DATALOGISK INSTITUT
DET NATURVIDENSKABELIGE FAKULTET
AARHUS UNIVERSITET

Hovedopgave

Diplom i Informationsteknologi
linien i Softwarekonstruktion

Refactoring af integrationssystem til kompositionel design

af

Katarzyna Rodziewicz

15. juni 2010

Katarzyna Rodziewicz,
studerende

Henrik Bærbak Christensen,
vejleder

Datalogisk Institut
Aarhus Universitet
Åbogade 34
8200 Århus N

Tlf.: 89425600
Fax: 89425601
E-mail: cs@cs.au.dk
http://www.cs.au.dk/da

Indholdsfortegnelse

1 Forord

Denne opgave handler om refactoring af et eksisterende system udviklet med hovedvægt på det funktionelle paradigme i c/c++, til et system i OO paradigme i kompositionelt design. Det er ikke ret tit man i dette erhverv får lov til at omskrive et system, som set fra forretningens side fungerer efter hensigten. Derfor er det vigtigt at udnytte denne mulighed for at skrive dette system ved hjælp af ny teknologi/metode for at afdække fremtidige behov bedst muligt. I denne konkrete opgave demonstreres anvendelse af en række teknikker og metoder, for at nå til et overskueligt design. Det nye design skal give mulighed for nemt og hurtigt at udvide med ny funktionalitet og forbedre læsbarheden.

2 Introduktion

Dette kapitel indeholder baggrunden for opgave, forklaring af en række begreber og en kort beskrivelse af de involverede systemer, samt problemformulering/hypotese. Formålet med kapitlet er at forklare, hvad det er for et problem, jeg ønsker at løse og hvilke metoder og tekniker vil jeg anvende til det. Det skal også skabe en forståelse for udgangspunktet for opgaven.

2.1 Baggrund

Jeg vil gerne tage udgangspunkt i mit arbejde, hvor jeg arbejder på et projekt, der går ud på at forny et adaptersystem, som integrer et købesystem med et legacysystem. Det er en typisk integrationsadapter, der kobler to vidt forskellige systemer sammen. Systemerne har forskellige API'er, kommunikationsmønstre, sikkerhedskrav osv. Adaptersystemet synkroniserer dem begge i realtime, hvilket giver mange specielle udfordringer pga. forskellige sikkerhedsmekanismer i de to systemer. Da kravet til adaptersystemet fra starten var meget begrænset, var der ikke anvendt et fremtidsrettet design. Løbende udvidelse af adaptersystemet har gjort det uoverskueligt, og videreudvikling og tilføjelse af nye funktionaliteter har altid medført risiko for fejl i eksisterende funktionalitet.

Der er opstået et meget stort behov for at redesigne og omskrive adaptersystemet, således det kan håndtere eksisterende funktionalitet, nye krav og udvidelser i fremtiden, som jeg ved bliver aktuelle i en overskuelig fremtid. Udvidelser i fremtiden må ikke påvirke eksisterende funktionalitet. Desuden er der en klar mulighed for, at en del funktionalitet kan genbruges af andre systemer/projekter og generaliseres til et selvstændigt framework.

En anden vinkel for projektet er et ønske om at løfte den fra c/c++ til c# og bruge TDD under udviklingen. Det er meget vigtigt at inkludere NUnit test for at projektet giver bedre sikkerhed ved fremtidens udvidelser og hurtigere overblik over ændringers indflydelse på eksisterede funktionalitet.

2.2 Forklaring af systemer og begreber

Der er to systemer der er integreret med adaptersystemet:

- Købesystemet, der er et handelssystem til aktiedealere. Systemet er opbygget af flere moduler, som man kan tilkøbe og på den måde udvide funktionaliteten. API'et er udstillet i c++ og kommunikationen med systemet sker via en protokol kaldet *The Transaction Network Protocol* (TNP). TNP kommunikation er asynkron og eventstyret. Der er en ”*callback*” metode, hvor man modtager beskeder og en ”*send*” metode, hvor man sender beskeder. Konkret funktionalitet er styret af forskellige typer af beskeder.
- Legacysystemet, blev tidligere også brugt som handelssystem, men bruges i dag primært til bogføring. Kommunikation til den forgår igennem Message Queue (MQ) Series. Beskeder sendes i et XML format.

Adaptersystemet, som er et typisk integrationssystem, dækker over funktionalitet til konvertering af data mellem de to, tilpasning af kommunikation og synkronisering i realtime. Det er den, som tage hensyn til forskellen imellem dem og afdækker særlige behov fra systemerne. Fra starten skulle den modtage data fra et system, konvertere data og sende til det andet system. Det har dog vist sig, at der er behov for en hel del forretningslogik omkring data konvertering. Der er væsentlige forskelle i hvordan man behandler forskellige typer af ordre og handler. På den måde voksede adaptersystemet langsomt med flere og flere udvidelser via ”if sætninger”.

2.3 Problemformulering

Jeg har tænkt mig at fremstille en arkitektur for adaptersystemet, der dækker den nødvendig funktionalitet og giver bedre udvidelsesmuligheder. Der skal være mulighed for at udvide den med nye typer af ordrer/handler, uden at påvirke implementering af eksisterende typer. En del af funktionaliteten vil jeg omskrive til et framework, for på den måde at gøre den genbrugelig. Det drejer sig først og fremmest om connection funktionalitet, som kan indkapsles og genbruges. Ud over det vil jeg omskrive det fra c++ til c#.

For at opnå det bedst mulige resultat under processen med refactoring, vil jeg anvende en række teknikker, som skal hjælpe med at opnå den bedste arkitektur for lige dette system. Jeg vil kigge på software kvalitets attributter som modificerbarhed og testbarhed, som jeg mener er de vigtigste for denne refactoring. Det vil jeg gøre for at klargøre fordeler ved det nye design i forhold til det gamle system, hvor der er anvendt et funktionelt paradigme.

For at forbedre processen vil jeg arbejde med Test Driven Development og beskrive erfaringer, fordele og ulemper med at arbejde med TDD.

2.4 Afgrænsning

Da opgaven omfatter hele processen for omskrivning af adaptersystemet – analyse af eksisterede funktionaliteten, ny design og udvikling, vil jeg begrænse mit arbejde til et udvalgt område af systemet. Jeg vil i min opgave vælge en del af systemet og med udgangspunkt i dette, vil jeg fremstille ny arkitektur, samt omskrive denne til et OO paradigme med understøttelse af TDD i processen.

2.5 Proces og forventede resultater

Det grundlæggende mål for denne opgave er at fremstille ny arkitektur og implementation for en del af funktionaliteten i adaptersystemet. Den nye arkitektur skal understøtte arkitektur attributter som modificerbarhed og testbarhed.

Opgaven skal kort beskrive og præsentere adaptersystemets funktionalitet, hvad dens opgaver er og dens ansvarsområder. Der skal præsenteres krav til variabilitet, som er udgangspunktet for en refactoringopgave. Ud fra det præsenterede variabilitetskrav, vil jeg vælge en, som jeg vil arbejde videre med. Med udgangspunkt i det valgte krav, vil jeg skitsere flere arkitekturmuligheder. Jeg vil analysere fordele og ulemper ved de forskellige arkitekturer og vælge en af dem til at arbejde videre med. Jeg vil forsætte med at lave implementation for den arkitektur for det valgte variabilitetskrav. I implementationsfasen vil jeg bruge TDD i processen. Brugen af TDD metoden i implementationsfasen skal hjælpe med at understøtte modificerbarhed og testbarhed i adaptersystemet.

Med det forløb for opgaven, vil jeg med et lille udsnit af adaptersystemets funktionalitet, præsentere hele processen for udvikling af store systemer som dækker: analyse, design, implementation og delvis testfasen.

Jeg forventer, at resultatet jeg kommer frem, til kan bruges til at lave en omfattende refactoring af adaptersystemet. Da jeg arbejder løbende på adaptersystemet på min arbejdsplads, vil jeg bruge mit resultat fra opgaven og overføre det direkte til arbejdet.

3 Analyse

3.1 Kort beskrivelse af eksisterende funktionalitet

Funktionaliteten i adaptersystemet kan man opdeler i tre hovedområder:

1. Kommunikation.

   Adaptersystemet kommunikerer med to systemer – købesystemet og legacysystemet via to forskellige protokoller:

   - MQ – med legacy systemet. Adaptersystemet kommunikerer med en række MQ køer, som man kan opdeler i to grupper: dem som adaptersystemet læser fra og dem som den skriver til.
   - TNP – med købesystemet.

2. Konvertering

   - fra TNP til XML
   - fra XML til MQ

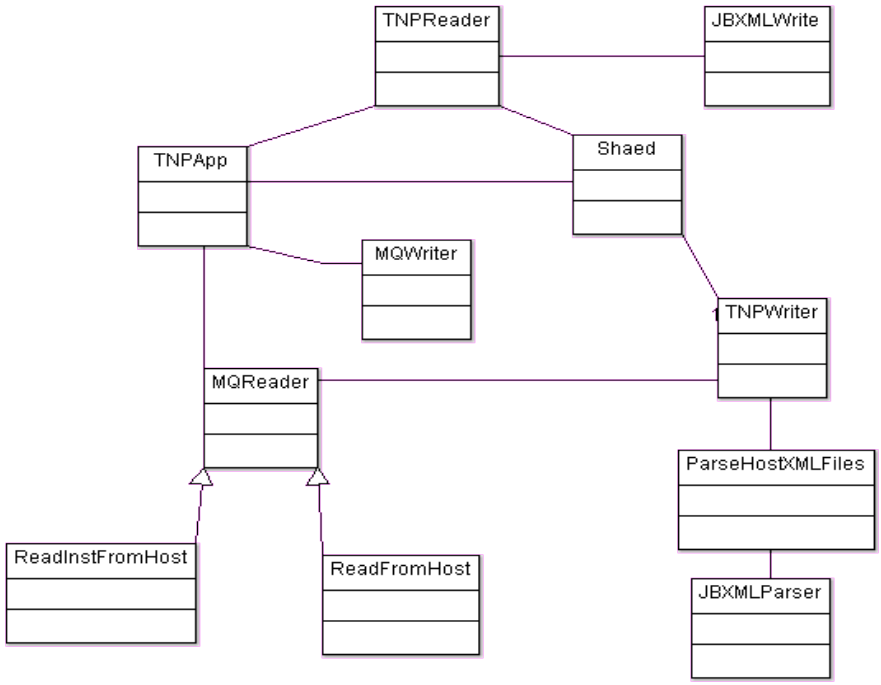
   Konvertering sker direkte fra et til et andet format.

3. Behandling – forretningslogikken.

   Forretningslogikken er flettet sammen med konverteringsfunktionalitet.

3.1.1 Nuværende implementation

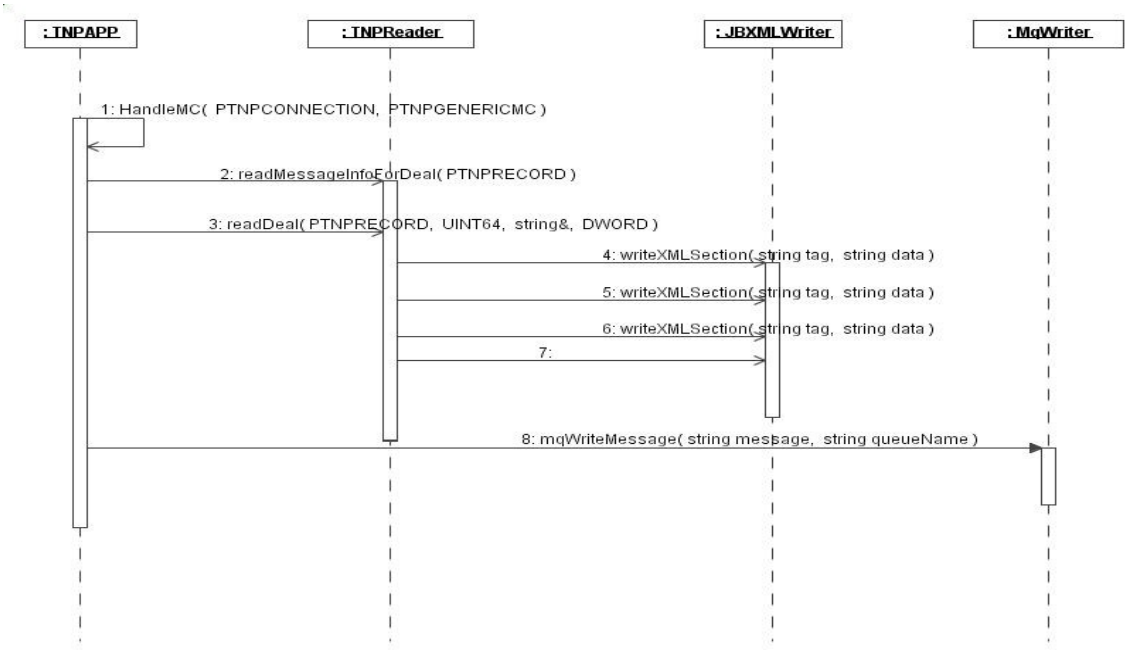
Opdelingen nævnt ovenfor er ikke afspejlet i den nuværende implementation, hvor alle tre områder er blandet sammen i forskellige klasser.



Figur 1:Oversigt over de vigtigste klasser i den nuværende implementation.

Fx har TNPApp klassen ansvar for at etablere forbindelse til købesystemet, modtage message fra købesystemet, implementere dele af forretningslogikken, og sende message til legacysystemet igennem MQ.

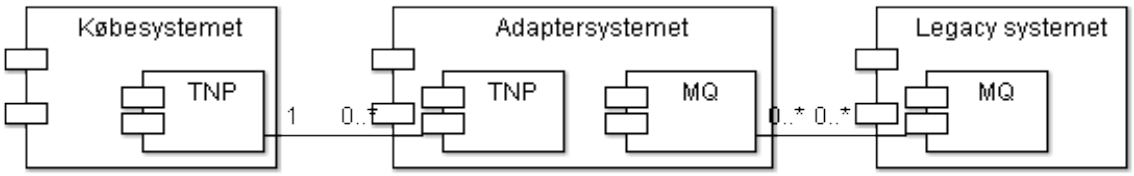
I en forenklet udgave kan man præsentere forløbet fra modtagelse af message fra købesystemet til at skrive på MQ som i nedstående sekvensdiagram:



Figur 2:Sekvensdiagram for afsendelse af message til legacysystemet - nuværende implementering

writeXMLSection metoden på *JBXMLWriter* bliver kaldt for hver variabel/felt for en trade message.

Når man snakker om kommunikation i integrationssystemer er der altid, i deres natur, to kommunikations kanaler/retninger.



Figur 3:Oversigt over komponenter i adaptersystemet opsætning

Adaptersystemet kommunikerer med TNP protokollen med købesystemet og med MQ protokollen med legacysystemet.

Begge protokoller er asynkrone:

TNP er event håndteret, hvor man implementerer en *callback* metode. Ved oprettelse af forbindelse til TNP, giver man pointer til denne metode. Der er sendt ny event til *callback* metoden, hver gang når købesystemet er klar med ny event.

MQ er implementeret med en fire/forget protokol. Dvs. en klientapplikation, som adaptersystemet afleverer en message til MQ køen. Applikation, som modtager message – i denne situation legacysystemet, har ingen kendskab til afsender.

TNP introducerer sin egen datastruktur, med egne metoder til at pakke data ud. Eksempel på opbygning af TNP message vises i Tabel 1. Message skal slette en handel i købesystemet. Den har

et XML parser objekt, som parameter. Man læser værdier direkte fra XML.

```
void deleteDeal(ParseHostXMLFiles*phxml, string name){
        PTNPDELETEDEAL ptnpDeleteDeal;
        PTNPCONNECTION pTnpConnection = g_pTnpConnections[0];
        DWORD dwRet;
        DWORD dwMessageSeqNo;
        HTNPMESSAGE hMessage;
        string matchId = phxml->tryReadTag("MatchId", true);
       string OrdrenrFA = phxml->tryReadTag("OrdrenrSys", true);
        string ordrenrJB = phxml->tryReadTag("OrdrenrJB",  true);
        dwRet = TnpCreateMessage( 0, &hMessage );
        if( dwRet != NO_ERROR )
                throw "writeOrder - TnpCreateMessage - Failed " + intToString(dwRet);
        dwRet = TnpAppendEmptyMessageRecord( &hMessage, RTY_DELETEDEAL,
        sizeof(TNPDELETEDEAL), (PTNPRECORD*) &ptnpDeleteDeal, 0 );
        if( dwRet != NO_ERROR )
                throw "deleteDeal - TnpAppendEmptyMessageRecord - TNPDELETEDEAL -    Failed " +
intToString(dwRet);
        ptnpDeleteDeal->afDeleteDealAttributes = TNP_DELETEDEAL_AF_INTERNAL;
        safestrcpy(ptnpDeleteDeal->szMatchId, matchId.c_str());
        AddBlob(&hMessage, phxml->tryReadTag("FondskodeIntern",         false), RTY_DELETEDEAL);
        AddBlob(&hMessage, ordrenrJB,         RTY_DELETEDEAL);
        AddBlob(&hMessage, phxml->tryReadTag("Levelnr",true), RTY_DELETEDEAL);
        doAppendUserName(&hMessage, userName, RTY_DELETEDEAL);
        dwRet = sendMessage( pTnpConnection->hServer , &hMessage, &dwMessageSeqNo );
        if( dwRet != NO_ERROR )
                throw "deleteDeal - SendMessage - Failed " + intToString(dwRet);
}
```

Tabel 1:Nuværende implementation - eksempel 1

Man bruger TNP strukturen hele vej igennem i systemet, dvs. forretningslogikkens del af adaptersystemet manipulerer data direkte på TNP strukturen. På den måde, er kendskabet til TNP protokollen spredt over hele adaptersystemets implementering.

Til at skrive og læse data fra MQ bruger man beskeder i et XML format. XML, som *string* bliver også sendt igennem systemet. På den måde er den indbygget i alle lag i adaptersystemet. I Tabel 2 vises eksempel på en sådan afhængighed.

```
void addDataForLimitOrder(ParseHostXMLFiles*phxml, TNPORDER*ptnpOrder){
      ptnpOrder->dPrice =StringToDouble_Comma(phxml->tryReadTag("KursLim", false));
    double stk = StringToDouble_Comma(phxml->tryReadTag("StkNomi", true)) -
StringToDouble_Comma(phxml ->
                    tryReadTag("StkNomiRest", false));
          ptnpOrder->dQuantity = stk;
}
```

Tabel 2:Nuværende implementation - eksempel 2

Som man kan se i eksemplerne fra den nuværende løsning, mangler adaptersystemet adskillelse af implementering af de to protokoller fra hinanden og fra forretningslogikken. Det gør det, at hver ændring medfører stor risiko for at introducere fejl i adaptersystemet.

En anden stor ulempe i den nuværende implementering er at man bruger *string* konstanter i adaptersystemet, som er navne på tags i XML filen. Det gør det meget besværligt at ændre XML filen og en ændring medfører risiko for at introducere fejl.

3.2 Krav til variabilitet

Adaptersystemet er et system, som udvides løbende. Der er forskellige interessenter i forhold til variabiliteten i det. På en side er der forretningen, der er i gang med at købe nye komponenter og på den måde opnår den ønskede understøttelse for deres arbejde. På den anden side er der udviklere, som skal sikre understøttelse for nye komponenter i adaptersystemet.

Funktionaliteten i c++ udgaven af adaptersystemet er tæt koblede. Der er ingen afgrænsning imellem forskellige typer af funktionalitet. Det giver store problemer ved ændringer og udvidelser. Hver gang man laver ændringer, medfører det stor risiko for fejl mange andre steder og kræver stor test af hele adaptersystemet. Man kan identificere følgende krav til variabilitet:

1. Nye typer af ordrer/handler

   Købesystemet kan udvides med ny komponent, der svarer til et bestemt marked. Det kan resultere i nye typer ordrer/handler, og nye måder at behandle de nye typer. Det skal være muligt at tilføje dem til systemet uden at påvirke behandlingen af eksisterende typer.

2. Ny forretningslogik

   Forskellig forretningslogik afhængig af kommunikation til et bestemt system. Når man sender en ordre eller en handel fx fra legacysystemet skal de behandles på en anden måde end når de sendes til legacysystemet fra købesystemet. Det skal være nemt at tilføje ny forretningslogik for andre systemer, som er interesserede i data fra købesystemet eller i at sende data til købesystemet. Der skal også være mulighed for at lave ændringer i eksisterende forretningslogik uden at påvirke en stor del af adaptersystemet.

3. Kommunikationsprotokoller og transportlag

   Der kan være behov for fx udskiftning af MQ kø'er med en WebService, som er et stort ønske i fremtidens udvidelser. Det kan også være en udvidelse til adaptersystemet med en WebService til andre interessenter ved siden af eksisterende MQ køer.

3.3 Vælg af variabilitet punkt

Jeg vil i denne opgave fokusere på et af de nævnte variabilitetspunkter og arbejde videre med det, som udgangspunkt. Det er nødvendigt af hensyn til afgrænsning af hovedopgaven, da et grundigt arbejde på alle områder vil være for omfattende. Det område, som jeg synes er mest interessent er kommunikationsprotokollen og transportlaget. Der er flere grunde til at jeg synes at lige det område er meget interessent at arbejde videre med.

- Det er et meget vigtigt område i adaptersystemet, da dens primære opgave er at skabe kommunikation imellem to forskellige verdner – købesystemet på en side og legacysystemet på den anden side.
- Dette område er en oplagt kandidat til at isolere fra resten af funktionalitet. Det er et område,

der eksisterer i mange systemer og problemet er mere generelt og ikke specifik for adaptersystemet. Dvs. løsningen kan være relevant for mange systemer.

- Det er vigtigt at gøre adaptersystemet protokoluafhængig – dvs. isolere det område på den måde for at kunne udskifte kommunikation med en anden eller tilføje flere kommunikationskanaler. Hverken udskiftning eller udvidelse må berøre andre områder i adaptersystem.
- Det er et komplekst område, som omfatter funktionalitet til konvertering af ordrer/handler og er delvis variabilitetspunkt for nye typer af ordrer/handler.
- For opgaven giver det mulighed for at isolere og generalisere den del af funktionaliteten. Det giver frihed fra forretningsspecifik funktionalitet, som ofte er afhængig af bankens forretningspolitik. Jeg tror, at valg af netop dette område, giver mig bedst mulighed for at koncentrere mig om softwarekonstruktion i stedet for forretning.

4 Design

Som det er beskrevet i afsnittet ”Nuværende implementation”, har vi her to kommunikation protokoller – en mod købesystemet og en mod legacysystemet.

Kommunikation mod købesystemet er ”et til et system” kommunikation. Adaptersystemet skal håndtere et købesystem med et bestemt API. API'et mod købesystemet er en statisk faktor, der ikke forventes at ændre sig (medmindre man udskifter hele købesystemet eller leverandøren laver radikale ændringer, der alligevel vil medføre store ændringer i adaptersystemet). Den del af adaptersystemet skal isoleres, således at alle TNP relateret referencer er indkapslet.

Kommunikation mod legacysystemet har tilgengæld en række variabilitets krav. Der kan være behov for kommunikation mod flere legacysystemer, behov for at understøtte flere kommunikationsprotokoller og behov for at understøtte forskellige kommunikationskanaler under den samme protokol. Da den del af adaptersystemets kommunikation kræver stor variabilitet og en af de hovedattributter, som jeg vil arbejde med er modificerbarhed, vil jeg arbejde med refactoring på den del af adaptersystemet.

Modificerbarhed i den del af adaptersystemet er et mål for, hvor nemt det er, at udskifte en protokol med en anden eller at tilføje en ny protokol, samt hvor få steder der bliver berørt af den type ændringer.

Da skrivning og læsning fra MQ køer er håndteret meget anderledes og har vidt forskellige procedurer, vil jeg i første omgang tage mig af den del af kommunikation, som sender data til legacystemer.

4.1 Metoden

Jeg vil arbejde med arkitektur prototyper, der i forenklede udgaver afprøver en ny løsning.

Arkitektur prototyper [3] ser ud til at være den helt rigtige metode at bruge i min opgave. I arbejdet med dem, eksperimenterer man med den valgte arkitektur. Man implementerer den i en forenklet udgave, hvor man holder forretningslogikken udenfor prototypen. Den forenklede implementering evalueres i forhold til kvalitetsattributter og risici for videre implementering af arkitektur.

For at mindske kompleksitet vil jeg ikke implementere hele den nødvendige integration for adaptersystemets kommunikationsfunktionalitet, men koncentrere mig om opkobling fra adaptersystemmet til legacysytemet. Både prototypen og adaptersystemet tager kun hensyn til implementering set fra adaptersystemets side. Jeg forholder mig altså ikke til, hvad der sker i implementering i legacysystemet.

Jeg vil bygge små prototyper med udsnit af kommunikationsfunktionalitet. Jeg vil gøre det i flere iterationer. Første iteration vil tage udgangspunkt i den nuværende implementation og de næste tage udgangspunkt i de forrige iterationer. På den måde vil jeg gennemgå gradvise ændringer fra den nuværende implementation til en ny arkitektur. Under hver itaration vil jeg gennemgå fordele og ulemper for den bestemte løsning.

4.2 Iteration 1

I første step i refactoring af adaptersystemet er målet at isolere den del, som har med skrivning til legacy systemet at gøre. Her snakker vi om selve oprettelsen af forbindelse og isolering af den datastruktur, som skal sendes til legacy systemet.

For at opnår dette mål vil jeg bruge 3-2-1 principper [2]:

- identificere de områder i applikation, som skal være modificerbare og er variable
- programmer mod interface i stedet for konkret implementation
- brug delegation i stedet for nedarvning

4.2.1 Arkitektur

Der er to hovedområder, som er variable og som skal isoleres:

- kommunikationsprotokollen, kommunikationskanalen
- datastruktur, som skal sendes eller modtages og dens konvertering

4.2.2 Løsning for kommunikation protokollen

Variabiliteten for kommunikation isoleres i *Writer* interfacet. Klientapplikationen er ikke interesseret i at vide hvordan information bliver sendt og hvordan hele kommunikationen oprettes. Den har kun brug for en metode *sendMessage,* derfor er *Writer* interface ret enkelt:

```
public interface Writer{
	void sendMessage(Trade info);
}
```

Data, der afgør, hvilken protokol eller til hvilket system den skal til er opsamlet i ordrer/handler, som sendes fra købesystemet igennem adaptersystemet/klienten. Den vil jeg kalde *Trade* og forenkle den for prototypens formål. Den får kun de variabler, som er relevante for den del af funktionaliteten.

```
public class Trade{
	private string id;
```

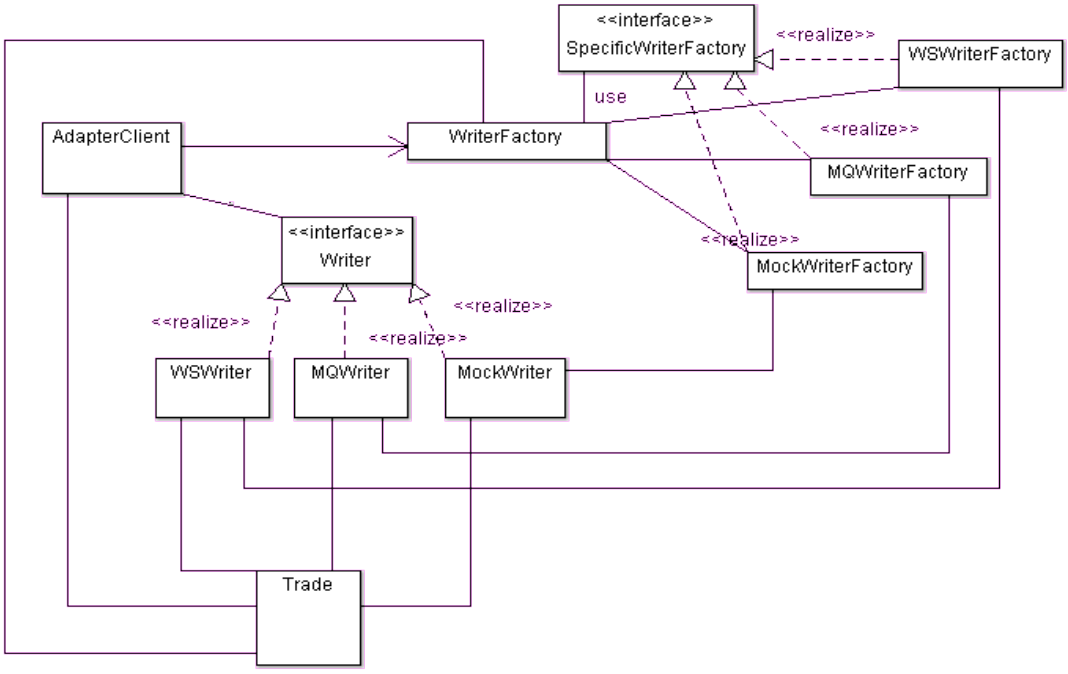
```
    private string systemUserName;
    private string tradsMarked;

    public string ID {
     set{ id = value;}
     get{ return id;}
   }
    public string SystemUserName{
        set { systemUserName = value; }
        get { return systemUserName; }
    }
    public string TradsMarked {
        set { tradsMarked = value; }
        get { return tradsMarked; }
    }
}
```

Ud fra variabilitetskrav for skrivningskommunikation, kan man identificere to abstraktionsniveauer:

- valg af kommunikationsprotokol. Logikken til hvilken kommunikations protokol man vælger afhænger af SystemUserName, der indikerer hvilken kommunikationsprotokol skal der anvendes.
- valg af kommunikationsforbindelse/kanal – det kan være forskellige MQ køer eller WebServices. I forbindelse med MQ Series, vil der fx være mulighed for at sende *Trade* til forskellige køer afhængig af, hvilket marked ordrer/handler kommer fra: fx tilhører danske aktier en INT markedsserver, der håndterer kommunikation med Københavns Fondsbørs.

For at holde implementeringen enkel, er der behov for at adskille de to abstraktionsniveauer.



Figur 4:Arkitektur for løsning for at finde frem til den rigtigt kommunikationsforbindelse for skrivning til legacy sytemet

Det første abstraktionsniveau isoleres i en *WriterFactory.* Den har ansvaret for at afgrænse og indkapsle funktionaliteten, som afgør hvordan en transaktion skal sendes. Den instantierer ikke en ny *Writer,* men vælger den relevante konkrete *Factory* og uddelegerer opgaven til den.

Det andet abstraktionsniveau isoleres med *Factory Method Pattern* [10]. Hver konkrete *Factory* implementerer *SpecificWriterFactory,* som definerer en *create* metode *createWriter(Trade)*. Den konkrete *Factory* har ansvar for den del af funktionaliteten, der afgør den konkrete forbindelse. Hver kommunikationsprotokol har sin egen *Factory,* som bestemmer hvilken konkrete *Writer* der skal produceres og returneres til *WriterFactory.*

4.2.3 Løsning for datakonvertering

For at sende *Trade* message kræver det at *Trade* objektet er konverteret til et format, der er forståeligt for modtagersystemet (legacy systemet). Det præcise format kan være afhængig af hvilket legacysystem man skal sende message til og hvilken tilstand den er i – er det hele message man skal sende eller er det kun kvittering. Konvertering af *Trade* objektet til XML format hænger sammen med hvilken kommunikationskanal man skal sende message til.

Ud fra det kan man konkludere at datakonvertering:

- hænger sammen med *Writer*
- skal konvertere til flere forskellige XML formater,
- der skal bruges den samme logik til at finde data format og kommunikationskanal

Variabilitet for forskellige XML formater indkapsles i en interface *Converter.* Den skal have to metoder: en til at konvertere data til XML og en anden til at konvertere indkommende data fra XML til *Trade* objektet.

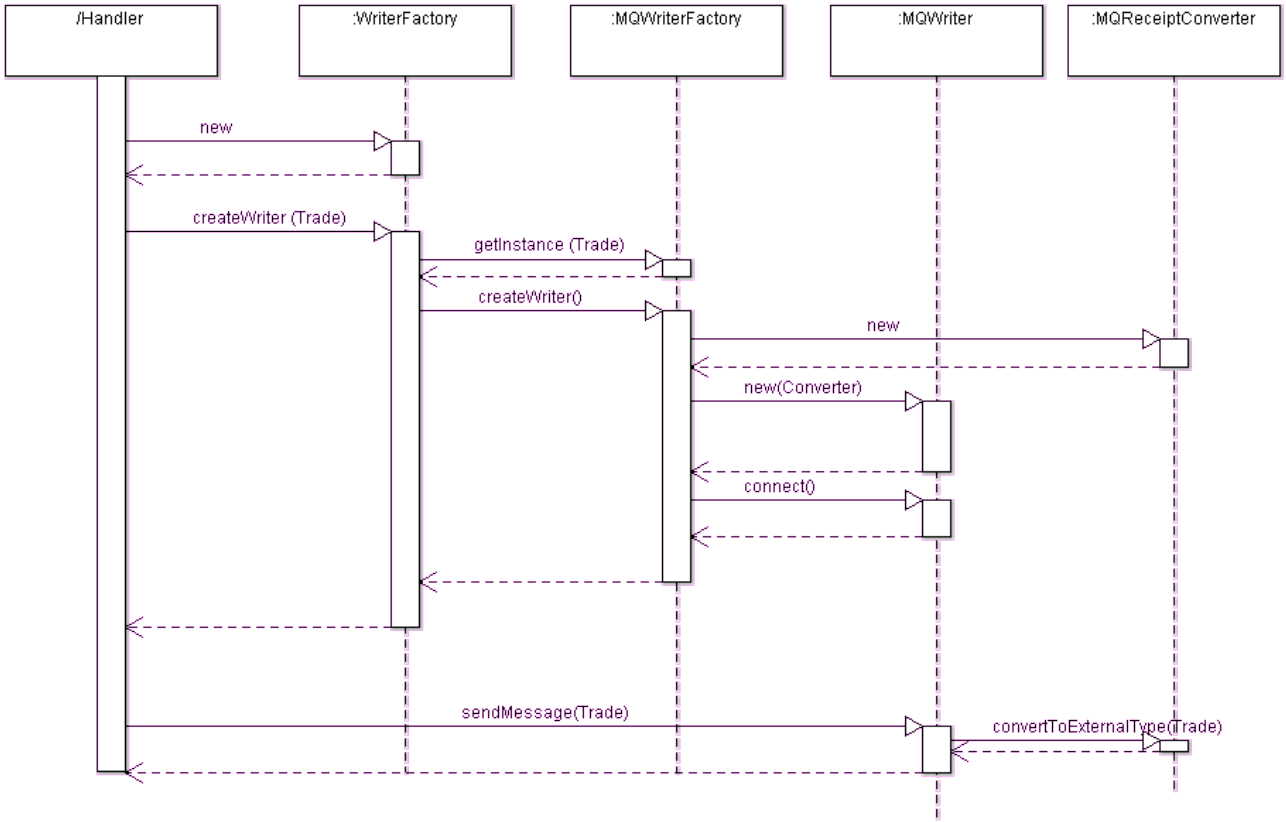
```
interface Converter {
       String convertToExternalType(Trade t);
      Trade convertToInternalType(String s);
}
```

Alle konkrete implemetationer af konverteringsfunktionalitet skal implementere *Converter* interfacet. Nu har jeg to typer *Writer* og *Converter*, som skal kombineres. Det er *Writer*, som har brug for *Converter* funktionaliteten og det er det samme forretningslogik, som afgør hvilken *Converter* og *Writer* der skal bruges. Derfor er det oplagt at give en *Converter*, som en konkret strategy[2] til *Writer* via en constructor parameter. Den konkrete factory på *Writer* initialiserings tidspunktet har alle informationer til at afgøre hvilken stratagy den har brug for.

4.2.4 Prototype

Som nævnt før, er denne prototype bygget med udgangspunkt i arkitektur diagrammet fra Figur4. Den afprøver en løs kobling mellem skrivningskommunikation, konvertering af internt *Trade* objekt og resten af systemet. De konkrete protokoller implementeres ikke færdige, da selve forbindelsens implementering i en bestemt protokol er irrelevant for problemstillingen.

Al nødvendig logik til at vælge den bestemte kommunikationskanal og tilsvarende konverteringsimplementation er indkapslet i de forskellige *Factory* klasser.



Figur 5:Send message - iteration1

I det rigtigte adaptersystems implementering vil der være et *Handler* objekt, som skal have ansvar for at behandle den pågældende handle/ordre efter bestemte forretningsregler. Når handel/ordre er klar skal den sendes videre til legacysystemet. Med den implementering, som protypen afprøver, har *Handler* objektet ikke længere kendskab til eller ansvar for at navigere message til den rigtige kanal og konvertere den til det korrekte format.

Konkret factory – i dette tilfælde *MQWriterFactory,* er implementeret som *Singelton.* Grunden til, at jeg har valgt denne konstruktion, er at det giver mulighed for at holde styr på alle oprettede *Writers* og på den måde genbruger dem.

Prototypen implementerer ikke en *Handler* klasse, men tester denne arkitektur fra NUnit. Jeg har startet implementering med at skrive test til *WriterFactory* klassen for at teste logikken i den. Da den klasse ingen tilstand har og ikke selv skaber retur objekter, fik jeg først min test grøn efter implementering af test til *MQWriterFactory* og selve factory klassen. Selvom prototypen virker meget begrænset, kan man hurtig miste fokus på den opgaver, som man lige er i gang med at implementere. Brugen af TDD metoden hjælper med at holde fokus og struktur i implementationsfasen. Det var meget mere klart i hvilken rækkefølge jeg skulle implementere funktionaliteten, og jeg kunne være sikker på, at jeg ikke glemte en del undervejs pga mistet fokus.

Selvom prototypen er forenklet i forholdt til selve adaptersystemet funktionaliteten, kan man bruge prototypen som struktur for adaptersystemets implementering.

Prototypen er bygget i C# med .NET Framework v.4 i WS2010. Der viste sig at være problemer med at få NUnit til at fungere sammen med .Net Framework v4. Hvis man vil bruge NUnit GUI eller consolen, er man nødt til at rette den relevante config filen med:

<startup><requiredRuntime version="v4.0.21006" /></startup>

4.2.5 Evaluering

4.2.5.1 Fordele

Hvis man sammenligner implementering i prototypen med den nuværende implementering, kan man finde en række forbedringer.

Der afkobles konkret implementering af *Writer* fra klienten/resten af adaptersystemet. Klienten ved faktisk slet ikke hvordan transaktionen bliver sendt eller konverteret. Når der er ingen reference til en konkret implementering af *Writer* på klientens side, giver det mulighed for at lave rettelser i eller udskiftning af kommunikationsprotokollen uden at rette i øvrige dele af adaptersystemet.

I modsætning til den oprindelige implementering er implementering af message konvertering isoleret fra resten af adaptersystemet. Den er indkapslet med et fælles interface og med et Strategy Pattern, så man på den måde frit kan kombinere konverteringer med kommunikationsprotokoller.

Logikken omkring protokoller og konverteringer er indkapslet i de forskellige factory klasser.

4.2.5.2 Ulemper

Ny fleksibel arkitektur medfører også en del ulemper. Der er skabt mange klasser til at håndtere den fleksibilitet. Det kan skabe uoverskuelighed. Ud fra klientens klasser kan man ikke se hvilken *Writer* udgave man arbejder med. Det kan være besværligt at finde eventuelle fejl eller bare at følge flowet.

En anden ulempen kan være, at logikken omkring valgt af protokoller og kanaler, er spredt i flere factories. I factories er der også implementeret logik omkring konverteringer og sammen gør factories implementering ret kompleks.

4.2.5.3 Modificerbarhed

I forhold til det vigtigste variabilitetspunkt er der opnået gode muligheder. Modificerbarheden blev forbedret drastisk. Fra 22 direkte kald i den nuværende implementering er der reduceret til ingen direkte kald på klintens side. Logikken omkring kommunikationsprotokollen er indkapslet i *Factory* klasser. På den måde vil ændringer af denne funktionalitet kun påvirke den konkrete protokol. Der er også mulighed for at tilføje en ny protokol. Ændringer i sådanne situationer er afgrænset til et sted i *FactoryWriter*.

Da konvertering er isoleret af resten adaptaersystemet, har man mulighed for at ændre afsendelsesformatet eller udskifte den med en ny. Det vil kun kræve en ny konverteringsimplementation og rettelse i en enkelt factory klasse.

4.3 Iteration 2

I iteration 2 arbejder jeg videre med løsningen fra iteration 1. Jeg vil prøve at forbedre løsningen og skabe løsere koblinger.

To hovedmål for denne iteration:

- at forenkle factory implementering, ved at flytte logikken fra factory til konfiguration ved hjælp af Dependency Injection Pattern [6]. Dependency Injection Pattern flytter afhængigheder mellem objekter til konfiguration og ansvaret for instancering er flyttet til et tredje objekt. Oftest bliver dette pattern brugt i forbindelse med en applikationscontainer, hvor sammenhænge mellem objekter er beskrevet i konfiguration. Jeg vil bruge en forenklet udgave af Interface Injection udgaven af dette pattern.
- at isolere kommunikationsimplementation til et selvstændig library – i dette tilfælde en dll fil. Formål med denne isolation er at bruge implementering af de konkrete protokoller på tværs af applikationer og på den måde spare udviklingstid i fremtiden.

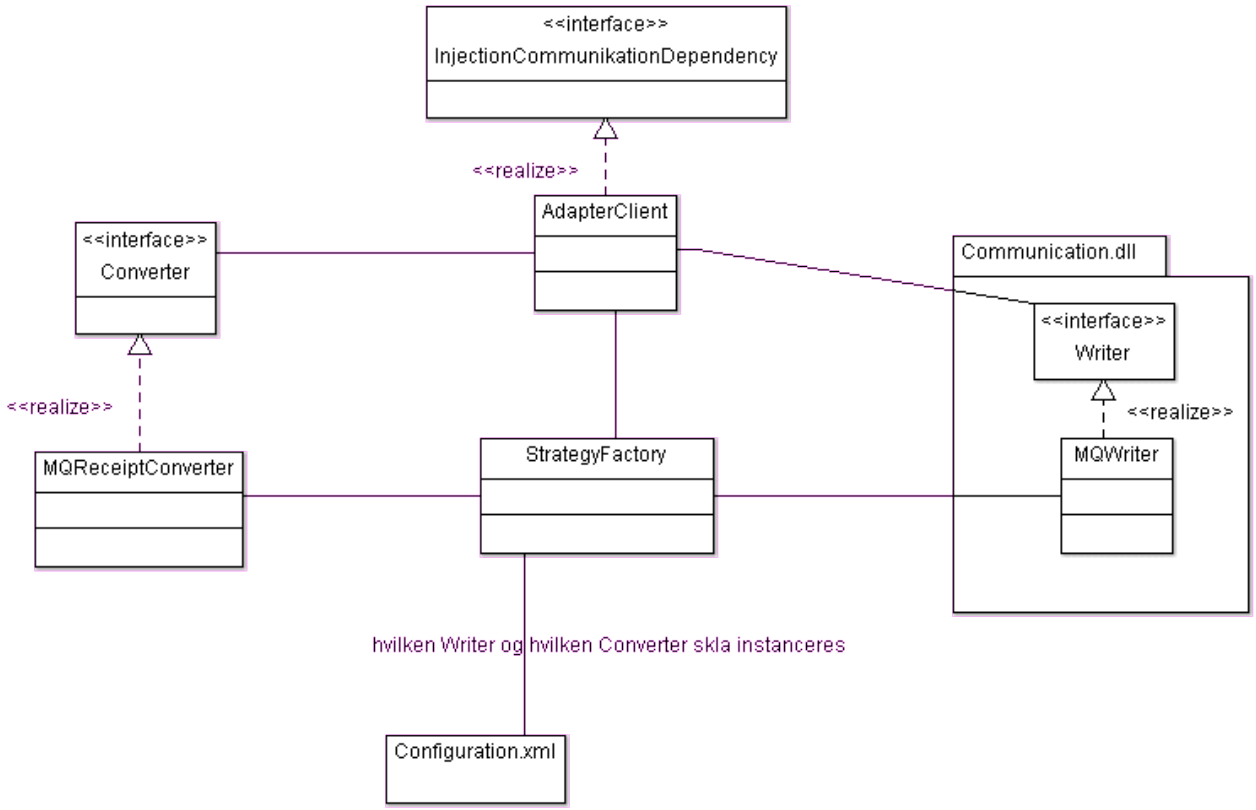
4.3.1 Arkitektur

Kommunikationsimplementationen er allerede isoleret og afkoblet fra resten af adaptersystemet ved hjælp af *WriterFactory*. Den eneste afhængighed til adaptersystemet er kendskabet til en fælles info klasse *Trade,* som er parameter til *sendMessage* metode. Jeg vil generalisere *Writer* interface til:

```
public interface Writer{
	void sendMessage(String info);
}
```

hvor *Trade* allerede er konverteret til en XML *String*. På den måde gør jeg *Writer* implementering uafhængig af adaptersystemets implementation og kan isolere den del som et selvstændigt library. Eneste ansvarsområdet for dll'en skal være at implementere connection til en række specifikke protokoller. På den måde kan den genbruges af andre applikationer, som skal etablere forbindelse med de protokoller som dll'en vil indeholde. Med en general *String* parameter gøres den domain uafhængig og kan genbruges i alle .Net applikationer – c# samt c++.

Når *Writer* implementering modtager en konverteret message, har den ikke brug for en konverterings strategy. Derfor skal konverterings- og kommunikationsfunktionalitet splittes og deres implementering metode uafhængige af hinanden. Selvom de bliver instantieret uafhængigt, er de stadig afhængige af hinanden i den sammen grad som i iteration 1. Denne afhængighed vil jeg flytte til konfiguration og implementere ved hjælp af Dependency Injection.



Figur 6:Arkitektur for afhængigheder mellem klienten, Writer og Converter - iteration2

AdapterClient skal bruge *StrategyFactory* til at finde de rigtige *Writer* og *Converter.* Når *AdapterClient* er færdig med at behandle *Trade,* har den brug for en bestemt strategy for hvordan *Trade* skal konverteres og sendes. Denne strategy er afhængig af *Trade* objektet og ændrer sig mange gange i løbet af *AdapterClienten's* livscyklus. Derfor har jeg valgt at designe den del af systemet med hjælp af *Strategy Pattern* [2], hvor strategy er leveret for hver behandlet *Trade.*

Forretningslogikken, som i iteration 1 var spredt i to lag af factories, placeres nu i *StrategyFactory* og konfiguration. *StrategyFactory* får ansvar for at finde frem til de rigtige *Writer* og *Converter,* men afhængigheden mellem dem er placeret i konfiguration. Til gengæld får *AdapterClient* brug for kendskab til både *Writer* og *Converter* – det er den som får ansvar for at fortage konvertering af *Trade* før afsendelse.

I denne arkitektur bruger jeg Strategy Pattern, som implemeteres med hjælp af Dependency Injection Pattern. *StrategyFactory* leverer en bestemt strategy afhængig af hvilken *Trade* der er under behandling, men til at finde og aflevere den rigtigt strategy (objekter som passer til *Trade's* tilstand) bruger den Injection Pattern med Interface Injecton.

Alle klasser, som skal bruge *StrategyFactory* skal implemetere *InjectionCommunicationDependency* interfacet.

```
public interface InjectionCommunicationDependency {
    void injectWriter(Writer w);
    void injectConverter(Converter c);

}
```

Ved hjælp af de to metoder kan factoryen sende de to nødvendig referencer tilbage til klienten (*AdapterClient*). Da valg af *Writer* og *Converter* afhænger af *Trade* objektet, kan de to referencer ændre sig i *AdpaterClient's* livscyklus. Derfor kan de ikke være sat, som parameter i constuctor på instanceringstidspunktet, men som en strategy for at færdigbehandle *Trade.*

4.3.2 Prototype

Første step i implementering af denne iteration er at isolere kommunikationsimplementation. Man skal isolere den i et selvstændigt dll-projekt og rette referencer rundt om i projektet. Denne ændring medfører en ”rød” NUnit test, men efter rettesler af referencer i Test projektet er testen ”grøn” igen. Ud over isolering af bestemte klasser er funktionalitet ikke ændret.

Næste step er implementation af Dependecy Injection Pattern.
Dependency Injection pattern har jeg valgt at implementere med eksisterende .Net konfigurations library og selv implementerer den nødvendig struktur. Det giver frihed til at strukturere konfigurationen, som man har brug for og som man selv synes det passer bedst til applikation. Det kræver dog et ret stort arbejde, da hvert tag i XML konfigurationen kræver en implementations klasse. Denne lille konfiguration kræver 8 klasser. Konfigurationen i XML filen har jeg struktureret således:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="CommunicationConfiguration"
             type="AdapterRefactoring_HO.Configuration.CommunicationConfiguration,
                                                    AdapterRefactoring_HO"/>
    <section name="ConnectionConfiguration"
             type="ConnectionLibrary.Configuration.ConnectionConfiguration,
                                                    ConnectionLibrary"/>
  </configSections>
  <CommunicationConfiguration>
    <SendConnections>
      <SendConnection name="Receipt-OMX" source="OMX" systemUserName="HOST"
                      converterType="AdapterRefactoring_HO.MQReceiptConverter,
                                                    AdapterRefactoring_HO"
                      writerType="ConnectionLibrary.MQWriter, ConnectionLibrary"
                      writerInfo="ReceiptFromOMX"/>
      <SendConnection name="Receipt-ROR " source="ROR" systemUserName="HOST"
                      converterType="AdapterRefactoring_HO.MQReceiptConverter,
                                                    AdapterRefactoring_HO"
                      writerType="ConnectionLibrary.MQWriter, ConnectionLibrary"
                      writerInfo="ReceiptFromOMX"/>
      <SendConnection name="Info-OMX" source="OMX" systemUserName="CLIENT"
                      converterType="AdapterRefactoring_HO.WSTradeConverter,
                                                    AdapterRefactoring_HO"
                      writerType="ConnectionLibrary.WSWriter, ConnectionLibrary"
                      writerInfo="InfoFromOMX"/>

    </SendConnections>
   </CommunicationConfiguration>
  <ConnectionConfiguration>
    <Writers>
      <Writer name="ReceiptFromOMX">
        <mqConnection channel="CLIENT.MQ1" hostName="MQU" port="5555"
                                              queueName="JB.RECEIPT.EUR"/>
      </Writer>
      <Writer name="InfoFromOMX">
        <mqConnection channel="CLIENT.MQ1" hostName="MQU" port="5555"
                                              queueName="JB.RECEIPT.EUR"/>
      </Writer>
    </Writers>
 </ConnectionConfiguration>
```

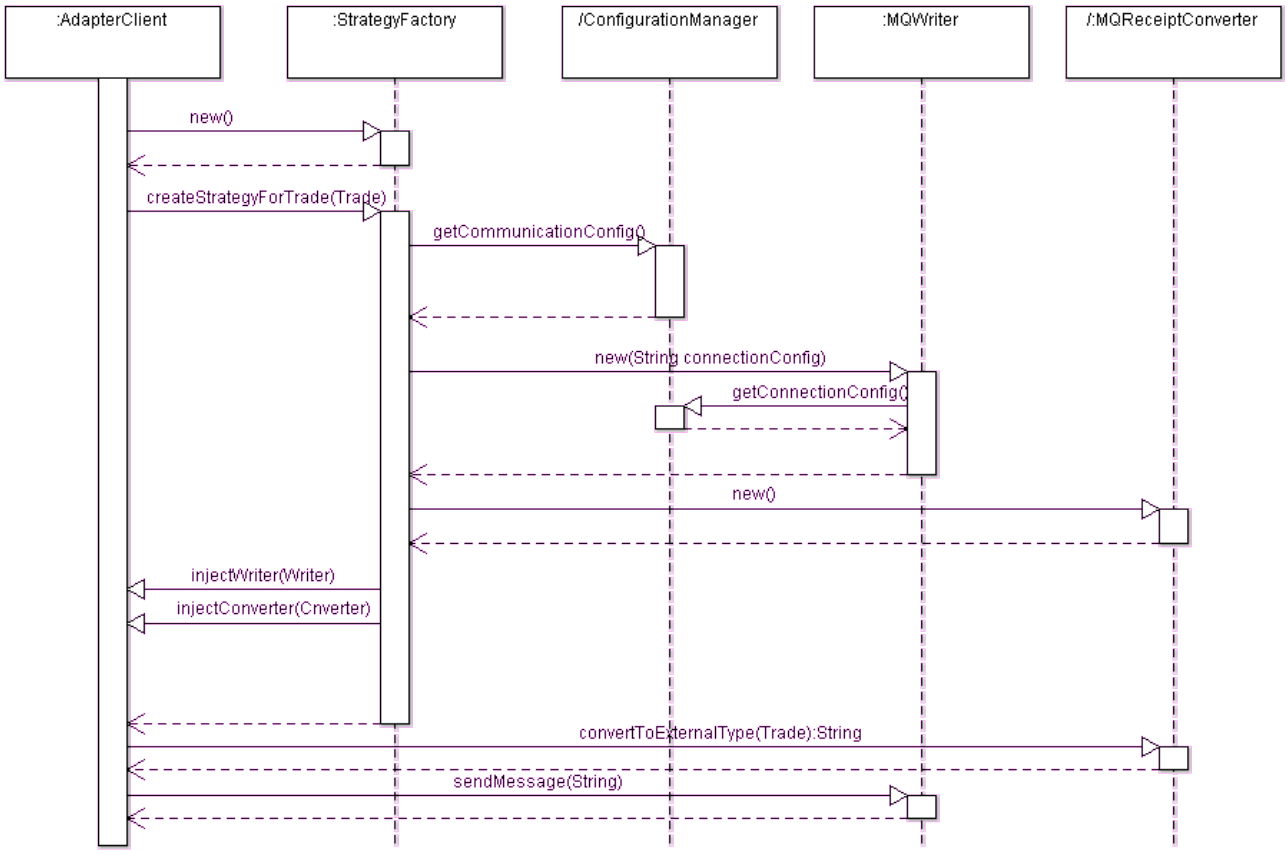
Tabel 3: Konfigurations strukturen

Jeg har defineret to hoved sectioner:

- CommunicationConfiguration, der har ansvaret for at sammensætte *Writer* og *Converter,* som skal bruges sammen,
- ConnectionConfiguration, som har ansvar for selve forbindelsens tekniske konfiguration.

I *Writer* konfiguration er *name* attribute bare en *String,* som man referer til fra sin implementering. Attributter *source* og *systemUserName* i *SendConnection* er de samme attributter, som var brugt i iteration 1 i de to factories, som var ansvarlige for forretningslogikken. Hvis man vil udskifte en af referencerne, er det nok at rette konfigurationen. Det samme gælder for en ny implementering af *Writer* eller *Converter.* På den samme måde kan man tilføje ny *SendConnection*, hvor man skal konfigurere et par *Writer/Converter* og selve protokollen. For at holde prototypen enkelt har jeg kun lavet MQ konfigurationen, men konfigurationen for *Writer* kan generaliseres så den kan være brugbar for en række protokoller.

SendMessage kald ser nu således ud:



Figur 4:Send Message - iteration 2

Som man kan se i diagrammet i Figur 4, har ansvarsområdet for de forskellige objekter ændret sig. Det kunne se ud til, at det nu er mere kompliceret for AdapterClient, men det giver mening hvis man vil isolere *Writer* til et selvstændigt library. Det er klientens ansvar at konvertere objektet til det forventede format og klienten har også frihed til at implementere konvertering på sin egen måde. Konvertering er domain specifik og *Writer* skal være domain uafhængig, hvis den skal fungere som et generelt library. *Writer* objektet er nu fri fra konvertering og har udelukkende ansvar for at sende message.

Brugen af .Net's konfiguration library og den måde som den skal implementeres, gør at man ikke kan implementere selvstændig test af konfigurationen i sit Test projekt. Det kan man dog teste igennem *StrategyFactory* tester. Man skal huske at denne test klasse tester implementering af *StrategyFactory* samt konfiguration i *CommnicationConfiguration* fra konfiguration filen. Det kan stærkt anbefales at implementere en ny test for hver *SendConnection* section, som beskriver konfiguration for ny sammenhæng mellem *Converter* og *Writer*. Selvom det ser ret banalt ud, er det et oplagt sted til at plante ”copy past” fejl og de er ikke nemme at finde bagefter.

Jeg har valgt at implementere en ny test i stedet for at rette testen fra iteration 1 og på den måde beholdt implementering fra iteration 1 parallel med implementering i iteration 2. Som under iteration1 gjorde brugen af TDD metoden implementationsarbejdet struktureret og målrettet. Man er mere bevist om, hvordan man skal strukturere kode, da man ved hvilket resultat man forventer. Det kræver dog tilvending og en vis disciplin, særligt i sådan et lille projekt som prototypen. Man sysnes, at man ved det hele på forhånd og at det er åbenlyst, da den ikke er så omfattende, men man kan dog være overrasket over det nye struktureret input, som man får med at bruge TDD.

4.3.3 Evaluering

4.3.3.1 Fordele

Når man tænker på fordele ved denne implementation i forhold til iteration 1, er det først og fremmest den større fleksibilitet, som man opnåede ved at flytte logikken til konfiguration. Hvis man vil udskifte implementering af *Writer* eller *Converter,* skal man lave ny implementering og tilføje den til konfiguration. Hvis man vil udskifte en eksisterende implementering med en anden, skal man kun rette i konfiguration uden at rette i implementering af adaptersystemet.

Isolering af protokolimplementation til en selvstændig dll, giver genbuglighed på tværs af mange projekter. Det kan fremtidssikre ensrettet udvikling og hurtigere udvikling.

4.3.3.2 Ulempe

Selvom man får større fleksibilitet i forbindelse med brug af *Writer* og *Converter,* kan man finde ændringer i *sendMessage* metodens parameter mindre fleksibel. Man antager at det skal være XML format, som man sender – derfor S*tring* parameter.

Største ulempen i at bruge Dependency Injection er kompleksiteten. Det at man flytter en del af referencerne ud til konfiguration, gør implementering mindre læsbar. Det kan være ret svært at finde en fejl, da det er ikke specielt tydeligt hvilke objekter der kommunikerer med hinanden. En anden ulempen kan være en større kompleksitet ved udvikling. Det kræver implementering af mange klasser, som man nemt kan blive forvirret af. Hele implementering kræver et ret stort forarbejde for at få det til at fungere. Jeg har selv oplevet under arbejde med prototypen, at det er meget svært at finde fejl i kobling mellem de forskellige sektioner og tags i konfigurationen og tilsvarende klasser. Det er dog selvfølgelig ”en gangs” opgaver, som giver stor glæde senere.

4.3.3.3 Modificerbarhed

Som jeg allerede har skrevet i afsnittet om fordele giver denne implementring ret stor frihed i forhold til modificerbarhed. Man har mulighed at fortage ændringer i runtime ved at rette konfiguration. Man kan genanvende eksisterende implementeringer af *Writer* og *Converter.* I denne arkitektur har man faktisk et set af *Writer* og en set af *Converter*, som man kan sammensætte på en logiske måde. *StrategyFactory* klassen er forenklet i forhold til implementering af factories i iteration1. Den looper bare *SendConnections* kollektion for at finde den, som passer med det bestemte *Trade* objekt.

4.4 Andre implementerings muligheder

Den præsenterede design og implementering er gennemført med ”selv kodet” løsning. Men hvis man undersøger emnet lidt, kan man nemt finde en del af frameworks, som tilbyder Injection of Control/Depandecy Injection funktionalitet. For at give nogle eksempler på sådanne frameworks, kan man nævne Spring[12], CastleWindsor[13] eller StructureMap[14].

Alle tilbyder lingende funktionalitet med konfiguration i XML eller andre formater (Spring er uafhængig af konfigurations format) og funktionalitet, som kobler konfiguration sammen med implementering. I sin konfiguration definerer man en bestemt plugin/object ved at identificere en

key/name/id, en type for det object og en assambly for dem.

Spring Eksempel:

```
<objects xmlns="http://www.springframework.net">
  <object id="InfoFromOMX" type="ConnectionLibrary.MQWriter, ConnectionLibrary">
    <!-- collaborators and configuration for this object go here -->
  </object>
</objects>
```

StructurMap Eksempel:

```
<PluginFamily Type="ConnectionLibrary.Writer"
              Assembly="ConnectionLibrary"
             DefaultKey="Info">
      <Plugin Type="ConnectionLibrary.MQWriter"
             Assembly="ConnectionLibrary"
              ConcreteKey="InfoFromOMX" />
</PluginFamily>
```

CastleWindsor Eksempel:

```
<components>
         <component id="InfoFromOMX"
              service="ConnectionLibrary.Writer, ConnectionLibrary"
              type="ConnectionLibrary.MQWriter, ConnectionLibrary">
         </component>
 </components>
```

Tabel 4: Eksempler på konfiguration for MQWriter i forskellige frameworks

I tabel 4 viser jeg forenklede eksempler på, hvordan konfigurationen kunne være for en implementering af *Writer* interface i de forskellige framewoks. Hele konfiguration kræver selvfølgelige mere end det, men som kan finde i deres dokumentation kan man kombinere frameworks konfiguration med customized konfiguration. Jeg har kigget lidt nærmere på Spring konfigurationsmuligheder. Man kan kombinere .Net's konfigurations library, som er brugt i implementering i iteration 2, med Springs konfiguration ved at definere *Springs configSection* og refererer til den i resten af konfiguration.

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler,
                                                          Spring.Core"/>
      <section name="objects"
       type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>
  <spring>
    <context>
      <resource uri="config://spring/objects"/>
    </context>
    <objects xmlns="http://www.springframework.net">
        ...
    </objects>
  </spring>
</configuration>
```

Tabel 5: Eksempel på definition for sectionGroup for Spring framework

Som man kan se i eksempel i Tabel 5, er sektioner defineret på den samme måde som sektioner i konfiguration i iteration 2 – Tabel 3. På den måde ville det være muligt at forbinde konfiguration for selve forretningslogikken for *StrategyFactory* med *Dependency Injection* konfiguration, som frameworket tilbyder. Med brug af et framework slipper man for at implementere den del af konfigurations klasserne, der har med Dependency Injection at gøre og på den måde forenkler egen implementering.

Jeg har ikke implementeret en prototypen med Spring eller et andet framework, da der ikke var tid til det under opgaven. Jeg har dog undersøgt dokumentationen for nogle af dem og Dependency Injection var en standard funktionalitet i dem. Derfor vil brug af et framework helt sikkert være en mulighed for implementering af adaptersystemet.

5 Slutevaluering og konklusion

Det overordnede mål med denne opgave var at fremstille en arkitektur for adaptersystemets kommunikationfunktionalitet ved hjælp af arkitekturprototyper og med vægt på kvalitetsattributterne modificerbarhed og testbarhed.

5.1 Akitektur

Nuværende implementering af adaptersystemet karakteriserer sig med blandende funktionalitet for flere ansvarsområder flettet sammen. Jeg har præsenteret forskellige arkitekturer for den udvalgte del af adaptersystemet og afprøvet dem i prototyper. De har været designet med 3-2-1 principper [2], derfor kan man identificere grundlæggende fælles fordele i forhold til den nuværende implementering :

- de forskellige ansvarsområder er isoleret fra hinanden og indkapslet med interfaces; afsendelse og konvertering er helt uafhængig fra hinanden og fra intern datastruktur
- kommunikationslogikken er isoleret i få klasser og uafhængig af datakonvertering og kommunikationsprotokoller

Der er selvfølgelige også forskel mellem arkitektur i iteration 1 og 2. Valg mellem dem vil være afhængig af størrelsen af systemet og sandsynlighed for modificering i fremtiden.

Arkitekturen i iteration1 placerer hele kommunikationslogikken i selve implementering, isoleret i forskellige factory klasser. Det giver hurtigt overblik over funktionaliteten, men hvis systemet vil vokse meget er der risiko for ”grime” og lange ”if” sætninger i factories. Det vil dog stadig være isoleret og rettelser begrænset til de få factory klasser.

Arkitekturen i iteration2 flytter logikken til konfiguration. Det giver muligheder for hurtige tilføjelser af ny logik uden at rette selve implementering. Denne frihed medfører, at det kan være svært at gennemskue referencer mellem de forskellige klasser. Implementering af konfiguration kan også være omfattende. Det kræver mange implemetationsklasser, som afspejler konfigurationsstrukturen. Denne klasse opbygger man kun en gang. Tilføjeler til konfiguration kræver selvfølgelig ikke ændringer i eksisterende implementering. Hvis man på et senere tidspunkt beslutter at ændre strukturen i konfigurationen, kræver det også rettelser i implementering i tilsvarende klasser og alle steder hvor de er brugt. Derfor er det vigtigt at finde den rigtigt struktur, som man kan udvider med ny konfiguration og ikke ændre i strukturen ved udvidelser af adaptersystemet.

Hvis man ville vælge at bruge framework vil arkitekturen for adapteren ikke være anderledes end i iteration2 udgaven, men kun selve implementering af konfiguration. Man ville undvære implementering af en del af konfigurationsklasser, som man ville bruge fra frameworket. Det kunne være nødvendigt at implementere den del, som er specifik for adaptersystemts logik.

Ud fra erfaringen opsamlet under opgaverne og overvejelser under evaluering, vil jeg konkludere at den bedste løsning for adaptasystemt vil være iteration2's løsning. Og det gør jeg fordi det er et større system, som fungerer i dag i produktion, men for hvilken der er større udvidelsesplaner og man kan forvente udvidelser i flere omgang. Selvom konfigurationsimplementering er ret omfattende, gør man det kun en gang. Man kan strukture konfiguration præcis efter adaptersystemts behov- i modsat til et færdigt framework, hvor man skal tilpasse adaptersystemet til mulighederne. Jeg ville dog anbefale, at undersøge muligheder nærmere for at bruge framework i adaptasystemet.

5.2 Modificerbarhed

Et af det vigtigste mål for opgaven var at skabe muligheder for modificering og udvidelser i adaptersystemet. Det var også grunden til, at jeg har valgte at basere mine opgave på den del af funktionaliteten – kommunikationsprotokoller, da jeg ved, at det er den del, som kræver udvidelser i den nærmeste fremtid.

Uanset hvilken løsning, som er præsenteret i opgaven, man ville vælge, øger den væsentligt modificerbarheden i adaptersystemet.

I nuværende implementering kan man finde 22 direkte reference i adaptersystemets implementering til skrivning til MQ. XML opbygning og navne på felter i XML filen er indkodet og spredt rundt i implementering. Derfor kan enhver ændring skabe stor risiko for at medføre en fejl og det er svært at overskrue alle steder, som skal rettes.

I opgaven blev kommunikationsreferencer isoleret og klientapplikationen/handler har ingen direkte reference længere. Klient/handler har faktisk slet ikke viden om, hvor og hvordan beskeden sendes. Det samme gælder konvertering. Selvom klient/handler i iteration 2 har kendskab til selve *Converter* interfacet, ligger logikken for at finde den rigtigt konverteringsimplementering i konfiguration og er fundet igennem factory.

I begge løsninger gælder det at :

- tilføjelse af ny funktionalitet medfører ny implementeringsklasser for ny funktionalitet – ny *Writer* og/eller *Converter* implemeteringsklasser
- ændringer i funktionalitet medfører ændringer isoleret til factory klasser for iteration 1 og ændringer i konfiguration (ikke i konfigurations strukturen) for iteration 2.

5.3 Testbarhed

En anden meget vigtig parameter for opgaven var testbarhed. I den nuværende implementation eksisterer muligheden for automatisk test ikke. Test foregår manuelt og kræver etablering af forbindelse til alle involverede systemer.

Med ny arkitektur blev der skabt mulighed for at mocke implemetation af de forskellige komponenter af systemet ud med test implementation. Det kan man gøre for både forbindelser og konvertering.

Ved at bruge TDD, som udviklingsmetode dækker man implemetation med test. Det giver sikkerhed ved ændringer, samt et stærk værktøj for analyse af ændringsønsker, da man på den måde hurtigt kan afsløre omfang og indflydelse af ændringer. Det kræver selvfølgelige konsekvens i brugen af metoden.

Man kan konkludere, at med ny arkitektur og TDD implementeringsmetoden skaber man testbarhed, som ikke eksisterer for den nuværende implementering.

6 Referencer

[1] Software Architecture, Object-Oriented Modeling and UML, Henrik Bærbak Christensen, version 1.5.1, October 2005

[2] Reliable and Flexible Software Explained:architecture, Patterns and Frameworks, Henrik Bærbak Christensen, August 2006

[3] Architectual Prototyping: An Approach for Grounding Architectural Design and Learning, Jakob Eyvind Bardram, Henrik Bærbek Christensen, Klaus Marius Hansen

[4] An Empirical Investgation af Architectural Prototyping, Henrik Bærbek Christensen, Klaus Marius Hansen, August 2009

[5] Software Architecture in Practice, 2nd edition, Len Bass, Paul Clements, Rick Kazman

[6] Martin Fowler, Inversion of Control Containers and the Dependency Injection pattern http://martinfowler.com/articles/injection.html#ANaiveExample

[7] Neal Ford, Refactoring toward design

[8] Neal Ford, Composed method and SLAP

[9] Neal Ford, Emergent design through metrics

[10] Design Patterns Explained – A New Perspective on Object-Oriented Design, Alan Shalloway and James R. Trott

[11] Martin Fowler, Refactoring

[12] http://springframework.net/

[13] http://www.castleproject.org/index.html

[14] http://structuremap.github.com/structuremap/index.html