Alex Thissen
is werkzaam bij Twice IT Training.

Illustraties: BartZelf, Bart Jansen

Exception Management

PLANNING EN AFHANDELING VOLGENS EEN EXCEPTION-STRATEGIE

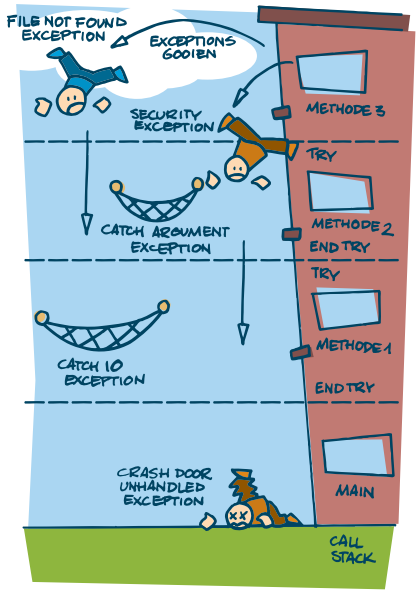
Exceptions treden op. Dat zegt niets over de kwaliteit van de programmeur of diens code. Die wordt eerder bepaald door de manier waarop met exceptions wordt omgegaan. De correcte afhandeling van exceptions is de eerste stap naar een robuustere applicatie. De volgende stap is het werken volgens een strategie voor het detecteren, loggen en rapporteren van exceptions. Dat is exception management: de planning voor het optreden van exceptions en ze volgens de geteste strategie afhandelen.

Dit artikel gaat er van uit dat u al enigszins bekend bent met Structured Exception Handling (SEH). In het artikel Exceptions en Events in de derde uitgave van Microsoft .NET Magazine is SEH al behandeld. Deze kunt u er eventueel nog op nalezen.[AT1]

Propageren van exceptions

Het krachtige aan exceptions is dat zij door het systeem heen propageren [RM2]naar een potentiële afhandelingroutine. Daarin zijn exceptions anders dan returnwaarden, die niet propageren en bovendien genegeerd kunnen worden. Wanneer binnen een try/catch-constructie de exception door geen enkel van de catch-blokken wordt afgevangen, dan zal de callstack worden afgewikkeld. Door deze propagatie zullen lager op de stack gelegen methoden een kans krijgen de exception af te vangen. Afbeelding 1 toont de callstack zoals deze in Visual Studio .NET tijdens het debuggen te zien is.

Afbeelding 1. 1 Callstack in Visual Studio.NET



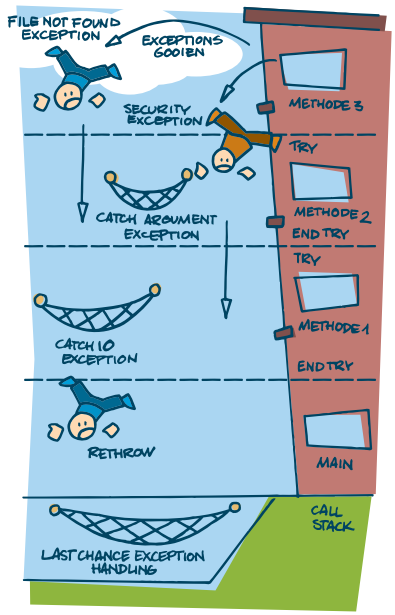
Afbeelding 2. Exception-propagatie en unhandled exceptions

Een opgetreden exception zal de callstack steeds verder omlaag gaan, naar de aanroeper van de aanroeper, enzovoort. Er ontstaat een unhandled exception wanneer de callstack volledig is afgewikkeld zonder dat enige applicatiecode met een try/catch-constructie de opgetreden exception afhandelt. Het is zeer goed mogelijk dat tijdens het propageren de CLR verscheidene try/catch-constructies passeert, maar dat nergens het specifieke exception-type wordt afgevangen. In afbeelding 2 is op vereenvoudigde wijze aangegeven hoe de afwikkeling van de callstack plaatsvindt en hoe unhandled exceptions ontstaan.

Unhandled exceptions

In het geval van unhandled exceptions zal de CLR ingrijpen. Voor console- en Windows-applicaties en Windows Services betekent dit dat de default system handler van de CLR de exception afvangt, aan de gebruiker meldt of logt en vervolgens het proces beëindigt. Bij ASP.NET webapplicaties en webservices zal het proces niet beëindigd worden. De CLR zal met een top-level exception handler de exception afvangen en een error-pagina of SOAP error-message terugsturen via Internet Information Server. Unhandled exceptions zijn altijd ongewenst. Op een ongecontroleerde manier wordt een request of proces beëindigd. De CLR biedt diverse handvatten om unhandled exceptions alsnog af te vangen. Dit zou ik last-chance exception handling willen noemen. Daarmee kan in ieder geval op een sierlijke manier afscheid genomen worden, zoals met een nette, duidelijke melding naar de gebruiker en het loggen en melden

van nuttige debugging-informatie. Afbeelding 3 toont een voorstelling van een last-chance exception handler.



Afbeelding 3. Last-chance exception handler en rethrow

Last-chance exception handling voor application domains

Voordat de CLR een application domain (AppDomain) termineert zal deze het UnhandledException event van het betreffende AppDomain-object afvuren. Door een event handler toe te voegen kan de exception worden afgehandeld. Listing 1 toont een consoleapplicatie die het event afvangt. De UnhandledExceptionEventArgs-parameter heeft twee properties: ExceptionObject en IsTerminating. Het ExceptionObject van het type Object (zodat ook non-CLS compliant exceptions afgevangen kunnen worden) moet worden gecast naar Exception. De Boolean property IsTerminating geeft aan of het proces zal worden beëindigd. Dit wordt in de meeste situaties gedaan, behalve bij exceptions die worden veroorzaakt in worker-threads.

In Windows-applicaties is het bovendien nog mogelijk en nodig om exceptions van de window-thread op te vangen. Het gaat hierbij om exceptions die ontstaan tijdens het uitvoeren van window-procedures (wndprocs) zoals in control event handlers. De System.Windows.Forms.Application-klasse heeft daarvoor een ThreadException-event dat net als bij de AppDomain.UnhandledException een handler kan krijgen. De signature van de eventhandler bevat nu echter een ThreadExceptionEventArgs-parameter, die een Exception-property heeft. Daarmee heeft u beschikking over de exception die is opgetreden.

Last-chance exception handling in ASP.NET runtime

In ASP.NET -webapplicaties kunnen op meer niveaus unhandled exceptions worden afgevangen. Dit kan binnen een pagina gedaan worden, via het Page.Error-event. In de van Page afgeleide klasse moet een handler voor dit event zijn opgenomen, zoals listing 2 laat zien.

In dit geval dus geen try/catch-logica, maar eerder traditionele error handling. Daarom moet de errormelding worden gewist doormiddel van het aanroepen van HttpServerUtility.ClearError(), zodat de ASP.NET runtime niet alsnog ingrijpt. Op applicatieniveau zijn alle unhandled exceptions te vangen met behulp van het Error-event van de HttpApplication-klasse. U neemt daarvoor een subroutine Application_Error op in de Global.asax-file, waarin u net als in de Page.Error event handler de exception-informatie met HttpServerUtility.GetLastError() benadert. Overigens kan met behulp van het ErrorPage-attribuut van het @Page-directive per pagina een custom error-pagina worden aangewezen, of voor de hele applicatie met een <customErrors> sectie in web.config. In de errorpagina is dan ook GetLastError te gebruiken. Bij ASP.NET Web Services worden unhandled exceptions ook afgevangen door de CLR. Naar de client wordt een XML-geserialiseerde SoapException teruggestuurd. De client kan de aanroepen naar het proxy-object in een try-block plaatsen en de potentiële

```
Imports System
Public Class TheApp
  Public Shared Sub Main()
    AddHandler AppDomain.CurrentDomain.UnhandledException, _
        AddressOf LastChanceExceptionHandler
    Throw New ApplicationException("Bewust gegooide exception")
  End Sub
  Private Shared Sub LastChanceExceptionHandler(ByVal sender As _
  Object, ByVal args As UnhandledExceptionEventArgs)
    Dim ex As Exception
    ex = DirectCast(args.ExceptionObject(), Exception)
    Console.WriteLine("Meld het volgende aan de helpdesk: " & _
    Environment.NewLine & ex.ToString())
    If args.IsTerminating Then
      Console.WriteLine("De applicatie wordt nu beeindigd.")
    End If
  End Sub
End Class
```

Listing 1. Last-chance exception handling per application domain

```
Public Class MyWebForm
  Inherits System.Web.UI.Page
  Private Sub Page_Error(ByVal sender As Object, _
  ByVal e As EventArgs) `Handles MyBase.Error
    Dim ex As Exception
    ex = Server.GetLastError()
    'Verdere afhandeling exception
    Server.ClearError()
  End Sub
End Class
```

Listing 2. Last-chance exception handling in een webpage

SoapExceptions afvangen. De InnerException bevat het oorspronkelijke exception-object, mits deze serialiseerbaar is.

Strategieën voor afhandeling van exceptions

Er is een aantal mogelijkheden om met de opgetreden exceptions om te gaan. In grote lijnen zijn er een viertal opties:

- Exceptions laten propageren
- Afhandelen
- Afvangen en opnieuw gooien
- Afvangen en een andere gooien (wrappen van exceptions)

Bij ieder van deze opties komt een aantal specifieke zaken om de hoek kijken.

Laten propageren

De allersimpelste oplossing is niets te doen. De exception gaat dan automatisch de callstack af. Hiermee wordt de aanname gemaakt dat een eerdere aanroeper de afhandeling zal doen. Het grote risico is dat niemand de exception afhandelt en dat er een unhandled exception ontstaat. Toch zijn er momenten dat u de exceptions laat propageren; waarover zo meteen meer. Eventueel kan bij deze optie een try/finally-constructie (dus zonder catch-blokken) worden gebruikt. Er is dan de gelegenheid om gegarandeerde code in het finally-block uit te voeren, voordat de exception gaat propageren.

Interventie

De laatste drie opties maken gebruik van interventie na het optreden van de exception. Het is denkbaar dat de afhandeling bestaat uit het 'annuleren' van de exception. In dat geval vangt u de exception af en onderneemt daarna geen verdere actie, behalve wellicht het melden aan de gebruiker of het loggen van de exception. Lager gelegen methoden op de callstack merken in dat geval niets van de exception. Probeer bij het inzetten van deze optie het 'catch all' block (Catch ex As Exception) zo weinig mogelijk te gebruiken. Er zijn namelijk systeem-exceptions die gegooid worden in situaties waarin de CLR niet meer correct kan functioneren: ExecutionEngineException, OutOfMemoryException en StackOverflowException. Het afvangen en annuleren van deze exceptions zou de CLR in een onstabiele toestand laten functioneren. De enige plek waar u deze drie exceptions afvangt is in een last-chance exception handler, waar de CLR het proces toch al zal beëindigen.

Idealiter kan het probleem na het afvangen worden opgelost of gecorrigeerd. Daarna kan de veroorzakende code opnieuw proberen of er verder kan worden gegaan. Wanneer het probleem niet te verhelpen is, of als daarin niet is voorzien, of indien de aanroeper ook de op de exception moet ontvangen, kan de exception alsnog worden gepropageerd. Dit wordt een rethrow genoemd. De afgevangen exception wordt dan bewust nogmaals gegooid vanuit het catch-block dat de exception zojuist afving; zie ook afbeelding 3. Voorafgaand aan de hervatte propagatie kan de op dit punt vergaarde informatie worden gelogd of verstuurd. Het is belangrijk om op de juiste manier exceptions te propageren nadat ze afgevangen zijn. Listing 3 toont een tweetal manieren om een afgevangen exception opnieuw te gooien. De CLR zal bij de eerste manier de oorsprong van de exception niet aanpassen, terwijl deze bij de tweede wordt verlegd naar de plek waar de throw wordt gedaan, alsof deze daar juist opgetreden is. De oorspronkelijke context gaat daarmee verloren.

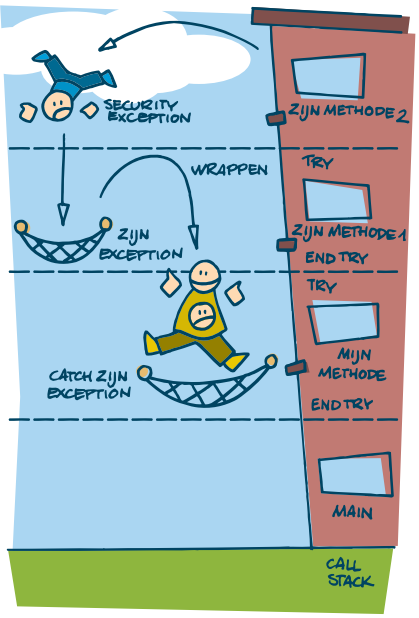
```
Try
  'Code die exception kan veroorzaken
Catch ex As Exception
  'Manier 1: Rethrow: oorspronkelijke context blijft behouden
  Throw
  'Manier 2: Oorsprong van exception wordt deze methode
  Throw ex
End Try
```

Listing 3. Twee methoden om een rethrow uit te voeren

> Het is verder gebruikelijk om de oorspronkelijke exception te verpakken in de nieuwe exception.

Wrappen van exceptions

De laatste mogelijkheid voor propagatie wordt gebruikt in situaties waar het nuttig is de exception af te vangen, maar een ander type exception daarvoor in de plaats te gooien. Dit is met name zinvol als het voor de aanroeper van uw code niet relevant of onduidelijk zou zijn om de echte exception te zien. Bijvoorbeeld, uw datalaag bevat een klasse met een GetCustomerByID-methode. De encrypted connection string wordt hierin uit de registry gelezen, maar de betreffende registrykey is niet aanwezig. De aanroeper van GetCustomerByID zou een ArgumentException zien, ondanks dat diens argumenten correct waren. Het is daarom zinvoller om in uw code de ArgumentException op te vangen en een andere exception (bijvoorbeeld DataAccessException) te gooien. De oorspronkelijke exception-informatie mag echter niet verloren gaan, omdat deze u kan helpen bij het oplossen van de problemen in de configuratie of data access-code. De eerste actie kan bestaan uit het loggen van de exception-informatie. Het is verder gebruikelijk om de oorspronkelijke exception te verpakken in de nieuwe exception. Iedere exception-klasse heeft daarvoor een readonly InnerException-property die refereert naar de gewrapte exception. Als u een exception vangt, kunt u altijd de InnerException-property toetsen. Is deze niet gelijk aan Nothing dan is er een gewrapte exception. De InnerException is alleen toe te wijzen in een speciale overloaded constructor die deze exception als laatste argument accepteert. Een gewrapte exception wordt als volgt gegooid: Throw New DataAccessException("Foutmelding", ex)



Afbeelding 4. Wrappen van exceptions

Omdat het wrappen meer keren kan worden uitgevoerd, kan er een diepe nesting van inner exceptions ontstaan. Om te weten te komen welke exception als eerste gewrapt werd, kan de methode Exception.GetBaseException worden gebruikt. Deze geeft de diepst geneste exception terug. De CLR past de strategie van het wrappen geregeld toe, zoals bij de eerder besproken unhandled exceptions in ASP.NET. Daar wordt de unhandled exception al door ASP.NET afgevangen, voordat het HttpApplication.Error-event plaatsvindt. In plaats daarvan wordt een HttpUnhandledException gegooid met als InnerException de oorspronkelijke exception. Hetzelfde wordt toegepast bij .NET Remoting, waar de nieuwe exception van het type RemotingException is.

Afhandelen of niet

Het is niet eenvoudig om te beslissen of een exception moet worden afgehandeld, of deze juist moet propageren, of dat een combinatie van beide nodig is. Wellicht dat een aantal vuistregels hierbij kan helpen.

Handel exceptions af wanneer u:

- relevante informatie toe kunt voegen aan de exception. Dit wordt met gedaan door een ander type exception te gooien vanuit het catch-block en de oorspronkelijke exception wordt gewrapt.
- resources wilt opruimen na het optreden van de exception. Deze cleanup wordt uitgevoerd in het catch-block.
- informatie over de exception wilt loggen.
- de oorzaak van de exception kunt herstellen en/of daarna kunt vervolgen.
- een niet-kritische exception verwachtte, maar niet kon voorkomen. Voorkomen is nog steeds beter dan genezen.

Verder is het van belang de rol van de code te betrekken in deze beslissing. Met name voor de ontwikkelaars van herbruikbare code, zoals class-library's, (web) control-library's en webservices is het belangrijk om niet altijd alle exceptions af te handelen, maar de consumer/client daarvan op de hoogte te stellen. De herbruikbare code kan niet altijd aannames maken over de manier waarop bij het optreden van een exception moet worden gehandeld. Dit kan de client, die bedacht dient te zijn op exceptions, vaak beter inschatten. In clientcode daarentegen, die gebruik maakt van onder andere library-code is het eigenlijk altijd zaak de exceptions af te vangen. De last-chance exception handlers kunnen een vangnetconstructie geven voor de exceptions die toch niet afgehandeld werden.

Maak er een gewoonte van om bij catch-blokken zo specifiek mogelijke exception-typen te gebruiken. Concretere typen exceptions geven specifiekere informatie. Door het afvangen van een FileNotFoundException kan de FileName-property worden geëvalueerd, die aan-

```
Imports System
Imports System.Runtime.Serialization
Imports System.Diagnostics
 <Serializable()> _
Public Class YourFrameworkBaseException
  Inherits ApplicationException
  Public Sub New()
    MyBase.New()
  End Sub
  Public Sub New(ByVal message As String)
    MyBase.New(message)
  End Sub
  Public Sub New(ByVal message As String, _
  ByVal innerException As Exception)
    MyBase.New(message, innerException)
  End Sub
  Public Sub New(ByVal info As SerializationInfo, _
  ByVal context As StreamingContext)
    'Voer hier custom deserialisatie uit
    MyBase.New(info, context)
  End Sub
  Protected ReadOnly ExceptionID As Guid = Guid.NewGuid
  Protected RealStackTrace As StackTrace = New StackTrace(True)
  Protected ResourceID As String ' Voor localisatie van foutmelding
  Public Overrides Function ToString() As String
    Return MyBase.ToString() & Environment.NewLine & _
    "ExceptionID: " & Me.ExceptionID.ToString("P") 'Enzovoorts
  End Function
  Public Overrides Sub GetObjectData(ByVal info As SerializationInfo, _
    ByVal context As StreamingContext)
    'Voer hier custom serialisatie uit
  End Sub
End Class
```

Listing 4. Custom exception-klasse

geeft welk bestand niet werd gevonden. Een catch-block met een IOException is niet in staat deze informatie te geven. Als u een eenvoudige manier zoekt om alle exceptions op te zoeken, gebruik dan de Class Viewer uit de Framework SDK (wincv.exe). Als zoekterm geeft u dan exception op. Dit zal alle exception-klassen in de Framework Class Library (FCL) tonen.

Eigen exception-klassen maken

In uw code zult u zelden voldoende hebben aan de bestaande exceptions uit de FCL. In uw applicatie en diens specifieke situaties heeft u behoefte aan eigen exception-klassen, omdat de bestaande exception-klassen niet of minder geschikt zijn. Uiteraard kunt u exceptions van het type Exception gooien, maar dit is af te raden. Beter is het om eigen exception-klassen te maken. De FCL heeft een tweetal basisklassen waarvan u uw custom exception-klassen kunt afleiden: System.SystemException en System.ApplicationException (die beide zijn afgeleid van System.Exception). SystemException is de basisklasse voor alle FCL-exceptions. De ApplicationException is de meest aangewezen basisklasse waarvan u uw applicatiespecifieke exceptions zult afleiden. Het is echter ook denkbaar dat u bij het ontwikkelen van een eigen framework SystemException als basisklasse zult nemen.

Let er op dat u bij het afleiden van de gekozen basisklasse alle aanbevolen constructors aanbrengt. Dit zijn er standaard drie, met een optionele vierde, indien de exception custom-serialisatie nodig heeft. Listing 4 noemt alle vier constructors.

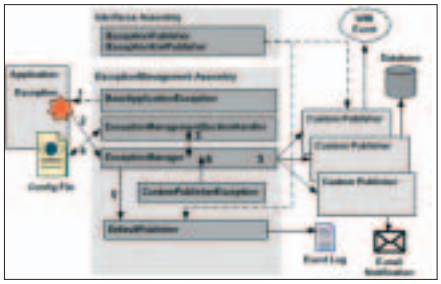
Een custom exception-klasse is extra waardevol als deze, behalve door zijn type, ook door properties extra informatie geeft over de opgetreden exception. U geeft de nieuwe exception-klasse daartoe public of protected properties. Bepaalde informatie en functionaliteit wilt u waarschijnlijk in iedere exception terug laten komen. Daarvoor maakt u een nieuwe basisklasse aan waarvan al uw custom exception-klassen weer afgeleid zullen worden. Door de overerving beschikken dan alle custom exception-klassen over de extras. Overigens, de CLR geeft al meer informatie dan u wellicht dacht. Een aantal properties van de basisklasse Exception wordt door de CLR gezet met gegevens over de exception. De readonly TargetSite-property bijvoorbeeld geeft een MethodBase-object terug dat de methode aangeeft waar de exception optrad. Via reflection kunt u dan meer over de methode, het bevattende type en dergelijke te weten komen. De Source-property bevat de naam van de applicatie waarin de exception ontstond. Deze moet daarvoor door uw of andermans code gezet worden alvorens de exception te gooien. Is dit niet gebeurt, dan zal de CLR Source gelijk stellen aan de naam van de veroorzakende assembly.

> Het EMAB biedt een infrastructuur om op een consistente wijze om te gaan met het managen en loggen van exceptions

Listing 4 bevat een aantal suggesties voor extra informatie die u op zou kunnen nemen in een eigen basisklasse-exception. Hierin zijn alleen de velden opgenomen en niet de properties. Het overriden van de ToString-implementatie is ook aan te raden bij het toevoegen van extra properties aan een exception-klasse.

Exception Management Application Block

Wanneer u consequent met exception management aan de gang gaat, zult u een strategie hebben ontwikkeld waarmee u de exceptions te lijf gaat. Daarbij zult u uw eigen exception-classes inzetten en library's gaan ontwikkelen met herbruikbare code, zoals voor het loggen van exception-informatie. U bent dan



Afbeelding 5. Infrastructuur van Exception Management Application Block

een eigen Enterprise Instrumentation Framework (EIF) aan het ontwikkelen, en dat is een goede zaak. Alvorens u alles zelf gaat ontwikkelen is het zeer nuttig om het Exception Management Application Block (EMAB) van Microsoft eens te bekijken. Het EMAB is één van acht application-blocks die Microsoft tot dusver heeft uitgebracht. Application-blocks zijn door Microsoft en de GotDotNet-community ontwikkelde en geteste mini-frameworks, bestaande uit broncode en documentatie. Het EMAB biedt een infrastructuur om op een consistente wijze om te gaan met het managen en loggen van exceptions. Het zorgt er voor dat de tijd die u spendeert om uw exception managementcode te ontwikkelen en te testen aanzienlijk wordt beperkt. De aangeboden infrastructuur, die in afbeelding 5 wordt getoond, is bovendien open en uitbreidbaar.

Het EMAB bevat een tweetal belangrijke klassen: de ExceptionManager en de BaseApplicationException. De ExceptionManager is in staat om exception-informatie te publiceren naar een aantal publishers. De publishers zijn klassen die de informatie wegschrijven naar bijvoorbeeld het event-log van het besturingsysteem. Eigen publishers zijn eenvoudig te ontwikkelen via implementatie van de IexceptionPublisher-interface. Nieuwe publishers zouden naar een database, WMI-event, e-mail of een Message Queue kunnen loggen. De BaseApplicationException wordt de nieuwe basisklasse voor alle custom exception-klassen die door u worden ontwikkeld. Deze basisklasse heeft een aantal extra properties dat veel meer informatie over de exception aanreikt, zoals de identiteit

van de gebruiker en de veroorzakende thread en AppDomain, het tijdstip van optreden en de machinenaam. Wanneer u behoefte heeft aan een eigen, betere en uitgebreidere basisklasse, dan leidt u deze van BaseApplicationException af in plaats van ApplicationException. In een aparte sectie in het configuratiebestand (*.config) van uw applicatie wordt geconfigureerd welke publishers er zijn en of deze aanstaan. Bovendien kan per publisher worden aangegeven van welke exception-typen er informatie wordt gepubliceerd.

Betere en robuustere applicaties

Exceptions zullen optreden en goed exception management in uw code getuigt van planning voor deze onvoorziene omstandigheden. Structured Exception Handling is het juist middel om met exceptions om te gaan. Hoe de afhandeling geschiedt verschilt per situatie en is afhankelijk van de intentie en de rol van de code. Verder is het goed om last-chance exception handling toe te passen voor gevallen waar een exception toch niet wordt afgehandeld. Wie een eigen exception management op wil nemen in een eigen Enterprise Instrumentation Framework, kan eigen exception-klassen schrijven en een infrastructuur voor het loggen en afhandelen. Het Exception Management Application Block van Microsoft Patterns and Practices biedt een dergelijke infrastructuur en broncode, en kan daarmee als startpunt dienen. Met de juiste toepassing van exception management ontwikkelt u betere en robuustere applicaties met het .NET Framework.

Interessante links en boeken

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp
http://www.gotdotnet.com/Community/Workspaces/Workspace.aspx?id=c1146b3a-3f9b-47b8-899e-f42e667cdccf
Applied .NET Framework, Jeffrey Richter, Microsoft Press, ISBN: 0-7356-1422-9
[AT1]Mag ik refereren naar eerdere artikelen uit het .NET magazine? Jazeker/RM.
[RM2]Is 'propageren' voor iedere developer hetzelfde als 'propagate'?

Microsoft Press

Titel: Programming with Managed Extensions for Microsoft® Visual C++® .NET--Version 2003
ISBN: 0-7356-1782-1
Author: Richard Grimes