Simson Fennikel Collector – datahandler designdokument
Versjon 1.0

Amund Gjersøe og Svein Alsterberg

Forsvarets forskningsinstitutt (FFI)

26.08.2008

FFI-rapport 2008/00461

899

P: ISBN 798-82-464-1433-1
E: ISBN 798-82-464-1434-8

Emneord

Simson

Quick-look

Datainnlesning

Datahandler

Godkjent av

Arne Cato Jenssen	Prosjektleder
Elling Tveit	Forskningssjef
John-Mikal Størdal	Avdelingssjef

Sammendrag

Marinen anskaffer fem nye fregatter innen 2010. Den første, KNM Fridtjof Nansen, ble levert fra Navantia i mai 2006. Fregattene er utstyrt med to sonarer i tillegg til Flash dyppesonaren som skal monteres på et NH-90 helikopter. Den lavfrekvente aktive tauesonaren, CAPTAS, er i stand til å detektere små, diesel-elektriske undervannsbåter i kystnære farvann på flere titalls kilometers avstand. Den skrogmonterte multirollesonaren, Spherion MRS2000, kan detektere miner og innkommende torpedoer så vel som undervannsbåter. Kombinert gjør disse systemene kampsystemet Fridtjof Nansen til muligens den mest avanserte anti-undervannsbåt plattformen som finnes.

Prosjektet *Nansen-klasse fregatt, evaluering* (P899) har bland annet som oppgave å levere et quick-look analyseverktøy til Marinen, basert på verktøyet utviklet i prosjekt Simson (P849). Dette verktøyet skal i første omgang brukes til å analysere data fra sonarene til kampsystemet Fridtjof Nansen. I denne sammenheng trenger man et verktøy for å enkelt tilrettelegge dataopptak for analyse. Denne oppgaven dekkes av Simson Fennikel Collector.

Simson Fennikel Collector (FC) ble utviklet for å gjøre det mulig å samle alle datalogger på ett lagringsformat. Dessuten ønsket man å bedre innlesningsprogramvaren for Simson databasen. FC er i dag i bruk ombord i Fridtjof Nansen klassen fregatter, og fungerer primært som et verktøy for å videredistribuere data fra fregattene til FFI. På FFI brukes både FC og andre innlesningsprogram for Simson databasen.

Dette dokumentet beskriver ideene bak det vi kaller en *datahandler* (no. datahåndterer) til FC. Det beskrives også hvilke grensesnitt som må implementeres, og det blir gitt korte eksempler på implementering.

Dokumentet er ment som et hjelpemiddel for programmerer som skal utvikle nye datahandlere.

English summary

The Norwegian navy will procure five new frigates within 2010. The first frigate, KNM Fridtjof Nansen, was delivered by Navantia in May 2006. The frigates are equipped with two sonars in addition to the Flash dipping sonar on the helicopter. The low frequency acoustic array, CAPTAS, is able to detect small, diesel electro submarines at tens of kilometers in littoral as well as open waters. The hull-mounted sonar, Spherion MRS2000, makes the frigate capable of detecting mines and incoming torpedoes as well as submarines. The combined qualities of these acoustic systems make the Fridtjof Nansen-class perhaps the most advanced anti-submarine warfare platforms for littoral waters ever built.

Project *Nansen-klasse fregatt, evaluering* (Fridtjof Nansen-class evaluation) will deliver a quick-look analysis software for analyzing anti submarine warfare (ASW) exercises. As part of this software package Simson Fennikel Collector will prepare the recorded data for analysis.

Simson Fennikel Collector (FC) is developed to support the archiving and database storage of data from the Norwegian Fridtjof Nansen class frigates. The data are analyzed using the Simson ASW analysis software for active sonars.

This document describes the design of the interface the classes must implement to be compatible with FC, and be treated as a *data handler*. The document includes the ideas, all the interfaces and some simple examples. The developer do not have to think about the graphical user interface when a reader has been implemented using the interface specification.

Innhold

1 Innledning

Prosjektet *Nansen-klasse fregatt, evaluering* (P899) har som oppgave å levere et quick-look analyseverktøy til Marinen, basert på verktøyet utviklet i prosjekt Simson (P849). I denne sammenheng trenger man et verktøy for å enkelt laste dataopptak fra et lagringsmedium inn på en harddisk. Derfra må data enkelt kunne lastes inn i Simson databasen slik at data kan analyseres ved hjelp av Simson analyseprogramvare for aktiv sonar. Denne oppgaven dekkes av Simson Fennikel Collector.

Simson programvare ble overlevert KNM Tordenskjold i 2005. Simson er et analyseverktøy for å evaluere og simulere fregatters aktive sonarsystem [1]. Siden Simson ble levert før første Fridtjof Nansen-klasse fregatt kom, var verktøyet tilpasset Oslo-klassen fregatter. Likevel ble verktøyet klargjort for data fra Fridtjof Nansen-klassen basert på spesifikasjoner og testdata.

Simson verktøyet viste seg nyttig som et analyseverktøy i debrief etter øvelser med aktiv sonar [2]. Dette til tross for at verktøyet i utgangspunktet var tiltenkt brukt i langtidsanalyser. Det ble et ønske at prosjekt P899 skulle utvikle en versjon av Simson som kunne gi en ”quick-look” av øvelser for bruk i debrief ombord Fridtjof Nansen-klasse fregatter. Quick-look versjonen av Simson blir kalt Simson Fennikel [3]. Navnet Fennikel kommer fra forkortelsen FNQL, som står for Fridtjof Nansen Quick Look.

Simson Fennikel Collector (FC) ble utviklet for å gjøre det mulig å samle et vidt spekter av datalogger på ett lagringsformat. Dessuten ønsket man å bedre innlesningsprogramvaren for Simson-databasen. FC er i dag i bruk ombord i Fridtjof Nansen-klassen fregatter, og fungerer primært som et verktøy for å videredistribuere data fra fregattene til FFI. Ved FFI brukes både FC og andre innlesningsprogram for Simson databasen.

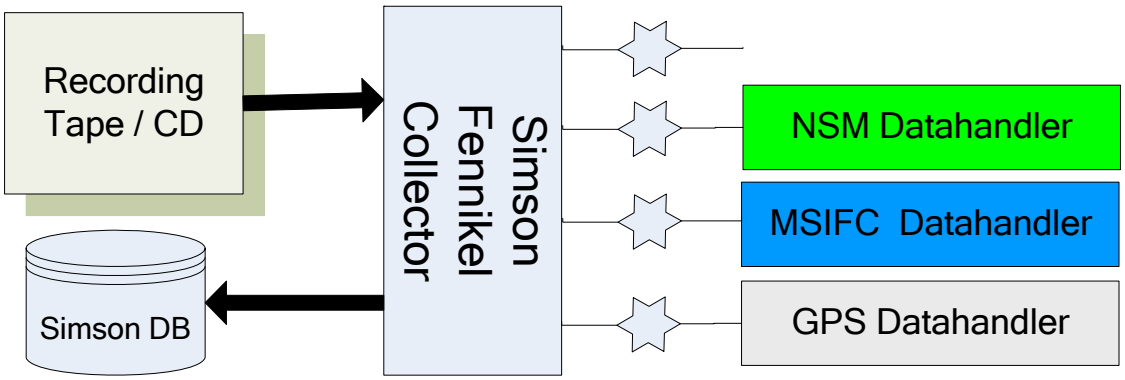
Vi ser at Simson Fennikel Collector kan bli utgangspunktet for å samle inn og lese flere dataformat enn de som kommer fra Fridtjof Nansen-klasse fregatter. Derfor har det blitt behov for å omorganisere FC for å bedre mulighetene for effektiv utvidelse av applikasjonen, og også gjøre grep for å bedre vedlikeholdsvennligheten av programvaren.

Slik FC er organisert nå, vil ikke utvikleren behøve å tenke på annet en sin egen kode ved innføring av nye dataopptak. Det er da en forutsetning at databasen ikke må oppdateres for å huse de nye dataopptakene.

Dette dokumentet beskriver ideene bak det vi kaller en *data handler* til FC. Det beskrives også hvilke grensesnitt som må implementeres for at en *data handler* skal bli akseptert av rammeverket, og det blir gitt korte eksempler på implementering.

2 Design av en datahandler

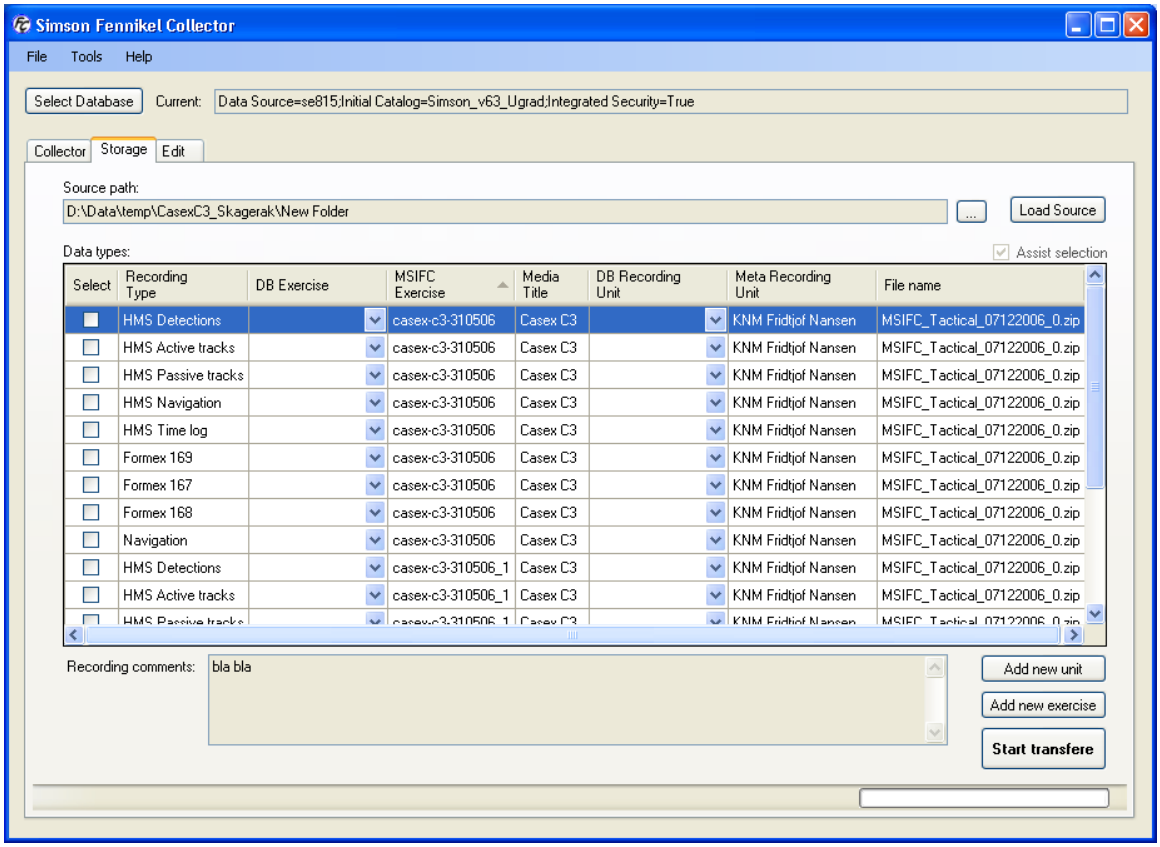
FC har to hovedoppgaver. Den første er å lagre data lokalt på en disk sammen med en enkel tolkning av dataene. Den andre oppgaven er å konvertere data fra et spesifikt format over til et Simson format som lagres i Simson databasen.



Figur 2.1 Oversikt over en datahandlers plass i FC. Koplingen mellom infrastrukturen i FC og datahendlerene er et definert grensesnitt som spesifiseres i dette dokumentet.

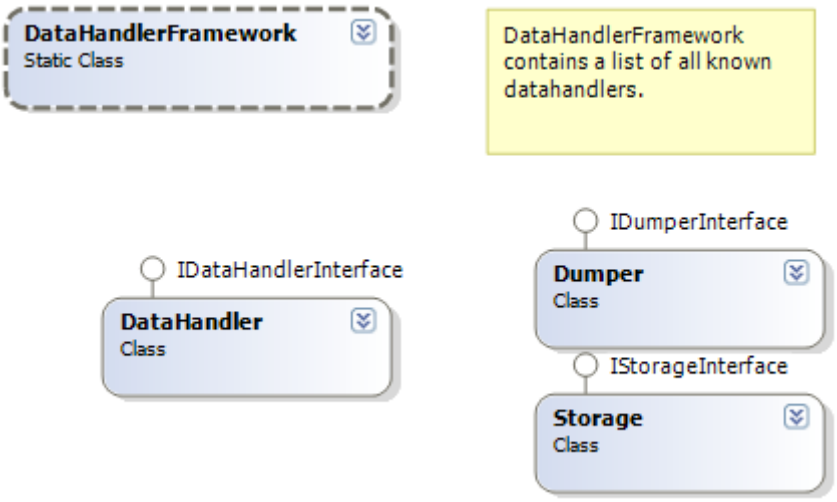
For å gjøre utvidelser av FC så oversiktlig som mulig, vil det bli lagd én datahandler for hvert dataformat. Forholdet mellom datahandlerne og FC er vist i Figur 2.1. Tilgangen til blant annet tape drev og Simson database blir formidlet til datahandlerene gjennom FC.

Et dataformat vil normalt være knyttet til en fil. En fil eller dataformat kan inneholde flere datatyper. I FC vil en datatype tilsvare en linje i *Storage DataGridView*, mens dataformater er knyttet til en hel metafil. *Storage DataGridView* er vist i Figur 2.2.



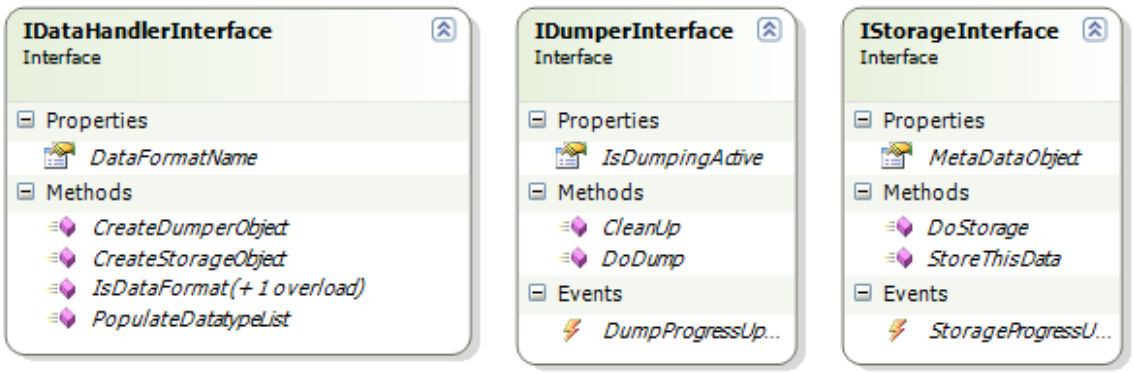
Figur 2.2 *Skjermbilde av Simson Fennikel Collector. I arkfanen "Storage" finnes tabellen kalt Storage DataGridView. Denne lister opp alle datatypene som en datahandler kan lese ut av én type fil*

Koblingsklassen vil sammen med *DataHandlerFramework*-klassen være grensesnittet mellom datainnlesing og GUI delen av programmet. Klassen vil ha metoder for gjenkjenning av dataformatet, metode for å fylle ut listen i *Storage DataGridView* og metoder for å lage et objekt med Dump-grensesnitt eller Storage-grensesnitt. Dump grensesnittet har metoder for innlesningen av data fra mediet (tape, disk, diskett) til disk, og Storage-grensesnittet har metoder for valg av data for innlesing samt for start av selve innlesningen fra disk til database. Betydningen av ordet *grensesnitt* i dette dokumentet er det samme som det programmatiske *interface* [4]. I Figur 2.3 er det en oversikt over de deltagende klasser som må til for å lage en komplett innlesning av et dataformat. Her er de implementert som separate klasser, men kan også implementeres som én klasse.



Figur 2.3 Deltagere i innlesningsfunctionalitetet til FC

En datahandler består av en eller flere klasser som har implementert de tre grensesnittene IDataHandlerInterface, IStorageInterface, IDumperInterface for det aktuelle dataformatet. Klassen som implementerer IDataHandlerInterface skal så føres opp i den statiske listen som finnes i klassen DataHandlerFramework (se Figur 2.5). All tilgang til klassen(e) som har implementert IStorageInterface, IDumperInterface skal gå via klassen som har implementert IDataHandlerInterface. Hensikten med å dele opp i tre grensesnitt er at de har hver sine oppgaver og benyttes i forskjellige deler av FC. Det gir også muligheten til å implementere grensesnittene i hver sin klasse.



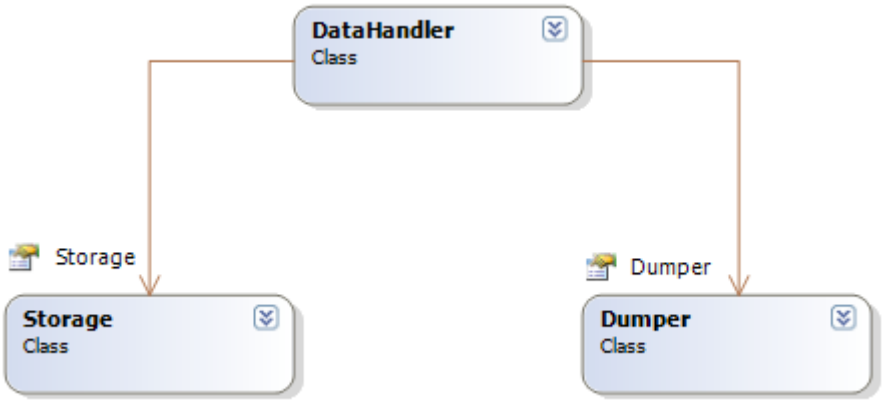
Figur 2.4 Grensesnitt (Interfaces) som må implementeres

IDataHandlerInterface skal hjelpe GUI delen av applikasjonen med å tolke datafilene og fylle ut dialogen hvor valg av data for innlesing til databasen blir foretatt. I tillegg skal den levere et Dumper-objekt eller Storage-objeket for dette dataformatet. Hensikten er å få et felles grensesnitt mot klassene/objektene som håndterer det aktuelle dataformatet. Når en data handler klasse er implementert (klasse med IDataHandlerInterface) må denne legges til i data handler rammeverks-klassen. Se Figur 2.5. Da *DataHandlerFramework* – klassen har et *using*[4] direktiv for namespace[4] *FennikelCollector.DataHandle* er det lettere å lage de nye *DataHandle* klassene i samme namespace.

```
public static class DataHandlerFramework
{

    public static IDataHandlerInterface FindDataHandlerForType(string dataTypeName)...


    public static IDataHandlerInterface[] FindAllDataHandlerForType(byte[] dataExample,
                                                                    string deviceName,
                                                                    object sourceDevice)...


    private static IDataHandlerInterface[] _dataHandlerList =
        {
            new MSIFCTacticalDataHandler(),
            new NavigationRmcDataHandler(),
            new NavigationUvbDataHandler(),
            new MSIFCTarDataHandler(),

            // Unknown should always be the last handler.
            new UnknownDataHandler()
        };

}
```

Figur 2.5 DataHandlerFramework – klassen med markering av hvor nye data handlere skal kreeres.

En typisk organisering kan være tre klasser som må implementeres hvert sitt grensesnitt, som vist i Figur 2.6. Her vil DataHandler objektet bli kreert av rammeverksklassen og de andre klassene (Storage og Dump) blir kun kreert ved behov ved hjelp av DataHandler klassen.

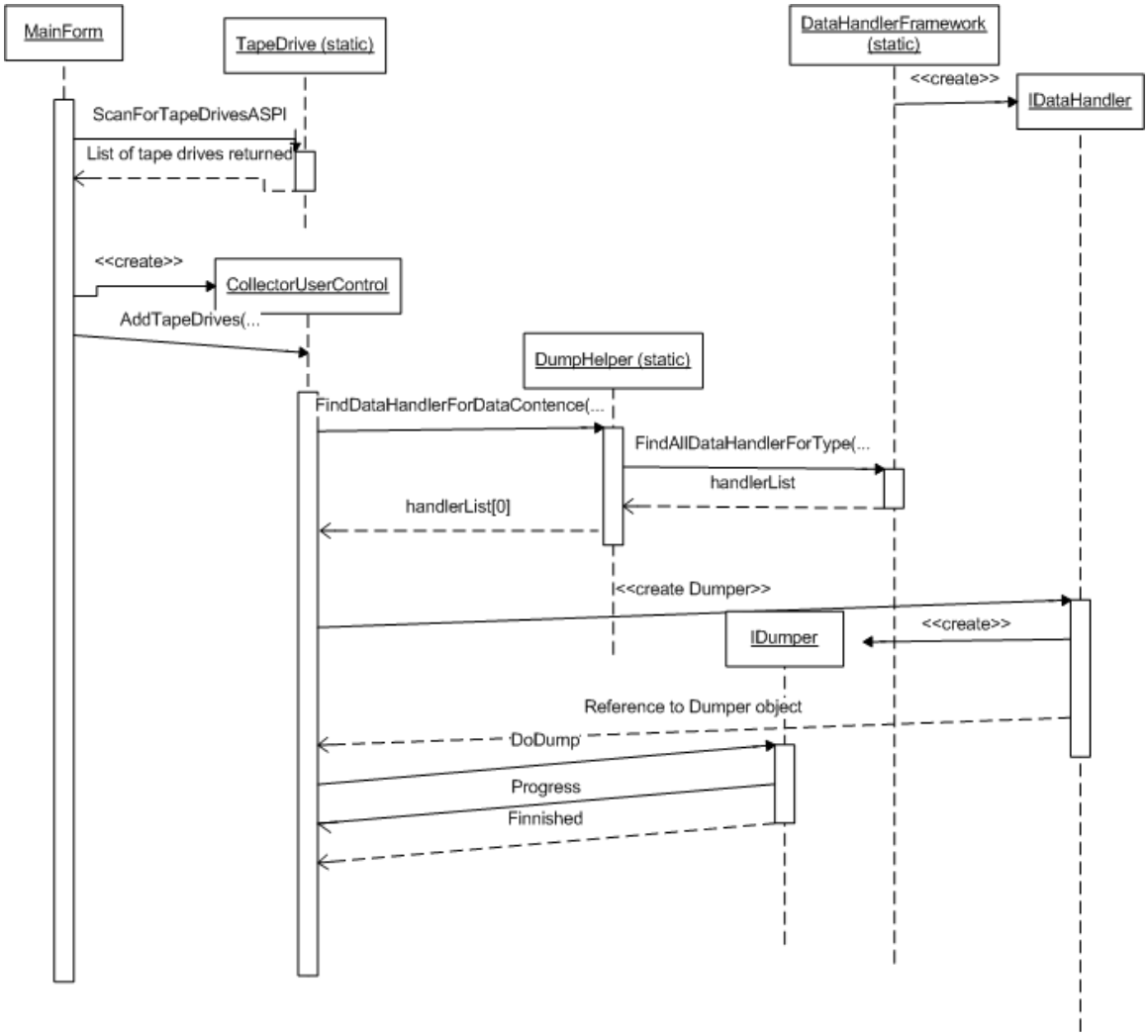


Figur 2.6 Typisk organisering av et datainnlesingsoppsett

De neste kapitlene beskriver først kommunikasjonen mellom rammeverket og den øvrige programvaren, og så rammeverket samt dets funksjon, og deretter beskrives grensesnittet som må implementeres med forklaring av parameterene.

2.1 Interaksjonsdiagrammer

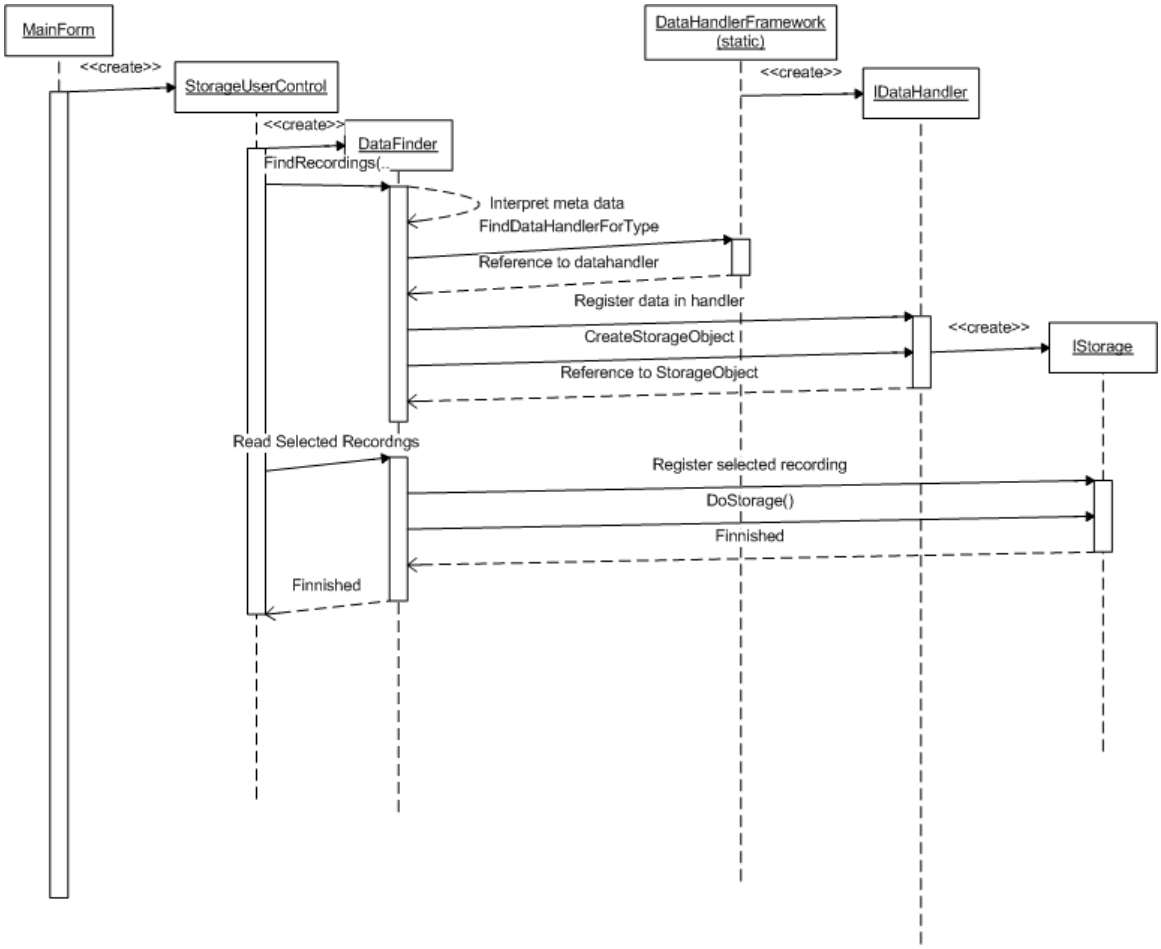
For å gi en grafisk fremstilling av kommunikasjonen mellom hovedklassene i tid er det tegnet sekvensdiagrammer inspirert av UML teori [5].



Figur 2.7 Sekvensdiagram for Dumper grensesnittet

Figur 2.7 viser kommunikasjonen mellom DataHandlerFramework og FC's brukergrensesnitt i forbindelse med lagring av dataopptak til harddisk.

Ved oppstart av applikasjonen blir eventuelle tape drev detektert og en liste av drev blir opprettet. Brukergrensesnittet for innsamling av data blir opprettet, og listen over tape drev levert. Brukeren setter i gang innlesning fra enten tape eller annet lagringsmedia. Da blir den statiske klassen DumpHelper kalt for å levere en passende datahandler-klasse til FC. DumpHelper kaller DataHandlerFramework og for levert en liste med referanse til aktuelle klasser, og den første blir videresendt til FC. FC kan nå direkte be datahandeleren om den instans av en klasse som kan lagre dataene til harddisk.



Figur 2.8 Sekvensdiagram for Storage grensesnittet

Figur 2.8 viser igjen kommunikasjonen mellom DataHandlerFramework og FC's brukergrensesnitt, men her i forbindelse med lagring til Simson databasen.

Klassen DataFinder leser alle meta data som måtte ligge i en gitt mappe. For alle datatyper som er funnet i disse meta dataene blir det opprettet en DataHandler. DataHandleren gir så en referanse til en Storage-klasse for den aktuelle datatypen.

Etter at brukeren av FC har valgt hvilke data som skal leses inn i databasen vil disse registreres, og innlesningen aktiviseres.

2.2 DataHandlerFramework

Denne klassen skal hjelpe GUI delen av Simson Fennikel Collector med håndteringen av de forskjellige dataformatene. Det er Collector (dumper) og Storage delen av programmet som benytter denne kassen. Disse har hver sin arkfane i Simson Fennikel Collector.
Collector delen benytter DataHandlerFramework klassen for finne ut hvilken type data som er på filen eller tapen. Denne informasjonen skal legges i metadatafilen og blir benyttet av Storage delen av programmet når disse dataene skal overføres til databasen. I tillegg til å forstå alle grensesnittene må en som implementerer en ny leser/skriver, forstå *DumpMetaData* klassen.

Denne klassen benyttes når metadata om et datasett, skal lagres på fil. Filen er en XML serialisering av selve DumpMetaData klassen. Denne metadata-filen blir senere benyttet for å finne riktig skriverobjekt for overføring av data til databasen. *DumpMetaData* klassen kan ha en liste av *DumpMetaDataElement* klasser. Disse elementklassene kan benyttes til intern informasjon mellom dumperfunksjonen og storagefunksjonen. Data i *DumpMetaDataElement* blir ikke tolket av selve GUI delen av FC. Denne funksjonaliteten er benyttet av to lesere i dag

MSIFCTacticalDataHandler
MSIFCRawDataHandler

MSIFC taktiskedata består ev en ZIP-fil som inneholder flere datafiler. Her benyttes *DumpMetaDataElement* til å liste opp filnavnene på filene i ZIP-filen. I rådatainnleseren blir offseten til hver enkel datapakke lagret i dette arrayet med elementer.

En av funksjonene i IDataHandlerInterface benytter en delegate som også er deklarert i *DataHandlerFramework* -klassefilen. Signaturen er vist i Figur 2.9. Hver gang denne kalles kommer det en ny linje i FC Storage-liste.

```
public delegate void PopulateListDelegate(  DumpMetaData myDumpFileInfo,
                                            string[] possibleRecordingTypes,
                                            string msifcExerciseName,
                                            int dataHandleListIndex);
```

Figur 2.9 Signaturen på funksjonen som blir benyttet til å fylle elementer i datatypelisten

2.3 Grensesnittene

Dumper klassen skal bidra i lesing fra ytre enhet til disk og lagre tilhørende metadata. Storage skal tolke metadata og overføre data fra fil til databasen. Datahandler har til oppgave å lage kun ett grensesnitt til Storage – Dumper paret, og tolke datatypen og returnere navnet på datatypen. DataHandler klassen vil være koblingspunktet mot FC.

ToString funksjonen, som alle objekter har, kan bli *overridet*[4] for å gi en beskrivelse av selve datahandlerklassen. Denne kan benyttes til å gi en beskrivende tekst. Behovet for en slik tekst finnes hvis det blir implementert flere alternative datahandlere for et dataformat. Da blir det behov for å velge datahandler fra en liste.

DataHandlerFramework klassen er en hjelper klasse som binder det hele sammen, slik som vist i Figur 2.5.

For å implementere de enkelte grensesnittene, må en forstå hensikten med grensesnittet og hva de enkelte funksjonene gjør.

2.3.1 IDataHandlerInterface

Dette grensesnittet blir benyttet av både lese og skrive delen av GUI. Grensesnittet har følgende signatur.

```
public interface IDataHandlerInterface
{
    string DataFormatName { get;}
    /// <summary> ...
    bool IsDataFormat(string dataFormatName);

    /// <summary> ...
    bool IsDataFormat(Byte[] dataExample, string deviceName, object sourceDevice);

    /// <summary> ...
    void PopulateDatatypeList(DumpMetaData metaData, PopulateListDelegate PopulateList, int handleListIndex);

    /// <summary> ...
    IStorageInterface CreateStorageObject();

    /// <summary> ...
    IDumperInterface CreateDumperObject();

}
```

Flere av funksjonene i dette grensenittet er banale å implementere, men gjør at det er enkelt for FC å finne riktige klasser under kjøring.

2.3.1.1 `string DataFormatName { get;}`

Denne funksjonen skal returnere navnet på datatypen i form av en streng. Denne vil bli brukt til å gjenkjenne datatypen ved senere anledninger.

Eksempel:

```
public string DataFormatName
{
      get { return "MSIFC_Tactical"; }
}
```

2.3.1.2 `bool IsDataFormat(string dataFormat)`

Denne funksjonen skal returnere true hvis vi kan lese inn denne datatypen. Med andre ord skal den returnere true hvis dataFormat parameteren er lik DataFormatName propertien. Den eneste implementasjonen som trengs for denne typen er som vist i eksemplet.

Eksempel:

```
public bool IsDataFormat(string dataFormat)
{
      return dataFormat == DataFormatName;
}
```

2.3.1.3 `public void PopulateDatatypeList(DumpMetaData metaData, PopulateListDelegate PopulateList, const int dataHandleListIndex)`

Denne funksjonen skal ved hjelp av *metaData* kalle funksjonen *PopulateList* (Figur 2.9) med en liste med navn over de formater som denne DataHandler-klassen kan håndtere. Samtidig skal den videresende metadata og *dataHandleListIndex* som parametere til *PopulateList*. Det er viktig å sende parameteren *dataHandleListIndex* videre til *PopulateList*. Den siste parameteren til

PopulateList er *msifcExcersiseName* og benyttes av *MSIFCTactial* leseren. I eksemplet under er denne parameteren satt til null og listen inneholder bare *DataFormatNavnet*. Se eksemplet under.

Eksempel:

```
public void PopulateDatatypeList(DumpMetaData metaData,
                        PopulateListDelegate PopulateList,
                        const int handleListIndex)
{
PopulateList(metaData, new string[] { DataFormatName },
                        null, handleListIndex);
}
```

2.3.1.4 `IStorageInterface CreateStorageObject();` `IDumperInterface CreateDumperObject();`

Disse to funksjonene returnerer hvert sitt objekt som implementerer de aktuelle grensesnittene. Det kan være forskjellige klasser, eller så kan det være samme klasse som implementerer alle grensesnittene. Dette tillater at man for eksempel bruker samme dumper klasse for flere datatyper.

Eksempel:

```
public IStorageInterface CreateStorageObject()
{
      return new MyStorageDataHandler();
}
public IDumperInterface CreateDumperObject()
{
      return new MyDumperDataHandler();
}
```

2.3.2 IDumperInterface

Klasser som implementerer IDumperInterface har to oppgaver. Den første oppgaven er å overføre data fra enheten (fil eller tape) til en harddisk, og deretter lage en metadatafil til det overførte datasettet. Genereringen av metadata er nesten like viktig som å kopiere data fra ett media til et annet.

En viktig ting å huske er at målnavnet på filene må endres hvis det finnes samme navn fra før. Det finnes en funksjon som heter *DumpHelper.CreateDestinationFileNames* som kan hjelpe deg med dette. For å lage en metadatafil benyttes klassen *DumpMetaData.* Den enkleste formen for metadata blir laget ved hjelp av to linjer kode.

Eksempel:

```
// Creating Meta file
DumpMetaData totalFileDumpInfo =
         new DumpMetaData(date, mediaTitle, unit,datasetComments,
         destFilePath, recordingFormat);
totalFileDumpInfo.Serialize(xmlFilename);
```

Alle nødvendige data for å opprette en korrekt metadatafil kommer inn med funksjonskallet *DoDump*. For avanserte brukere er det mulig å manipulere med standard funksjonaliteten. I MSIFCTactical finnes det eksempler på avansert bruk. Her blir en datakilde (tape) splittet i flere målfiler med én tilhørende metafil.

```
public interface IDumperInterface
{
    event EventHandler<FenikkelCollector.FileDumping.TAPI.DumpProgressEventArgs> DumpProgressUpdate;

    bool IsDumpingActive {get;}

    bool DoDump(DateTime date, string mediaTitle, string unit,
            string datasetComments, string dumpDirectory, string recordingFormat, object sourceDevice);

    void CleanUp();
}
```

Figure 2.1 Kildekoden til IDumperInterface. Grensesnittet inneholder ett event, en variabel og to metoder.

2.3.2.1 `event EventHandler<DumpProgressEventArgs> DumpProgressUpdate`

DumpProgressUpdate er et *event*[4] som er tiltenkt å gi oppdateringer i GUI for å vise fremgangen av dumpingen til bruker. Denne bruker klassen DumpProgressEventArgs for å formidle data. Her har man mulighet til å sende tekst og to verdier.

Eksempel:

```
// Sending progressupdate if the event is listened to by i.e. GUI
if (DumpProgressUpdate != null)
{
      TimeSpan spTid = new TimeSpan(DateTime.Now.Ticks -
      lastTime.Ticks);
      // If there is over 2 sec since last update was sendt
      if (spTid.Seconds >= 2)
      {
            // In case of new file
            if (_currentFileSize < lastCurrenFileSize)
            lastCurrenFileSize = 0;

            lastTime = DateTime.Now;
            double dumpspeed = (_currentFileSize -
            lastCurrenFileSize) / (1024 * spTid.TotalSeconds);
            DumpProgressUpdate(this,
                  new DumpProgressEventArgs(-1, dumpspeed));
            lastCurrenFileSize = _currentFileSize;
      }
}
```

Hvis det ikke er mulig eller ønskelig å gi et tall som gir fremgangen så kan man sende verdien ”-1”. Da vil fremgangen vises som en pågående prosess i GUI’et.

2.3.2.2 `bool IsDumpingActive`

Dumping prosesser anbefales at settes ut i tråder slik at GUI vil være responsivt. Denne propertien vil kunne si om tråden kjører eller ikke.

Eksempel:

```
/// <summary>
/// Checks if the dumping thread is active
/// </summary>
public bool IsDumpingActive
{
	get
	{
		if (_thread != null)
			return _thread.IsAlive;
		else
			return false;
	}
}
```

2.3.2.3 `bool DoDump(DateTime date, string mediaTitle, string unit, string datasetComments, string dumpDirectory, string recordingFormat, object sourceDevice)`

DoDump er hovedmetoden i dumper grensesnittet. Dette kallet starter dumpingen av data fra ett media til et annet. Innparametrene kommer hovedsaklig fra Dumper GUI, mens sourceDevice er en referanse til klassen som gir aksess til data som skal dumpes. Per i dag er det en instans av klassen TapeDrive[1].

Eksempel:

```
/// <summary>
/// Starts dumping tape to disk
/// </summary>
public bool DoDump(DateTime date, string mediaTitle,
 string unit, string datasetComments,
 string dumpDirectory, string recordingFormat,
 object sourceDevice)
{
	TapeDrive tape = sourceDevice as TapeDrive;
	if (tape == null)
		return false;
	this._tape = tape;
	this._dumpDir = dumpDirectory;
	this._exercise = mediaTitle;
	this._unit = unit;
	this._comments = datasetComments;
	this._recFileType = recordingFormat;
	this._date = date;

	_thread.Start();
	return true;
}
```

[1] TapeDrive fungerer på tapedrev, CD/DVD-drev og harddisker.

2.3.3 IStorageInterface

Klasser som implementerer IStorageInterface vil inneholde metoder for tolkning av data opptak som er lagret på harddisk for videre lagring i databasen.

Grensesnittet inneholder et event for tilbakemelding til GUI om fremgangen på lagringen, en klasse som inneholder informasjon om dataopptaket og to metoder: *StoreThisData* som samler opp alle ønsket utleste datatyper som støttes av klassen som implementerer grensesnittet. Og *DoStorage* som starter konverteringen av alle valgte data og lagrer dem i databasen.

```
/// <summary>
/// This is the interface of data converters (rec => database ) used by the datahandlers
/// </summary>
public interface IStorageInterface
{
    event EventHandler<FenikkelCollector.FileDumping.TAPI.DumpProgressEventArgs> StorageProgressUpdate;

    DumpMetaData MetaDataObject { get; set;}

    void StoreThisData( int excerciseID, int unitID, string recordingType,
                        string sourceFilename, string extraExerciseInfo);

    void DoStorage();
}
```

Figure 2.2 Kildekoden av grensesnittet. Grensesnittet inneholder ett event, en klasse for informasjon og dataopptaket og to metoder. StoreThisData for registrering av hvilke data som skal inn i databasen, og DoStorage som starter selve innlesningen.

2.3.3.1 `event EventHandler<DumpProgressEventArgs> StorageProgressUpdate`

StorageProgressUpdate er en hendelse som er tiltenkt å gi oppdateringer i GUI for å vise fremgangen av konverteringen og databaselagringen til bruker. Denne bruker klassen DumpProgressEventArgs for å formidle data. Her har man mulighet til å sende tekst og to verdier.

Eksempel:

```
/// <summary>
/// Sends progress event
/// </summary>
/// <param name="message">Text message of progress</param>
/// <param name="progress">Value of progress [0..1]</param>
private void SendProgressUpdate(string message, double progress)
{
  if (StorageProgressUpdate != null)
  StorageProgressUpdate(this, new FenikkelCollector.FileDumping.
                    TAPI.DumpProgressEventArgs(message, progress));
}
```

I eksemplet over har man lagd en egen hjelpemetode for å sende hendelsen videre. Utfordringen med oppdatering av fremgangen ved innlesning er å finne den faktiske fremgangen. Hvis det ikke er mulig eller ønskelig å gi et tall som gir fremgangen så kan man sende verdien ”-1”. Da vil fremgangen vises som en pågående prosess i GUI’et.

2.3.3.2 `DumpMetaData MetaDataObject { get; set;}`

Denne variabelen er en instans av klassen *DumpMetaData* som er den deserialiserte XML-filen generert gjennom *IDumperInterface*. Denne klassen inneholder informasjon som brukes for å definere dataopptaket i databasen. Instansen blir lagd av GUI'et, så klassen som implementerer *IStorageInterface* behøver ikke å opprette dette objektet siden det gjøres automatisk. Det man må gjøre er å opprette en lokal versjon av variabelen som vist i eksemplet under.

Eksempel:

```
private DumpMetaData _metaDataObject = null;
public DumpMetaData MetaDataObject
{
  get { return _metaDataObject; }
  set { _metaDataObject = value; }
}
```

2.3.3.3 `public void StoreThisData(int excerciseID, int unitID, string recordingType, string sourceFilename, string extraExerciseInfo)`

Metoden *StoreThisData* er metoden for å registrere alle data som skal leses inn i databasen ved å bruke den gjeldende klassen som implementerer grensesnittet *IStorageInterface*. Informasjonen som sendes er fylt ut av GUI-laget av FC. Dataene som sendes her er nødvendige for å lagre dataopptakene korrekt i databasen.

En klasse som implementerer grensesnittet kan enten lese inn kun én type data, eller så kan det hende at det er naturlig å lese inn flere typer data som for eksempel ligger i samme fil. Hvis det første er tilfellet behøver man kun å lagre dataene som kommer inn i metoden til lokale variable i klassen.

Eksempel:

```
public void StoreThisData(int excerciseID, int unitID, string
recordingType, string sourceFilename, string extraExerciseInfo)
{
  _excerciseID = excerciseID;
  _unitID = unitID;
  _sourceFilename = sourceFilename;
  _recordingType = recordingType;
  _extraExerciseInfo = extraExerciseInfo;

  _intitiated = true;
}
```

Hvis en fil kan inneholde flere datatyper må man utvide metoden til å håndtere dette. Eksemplet under viser håndteringen av en fil som inneholder tre typer data. Først sikrer man at den felles informasjonen om opptaket er registrert, og deretter tester man på datatypen som er avhuket i GUI og som har trigget metodekallet. I eksemplet er leseren (*_reader*) skilt ut i en egen klasse.

Denne inneholder flagg man kan sette for å definere hvilke datatyper som skal leses inn fra filen beskrevet i opptaksinformasjonen (*_recording*).

Eksempel:

```
public void StoreThisData(int excerciseID, int unitID, string
recordingType, string sourceFilename, string extraExerciseInfo)
{
  if (_reader == null)
  {
    _reader = new MsifcHmsRawReader(MetaDataObject);
    _reader.StorageProgressUpdate += new EventHandler<
            FenikkelCollector.FileDumping.TAPI.DumpProgressEventArgs>
            (_reader_StorageProgressUpdate);
    _recording = new Recording();
    _recording.ParentExercise = new Exercise();
    _recording.ParentExercise.ID = excerciseID;
    _recording.RecordingUnit = new Unit();
    _recording.RecordingUnit.ID = unitID;
    _recording.SourceFile = new System.IO.FileInfo(sourceFilename);
    _recording.SourceDetails = "Binary data";
  }

  switch(recordingType)
  {
    case HMS_BMF_SPY:
      _reader.ReadBeamFormed = true;
      break;
    case HMS_Raw:
      _reader.ReadRawVideo = true;
      break;
    case HMS_Reverb:
      _reader.ReadReverberation = true;
      break;
  }
}
```

2.3.3.4 public void DoStorage()

DoStorage metoden setter i gang konverteringen av dataopptak til klassene gikk i en definert datamodell for så å lagre dataene i databasen.

Etter at alle valgte dataopptak er valgt i GUI og *StoreThisData* er kalt for hver av dem, vil FC kalle metoden *DoStorage*. Denne metoden vil inneholde et kall til en metode i leserklassen som starter selve innlesningen og lagringen.

Eksempel:

```
public void DoStorage()
{
  try
  {
    _reader.Read(_recording);
  }
  catch (Exception ex)
  {
    throw ex;
  }
```

```
  finally
  {
    // Resetting the reader used by the Data Handler
    _reader.StorageProgressUpdate -= new EventHandler<
             FenikkelCollector.FileDumping.TAPI.DumpProgressEventArgs>
             (_reader_StorageProgressUpdate);
    _reader = null;
  }
}
```

Feil som kommer under innlesning vil sendes videre til GUI-laget for visning. Mer om registrering og sending av feil i 2.3.4.

2.3.4 Håndtering av feil i en data handler

I en data handler burde feilmeldingene til brukeren være så detaljert at man kan spore feilen ned til metoden den kom fra. Dette er fordi det da blir enklere for programmereren å finne metoden som feiler. Feil som ”Object not set to an instance of an object” har ingen mening alene og burde aldri bli vist uten noe mer informasjon.

Dette løses i FC ved at data handlerne har *try-catch* rundt alle metoder som kan feile. I metoder som er nederst i strukturen kan man teste og kaste feil selv eller sende feilmeldingen mottatt. Videre oppover i strukturen legger metodene som har kalt metoden som feilet på sitt eget metode navn, og kaster feilmeldingen videre. Tilslutt fanges feilen opp av GUI-laget og feilmeldingen vises. Data handleren sørger for at den blir resatt slik at FC ikke må startes på nytt.

Eksemplet som følger viser en forenkling av hvordan sending av feilmeldinger i en data handler er. Som man ser legger hver metode opp til et fornuftig nivå med informasjon slik at feilmeldingen kan spores til den metoden som feilet. Man må huske på at det er vanlig å ha likt navn på metodene i ulike data handlere. For eksempel vil en feil fra en metode kalt *Read* kunne komme fra mange forskjellige data handlere. Derfor er det viktig å få med hvilken data handler som sender feilmeldingen til GUI-laget.

Eksempel:

```
private void MetodeSomFeiler()
{
  try
  {
    double tall = Double.Parse("tolv");
  }
  catch(Exception ex)
  {
    throw new Exception("Feil i MotodeSomFeiler:\n" + ex.Message);
  }
}

private void MetodeSomKallerEnAnnenMetode()
{
  try
  {
    MetodeSomFeiler();
  }
  catch(Exception ex)
  {
    throw new Exception("MetodeSomKallerEnAnnenMetode\n" + ex.Message);
  }
}

private void button1_Click(object sender, EventArgs e)
{
  try
  {
    MetodeSomKallerEnAnnenMetode();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message, "Feil",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
  }
}
```

Feilmeldingen man får når man klikker på *button1* er vist i Figure 2.3.

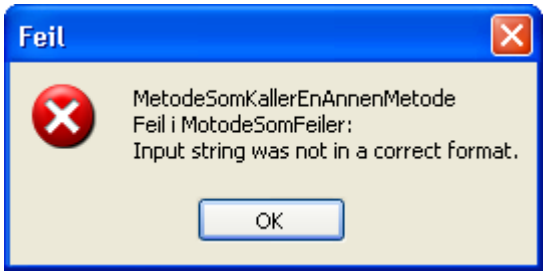


Figure 2.3 Resulat av en feilmeldingskjede som er lik den som implementeres i en data handler. Dette gjør det mulig å spore feilen ned til opphavet.

2.3.5 Dokumenterte løsninger basert på datahandler designet

Sommerstudenten Anders Båtstrand dokumenterte sin implementering av en datahandler for behandling av data fra Dacula [6]. Dette er en avansert løsning på en datahandler. Den tar inn

flere filer som dekkes av ett dataformat. Dataformatet inneholder mange datatyper som blir lest inn i minnet. Arbeidet inkluderer ikke databaselagring, men forbereder data for lagring i database på en god måte.

2.3.6 Forslag til fremtidige endringer av grensesnittene

Løsningen med grensesnitt er mest aktuell hvis metodene, som skal støttes av klassen for å implementere grensesnittet, har forskjellig innhold. Et eksempel er hvis en metode skal kopiere klassen, så løses dette forskjellig fra klasse til klasse. DataHandlere som lages for FC vil i stor grad ha lik kode i metodene, og derfor kunne det vært hensiktsmessig å bruke en abstrakt klasse som grunnlag for DataHandlerene. Dette ble vurdert i den tidlige designfasen, men ble vurdert som unødvendig. Etter at DataHandler designet har blitt brukt til å utvikle flere DataHandlere, ser man at spesielt IDataHandlerInterface kan erstattes med en abstrakt klasse. IDumperInterface og IStorageInterface er det fremdeles aktuelt å ha utskilt i egne grensesnitt.

Referanser

[1] J. Wegge and E. Tveit, "**OVERALL DESCRIPTION OF AN ANALYSIS AND SIMULATION TOOL FOR ACOUSTIC ASW SENSORS**,"**FFI/RAPPORT-2003/01928**, 2003.

[2] A. L. Gjersøe, S. Alsterberg, and J. Wegge, "**Prosjekt 899 "Nansen-klasse fregatt, evaluering" med på "Marinen i Nord" 2005 - Toktrapport**,"**2006/02728**, 2008.

[3] J. Wegge, S. Alsterberg, and K. T. Hjelmervik, "**DESIGN DOKUMENT FOR SIMSON FENNIKEL - ET QUICKLOOK VERKTØY FOR AU-VIRKSOMHET MED FRIDTJOF NANSEN-KLASSEN FREGATTER**," FFI rapport **2006/02267**, 2006.

[4] Christian Nagel, Bill Evjen, Jay Glynn, Morgan Skinner Katrli Watson, and Allen Jones, *Professional C# 2005* Wiley Publishing, 2006.

[5] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* 1999.

[6] A. L. Båtstrand, "**Fra Dacula til Simson Fennikel** - en innleser til Simson Fennikel Collector,"08/00197, 2008.

Forkortelser

ASW	Anti Submarine Warfare
FC	Simson Fennikel Collector
FFI	Forsvarets Forskningsinstitutt
GUI	Graphical User Interface – grafisk brukergrensesnitt
MSIFC	Multi Sensor Integrated Fire Control – kampsystemet for bl.a. sonar
UML	Unified Modeling Language