modellen prenumererar gränssnittet på de händelser som skickas ut av modellen. Prenumerering av händelser förklaras i kapitel 2.2.5, och en lista över händelserna i modellen finns i bilaga D.2. När en händelse från modellen tas emot, uppdateras gränssnittet med den nya informationen.



Figur 20 – Uppstart av användargränssnitt

När gränssnittet, se Figur 20, startas är alla larm påslagna, kraften på motorerna är av och nedräkningsklockan är satt till 10 sekunder. Bilden som visas ändras utefter vilken fas uppskjutningen befinner sig i. I Figur 21 är simulationen i rullningsfasen. De två larm som visas talar om att motorerna är överhettade samt att raketen måste börja rulla runt sin längsgående axel för att stabilisera flygningen.



Figur 21 – GUI under körning

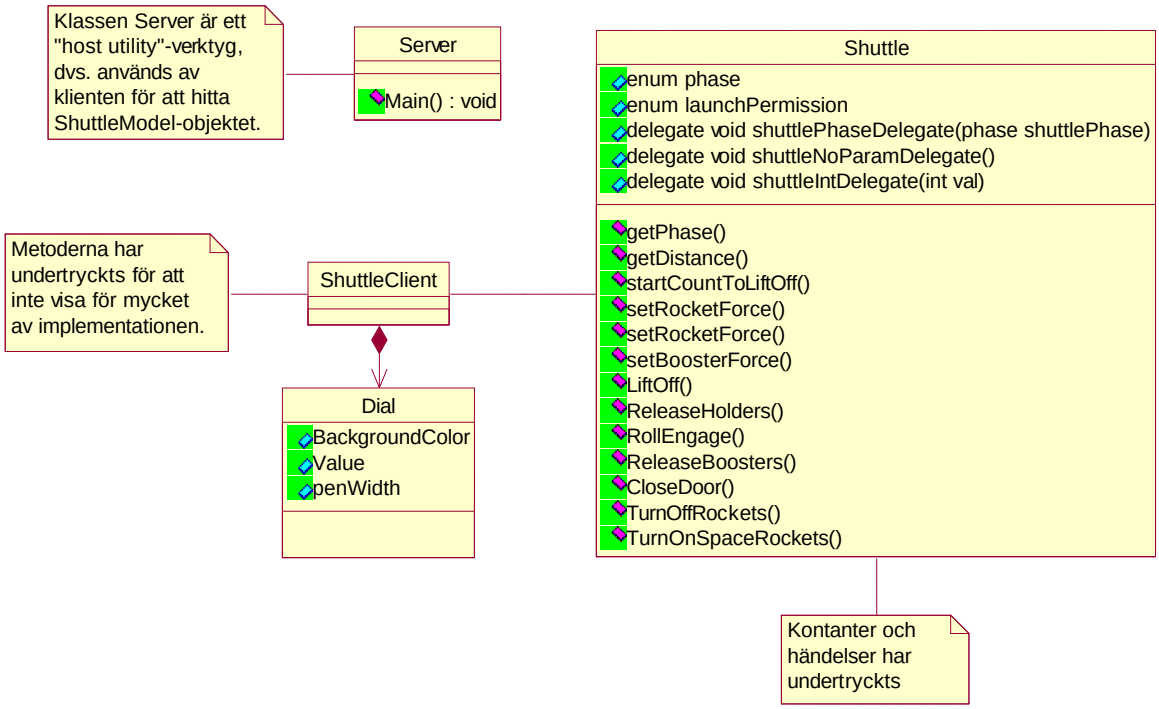
Det grafiska användargränssnittet är indelat i ett antal grupper av komponenter, där varje grupp avdelas med en GroupBox. De komponenter som hör ihop har samlats i samma gruppering. Nedräkningen utgörs av en etikett (eng. Label) och en knapp (eng. Button), där knappen startar nedräkningen och etiketten visar återstående tid. I modellen finns ett förvillkor som säger att nedräkningen endast får startas en gång. Detta måste kontrolleras i gränssnittet som anropar metoden. När knappen för att starta nedräkningen blivit nedtryckt dimmas den vilket medför att den endast kan bli nedtryckt en gång, och på sätt upprätthålls kravet. Detta görs genomgående i gränssnittet.

För att visa temperaturen på motorerna används en etikett för att visa den exakt och en ProgressBar för att visa den procentuellt. Kraften på motorerna sätts med två TrackBar-objekt, och för att visa den aktuella kraften används två Dial-objekt. Dial-komponenten skapades för den här implementationen och hur den är uppbyggd beskrivs i bilaga A. De nio larmen visas med hjälp av CheckBox-objekt. Dessa är gröna när larmen är påslagna, annars grå, och när larmen löser ut blir de röda.

4.3 Sammanslagning

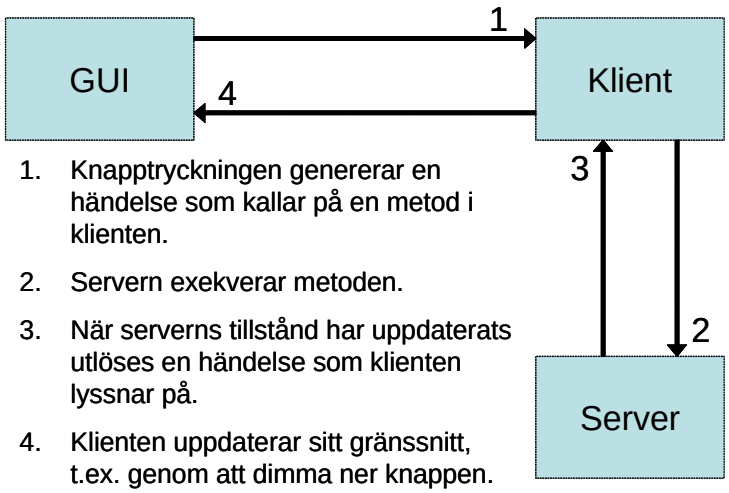
Efter att ha beskrivit server- och klientlösning, kommer nu båda att diskuteras tillsammans mer ingående. Först kommer ett klassdiagram med de viktigaste medlemmarna att presenteras och sedan förklaras interaktion mellan objekten.

I Figur 22 visas ett enkelt klassdiagram över hur konstruktionslösningen ser ut. Som bilden visar, har alla medlemmar i ShuttleClient, som är klassen som hanterar gränssnittet, undertryckts för att inte visa viktiga implementationsdetaljer. Eftersom syftet med laborationen är att implementera en klient med ett grafiskt gränssnitt utelämnas viss information.



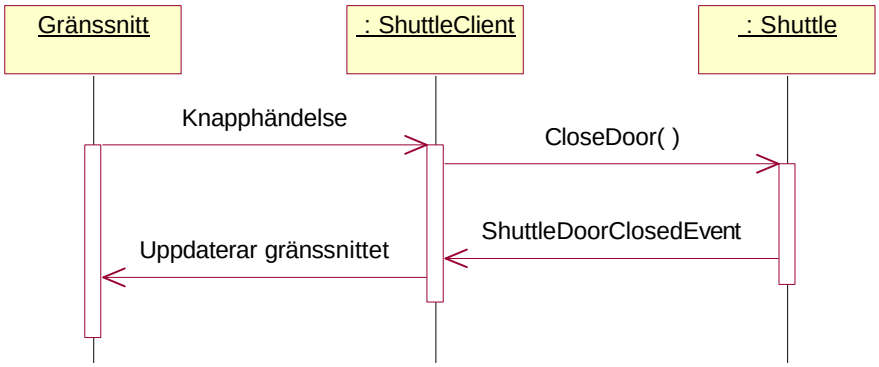
Figur 22 - Klassdiagram

Resten av detta kapitel kommer att inriktas på interaktionen mellan klient- och serverobjekten. Kommunikationen, som redan nämnts i kapitel 3.3.2, sker med hjälp av en teknik som kallas remoting.



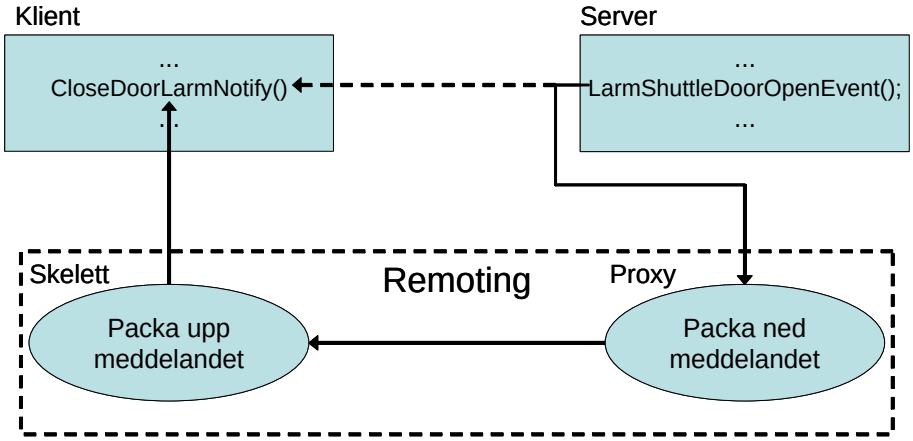
Figur 23 – Kommunikation mellan klient och server

Figur 23 visar en exempelinteraktion mellan ett grafiskt gränssnitt, en klient som implementerar det och en server. Det som illustreras är hur kommunikationen kan gå till för händelsen att rymdfärjans dörr stängs i laborationen. Användaren av programmet trycker på en knapp vars funktion är att stänga dörren. Knapptryckningen genererar en händelse som klienten lyssnar på. Klientens metod kallar på metoden CloseDoor() i serverobjektet. Under exekveringen av CloseDoor() kommer händelsen ShuttleDoorClosedEvent att utlösas. Denna händelse har en lyssnande metod i klientobjektet. När denna metod kör uppdateras gränssnittet att spegla att dörren har stängts i simulatorn. Figur 24 visar denna interaktion med ett sekvensdiagram.



Figur 24 – Sekvensdiagram över dörrstängning

Ovan nämnda händelse har ett larm som utlöser om rymdfärjan går in i LAUNCHING-fasen, se Figur 15, och dörren fortfarande är öppen. Det är tänkt att klienten ska fånga detta larm och visualisera det i gränssnittet. I konstruktionen används kryssrutor som byter bakgrundsfärg när ett larm tas om hand. Knappen som åtgärdar problemet byter också färg. För mer information om hur händelser, såsom larm, ska tas om hand i klienten, se kapitel 2.2.5.



Figur 25 – Principen för hur remoting fungerar

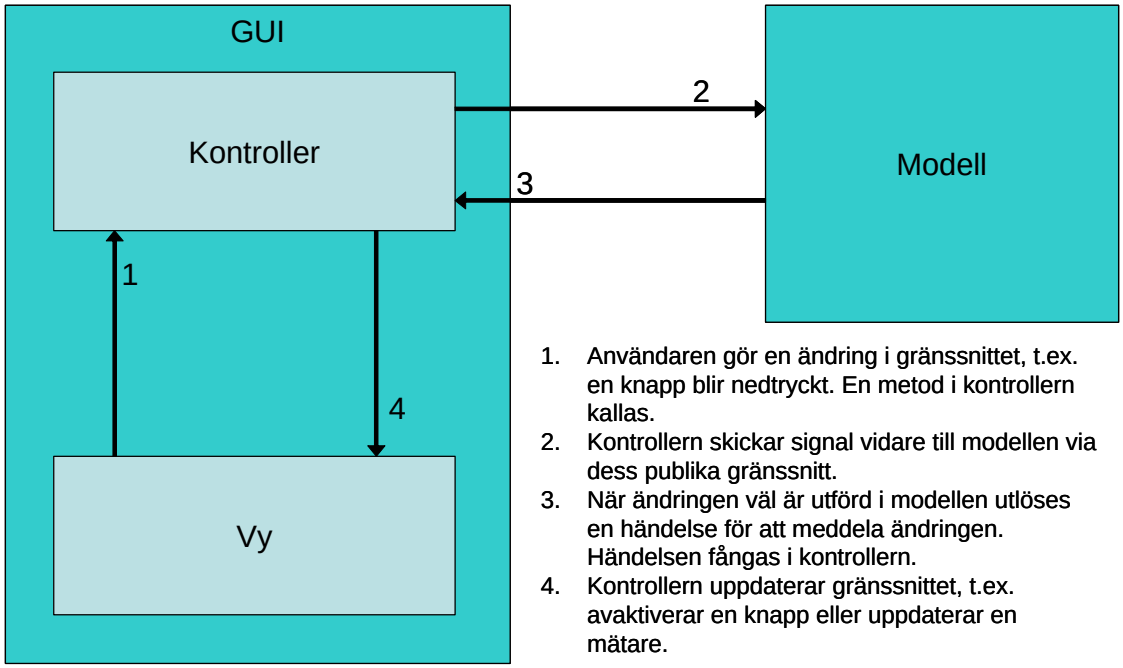
Själva kommunikationen mellan objekten sker automatiskt med remotingen, se Figur 25. Remotingen tar anropet och dess parametrar och paketerar ner dem i ett SOAP-meddelande. Detta meddelande skickas sedan över den specificerade kanalen till den mottagande sidan, som packar upp informationen. Det som är till stor hjälp med kommunikationsutbytet i .NET är att allt detta är transparent. Programmeraren kan lika gärna programmera som om det distribuerade objektet vore lokalt.

En detalj som måste nämnas är att eftersom prenumeration används i klienten för att modellen ska kunna skapa ett proxy till klienten, får implementeraren inte glömma att avprenumerera när denne är klar. Detta för att förhindra feluppkomst.

4.3.1 Hur MVC kan implementeras i laborationen

Efter det korta exemplet på hur rymdfärjans dörr kan stängas i föregående kapitel kommer detta kapitel att inrikta sig på hur interaktionen mellan objekten bör se ut i ett mer MVC-aktigt perspektiv, se kapitel 3.2. I en MVC-lösning bör klienten egentligen bestå av två separata klasser, kontrollern och gränssnittet (eller vyn). Dessa skulle samverka som Figur 26 visar. Från en användares initiativ skickas en signal från en gränssnittskomponent till kontrollern. Denna tar emot signalen och undersöker om alla villkor är uppfyllda för att kunna skicka signalen vidare till modellen. Om så är fallet kallas en metod i modellen som utför den specifika uppgiften. När modellen är klar triggas en händelse som kontrollern lyssnar på. Den fångar händelsen och uppdaterar gränssnittet för att spegla det nya tillståndet i modellen.

I konstruktionen används en lite annorlunda lösning, men som konceptuellt är lika. Klienten består av endast en klass men har separerat de delar som konceptuellt kan ses som en kontroller till egna metoder i klassen. Dessa metoder har kontrollerns uppgift, dvs. att skicka signaler från gränssnitt till modell och att uppdatera vyn på externa händelser.



Figur 26 – MVC i laborationen

Alternativet att inte använda en lösningen baserad på MVC kan mycket väl försvåra ändringar i det grafiska gränssnittet, samt göra kommunikationen mellan server- och klientobjekten mycket mer komplicerad. Orsaken till detta förklarades i ett tidigare kapitel om MVC.

4.4 Test

Hela det publika gränssnittet i modellen är styrt av kontrakt, se [21], vilket innebär att varje metod har ett för- och eftervillkor. Eftervillkoret gäller för metoden endast om förvillkoret är uppfyllt vid anrop. Om förvillkoret inte är uppfyllt, finns inget krav på att metoden ska fungera korrekt. Det är något som studenterna som ska implementera laborationen måste tänka på.

Eftersom klienten direkt anropar modellen, ligger kravet på kontroll i gränssnittet. Detta kan manifestera sig som att en knapp är dimmad om den ej får tryckas ned osv. Det ligger då i implementerarnas händer att se till att alla krav är uppfyllda innan varje anrop till modellen sker. Vid implementeringen av klientlösningen har alla kontrakt uppfyllts innan anrop med hjälp av speciella metoder som har fungerat som en kontroller, se kapitel 3.2. De egentliga fel som kan inträffa är s.k. yttre fel, såsom fel vid kommunikationen mellan modell och klient. Dessa måste fångas med hjälp av undantag (eng. exceptions).

Den testning som har gjorts har främst inneburit att kontrollera kontrakt och göra tester på att gränssnittet fungerar som det ska. Ett av dessa tester innebar att multipla gränssnitt kopplades upp mot modellen för att se att ändringar i ett gränssnitt speglades i de andra.

5 Erfarenheter

Ett av målen för .NET och C# var att uppfylla MS krav på RAD. Detta anses ha uppfyllts bra, då det är mycket enkelt att skapa program. De egentliga problem som erfarits med detta har varit att mycket av informationen har varit bristfällig. Med detta menas att eftersom språket, tillsammans med plattformen, är så nytt, saknas mycket vital information. Vid implementering av remoting i konstruktionen fanns det mycket information att tillgå, men denna var alltid för generell för att förstå den bakomliggande strukturen och hur den fungerade. Förhoppningsvis kommer detta snart att ändras.

En annan bra erfarenhet av .NET har varit de nästan fullständiga klassbiblioteken som finns att tillgå. Dessa gör att utvecklingen kan gå mycket snabbare, då hjulet inte behöver uppfinnas varje gång det ska användas. Uppgiften har dock endast krävt att bara en liten delmängd av .NET har utnyttjats. Uppsatsen använder en mycket liten del av det hela. T.ex. används inte ASP.NET eller ADO.NET över huvudtaget för att bara nämna något.

I konstruktionslösningen valdes remoting istället för web services. Orsaken till detta har redan förklarats tidigare i arbetet, se 3.3.1. Kontentan blir att webbtjänster är bättre att använda för små distribuerade metoder som ska returnera ett värde. Remoting verkar också vara ett krav om kommunikation ska kunna initieras från servern.

Det går inte att undgå att C++-program är mycket snabbare än program skrivna för .NET, dock har detta inte upplevts som ett problem i vår konstruktion. Det som skulle kunna upplevas som långsamt i den föreslagna applikationen är kommunikationen mellan objekten och det faktum att applikationen är distribuerad.

Vid implementering har det uppmärksammats att MS Visual Studio .NET är till stor hjälp och det rekommenderas att programmet används i laborationen. Programmet har en mekanism, Intellisense, som hjälper till att avsluta programsatser. Verktyget ger en lista på möjliga alternativ för aktuell programsats och hur dessa ska användas. Förutom detta erbjuder programmet automatisk indentering och möjligheten att dölja kod som inte är aktuell samt andra bra hjälpmedel som underlättar vid programmering.

6 Slutsatser

Vi har lärt oss hur .NET och C# fungerar och hur det används. Detta var en av anledningarna till varför vi valde att skriva denna uppsats. Vi är glada över att .NET motsvarade våra förväntningar, då det är MS nya stora satsning på framtiden och det känns som att det är en fördel att idag kunna lite om det.

Vi har dessutom lärt oss hur man skriver distribuerade program samt lagt grunden till utökad kunskap om plattformen .NET. Detta kommer att kunna hjälpa oss att bli bättre på att programmera i .NET i framtiden. En nackdel med att använda MS nya ramverk för distribuerade system är just att det är nytt, och därför svårt att finna information om det.

Uppsatsen känns för oss som en framgång, då vi har konstruerat en distribuerad programvara som kan ligga till grund för en ny laboration i en kurs, vilket var en stor del av uppsatsens mål. Vi har också försökt att presentera .NET och språket C# på ett så enkelt och uttömmande sätt som möjligt, och även detta har uppfyllt ännu ett mål för arbetet.

Referenser

[1] *MSDN*, http://msdn.microsoft.com/netframework/productinfo/overview/default.asp, 2003-02-04.

[2] *MSDN*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconwhatiscommonlanguagespecification.asp, 2003-02-04.

[3] *C# Unleashed*, Joseph Mayo, Sams Publishing 2002, ISBN: 0-672-32122-x.

[4] *C# & .NET – Windowsprogrammering i Visual Studio .NET*, Erik Ronne, Docendo Sverige AB 2002, ISBN: 91-7882-564-4.

[5] *C# .NET – Web Developer's Guide*, Adrian Turtschi, DotThatCom.com, Jason Werry, Greg Hack, Joseph Albahari, Saurabh Nandu, Wei Meng Lee, Syngress Publishing, Inc., ISBN: 1-928994-50-4.

[6] *MSDN*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconjitcompilation.asp, 2003-02-10.

[7] *JAVA*, http://java.sun.com, 2003-02-10.

[8] *IEEE Computer Society*, http://www.computer.org/software/homepage/2003/s1lap_1.htm, 2003-02-12.

[9] *O`Reilly*, http://windows.oreilly.com/news/hejlsberg_0800.html, 2003-02-12.

[10] *W3C World Wide Web Consortium*, http://www.w3.org, 2003-02-17.

[11] *MSDN*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrftypesaccessmodifiers.asp, 2003-03-17.

[12] *MSDN*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cscon/html/vclrfcomparisonbetweenccsharp.asp, 2003-04-01.

[13] *Genamics*, http://genamics.com/developer/csharp_comparative.htm, 2003-04-01.

[14] *Kuro5hin*, http://www.kuro5hin.org/story/2002/6/25/122237/078, 2003-04-01.

[15] *Comparison of characteristics of my database systems*, http://www.garret.ru/~knizhnik/compare.html, 2003-04-01.

[16] *C# From a Java Developer's Perspective*, http://www.25hoursaday.com/CsharpVsJava.html, 2003-04-01.

[17] *Microsoft .NET XML Web Services*, Robert Tabor, Sams 2001, ISBN: 0-672-32088-6.

[18] *C# Help*, http://www.csharphelp.com/archives/archive187.html, 2003-04-02.

[19] *C# Station*, http://www.csharp-station.com/Articles/InterfacesInDotNetRemoting.aspx, 2003-04-02.

[20] *Model View Controller*, http://www.netchemistry.com/pdf/white_papers/NCModelViewController.pdf, 2003-05-15.

[21] *Method Description for Semla - A Software Design Method with a Focus on Semantics*, Martin Blom, Eivind J. Nordby, Anna Brunström

Akronymlista

ADO – ActiveX Data Objects

ASP – Active Server Pages

BCL – Base Class Library

CLS – Common Language Specification

DLL – Dynamic Link Library

GUI – Graphical User Interface

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

IDE – Integrated Development Environment

IIS – Internet Information Server

JIT – Just In Time

JVM – Java Virtual Machine

MS – Microsoft

MSDN – Microsoft Developer Network

MSIL – Microsoft intermediate language

MVC – Model View Controller

RAD – Rapid Application Development

RPC – Remote Procedure Call

SDK – Software Development Kit

SOAP – Simple Object Access Protocol

SQL – Structured Query Language

TCP – Transmission Control Protocol

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

XML – eXtensible Markup Language

A Dial

Här följer koden för den gränssnittskomponent som kan användas som mätare i klientlösningen.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace ShuttleControl
{
	/// <summary>
	/// En grafisk komponent för visning av t.ex. krafter och gaspådrag.
	/// </summary>
	public class Dial : System.Windows.Forms.UserControl
	{
		/// <summary>
		/// Required designer variable.
		/// </summary>
		private System.ComponentModel.Container components = null;
		private Color backgroundColor = Color.Empty;
		/// <summary>Sätter färgen på komponenten.</summary>
		/// <remarks>Tar emot ett Color-objekt.</remarks>
		public Color BackgroundColor
		{
			get
			{
				return backgroundColor;
			}
			set
			{
				backgroundColor = value;
				this.Refresh();
			}
		}
		private int val = 0;
		/// <summary>Sätter värdet på komponenten.</summary>
		/// <remarks>Tar emot ett heltal mellan 0 och 100.</remarks>
		public int Value
		{
			get
			{
				return val;
			}
			set
			{
				if(value < 0)
				{
					val = 0;
					this.Refresh();
				}
				else if(value > 100)
				{
					val = 100;
					this.Refresh();
				}
				else
```

```
                {
                    val = value;
                    this.Refresh();
                }

            }
        }

        public int penWidth = 2;
        public Dial()
        {
            // This call is required by the Windows.Forms Form Designer.
            InitializeComponent();
        }

        private int setX(int val, double size)
        {
            double angle = (1.25d - 0.015d * val) * Math.PI;
            return (int)(Math.Cos(angle) * size * Width + Width/2);
        }

        private int setY(int val, double size)
        {
            double angle = (1.25d - 0.015d * val) * Math.PI;
            return (int)(Height - (Math.Sin(angle)
                * size * Height + Height/2));
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics graphics = e.Graphics;
            graphics.SmoothingMode =↵
System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
            Pen pen = new Pen(Color.Black, penWidth);
            Pen pen2 = new Pen(Color.Black, 4);
            SolidBrush brush = new SolidBrush(backgroundColor);
            Point p1 = new Point(Width/2,Height/2);
            Point p2 = new Point();
            p2.X = setX(Value, 0.4d);
            p2.Y = setY(Value, 0.4d);

            graphics.FillEllipse(brush, 0, 0, Width, Height);
            graphics.DrawEllipse(pen, penWidth/2+5, penWidth/2+5,
                Width - penWidth-10, Height - penWidth-10);
            graphics.DrawLine(pen2, p1, p2);
            for(int i = 0; i < 101; i += 10)
                if(i == 0 || i == 100)
                    graphics.DrawLine(pen, new Point(setX(i, 0.45d),↵
setY(i, 0.45d)), new Point(setX(i, 0.55d), setY(i, 0.55d)));
                else
                    graphics.DrawLine(pen, new Point(setX(i, 0.45d),↵
setY(i, 0.45)), new Point(setX(i, 0.5d), setY(i, 0.5d)));
        }

        protected override void OnMove(EventArgs e)
        {
            this.Refresh();
        }

        protected override void OnResize(EventArgs e)
        {
            this.Refresh();
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
```

```
        {
            if( disposing )
            {
                if( components != null )
                    components.Dispose();
            }
            base.Dispose( disposing );
        }

        #region Component Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            components = new System.ComponentModel.Container();
        }
        #endregion
    }
}
```

B XML-dokumentation

<c>

Används för att indikera att text inom en beskrivning ska markeras som kod.

Ex.
```
/// <summary>
/// Metoden <c>MinMetod</c> är en metod i klassen
/// <c>MinKlass</c>
/// </summary>
public void MinMetod() {}
```

<code>

Används för att indikera att flera rader ska markeras som kod.

Se <example> för ett exempel på hur <code> kan användas.

<example>

Används för att visa exempel på hur en metod eller annan kod ska användas.

Ex.
```
/// <example>Så här används Add-metoden.
/// <code>
///  public static void Main()
/// {
///      Add(3, 14);
/// }
/// </code>
/// </example>
public int Add(int x, int y) {}
```

<exception>

Används för att tala om vilka undantag som kan kastas. cref pekar undantaget i fråga.

Ex.
```
/// <exception cref="System.Exception">Kastas när...</exception>
public void MinMetod() {}
```

<include>

Används för att referera till en XML-fil som innehåller XML-dokumentation. <include> har två parametrar, file och path, som används för att lokalisera filen respektive hitta i filen.

Ex.
```
/// <include file='Fil.xml' path='Doc/Class[@name="Class1"]/*'/>
public class MinKlass1 {}

*includeFil.xml*
<Doc>
    <Class name="Class1">
        <summary>Min första klass</summary>
```

```
        </Class>
        <Classer name="Class2">
            <summary>Min andra klass</summary>
        </Class>
    </Doc>
```

<list>

Används för att göra listor, och kan vara av typen bullet, number eller table. Ett element läggs till med taggen item. Listorna kan även ha en listheader.

Ex.
```
/// <list type="bullet">
/// <listheader>
///     <term>uttryck</term>
///     <description>beskrivning</description>
/// </listheader>
/// <item>
///     <term>uttryck</term>
///     <description>beskrivning</description>
/// </item>
/// </list>
public void MinMetod() {}
```

<para>

Används för att skapa stycken i texten.

Ex.
```
/// <summary>Lite text...
/// <para>Lite mer text i ett nytt stycke.</para>
/// </summary>
public void MinMetod() {}
```

<param>

Används för att beskriva parametrarna till en metod

Ex.
```
/// <param name="status">Används för att indikera status</param>
public void MinMetod(bool status) {}
```

<paramref>

Används för att visa att ett ord är en parameter.

Ex.
```
/// <remarks>MinMetod än en metod i klassen MinKlass.
/// Parametern <paramref name="var"/> är ett heltal.
/// </remarks>
public static void MinMetod(int var) {}
```

<permission>

Används för att dokumentera åtkomsten för ett medlem. Med System.Security.PermissionSet kan man specificera olika åtkomster.

Ex.
```
/// <permission cref="System.Security.PermissionSet">
```

```
    /// Alla kan komma åt den här metoden.
    /// </permission>
    public static void MinMetod() {}
```

<remarks>

Används för att ge extra information utöver den under <summary>.

Ex.
```
    ///<summary>Lite information.</summary>
    ///<remarks>Liter mer information.</remarks>
    public void MinMetod() {}
```

<returns>

Används för att dokumentera returvärden.

Ex.
```
    /// <returns>Returnerar noll.</returns>
    public static int GetZero()
    {
        return 0;
    }
```

<see>

Används för att lägga till en länk i en text. <see> har parametern cref som innehåller själva länken.

Ex.
```
    /// <summary>MinMetod skriver ut "Hello World!" på skärmen.
    /// Se <see cref="System.Console.WriteLine"/> för mer
    /// information om utskrifter.
    /// </summary>
    public void MinMetod()
    {
        System.Console.WriteLine("Hello World!");
    }
```

<seealso>

Används för att ge referenser till andra ämnen som kan vara av intresse för användaren.

Ex.
```
    /// <summary>MinMetod använder
    /// <see cref="System.Console.WriteLine"/>
    /// för att skriva till skärmen.
    /// </summary>
    /// <seealso cref="System.Console.ReadLine"/>
    public void MinMetod(string text) {}
```

<summary>

Används för att ge en kortare beskrivning av ett element.

Ex.
```
    /// <summary>Metoden Add() adderar två tal.</summary>
    public int Add(int x, int y) {}
```

<value>

Används för att beskriva egenskaper(eng. properties), och vad de representerar.

Ex.
```
private string namn;
/// <value>
/// Name returnerar och sätter värdet på variabeln name.
/// </value>
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
```

C Remotingexempel

HostedServer.cs

```
using System;

namespace Server{
     public class HostedServer : MarshalByRefObject
     {
          // Klassens enda publika metod. Returnerar strängen "Hello World!"
          public string getServerResponse(){
               return "Hello World!";
          }
     }
}
```

web.config

```
<configuration>
     <system.runtime.remoting>
          <application>
               <service>
                    <wellknown mode="SingleCall" type="Server.HostedServer,
HostedServer" objectUri="HostedServer.soap" />
               </service>
          </application>
     </system.runtime.remoting>
</configuration>
```

Client.cs

```
using System;
using System.Runtime.Remoting;
using Server;

namespace Client
{
     public class Client
     {
          public static void Main(string[] args)
          {
               // URL till serverobjektet.
               String url = "http://localhost:8000/HostedServer/HostedServer.soap";
               // Konfigurera remoting från filen "Client.exe.config".
               RemotingConfiguration.Configure("Client.exe.config");
               // Få en referens till HostedServerobjektet med hjälp av ett
               //Activatorobjekt. Metoden GetObject returnerar ett Objectobjekt
               //så det måste konverteras till ett HostedServerobjekt.
               HostedServer server =
(HostedServer)Activator.GetObject(typeof(HostedServer), url);
               // Skriv ut svaret från servern.
               Console.WriteLine(server.getServerResponse());
          }
     }
}
```

Client.exe.config

```
<configuration>
     <system.runtime.remoting>
```

```
          <application name="Client">
               <client url="http://localhost:8000/HostedServer">
                    <wellknown type="Server.HostedServer, HostedServer"
url="http://localhost:8000/HostedServer/HostedServer.soap" />
               </client>
               <channels>
                    <channel
type="System.Runtime.Remoting.Channels.Http.HttpChannel, System.Runtime.Remoting"
/>
               </channels>
          </application>
     </system.runtime.remoting>
</configuration>
```

RemotingHost.cs

```
using System;
using System.Runtime.Remoting;

public class RemotingHost
{
     public static void Main()
     {
          // Konfigurera remotingen från angiven fil.
          RemotingConfiguration.Configure("RemotingHost.exe.config");
          // Skriv ut informationstext.
          Console.WriteLine("Press any key to exit...");
          // Vänta på användarinmatning för att avsluta programmet.
          Console.ReadLine();
     }
}
```

RemotingHost.exe.config

```
<configuration>
     <system.runtime.remoting>
          <application name="HostedServer">
               <service>
                    <wellknown mode="SingleCall" type="Server.HostedServer,
HostedServer" objectUri="HostedServer.soap" />
               </service>
               <channels>
                    <channel port="8000"
type="System.Runtime.Remoting.Channels.Http.HttpChannel, System.Runtime.Remoting"
/>
               </channels>
          </application>
     </system.runtime.remoting>
</configuration>
```

D Beskrivning av modell

I denna bilaga följer alla delegater, händelser och metoder som används i modellen, bortsett från konstruktorn.

D.1 Delegater

Här följer de tre delegaterna som används för modellens händelser.

```
public delegate void shuttlePhaseDelegate(phase shuttlePhase)
public delegate void shuttleIntDelegate(int val)
public delegate void shuttleNoParamDelegate()
```

D.2 Händelser

Nedan beskrivs alla händelser som skickas ut från modellen.

`public event shuttlePhaseDelegate PhaseEvent`
Meddelar vilken fas färjan befinner sig i.

`public event shuttleIntDelegate SecondsToLiftOffEvent`
Meddelar antalet sekunder kvar av nedräkningen.

`public event shuttleIntDelegate DistanceChangeEvent`
Meddelar nuvarande avstånd från jorden.

`public event shuttleIntDelegate DistanceToSpaceEvent`
Meddelar nuvarande avstånd till rymden.

`public event shuttleIntDelegate RocketForceEvent`
Meddelar nuvarande kraft på huvudraketen.

`public event shuttleIntDelegate BoosterForceEvent`
Meddelar nuvarande kraft på huvudraketen.

`public event shuttleIntDelegate RocketTemperatureChangeEvent`
Meddelar temperaturen på motorerna.

`public event shuttleNoParamDelegate LarmShuttleDoorOpenEvent`
Larmar om att dörren är öppen.

`public event shuttleNoParamDelegate LarmReleaseShuttleHolderEvent`
Larmar om att rymdfärjan hålls fast på marken.

`public event shuttleNoParamDelegate LarmRocketOverheatEvent`
Larmar om att motorerna är överhettade.

public event `shuttleNoParamDelegate LarmStartShuttleRollEvent`
Larmar om att färjan måste påbörja sin rullning.

public event `shuttleNoParamDelegate LarmTurnRocketsOffEvent`
Larmar om att huvudraketen ska stängas av.

public event `shuttleNoParamDelegate LarmTurnSpaceRocketsOnEvent`
Larmar om att rymdraketen ska slås på.

public event `shuttleNoParamDelegate LarmIncreaseForceEvent`
Larmar om att kraften på raketerna måste ökas.

public event `shuttleNoParamDelegate LarmReleaseBoostersEvent`
Larmar om att hjälpraketerna ska släppas.

public event `shuttleNoParamDelegate LarmShuttleInSpaceEvent`
Larmar om att rymdfärjan nu är i rymden.

public event `shuttleNoParamDelegate CountDownStartedEvent`
Meddelar att nedräkningen påbörjats.

public event `shuttleNoParamDelegate LiftOffEvent`
Meddelar att rymdfärjan har lyft från marken.

public event `shuttleNoParamDelegate ShuttleHolderReleasedEvent`
Meddelar att hållaren av rymdfärjan har släppt.

public event `shuttleNoParamDelegate ShuttleRollEngagedEvent`
Meddelar att rullningen av färjan ha börjat.

public event `shuttleNoParamDelegate BoostersReleasedEvent`
Meddelar att färjans hjälpraketer har släppts.

public event `shuttleNoParamDelegate ShuttleDoorClosedEvent`
Meddelar att dörren är stängd.

public event `shuttleNoParamDelegate RocketsTurnedOffEvent`
Meddelar att huvudraketen har stängts av.

public event `shuttleNoParamDelegate SpaceRocketsTurnedOnEvent`
Meddelar att rymdraketen har slagits på.

public event `shuttleNoParamDelegate ForceIncreasedEvent`
Meddelar att tillräckligt med kraft finns i raketerna.

public event `shuttleNoParamDelegate RocketTemperatureOkEvent`
Meddelar att motorerna inte är överhettade.

D.3 Metoder

Här beskrivs alla metoder i modellen, inklusive kontrakten för de publika metoderna.

`public override` `object InitializeLifetimeService()`
Finns för att förhindra att tjänsten dör.

`private void` `connectTimers()`
Kopplar timerhändelserna till sina hanterare.

`private void` `stopTimers()`
Stoppar timrarna.

`private void` `resetShuttle()`
Återställer modellen.

`private void` `clientActionReset()`
Återställer klientens inverkan på modellen.

`private void` `enterPhase(phase newPhase)`
Får modellen att gå in i en ny fas.

`public` `phase getPhase()`
Returnerar nuvarande fas.
Pre: True.
Post: The current phase has been returned.

`public int` `getDistance()`
Returnerar nuvarande avstånd till jorden.
Pre: True.
Post: The current distance has been returned.

`private int` `calculateDistanceFromEarth()`
Räknar ut avståndet till jorden.

`public void` `startCountToLiftOff()`
Startar nedräkningen.
Pre: Not previously called.
Post: Countdown has been started.

`private void` `secondsToLiftOff(object sender, ElapsedEventArgs args)`
Hanterar rymdfärjans nedräkning.

`private void` `distanceChange(object sender, ElapsedEventArgs args)`
Hanterar förändringar i avståndet.

`public void` `setRocketForce(`**`int`**` force)`
Sätter raketens kraft.
Pre: 0 <= force <= 100.
Post: The rocket's force has been set.

public void `setBoosterForce(int force)`
Sätter hjälpraketens kraft.
Pre: 0 <= force <= 100.
Post: The booster's force has been set.

private void `forceChange()`
Hanterar ändringar i kraft i raketen och färjans hjälpraketer.

public void `LiftOff()`
Får färjan att lyfta.
Pre: Not previously called.
Post: Liftoff has been made possible & if force < 80 LarmIncreaseForceEvent emitted.

public void `ReleaseHolders()`
Släpper färjan från sin hållare.
Pre: Not previously called.
Post: Shuttleholders released & liftoff has been made possible.

public void `RollEngage()`
Får färjan att rulla runt sin längsgående axel.
Pre: Not previously called & getDistance() >= ROLL_LIKE_A_CROCODILE.
Post: Shuttle has been set to roll.

public void `ReleaseBoosters()`
Får färjan att släppa sina hjälpraketer.
Pre: Not previously called & getDistance() >= RELEASE_BOOSTERS.
Post: Shuttle has released its boosters.

public void `CloseDoor()`
Stänger färjans dörr.
Pre: Not previously called.
Post: The shuttle's door has been closed.

private void `RocketTemperatureChange(object sender, ElapsedEventArgs args)`
Hanterar temperaturförändringar i motorerna.

private void `calculateDistanceToSpace()`
Hanterar förändringar i avståndet till rymden.

public void `TurnOffRockets()`
Stänger av färjans huvudraket.
Pre: Not previously called & getDistance() >= SHUTTLE_IN_SPACE.
Post: Shuttle's rockets has been turned off.

public void `TurnOnSpaceRockets()`
Sätter på färjans rymdraketer.
Pre: Not previously called & getDistance() >= SHUTTLE_IN_SPACE.
Post: Shuttle's spacerockets has been turned on.