
Datavetenskap

---

**Mikael Hackman och Maria Höglund**

# Jämförelse av Exekveringstider vid Kontraktsprogrammering i Java

---

Examensarbete, C-nivå

02:01

# Jämförelse av Exekveringstider vid Kontraktsprogrammering i Java

**Mikael Hackman och Maria Höglund**

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.

Mikael Hackman och Maria Höglund

Mikael Hackman och Maria Höglund

Godkänd, 020215

Handledare: Hannes Persson

Examinator: Stefan Lindskog

# Sammanfattning

På ämnet datavetenskap vid Karlstads universitet finns forskningsgruppen SERG Software Engineering Group. SERG forskar i kontraktsbaserad programmering. Något som de ännu inte utvärderat är skillnaden i exekveringstid vid programmering med starka- respektive svaga kontrakt. Det här dokumentet redovisar hur de olika mekanismer som används vid kontraktsprogrammering påverkar exekveringstiderna. Arbetets fokus har varit att undersöka hur exekveringstiden för ett Java-program som helhet påverkas beroende på val av programmeringsstil. Det program som använts för mätningarna simulerar inläggning av ett element i en stack. Mätningarna visar att programmering med starka kontrakt alltid ger en kortare exekveringstid än programmering med svaga kontrakt.

# Comparison of Execution Times when Programming with Contracts in Java


## Abstract

The department of computer science at the University of Karlstad has a research group called SERG Software Engineering Research Group. In SERG they research upon contrakt-based programming. Something that they have not yet evaluated is the difference in execution time for contract-based vs. exception-based programming. This document records how the different mechanisms used with contract-based programming affect the execution times. The focus of this document is to investigate how the execution time for a program written in the programming language Java is affected dependent owing to the type of contract beeing used. The program beeing used simulates pushing an element on to a stack. The measurements show that programming whith strong conracts always gives a shorter execution time than programming with weak contracts.

# Innehållsförteckning

# Figurförteckning

# Tabellförteckning

# 1 Inledning

Den här uppsatsen är skriven som ett examensarbete på C-nivå i ämnet datavetenskap. Arbetet gick ut på att undersöka hur olika metoder för felhantering i Java påverkar exekveringstiden för ett program. En jämförelse av exekveringstider har gjorts vad gäller program skrivna dels med undantag och dels med för- och eftervillkor som felhanteringsmetod. Den slutsats som ska dras från dessa jämförelser är när det lönar sig att använda undantag och när det lönar sig att använda för- och eftervillkor. De slutsatser om lönsamhet som dras i uppsatsen avser enbart exekveringstid.

## 1.1 Bakgrund

Ämnet datavetenskap vid Karlstads universitet har en forskningsgrupp som arbetar inom ett projekt för ökad programvarukvalitet genom semantisk beskrivning (SERG). De utvecklar verktyg och metoder för semantisk beskrivning. Forskningen fokuseras på att utvärdera kontraktbaserad programmering kontra traditionella programmeringsmetoder. Effektiviteten och fördelarna med kontraktbaserad programmering är under utredning. Ett programs effektivitet kan ses ur en mängd olika aspekter. En aspekt som ännu inte har undersökts är hur exekveringstiden skiljer sig för de olika programmeringsstilarna. Denna uppsats redovisar några av de skillnader som kan uppstå i exekveringstid beroende på programmeringsstil. Dessa skillnader kan sedan vägas in i en jämförelse av effektiviteten för de olika programmeringsstilarna. Andra aspekter vad gäller ett programs effektivitet berörs inte i denna uppsats.

## 1.2 Mål

Målet med den här uppsatsen är att visa skillnader i exekveringsstid för de två programmeringsstilarna undantag respektive för- och eftervillkor. Syftet är att kunna säga när det är mest lönsamt vad gäller exekveringstid att använda undantag eller för- och eftervillkor.

## 1.3 Avgränsningar

Syftet med uppsatsen är inte att mäta exakta tider för olika mekanismer som används för de olika programmeringsstilarna. I stället för att fokusera på exakta tider ska de mekanismer som

används jämföras med varandra för att visa de skillnader som uppstår. De mekanismer som används för kontraktsprogrammering kan dock användas på olika sätt. För att kunna göra en allmän jämförelse ska mekanismernas egenskaper beskrivas, för att ge en bild av hur ett program kommer att bete sig genom användande av en viss mekanism.

Vissa saker som visat sig påverka de uppmätta tiderna har på grund av tidsbegränsning inte undersökts närmare.

### 1.4 Disposition

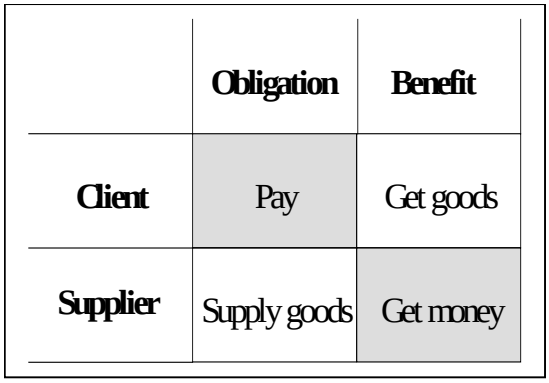
Detta kapitel visar hur denna uppsats är strukturerad. Kapitel 1 presenterar målet med uppsatsen och bakgrunden till varför den utförts. Kapitel 2 förklarar de begrepp som kontraktsprogrammering innebär. De förutsättningar som finns för att utföra de mätningar som sker diskuteras i kapitel 3. Utifrån dessa förutsättningar beskrivs de grundläggande tester som utförts i kapitel 4. Dessa tester ligger till grund för den testplan som presenteras i kapitel 5. Det exempel som tagits fram utifrån testplanen finns beskrivet i kapitel 6. De mätningar av exekveringstid som sker i kapitel 8 innefattar även tider för en utanpåliggande loopstruktur som bör räknas bort. Hur dessa beräkningar kan ske redovisas i kapitel 7. Utifrån testplanen analyseras i kapitel 9 de mätningar som genomförts. De problem som stötts på i samband med mätningarna redovisas i kapitel 10. Slutligen följer en sammanfattning av uppsatsen i kapitel 11, där även förslag på vidare forskning finns presenterat.

## 2 Kontraktsprogrammering

Kontrakt kan ses som ett formellt avtal mellan två parter, där varje sida har vissa skyldigheter och rättigheter. Dessa parter benämns oftast som leverantör och klient vilket även används i denna uppsats. Inom kontraktsprogrammering skiljer man på svaga respektive starka kontrakt. Bertrand Meyer är mannen som gav kontraktsprogrammering dess namn[1], men idéerna fanns hos en man vid namn Hoare redan i slutet av 1960-talet[3]. Detta är även vad forskningsgruppen SERG på ämnet datavetenskap på Karlstads universitet forskar i. Forskningsgruppen har utarbetat dokumentet SEMLA[6] som är en designmetod för semantisk beskrivning. Detta kapitel beskriver innebörden av ett designkontrakt samt hur programmering med starka respektive svaga kontrakt fungerar.

## 2.1 Designkontrakt

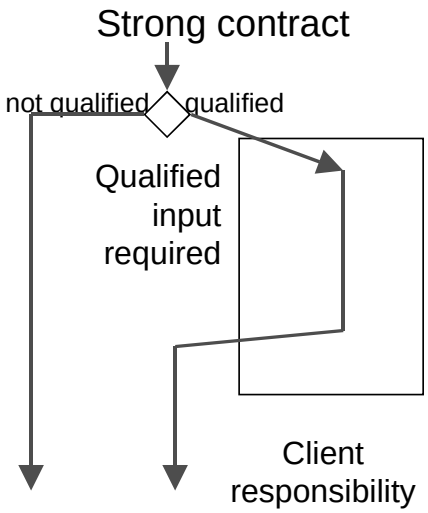
Kontrakt är ett formellt avtal mellan en klient och en leverantör. Klienten och leverantören har båda vissa rättigheter och skyldigheter, detta illustreras i Figur 2.1. De regler som gäller för klient och leverantör beskrivs med så kallade för- och eftervillkor. Förvillkoret talar om vilka skyldigheter en klient har gentemot leverantören, medan eftervillkoret talar om vilka skyldigheter leverantören har gentemot klienten om klienten håller förvillkoret. Det finns olika typer av kontrakt, skillnaden på dessa är att för- och eftervillkoren har olika styrka. Beroende på hur förvillkoren ser ut kan ett kontrakt vara starkt eller svagt.




*Figur 2.1: Rättigheter och skyldigheter för klient och leverantör.*

## 2.2 Starka kontrakt

För programmering med starka kontrakt används ett starkt förvillkor. Detta innebär att förvillkoret skyddar en metod från felaktigt användande, vilket skulle kunna skada objektet som helhet. Eftersom en klient binder sig att använda metoden på rätt sätt minskas antalet tester i metoden. Metoden behöver på så vis inte ta hand om värden som inte har någon mening. Väl inne i metoden antas att förvillkoret är uppfyllt och inga fler tester skall utföras. Eftervillkoret beskriver vad metoden garanterar när den returnerar. Så länge som metoden anropas korrekt så kommer alltid eftervillkoret att uppfyllas av metoden. Figur 2.2 nedan illustrerar möjliga exekveringsvägar[4].




*Figur 2.2: Programflöde för starka kontrakt.*

För att illustrera detta med ett exempel används metoden `top()` på en stack. Denna returnerar värdet på det översta elementet i stacken. För att detta ska vara meningsfullt så krävs det att stacken inte är tom. Används starka kontrakt förbinder sig användaren att kontrollera att det verkligen finns något element som kan returneras från stacken innan anropet till metoden sker. När ett giltigt anrop sker förbinder sig metoden att returnera elementet till användaren. Figur 2.3 visar hur för- och eftervillkoren ser ut för metoden `top()` som användes som exempel ovan.

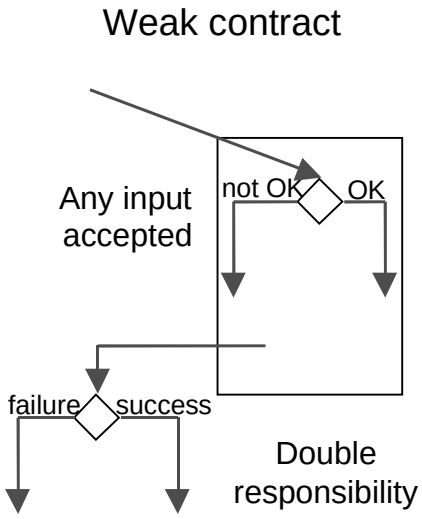
```
// pre:  !isEmpty().
// post: Värdet på översta elementet i stacken har returnerats.
```

*Figur 2.3 För- och eftervillkor för starka kontrakt.*

## 2.3 Svaga kontrakt

Vid programmering med svaga kontrakt kan alltid klienten anropa en metod, det vill säga det finns ingen kontroll på att tillåtna värden för metoden används. I metoden finns istället en eller flera metoder, så kallade undantag som har till uppgift att hantera den typ av fel som kan uppstå i det aktuella fallet. Detta är alltså leverantörens uppgift. Vissa vanliga undantag finns fördefinerade i Java, men möjligheten finns även att skapa egna. Vid implementation av en metod måste ett test utföras för det som kan tänkas gå snett och kasta det undantag som

definerats i metoden. Detta innebär att klienten måste kunna hantera eventuella undantag som kastats. Figur 2.4 nedan illustrerar detta[4].




*Figur 2.4: Programflöde för svaga kontrakt.*

Exemplet som användes för starka kontrakt i kapitel 2.2 skiljer sig vid programmering med svaga kontrakt. Metoden anropas utan att kontrollera om det finns ett element att returnera. Detta innebär att stacken får ansvara för att meddela användaren om när operationen inte är meningsfull. Detta sker vanligtvis i Java genom att ett undantag kastas. Figur 2.5 nedan visar hur för- och eftervillkoren ser ut för exempelmetoden `top()` ovan när svaga kontrakt används.

```
// pre:  True.
// post: Värdet på översta elementet i stacken har returnerats om
stacken ej är tom, annars har ett undantag kastats.
```

*Figur 2.5 För- och eftervillkor för svaga kontrakt.*

# 3 Förutsättningar för mätningarna

För att utföra mätningar av exekveringstid för ett program finns vissa yttre omständigheter som påverkar de uppmätta tiderna. Uppmätta tider beror på vilken maskinvara som används för mätningarna, samt vilken Javamiljö programmet exekveras i. Kompilering av programkoden till bytekod ger också variation i de uppmätta tiderna beroende på val av kompilator. I detta kapitel beskrivs den utrustning som använts och hur denna påverkar de mätningar som utförts. Följderna av detta diskuteras mer ingående senare i rapporten.

## 3.1 Maskinvara

En förutsättning för att kunna göra jämförelser mellan mätningarna är att alla mätningar utförs på samma dator, eller datorer med exakt samma konfiguration för att kunna bortse från eventuella skillnader i konfiguration. För de mätningar som utförts har en dator med följande konfiguration använts:

| |
|---|
| Processor: Pentium II Celeron 466 MHz<br>Internminne: 64 MB sdram PC100<br>Hårddisk: Seagate U8 ( 8 GB )<br>Operativsystem: Windows 98 v 4.10.1998<br>Javaversion: Java 2 SDK Standard Edition v 1.4.0 beta 2 |

*Figur 3.1: Specifikation av dator och programvara.*

Den version av Javamiljön som använts finns att hämta hos Sun[8]. Andra program som använts för de mätningar som utförts är TextPad[5] som är en texteditor för att skriva källkoden. De inbyggda verktyg som finns i Textpad har använts för att kompilera och köra de program som skrivits för att utföra mätningarna. Dessa funktioner är direkt kopplade till den Javaversion som används och har ingen inverkan på tiderna för de mätningar som utförts.

## 3.2 Tidsupplösning

För tidtagningen används ett systemanrop till proceduren `System.currentTimeMillis()`, som returnerar aktuell tid i millisekunder. Det första försök som gjordes var att ta reda på hur exakt tidsupplösningen var på den dator som användes till mätningarna. Det program som

användes för att mäta datorns tidsupplösning finns i bilaga A. Programmet gör upprepade anrop till `System.currentTimeMillis()`. Varje gång värdet som returneras skiljer sig från föregående värde sparas det i en array. Även skillnaden mellan aktuellt anrop och föregående anrop räknas ut och sparas. När tio värden uppmätts avbryts mätningarna och tiderna skrivs ut. En körning av programmet gav följande utskrift.

```
Result: 1007970742900 Difference: 1007970742900
Result: 1007970742960 Difference: 60
Result: 1007970743010 Difference: 50
Result: 1007970743070 Difference: 60
Result: 1007970743120 Difference: 50
Result: 1007970743180 Difference: 60
Result: 1007970743230 Difference: 50
Result: 1007970743290 Difference: 60
Result: 1007970743340 Difference: 50
Result: 1007970743400 Difference: 60
```

*Figur 3.2: Utskrifter för mätning av tidsupplösning.*

Det visade sig att tiden på den dator som användes för mätningarna ändras i intervaller om 50 eller 60 millisekunder. För att kunna bortse från denna variation krävs att de uppmätta tiderna är av en större storleksordning.

## 3.3 Optimering

När koden kompileras till bytekod utförs ingen större optimering. Genom att i kompileringen använda `javac -o` sker en viss optimering. Denna möjlighet har dock utlämnats i de mätningar som utförts. De optimeringar som sker beror i stället på hur den virtuella maskinen arbetar. Traditionellt används Just-In-Time ( JIT ) kompilatorer för Java. En JIT kompilator exekverar bytekoden och kompilerar varje metod första gången den exekveras. Sedan JDK 1.2.2 har emellertid Sun i sin utvecklingsmiljö använt sig av Java HotSpot Performance Engine. Denna miljö användes således för mätningarna. En närmare beskrivning av hur denna fungerar finns i The Java HotSpot VM Technical White Paper[7]. I stället för att kompilera koden metod för metod körs då programmet direkt. Koden analyseras under körningen för att

upptäcka kritiska ”hot spots” i programmet. Detta är de ställen i koden som upptar den största delen av exekveringstiden. När tillräckligt mycket information samlats in om programmets ”hot spots” så sker optimeringen på dessa programdelar. Där sker en kraftig optimering av koden, vilket innebär att den kompileras till maskinkod och metod-”inlining” sker. Detta innebär att koden för de metoder som ofta anropas, kopieras in på det ställe där de används så att inget metodanrop krävs. Detta ger upphov till ändrade exekveringstider för kod som utförs flera gånger. Hur exekveringstiderna påverkas av detta undersöks närmare i kapitel 4 där det undersöks vilket tillvägagångssätt som skall användas för att få stabila mätvärden.

# 4 Mätteknik

För att komma fram till en testplan gjordes en förstudie med en mängd mindre experiment. Detta gjordes för att upptäcka de problem som kan uppstå vid tidmätningar. De första testerna som utfördes var att mäta tider för så små delar av ett program som möjligt. Dessa smådelar refereras fortsättningsvis till som elementarpartiklar. Den uppsättning elementarpartiklar som undersöktes var tilldelning av primitiva datatyper, skapande av nya objekt och liknande. Liknande mätningar har tidigare utförts av Bruce Eckel i hans bok Thinking in Java[2]. Där har han i ett kapitel utfört mätningar av olika elementarpartiklar och presenterat hur dessa förhåller sig till varandra. Det här kapitlet tar upp de undersökningar som utförts för att komma fram till på vilket sätt mätningarna skulle utföras för att erhålla stabila mätvärden. Avslutningsvis diskuteras i kapitel 4.4 de slutsatser som drogs från de mätningar som utförts på elementarpartiklarna samt hur dessa samarbetar.

## 4.1 Loopstruktur

Eftersom mätning av elementarpartiklar tar kortare tid än vad som är mätbart krävs någon form av loop för att få en mätbar exekveringstid. Detta kapitel undersöker hur tiderna för det som ska mätas påverkas av att utföras i en loop.

### 4.1.1 Typ av loop

Ett antal mätningar av olika loopstrukturer genomfördes för att se hur tiderna påverkades av loopens utseende. Strukturerna visade sig ha olika egenskaper, dvs mätning av ett och samma anrop kunde ta olika lång tid beroende på vilken loopstruktur som användes. Att lägga det som ska mätas inuti en loop medför att den uppmätta tiden blir högre än den faktiska tiden.

Loopen medför alltså en overhead på exekveringstiden. Denna loopoverhead kan alltså skilja sig beroende på vilken typ av loop som används. Utifrån de mätningar som utförts visar det sig att en for-loop och en while-loop kan skrivas så att de tar lika lång tid. Figur 4.1 visar utseendet på dessa loopstrukturer.

```
for( int i = 0; i < Antal; i++ )
{
    //Utför testet här
}

int i = 0;
while( i < Antal )
{
    //Utför testet här
    i++;
}
```

*Figur 4.1: Kod för likvärdiga loopstrukturer.*

Eftersom dessa två loopar inte skiljer sig så spelar det ingen roll vilken av dem som används. Den loopstruktur som valdes blev ovanstående for-loop. Fortsättningsvis avses denna for-loop när ordet loop används i uppsatsen.

### 4.1.2 Storlek på loopvariabel

Eftersom det finns ett antal osäkerhetsfaktorer i tidtagningen så bör loopen köras ett stort antal varv. För att kunna avgöra hur många gånger loopen skulle köras för att få ett stabilt mätvärde utfördes mätningar på en tom for-loop. Koden som användes för att utföra dessa mätningar presenteras i bilaga B. Detta program mäter tiden för en tom loop som utförs med olika antal varv. Tabell 4.1 visar de tider som uppmättes vid en exekvering av programmet.

| Antal varv | Uppmätt tid |
|---:|---:|
| 10 | 0 ms |
| 100 | 0 ms |
| 1 000 | 0 ms |
| 10 000 | 0 ms |
| 100 000 | 0 ms |
| 1 000 000 | 60 ms |
| 10 000 000 | 110 ms |
| 100 000 000 | 1 260 ms |
| 1 000 000 000 | 12 360 ms |

*Tabell 4.1: Tider för mätning av loopvariabel.*

För att den variation som finns i datorns tidsupplösning som undersöktes i kapitel 3.2 inte ska påverka de uppmätta tiderna för mycket krävs tider som är några storleksordningar större än denna. Tabellen visar att när loopvariabelns storlek är en miljard är inverkan från tidsupplösningen endast några promille vilket kan bortses från i mätningarna. På grund av detta användes en loopvariabel med denna storlek för övriga mätningar. I de fall där ändringar förekommer förklaras anledningen till detta i det sammanhanget.

## 4.2 Påverkan från övrig kod

För att möjliggöra fler mätningar under samma exekvering av programmet så undersöktes möjligheterna att upprepa tidtagningen. Ett experiment utfördes där flera mätningar lades i sekvens. Dessa mätningar utfördes direkt i en statisk main-metod. Det visade sig att dessa mätningar varierade beroende av exekveringsordningen. Detta innebär att om koden för det som skall mätas placeras nedanför ett annat kodstycke varierar den uppmätta tiden. För att undvika denna effekt utfördes ett test där varje mätning i stället isolerats till en egen metod. Detta test visade att anropsordningen till dessa metoder inte längre hade någon inverkan på tiderna. Inte heller kod som placerats tidigare i den anropande metoden påverkade de uppmätta tiderna. Anrop till flera identiska metoder gav likvärdiga tider. Däremot visade det sig att kod som finns innan metoderna tex instansvariabler påverkar tiderna för mätningarna. På grund av detta lämnas alla instansvariabler mm som behövs för starka respektive svaga villkor kvar även i de körningar där de ej behövs, detta för att den jämförelse som gjorts skall

vara så konsekvent som möjligt. Första gången en metod anropades gav den dock ett högre värde än vid upprepade anrop. Detta kan antas bero på hur den virtuella maskinen arbetar. Eftersom en optimering av kod som ofta utförs sker, så innebär detta att vid upprepade anrop till tidtagningsmetoden så optimeras denna. Detta innebär att de tider som uppmätts i experimenten är optimerade i den virtuella maskinen. Genom att bortse från det första uppmätta värdet och i stället använda övriga optimerade värden kan ett stabilt mätvärde erhållas.

### 4.3 Stabilt mätvärde

Eftersom mätningarna utförs i en flerprocessmiljö, så innebär detta att andra program kan störa mätningarna. För att minimera de yttre omständigheterna så körs inga andra program samtidigt som mätningar sker. Trots detta kan operativsystemet utföra andra handlingar som kan påverka mätningarna. Detta kan resultera i att vissa värden avviker kraftigt från de övriga. Genom att använda ett medianvärde från de mätvärden som erhålls minimeras den inverkan dessa avvikande värden har.

### 4.4 Mätning av elementarpartiklar

Målet med att mäta tiderna för elementarpartiklar var att göra en uppskattning av förhållandet i exekveringstid mellan dessa. Den ursprungliga tanken var att genom att mäta tiden för olika elementarpartiklar kunna uppskatta exekveringstiden för ett större program genom att addera tiderna för de elementarpartiklar som ingår i programmet. De mätningar som utfördes visade dock att om tiderna för flera av dessa elementarpartiklar adderas så kommer de inte att överensstämma med den uppmätta tiden för samma program. I stället för att fokusera på dessa elementarpartiklar kommer denna uppsats fortsättningsvis att undersöka de typiska mekanismer som används vid kontraktsprogrammering genom att analysera ett större exempel.

## 5 Försöksplanering

Genom att undersöka ett större exempel för att se hur tiderna påverkas när vissa saker i koden förändras kan de skillnader som uppstår mellan programmeringsstilarna jämföras. Det exempel som används för att utföra mätningarna simulerar inläggning av ett element i en

stack och beskrivs närmare i kapitel 6. Detta exempel används för att utföra ett antal mätningar där olika mekanismer undersöks. De mekanismer som undersöks används på olika sätt beroende av programmeringsstil. Genom att undersöka de tidsskillnader som uppstår mellan olika mätningar kan skillnader mellan programmeringsstilarna analyseras. I detta kapitel presenteras de mätningar som utförs samt hur dessa ska jämföras för att redovisa de skillnader som finns mellan programmeringsstilarna.

## 5.1 Loopoverhead

För att möjliggöra jämförelser mellan olika mätningar krävs att overheaden för loopen räknas bort. Om det visar sig att den extra tid som loopoverheaden medför är konstant mellan de olika mätningarna kan ett konstant värde räknas bort från den totala tiden. Genom att undersöka hur tiderna påverkas om något upprepas flera gånger varje varv i loopen kan ett teoretiskt värde av loopens overhead beräknas. Dessa beräkningar beskrivs närmare i kapitel 7.3. Om dessa beräknade värden visar sig vara samma oavsett loopens innehåll kan detta räknas bort från de uppmätta tider som erhålls vid varje mätning. För att detta värde ska vara korrekt bör det även motsvara den uppmätta tiden för en tom loop. De experiment som utfördes för att beräkna loopens overhead, samt resultaten från dessa experiment beskrivs i kapitel 7.

## 5.2 Mekanismer som används vid kontraktsprogrammering

För att möjliggöra en jämförelse av programmeringsstilar identifieras i detta kapitel de mekanismer som används för kontraktsprogrammeringen.

### 5.2.1 Starka kontrakt

För starka kontrakt krävs en mekanism för att kontrollera att push-metoden får anropas. Denna mekanism innebär att klienten utför ett anrop till en metod i stacken för att kontrollera de förvillkor som angivits för push-metoden. I detta fall kontrolleras om stacken redan skulle vara full. Beroende av svar så kan klienten anropa push-metoden eller hantera den felsituation som uppstår. Detta styrs med en if-else sats. Om förvillkoren är uppfyllda kan klienten anropa push-metoden. Det enda som sker i push-metoden är att elementet läggs till på stacken. Om förvillkoren inte uppfylls får klienten inte anropa push-metoden utan i stället hanteras felsituationen.

### 5.2.2 Svaga kontrakt

För svaga kontrakt krävs inget test för att få anropa push-metoden. Däremot krävs det att klienten tar hand om eventuella undantag som kastas och därefter hanterar felsituationen. Detta styrs i ett try-catch block. Detta kräver att push-metoden kontrollerar att anropet går att genomföra. Denna kontroll utförs på ett liknande sätt som den kontroll klienten utför vid användande av starka kontrakt. Om anropet är giltigt läggs ett element till på stacken, annars kastas ett undantag som klienten sedan får hantera.

### 5.2.3 Sammanställning av mekanismer

Följande mekanismer används vid kontraktsprogrammering.

- Ett try-catch block används av klienten vid programmering med svaga kontrakt för att utföra anropet och hantera eventuella undantag som metoden kastar.
- En if-else sats används av klienten vid programmering med starka kontrakt för att styra programmet så att ett anrop till metoden endast sker om förvillkoren är uppfyllda, eller hantera eventuella felsituationer.
- En if-else sats används inne i metoden vid programmering med svaga kontrakt för att styra programmet så att operationen endast utförs om den är möjlig att utföra, eller att ett undantag kastas vid ett ogiltigt anrop.
- Kastande av undantag sker vid programmering med svaga kontrakt när klienten utfört ett ogiltigt anrop till metoden.
- En kontrollmekanism som kontrollerar om det är möjligt att utföra operationen. För starka kontrakt används denna av klienten, medan för svaga kontrakt används den inne i metoden. Dess uppgift är att kontrollera att stacken inte redan är full.
- För att hantera eventuella fel som uppstår krävs någon typ av mekanism för detta. Denna mekanism har till uppgift att ta hand om de felsituationer som uppstår och hanteras för båda kontraktstyperna av klienten.

## 5.3 Planering för mätning av mekanismer

Detta kapitel tar upp de mätningar som utförts för att kunna göra en jämförelse mellan starka och svaga kontrakt.

### 5.3.1 Direktanrop

För att utföra operationen behöver man i detta fall inte utföra några tester vid anropet eller i metoden. Anropet får alltid ske och kommer alltid att lyckas. Den här mätningen utförs för att

ha en referens till hur lång tid det tar att utföra själva metodanropet. Detta används som underlag vid jämförelsen mellan starka och svaga kontrakt.

### 5.3.2 Mätningar av fullständiga anrop

För en första jämförelse mellan programmeringsstilarna utfördes mätningar av ett progam som skrivits i två varianter, en variant med starka kontrakt och en med svaga kontrakt. Detta för att kunna avgöra vilken programmeringsstil som är effektivast vad gäller exekveringstid. För att se hur de olika stilarna beter sig beroende på antal fel som uppstår utfördes upprepade mätningar för varje programmeringsstil, där antalet fel som inträffade förändrades mellan mätningarna.

### 5.3.3 Skapande av undantag

Vid användande av dynamisk minnesallokering krävs att minne för objekt som inte längre används återlämnas till systemet. I Java sker detta automatiskt med hjälp av garbage collection. När minnet börjar ta slut går en mekanism i gång för att ”rensa upp” och återlämna det minne som inte längre används. I de mätningar som planerades med svaga kontrakt i kapitel 5.3.2 skapas ett nytt undantag varje gång ett fel inträffar. Detta medför att varje gång ett fel inträffar skapas ett nytt objekt, vilket tar minne i anspråk. För att återta minne som använts av dessa objekt så kan garbage collection komma att ske under körningen. Detta styrs automatiskt och påverkar tiden för körningen. Detta gör att tiden för varje mätning varierar beroende av när garbage collectorn går igång. Hur ofta detta sker har inte undersökts närmare då detta varierar beroende på val av Javamiljö. I stället undersöks hur tiden påverkas genom att använda samma undantag genom hela körningen. För att undvika att garbage collection ska påverka tiderna så har ytterligare en mätning utförts med svaga kontrakt. I stället för att skapa nya undantag varje gång ett fel inträffar så har ett undantag skapats som en medlemsvariabel i stackobjektet. Detta används varje gång ett fel uppstår i körningen. Detta gör att inga nya objekt skapas när ett fel uppstår och garbage collectorn går inte igång för att rensa upp gamla undantag.

### 5.3.4 Kontraktsoverhead

För starka kontrakt krävs ett test från klientens sida innan anrop till metoden för att se till att metoden anropas korrekt. Detta test utförs i en if-sats. Svaga kontrakt som alltid tillåter anrop till metoden kräver i stället att klienten tar hand om eventuella undantag som kastas av metoden. Hantering av detta sker i ett try-catch block. Båda dessa stilar innebär en overhead på klientsidan. För att se hur stor overheaden kan tänkas vara för de olika stilarna utförs

mätningar där stacken aldrig kan tänkas bli full. Dessa jämförs sedan med ett direktanrop till samma metod.

### 5.3.5 Testning av instansvariabler

I de fall när något kan gå fel så krävs olika typer av tester för att hantera detta. Beroende på implementation kan dessa tester utföras på olika sätt. För det exempel som används i denna uppsats så krävs det att stackens storlek testas när ett element ska läggas till. Detta kan ske på olika sätt. Vid användande av starka kontrakt krävs ett test innan anrop till push-metoden för att kontrollera att stacken inte är full. Vid användande av svaga kontrakt sker kontrollen i stället inne i metoden. Dessa test kan se ut på olika sätt men ändå ge samma resultat. För att undersöka vilken inverkan testets utseende har på tiden utförs ytterligare mätningar där kontrollens utseende förändrats. Testet för att kontrollera om stacken är full vid användande av svaga kontrakt kan ske direkt eller genom en metod som utför testet. För att undersöka om ett eventuellt metodanrop medför en extra overhead utförs ytterligare en mätning med svaga kontrakt där testningen sker direkt med instansvariablerna.

### 5.3.6 Felhantering

För att undersöka om de mekanismer som hanterar fel har inverkan på exekveringstiden även i de fall när inga fel uppstår har ytterligare några mätningar utförts. I dessa mätningar utför denna mekanism ingenting.

# 6 Implementering

För att se vad som skiljer sig åt mellan de olika programmeringsstilarna används metoden `push( int value )` som exempel. Metoden ska simulera inläggning av ett element i en stack. Detta kan hanteras på olika sätt beroende av programmeringsstil. Det enklaste fallet är om stacken klarar av att lägga in ett oändligt antal element och operationen alltid lyckas. Då krävs inga tester för att se om operationen kan genomföras. I detta fall kan ett direktanrop till metoden göras. Om antalet element som kan läggas in i stacken begränsas så måste ett kontrakt sättas upp för ett anrop till metoden. Kontraktet kan vara antingen starkt eller svagt. För att mäta exekveringstiden för olika typer av anrop har ett program skrivits som utför mätningarna. Programmet som används för att mäta tiderna består av två klasser. En testklass som mäter tiderna och utför anropet till metoden, samt en stackklass som innehåller implementationen för de metoder som används i exemplen.

## 6.1 Testklass

Testklassen består av metoden `time()` som har till uppgift att mäta tiden det tar att utföra operationen `push()` på ett Stackobjekt, se bilaga C. För att mäta tiden för olika typer av anrop ändras denna metod för att motsvara de krav som ställs för de olika typerna av kontrakt. Klassen består även av metoden `reset()` som har till uppgift att bestämma storleken på stacken, samt nollställa antal element och antal fel i stacken. Detta för att kunna styra antalet lyckade försök att lägga till ett element. Denna metod gör det möjligt att kunna använda samma stack för upprepade mätningar istället för att skapa en ny för varje mätning. Tiderna som mäts påverkas även av klassens utseende i övrigt se kapitel 4.2 om påverkan från övrig kod. Testklassen innehåller förutom ovanstående metoder ett antal instansvariabler som används för olika testfall. Instansvariablerna är `Loops` som styr hur många gånger anropet ska genomföras, `m_error` som är en heltalsvariabel som räknas upp varje gång ett fel inträffar, samt `m_stack` som är en instans av Stackklassen. Eftersom testklassen innehåller ett Stackobjekt innebär detta att Stackklassens utseende inte bör förändras mellan körningarna.

## 6.2 Stackklass

Stackklassen innehåller flera olika implementationer av metoden `push()` som används i de tester som utförts, se bilaga D. Implementationen mellan dessa skiljer sig åt och används för olika mätningar. Pushmetodens huvuduppgift är att lägga till ett element i en array. Varje gång ett element läggs till så ökas en heltalsvariabel med ett för att hålla reda på antalet element som lagts in i stacken. För att det ska vara möjligt att upprepa inläggningen av ett element i stacken en miljard gånger så läggs elementet alltid in på första positionen i arrayen. Därmed behöver inte stacken kunna lagra i miljard element rent fysiskt. För att kunna begränsa stackens storlek används heltalsvariabeln `m_size`. Denna variabel anger stackens tänkta storlek. När antalet element som lagts in i stacken är lika stor som sizevariabeln anses stacken vara full och ytterligare försök att lägga till element ska misslyckas. För en stack med obegränsad storlek kan sizevariabeln bortses. För att den anropande metoden ska kunna kontrollera om stacken är full har metoden `isFull()` implementerats.

### 6.2.1 Beskrivning av push-metoder

**`push1()`** : i denna metod läggs ett element till på stacken och stackens storlek ökas med ett. Denna metod används vid direktanrop och vid anrop med svaga kontrakt, då inuti en if-else sats.

**push2()**: den här metoden ser ut som push1() men med den skillnaden att den kan kasta ett undantag. Används för att utföra de mätningar som diskuteras i kapitel 5.3.4 om kontraktsoverhead för svaga kontrakt.

**push3()**:denna metod används för att illustrera svaga kontrakt. Metoden kan därför kasta ett undantag. I detta fall kastas ett nytt undantag varje gång anropet misslyckas, detta sker i else-grenen. I if-grenen läggs ett element till på stacken och stackens storlek ökas med ett.

**push4()**: den här metoden har samma uppgift och ser ut som push3(), men med den skillnaden att metoden kastar samma undantag varje gång ett anrop misslyckas.

**push5()**: den här varianten av push-metoden ser ut och har samma uppgift som push4(), men med den skillnaden att testet i if-grenen för att se om stacken är full ser ut på ett annat sätt än alla föregående varianter av metoden. Här testas om antalet element i stacken är mindre än stackens storlek. I de föregående varianterna testas däremot så att inte antalet element i stacken är lika med stackens maximala storlek.

**push6()**:denna metod ser ut som push5() men med den skillnaden att testet för att kontrollera om stacken är full sker direkt och inte genom ett anrop till en metod som utför anropet.

# 7 Loopoverhead

Detta kapitel utreder beteendet för den loop som används för de mätningar som genomförts. För att kunna beräkna den overhead som loopstrukturen medför krävs ytterligare tester av loopstrukturens egenskaper. Målet med dessa tester är att få fram ett konstant värde som motsvarar overheaden för ett bestämt antal varv som loopen utförs. Detta konstanta värde skulle då motsvara tiden för en tom loop. Mätningar av tiden för en tom loop presenteras i kapitel 7.1. Genom att upprepa det som ska mätas flera gånger inne i loopen kan ett teoretiskt värde för den tomma loopen beräknas. Dessa beräkningar beskrivs i kapitel 7.3. De mätningar som utförts för att användas för beräkningarna beskrivs i kapitel 7.2. Utifrån de beräkningar som erhållits presenteras i kapitel 7.4 slutsatsen om hur loopens overhead kan räknas bort.

## 7.1 Mätning av tom loop

De tider som uppmäts vid en exekvering av det testprogram som används räknar med den overhead som loopstrukturen medför. Tiden för en tom loop har uppmätts för att användas

som underlag för en jämförelse med beräknade teoretiska värden på loopens overhead. Om det visar sig att dessa överensstämmer kan tiden för en tom loop räknas bort från de uppmätta tiderna för alla mätningar. De uppmätta tiderna presenteras i Tabell 7.1 nedan:

| Anrop nr | Uppmätt tid |
|---|---|
| 1 | 15 440 ms |
| 2 | 9 220 ms |
| 3 | 9 180 ms |
| 4 | 9 170 ms |
| 5 | 9 170 ms |

*Tabell 7.1: Tider för tom loop.*

## 7.2 Teoretiskt värde

För att räkna fram ett teoretiskt värde utfördes mätningar på några av de exempel som planerades i kapitel 5.3. Mätningar utfördes där antalet anrop i loopen var ett,två och fyra. Dessa mätningar utfördes på några av de mätningar som presenteras i kapitel 8. De exempel som använts är direktanrop, samt starka och svaga kontrakt som utförts så att de alltid lyckas. Dessa finns beskrivna i kapitel 8.1 och 8.2. Nedanstående kod visar vilka ändringar som gjorts för att utföra dessa mätningar vid direktanrop.

```
for( int i = 0; i < Loops; i++ )
{
          m_stack.push1( i );
          m_stack.push1( i );
}
```

*Figur 7.1: Kod för mätning av två upprepade direktanrop.*

På samma sätt modifieras den kod som finns beskriven för starka och svaga kontrakt i Figur 8.1 och Figur 8.2, där hela den beskrivna programkoden dupliceras så att det utförs två eller fyra gånger. I de tre tabeller som följer nedan presenteras de uppmätta tiderna för dessa mätningar.

| Anrop nr | 1 * push1() | 2 * push1() | 4 * push1() |
|---|---|---|---|
| 1 | 32 350 ms | 40 320 ms | 60 690 ms |
| 2 | 22 020 ms | 33 450 ms | 53 500 ms |
| 3 | 22 030 ms | 33 510 ms | 53 440 ms |
| 4 | 22 020 ms | 33 500 ms | 53 450 ms |
| 5 | 22 030 ms | 33 510 ms | 53 440 ms |

*Tabell 7.2: Tider för upprepade direktanrop.*

| Anrop nr | 1 * push1() | 2 * push1() | 4 * push1() |
|---|---|---|---|
| 1 | 47 190 ms | 82 990 ms | 156 600 ms |
| 2 | 41 020 ms | 71 240 ms | 135 450 ms |
| 3 | 41 030 ms | 71 240 ms | 135 500 ms |
| 4 | 41 030 ms | 71 230 ms | 135 500 ms |
| 5 | 40 980 ms | 71 240 ms | 135 500 ms |

*Tabell 7.3: Tider för upprepade anrop med starka kontrakt - samtliga lyckade.*

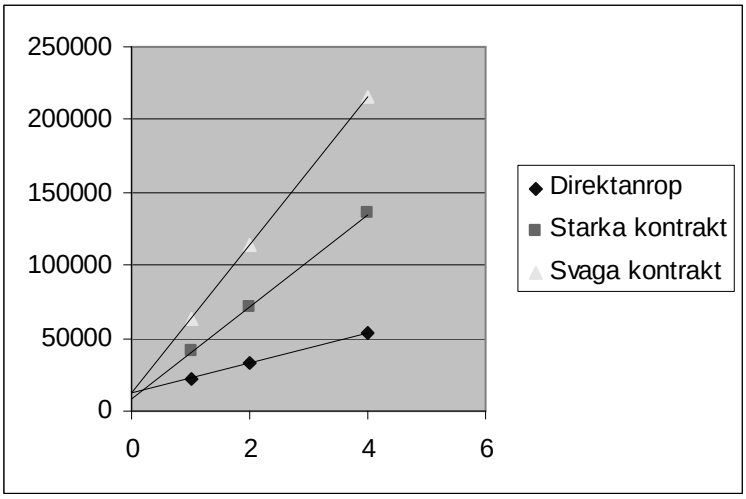
| Anrop nr | 1 * push3() | 2 * push3() | 4 * push3() |
|---|---|---|---|
| 1 | 63 060 ms | 114 200 ms | 216 240 ms |
| 2 | 63 110 ms | 114 130 ms | 216 300 ms |
| 3 | 63 050 ms | 114 140 ms | 216 240 ms |
| 4 | 63 110 ms | 114 130 ms | 216 240 ms |
| 5 | 63 110 ms | 114 140 ms | 216 080 ms |

*Tabell 7.4: Tider för upprepade anrop med svaga kontrakt - samtliga lyckade.*

## 7.3 Beräkning av teoretiska värden

Genom att undersöka skillnaden i tid mellan olika antal anrop kan tiden för ett anrop utan loopens overhead beräknas. Den ökning som kan utläsas genom att ta tiden när två anrop sker varje loopvarv minus tiden för ett anrop per varv ger den tid det tar att utföra en miljard anrop till metoden utan loopoverhead. För att undersöka om denna ökning håller i sig utfördes också en mätning med fyra anrop inuti loopen. För samtliga körningar beräknades medianvärdet av

alla mätvärden utom det första, se kapitel 4.3 Dessa värden lades in i ett diagram som tre punkter, där diagrammets x-axel representerar totala antalet anrop som utförts och y-axeln den uppmätta tiden. Detta diagram visas i Figur 7.2 nedan.




*Figur 7.2: Diagram över teoretiska värden för loopoverhead.*

Diagrammet visar att de tre punkterna för varje exempel verkar sammanfalla på en rät linje. Detta kan visas genom beräkning av förklaringsgraden, som är ett mått på hur väl punkterna representerar en rät linje. Vid beräkning av förklaringsgraden visar det sig att detta stämmer mycket väl. Förklaringsgraden $R^2$ är för alla tre kurvorna över 99%. Detta tyder på att antagandet om att mätvärdena ligger på en rät linje är korrekt. Den räta linjen ger följande ekvation:

$y = ax + b$

y = den totala uppmätta tiden

x = antal anrop till metoden varje varv i loopen

a = tiden för en miljard metodanrop

b = loopoverhead för det specifika anropet.

Linjens skärningspunkt med y-axeln representerar då tiden för loopens overhead. Kurvans lutning beskriver ökningen i tid för att utföra anropen till metoden. För att se om skärningspunkten med y-axeln gäller för samtliga metoder eller om den skiljer sig beroende

av loopens innehåll beräknas detta för samtliga mätningar som utfördes i kapitel 7.2. Den räta linje som beräknats för de tre exemplen visas i Tabell 7.5 nedan.

| | | |
|---|---|---|
| Direktanrop | y = 10401 x + 12055 | $R^2$ = 0,9987 |
| Starka kontrakt | y = 31583 x + 8895 | $R^2$ = 0,9998 |
| Svaga kontrakt | y = 51045 x + 12058 | $R^2$ = 1,0000 |

*Tabell 7.5:Beräkning av räta linjens ekvation för de olika mätningarna.*

## 7.4 Slutsatser

Från Tabell 7.5 kan skärningspunkten med y-axeln för de olika linjerna utläsas. Denna representerar loopoverheaden för de olika mätningarna. Det visar sig att skärningspunkterna för de olika linjerna avviker från varandra. Detta innebär att loopens overhead inte kan ses som ett konstant värde. Beräkningen av loopoverheaden för ett direktanrop gav en avvikelse på 30% från det uppmätta värdet för en tom loop. Detta gällde även för svaga kontrakt, medan beräkningen för starka kontrakt avvek med cirka 5%. Detta tyder på att för en grov uppskattning av loopens overhead kan ändå värdet för den tomma loopen användas. Om ett mera exakt värde krävs så måste detta beräknas för varje mätning. Fortsättningsvis kommer ändå ett konstant värde för loopens overhead att användas i beräkningarna. Det värde som används är 9,2 sekunder vilket är det värde som uppmätts för en tom loop. I vissa jämförelser i kapitel 9 ligger tiderna mellan de saker som jämförs nära varandra. Detta gör att den osäkerhet som uppstår när ett konstant värde används kommer att ge en stor osäkerhet i jämförelsen.

# 8 Mätningar

Detta kapitel tar upp de mätningar som utförts utifrån den planering som gjordes i kapitel 5.3 för jämförelse av kontraktstyperna. Ingen analys av mätvärdena sker i detta kapitel utan detta sker i kapitel 9. För de mätningar som utförts har medianen beräknats på alla värden utom det första på grund av de observationer som gjorts i kapitel 4. Från detta värde räknades även loopoverheaden bort enligt förslag i kapitel 7.4. Även detta beräknade värde presenteras i tabellerna.

## 8.1 Direktanrop

Tabell 8.1 nedan visar de uppmätta tiderna för en mätning av programmet där ett anrop till metoden `push1()` utfördes utan tester. Detta ska representera tiden för ett direktanrop till metoden. Eftersom stackens storlek inte har någon inverkan vid direktanrop så kommer aldrig ett fel att inträffa. Detta innebär att endast en mätning är nödvändigt för direktanropet.

| **Anrop nr** | **0% fel** |
|---|---|
| 1 | 32 350 ms |
| 2 | 22 020 ms |
| 3 | 22 030 ms |
| 4 | 22 020 ms |
| 5 | 22 030 ms |
| **Beräknad tid** | **12 825 ms** |

*Tabell 8.1: Tider för direktanrop.*

## 8.2 Fullständiga anrop med kontrakt

Om stackens storlek begränsas krävs ett kontrakt för att kontrollera varje försök att lägga till ett element på stacken. Med starka kontrakt sker alla förändringar på klientsidan, medan svaga kontrakt kräver förändringar både på klientsidan och på leverantörssidan. Detta kapitel beskriver hur anropen utförts samt de uppmätta tiderna för respektive kontraktstyp.

### 8.2.1 Starka kontrakt

Tabell 8.2 nedan visar de tider som uppmätts när metoden `push1()` anropades med starka kontrakt. För att kontrollera att förvillkoret är uppfyllt används metoden `isFull()` som returnerar `true` om stacken redan är full. Anropet som användes för detta visas i Figur 8.1 nedan.

```
if( !isFull() )
{
    m_stack.push1( i );
}
else
{
    m_error++;
}
```

*Figur 8.1: Kod för anrop med starka kontrakt.*

För att kontrollera antalet fel som inträffar vid varje anrop till tidtagningsmetoden så ändras metoden `reset()` mellan varje körning av programmet. Varje kolumn i tabellen visar en körning av programmet. Vid första körningen av programmet initierades stackens storlek till en miljard. Därefter minskades den med 250 miljoner mellan varje körning.

| **Anrop nr** | **0% fel** | **25% fel** | **50% fel** | **75% fel** | **100% fel** |
|---|---|---|---|---|---|
| 1 | 47 190 ms | 43 720 ms | 40 050 ms | 36 300 ms | 32 680 ms |
| 2 | 41 020 ms | 37 240 ms | 33 450 ms | 29 660 ms | 25 920 ms |
| 3 | 41 030 ms | 37 240 ms | 33 440 ms | 29 720 ms | 25 930 ms |
| 4 | 41 030 ms | 37 240 ms | 33 510 ms | 29 660 ms | 25 870 ms |
| 5 | 40 980 ms | 37 240 ms | 33 450 ms | 29 660 ms | 25 920 ms |
| **Beräknad tid** | **31 825 ms** | **28 040 ms** | **24 250 ms** | **20 460 ms** | **16 720 ms** |

*Tabell 8.2: Tider för anrop med starka kontrakt.*

### 8.2.2 Svaga kontrakt

Tidtagningen för svaga kontrakt utfördes på motsvarande sätt som för starka kontrakt ovan. Svaga kontrakt kräver dock att metoden som anropas hanterar de fel som kan uppstå och kastar ett undantag om ett fel uppstår. Därför anropades i stället metoden `push3()` som utför detta. Hanteringen av undantag sker i ett `try-catch` block. Nedanstående kod visar hur anropet utfördes med starka kontrakt.

```
try
{
    m_stack.push3( i );
}
catch( Exception e )
{
    m_error++;
}
```

*Figur 8.2: Kod för anrop med svaga kontrakt.*

Mätningar utfördes även i detta fall upprepade gånger med olika storlek på stacken för att mäta tiderna för olika antal fel som uppstår under en körning. Tabell 8.3 presenterar de tider som uppmättes vid de fem körningar som utfördes.

| **Anrop nr** | **0% fel** | **25% fel** | **50% fel** | **75% fel** | **100% fel** |
|---|---|---|---|---|---|
| 1 | 63 060 ms | 4 302 980 ms | 8 549 650 ms | 12 798 630 ms | 17 368 970 ms |
| 2 | 63 110 ms | 4 293 140 ms | 8 546 570 ms | 12 798 520 ms | 17 357 560 ms |
| 3 | 63 050 ms | 4 299 900 ms | 8 545 860 ms | 12 800 820 ms | 17 360 630 ms |
| 4 | 63 110 ms | 4 299 790 ms | 8 545 640 ms | 12 781 440 ms | 17 375 080 ms |
| 5 | 63 110 ms | 4 292 920 ms | 8 545 520 ms | 12 803 960 ms | 17 361 450 ms |
| **Beräknad tid** | **53 910 ms** | **4 287 265 ms** | **8 536 550 ms** | **12 790 470 ms** | **17 351 840 ms** |

*Tabell 8.3: Tider för anrop med svaga kontrakt.*

## 8.3 Undantag

För att mäta den tid det tar om samma undantagsobjekt används vid användande av svaga kontrakt krävs att en annan push-metod används. Metoden `push4()` skiljer sig från `push3()` endast på så sätt att samma undantagsobjekt returneras varje gång ett fel uppstår. Även koden på klientsidan behåller samma utseende bortsett från att anropet sker till en annan push-metod. Koden för anropet visas i Figur 8.3 nedan.

```
try
{
    m_stack.push4( i );
}
catch( Exception e )
{
    m_error++;
}
```

*Figur 8.3: Kod för användande av samma undantagsobjekt.*

Mätningar utfördes även i detta fall upprepade gånger med olika storlek på stacken för att mäta tiderna för olika antal fel som uppstår under en körning. Tabell 8.4 presenterar de tider som uppmättes vid de fem körningar som utfördes.

| **Anrop nr** | **0% fel** | **25% fel** | **50% fel** | **75% fel** | **100% fel** |
|---|---|---|---|---|---|
| 1 | 45 420 ms | 200 370 ms | 355 260 ms | 510 150 ms | 665 210 ms |
| 2 | 45 370 ms | 195 370 ms | 347 070 ms | 470 490 ms | 611 650 ms |
| 3 | 45 420 ms | 195 320 ms | 346 040 ms | 469 830 ms | 611 260 ms |
| 4 | 45 370 ms | 195 370 ms | 344 760 ms | 469 830 ms | 611 270 ms |
| 5 | 45 420 ms | 195 370 ms | 344 770 ms | 469 840 ms | 619 390 ms |
| **Beräknad tid** | **36 195 ms** | **186 170 ms** | **336 205 ms** | **460 635 ms** | **602 260 ms** |

*Tabell 8.4: Tider för användande av samma undantagsobjekt.*

## 8.4 Kontraktsoverhead

De mekanismer som används på klientsidan skiljer sig beroende av kontraktstyp. Vid starka kontrakt används en if-sats för att kontrollera om anropet får utföras eller om en felsituation uppstått. För att undersöka denna mekanism användes den kod som visas i Figur 8.4. Genom att använda metoden `isFull2()` som alltid returnerar falskt innebär detta att stacken aldrig ses som full och ett anrop alltid är tillåtet.

```
if( !isFull2() )
{
    m_stack.push1( i );
}
else
{
    m_error++;
}
```

*Figur 8.4: Kod för anrop med starka kontrakt till en stack som aldrig blir full.*

Svaga kontrakt använder sig av ett try-catch block för att kontrollera om metoden som anropas kastar ett undantag. För att undersöka denna mekanism användes den kod som visas i Figur 8.5 nedan. Genom att anropa metoden `push1()` innebär detta att undantag aldrig kastas och stacken kan ses att vara av oändlig storlek.

```
try
{
    m_stack.push1( i );
}
catch( Exception e )
{
    m_error++;
}
```

*Figur 8.5: Kod för anrop med svaga kontrakt till en stack som aldrig blir full.*

För båda mätningarna av kontraktsoverhead kommer aldrig ett fel att inträffa. Detta innebär att endast ett fall för varje kontraktstyp är möjligt. De tider som uppmättes vid mätningarna av kontraktsoverhead presenteras i Tabell 8.5 nedan.

| Anrop nr | Starka kontrakt | Svaga kontrakt |
|---|---|---|
| 1 | 39 880 ms | 34 440 ms |
| 2 | 26 920 ms | 34 380 ms |
| 3 | 26 910 ms | 34 440 ms |
| 4 | 26 970 ms | 34 380 ms |
| 5 | 26 910 ms | 34 440 ms |
| **Beräknad tid** | **17 715 ms** | **25 210 ms** |

*Tabell 8.5: Tider för mätningar av kontraktsoverhead.*

## 8.5 Instansvariabler

De tester av stackens storlek som tidigare utförts har skett genom ett anrop till metoden `isFull()` som kontrollerar om antalet element är lika många som stacken kan rymma totalt. Detta kapitel mäter tiderna för de övriga varianter som kan användas för att utföra detta test.

### 8.5.1 Förändrad metod för jämförelse av instansvariabler

Starka kontrakt använder sig av metoden `isFull()` som sedan negeras för att kontrollera att förvillkoret för push-metoden är uppfyllt. Genom att i stället använda metoden `isNotFull()` för att testa förvillkoret förändras koden för anrop med starka kontrakt på det sätt som visas i Figur 8.6 nedan.

```
if( isNotFull() )
{
    m_stack.push1( i );
}
else
{
    m_error++;
}
```

*Figur 8.6: Kod för ändrad jämförelse av instansvariabler för starka kontrakt.*

Mätningar av detta utfördes när samtliga anrop lyckades, samt när inga anrop lyckades. De tider som erhölls från dessa mätningar presenteras i Tabell 8.6 nedan tillsammans med de mätningar med svaga kontrakt som använde samma typ av kontroll och förklaras nedan.

Liksom för starka kontrakt har metoden `isFull()` tidigare används för svaga kontrakt. Detta för att kontrollera när ett undantag ska kastas. Genom att anropa metoden `push5()` för att lägga till ett element på stacken sker kontrollen av när ett undantag ska kastas med hjälp av metoden `isNotFull()`. Koden för anropet med svaga kontrakt beskrivs i Figur 8.7 nedan.

```
try
{
    m_stack.push5( i );
}
catch( Exception e )
{
    m_error++;
}
```

*Figur 8.7: Kod för ändrad jämförelse av instansvariabler för svaga kontrakt.*

Även för svaga kontrakt utfördes mätningar när samtliga anrop lyckades, samt när inga lyckades. Resultatet av mätningarna presenteras i Tabell 8.6 nedan.

| | **Starka kontrakt** | | **Svaga kontrakt** | |
|---|---|---|---|---|
| **Anrop nr** | **0% fel** | **100% fel** | **0% fel** | **100% fel** |
| **1** | 48 220 ms | 34 490 ms | 44 430 ms | 665 090 ms |
| **2** | 35 810 ms | 28 010 ms | 44 430 ms | 634 990 ms |
| **3** | 35 760 ms | 28 070 ms | 44 440 ms | 633 570 ms |
| **4** | 35 860 ms | 28 070 ms | 44 380 ms | 633 670 ms |
| **5** | 35 760 ms | 28 060 ms | 44 430 ms | 633 680 ms |
| **Beräknad tid** | **26 585 ms** | **18 865 ms** | **35 230 ms** | **624 475 ms** |

*Tabell 8.6: Tider för förändrad jämförelse av instansvariabler.*

### 8.5.2 Direkt användande av instansvariabler vid jämförelse med svaga kontrakt

De kontroller som utförs för att kontrollera om ett element kan läggas till på stacken sker för svaga kontrakt inne i stackobjektet. Detta ger möjligheten att direkt jämföra instansvariablerna för att utföra kontrollen. Metoden `push6()` implementeras på detta sätt. Figur 8.8 nedan visar den kod som används för anropet.

```
try
{
    m_stack.push6( i );
}
catch( Exception e )
{
    m_error++;
}
```

*Figur 8.8: Kod för anrop med direkt jämförelse av instansvariabler.*

Mätningar av detta utfördes när samtliga anrop lyckades, samt när inga anrop lyckades. De tider som erhölls från dessa mätningar presenteras i Tabell 8.7 nedan.

| **Anrop nr** | **0% fel** | **100% fel** |
|---|---:|---:|
| 1 | 57 620 ms | 896 110 ms |
| 2 | 57 620 ms | 877 430 ms |
| 3 | 57 610 ms | 877 330 ms |
| 4 | 57 570 ms | 877 600 ms |
| 5 | 57 610 ms | 878 640 ms |
| **Beräknad tid** | **48 410 ms** | **868 315 ms** |

*Tabell 8.7: Tider för direkt användande av instansvariabler för jämförelse.*

## 8.6 Felhantering

De mätningar som tidigare utförts har använt en godtyckligt vald mekanism som utförts när ett fel inträffar. För både starka och svaga kontrakt har en heltalsvariabel räknats upp för att det ska vara möjligt att utläsa hur många felaktiga försök att anropa push-metoden som utförts varje körning. De mätningar som utförs i detta kapitel utesluter denna mekanism. Koden för den mätning som utfördes med starka kontrakt visas i Figur 8.9 nedan.

```
if( !isFull() )
{
    m_stack.push1( i );
}
else
{
}
```

*Figur 8.9: Kod för anrop med utelämnad felhantering, starka kontrakt.*

For svaga kontrakt modifierades koden på ett liknande sätt. Den kod som användes visas i Figur 8.10 nedan. För denna mätning användes ett anrop till `push4()` vilket måste tas hänsyn till när dessa mätningar används för jämförelser.

```
try
{
    m_stack.push4( i );
}
catch( Exception e )
{
}
```

*Figur 8.10: Kod för anrop med utelämnad felhantering, svaga kontrakt.*

De mätningar som utfördes i detta kapitel används endast för att undersöka hur den mekanism som används vid felhanteringen påverkar tiden i de fall när inga fel uppstår. De mätvärden som presenteras i Tabell 8.8 är de tider som uppmätts i de fall när inga fel inträffar.

| **Anrop nr** | **Starka kontrakt** | **Svaga kontrakt** |
|---|---:|---:|
| 1 | 47 620 ms | 43 780 ms |
| 2 | 38 830 ms | 43 770 ms |
| 3 | 38 840 ms | 43 720 ms |
| 4 | 38 830 ms | 43 780 ms |
| 5 | 38 830 ms | 43 720 ms |
| **Beräknad tid** | **29 630 ms** | **34 545 ms** |

*Tabell 8.8: Tider för anrop med utelämnad felhantering.*

# 9 Analys av mätningar

I detta kapitel analyseras de mätningar som utfördes i kapitel 8. Genom att jämföra de tider som erhållits dras slutsatser om hur olika mekanismer påverkar exekveringstiderna för kontraktstyperna. Likheter och skillnader för kontraktstyperna tas också upp. Vid alla jämförelser används det värde där loopoverheaden räknats bort.

## 9.1 Jämförelse av fullständiga anrop

Genom att jämföra resultaten i Tabell 8.3 och Tabell 8.2 med resultatet för direktanrop i Tabell 8.1 visar detta att även när inga fel uppstår ger användande av kontrakt en längre exekveringstid. Tabellerna visar även att användande av svaga kontrakt alltid ger en längre exekveringstid än användande av starka kontrakt. Tabell 8.2 visar även att när fler fel uppstår minskar exekveringstiden för starka kontrakt, medan Tabell 8.3 visar att tiden kraftigt ökar för svaga kontrakt.

## 9.2 Undantag

Genom att jämföra Tabell 8.4 där samma undantagsobjekt används, med Tabell 8.3 där ett nytt undantagsobjekt skapas varje gång ett fel uppstår visar det sig att en stor del av exekveringstiden beror på skapandet av undantagsobjekten. Genom användande av samma undantagsobjekt minskar tiderna kraftigt. De största tidsskillnaderna uppstår när många fel inträffar. Hur stor del av ökningen som beror på garbage-collection som diskuterades i kapitel 3.3 har inte undersökts närmare. Något som är värt att notera är att även i det fall när inga fel uppstår är tiden lägre när samma undantagsobjekt används. Vid jämförelse av Tabell 8.4 och Tabell 8.2 framgår att den tidsvinst som användande av samma undantagsobjekt innebär inte är tillräcklig för att användande av svagt kontrakt ska vara lika effektivt som användande av starkt kontrakt.

## 9.3 Kontaktsoverhead

När de mekanismer som används på klient-sidan för kontroll av anrop till en metod används för en metod där anrop alltid lyckas och inga fel inträffar innebär detta en overhead. Genom att jämföra Tabell 8.5 med Tabell 8.1 kan denna overhead beräknas. Genom att räkna bort tiden för direktanropet från de tider där kontrakt används erhålls tiden för denna overhead.

Resultaten av dessa beräkningar visar att det `try-catch` block som används för svaga kontrakt ger en större overhead än `if-else` satsen som används för starka kontrakt.

## 9.4 Instansvariabler

De mätningar som utförts i Tabell 8.6 använder en förändrad metod för testning av instansvariabler. Dessa kan jämföras med Tabell 8.2 och Tabell 8.4 för att analysera vilken inverkan denna förändring kan ha på exekveringstiden. Det visar sig att tiderna för de förändrade anropen minskade exekveringstiden när inga fel uppstod, men gav en ökning i det fallet när samtliga anrop misslyckades. Detta gäller för båda kontraktstyperna. Den jämförelse som använts i dessa metoder har även använts för en direkt jämförelse vid användande av svaga kontrakt. De uppmätta värdena för detta finns i Tabell 8.7 och kan jämföras med det förändrade metodanropet för svaga kontrakt. Detta visar att metodanropet i sig inte innebär någon overhead, utan i stället en tidsvinst. Detta antas bero på att koden optimeras effektivare när den läggs i en egen metod.

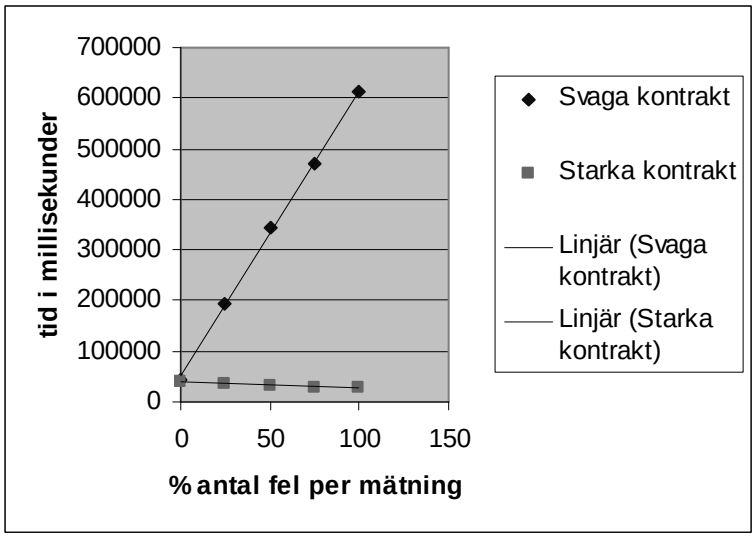
## 9.5 Felhantering

De mätningar som presenterades i Tabell 8.8 har utelämnat den mekanism som används vid felhanteringen. Genom att jämföra dessa tider med de tider där inga fel uppstår i Tabell 8.2 och Tabell 8.4 framgår att tiden ökar när felhanteringsmekanismen är närvarande trots att den inte används. Detta gäller för både starka och svaga kontrakt och antas variera beroende på vilken uppgift felhanteringsmekanismen utför.

## 9.6 Sammanfattning av mätningarna

Eftersom de tester som krävs i den anropande metoden är kostsammare för svaga kontrakt, samt att ytterligare ett test krävs inne i metoden för att kontrollera om anropet är giltigt, medför detta att även om inga fel inträffar så är overheaden som svaga kontrakt medför större än den för starka kontrakt. Ytterligare en sak som talar emot svaga kontrakt är kostnaden för att kasta ett undantag. Även om den största kostnaden i exekveringstid ligger i skapandet av ett undantag så medför även själva kastandet av undantaget en kraftig ökning av exekveringstiden när ett fel uppstår. Om i stället starka kontrakt används så behöver metoden inte ens anropas om det inte är tillåtet. Detta leder till att tiden i stället för att öka vid felhanteringen till och med kan minska. I Figur 9.1 nedan visar diagrammet de uppmätta

exekveringstiderna för starka och svaga kontrakt. Jämförelsen är mellan de ursprungliga mätvärden som uppmättes för starka kontrakt i Tabell 8.2 och de mätvärden som uppmättes för svaga kontrakt i Tabell 8.4. Trots att de mätningar som används för svaga kontrakt använder sig av samma undantag visar diagrammet att starka kontrakt är effektivare i alla lägen.




*Figur 9.1: Diagram över skillnader mellan starka och svaga kontrakt.*

En sak som inte framgår tydligt i diagrammet är att tiden för svaga kontrakt är högre än för starka kontrakt även i det fall när inga fel uppstår. De flesta program som skrivs kräver någon typ av felhantering. Ju mer felhantering som krävs i programmet så ökar betydelsen av valet av kontraktstyp. Om fel ofta uppstår i ett program ger svaga kontrakt en klart försämrad exekveringstid gentemot starka kontrakt.

# 10 Problem

De problem som uppkommit under projektets gång redovisas i detta kapitel.

- Genom att uteslutande använda samma exempel finns vissa aspekter i kontraktsprogrammeringen som inte undersökts. På grund av detta kan resultaten i denna undersökning inte ses som fullständiga, utan fokuserar bara på en del av de mekanismer som används vid kontraktsprogrammering.

- För att de tester som utförts ska vara direkt jämförbara med varandra innebär detta att alla metoder och instansvariabler som använts i de olika experimenten har lämnats kvar i koden vid de övriga mätningarna. Detta medför en overhead på de tider som uppmätts.
- Eftersom det första värdet som uppmätts vid varje mätning skiljer sig från övriga och anledningen till detta inte utretts i detalj är det osäkert vilka mätvärden som ska användas för beräkningarna.
- En metod för att räkna ut loopens overhead har tagits fram, men på grund av tidsbrist användes ett konstant värde för att räkna bort loopens overhead. Detta medför en osäkerhetsfaktor i de beräknade tiderna.
- Anledningen till tidsbristen var att de undersökningar som gjordes för att få en korrekt mätteknik tog längre tid än beräknat.

# 11 Sammanfattning

Målet med detta examensarbete var att undersöka om och när det är effektivare att använda starka respektive svaga kontrakt. De slutsatser som drogs från de mätningar som utförts finns redovisade i kapitel 9. Resultatet av dessa visar att det alltid lönar sig att använda starka kontrakt. Vi har med detta arbete lärt oss att mätning av exekveringstider inte är så enkelt som det låter. Det har visat sig att kod som finns med i programmet men inte används har inverkan på exekveringstiden. Även placeringen av kod i ett program har visat sig ha inverkan på den uppmätta tiden vilket gör att mätning av så kallade elementarpartiklar inte går att genomföra. På grund av dessa erfarenheter har mätning av tider för elementarpartiklar utelämnats. För att kunna utföra dessa mätningar och förstå hur dessa elementarpartiklar samarbetar krävs vidare forskning om hur kompilering och exekvering av program sker.

# Referenser


[1] Bertrand Meyer. *Object-oriented software construction. 2nd edition.* Prentice Hall. 1997.

[2] Bruce Eckel. *Thinking in Java.* 1st edition. Prentice Hall.1998.

[3] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming.* The queen´s University of Belfast, Nothern Ireland. 1969.

[4] Eivind J. Nordby, Martin Blom, Anna Brunström. *Error Management with Design Contracts.* http://www.cs.kau.se/cs/serg/publications.html, 2002-01-16.

[5] Helios Software Solutions. *TextPad – the text editor for Windows.* http://www.textpad.com, 2001-09-10.

[6] Martin Blom, Eivind J. Nordby, Anna Brunström. *Method Description for Semla.* Karlstad University, 2000.

[7] Sun Microsystems. *Java HotSpot™ Technology.* http://java.sun.com/products/hotspot/, 2001-09-10.

[8] Sun Microsystems. *The Source for Java™ Technology.* http://java.sun.com, 2001-09-10.

# A Tidsupplösning

```
class Tidsuppl
{
    public static void main(String[] args)
    {
        final int Reps = 10;
        long result[] = new long[Reps];
        long diff[] = new long[Reps];
        long time, last;
        int rep = 0;
        last = 0;
        while (rep < Reps)
        {
            time = System.currentTimeMillis();
            if (time != last)
            {
                diff[rep] = time - last;
                last = result[rep] = time;
                rep++;
            }
        }

        for (int i = 0; i < Reps; i++)
        {
            System.out.print( "Result: " + result[i] );
            System.out.println( " Difference: " + diff[i] );
        }
        System.out.println();
    }
}
```

# B Loopvariabel

```
class loop
{
    public static void main( String[] args )
    {
        int antal = 1;
        long start, stop, time;

        do
        {
            antal *= 10;
            start = System.currentTimeMillis();
            for( int i = 0; i < antal; i++ )
            {
            }
            stop = System.currentTimeMillis();
            time = stop - start;
            System.out.println( antal + " " + time );
        }while( antal < 1000000000 );
    }
}
```

# C Testklass

```
public class Test
{
    private final int Loops;
    private int m_error;
    private Stack m_stack;

    public Test()
    {
        Loops = 1000000000;
        m_error = 0;
        m_stack = new Stack();
    }

    public void reset()
    {
        m_error = 0;
        m_stack.reset( 0 );
    }

    public long time()
    {
        long start, stop;
        start = System.currentTimeMillis();
        for( int i = 0; i < Loops; i++ )
        {
            //Testa villkor och
            //utför anrop till pushmetoden
        }
        stop = System.currentTimeMillis();
        return stop - start;
    }


    public static void main( String[] args )
    {
        Test t = new Test();
        for( int j = 0; j < 5; j++ )
        {
            t.reset();
            System.out.print( "Tid: " + t.time() );
            System.out.print( " Antal varv: " + t.Loops );
            System.out.println( " Antal fel: " + t.m_error );
        }
        System.exit( 0 );
    }
}
```

# D Stackklass

```
public class Stack
{
    private int m_antal;
    private int m_size;
    private int[] m_array;
    private Exception m_exception;

    public Stack()
    {
        m_array = new int[1];
        m_exception = new Exception();
    }

    public void push1( int tal )
    {
        m_array[0] = tal;
        m_antal++;
    }

    public void push2( int tal ) throws Exception
    {
        m_array[0] = tal;
        m_antal++;
    }

    public void push3( int tal ) throws Exception
    {
        if( !isFull() )
        {
            m_array[0] = tal;
            m_antal++;
        }
        else
        {
            throw new Exception();
        }
    }

    public void push4( int tal ) throws Exception
    {
        if( !isFull() )
        {
            m_array[0] = tal;
            m_antal++;
        }
        else
        {
            throw m_exception;
        }
    }
```

```
    public void push5( int tal ) throws Exception
    {
        if( isNotFull() )
        {
            m_array[0] = tal;
            m_antal++;
        }
        else
        {
            throw m_exception;
        }
    }

    public void push6( int tal ) throws Exception
    {
        if( m_antal < m_size )
        {
            m_array[0] = tal;
            m_antal++;
        }
        else
        {
            throw m_exception;
        }
    }

    public boolean isFull()
    {
        return m_antal == m_size;
    }

    public boolean isFull2()
    {
        return false;
    }

    public boolean isNotFull()
    {
        return m_antal < m_size;
    }

    public void reset( int size )
    {
        m_antal = 0;
        m_size = size;
    }
}
```