



**SRW2000**

**Standaard Raamwerk Water - versie 1.0**

**Richtlijnen voor SR-componenten**

**Versie 1.1**

Blanco pagina.



**SRW2000**

**Standaard Raamwerk Water - versie 1.0**

**Richtlijnen voor SR-componenten  
Versie 1.1**

Kernteam SRW 2000

In opdracht van STOWA, ALTEERRA, RIVM, RIZA

September 2000

**Bibliografie:**

Tacke, J.H.P.M. (MX.Systems BV), Brinkman, R. (WL | Delft Hydraulics), Frieswijk, E. (EDS International BV), Levelt, D. (WL | Delft Hydraulics), Otjens, A.J. (WISL-Alterra), 2001, Standaard Raamwerk – versie 1.0, Richtlijnen voor SR-componenten, ISBN 90-5773-144-4, Stichting Toegepast Onderzoek Waterbeheer (STOWA), Utrecht, werkdocument 2001-W-08, Rijkswaterstaat, Rijksinstituut voor Integraal Zoetwaterbeheer en Afvalwaterbehandeling (RIZA), Lelystad, werkdocument 2001.127X, MX.Systems BV document P4118-R-34 58 p.

Blanco pagina.

## Voorwoord

Het project 'Standaard Raamwerk – Realisatie versie 1.0' wordt uitgevoerd in opdracht van STOWA (Stichting Toegepast Onderzoek Waterbeheer) en medegefinancierd door RWS – RIZA, RIVM, NITG TNO en Alterra. Het consortium van opdrachtnemers bestaat uit IT-organisaties die actief applicaties ontwikkelen voor het integraal waterbeheer in Nederland, te weten W!SL/Alterra, MX.Systems, Geodan IT, NITG TNO en WL | Delft Hydraulics.

De volgende personen zijn bij de uitvoering van het project 'Standaard Raamwerk – Realisatie versie 1.0' betrokken geweest.

Stuurgroep:	Jacques Leenen (vz)	STOWA
	Jan Anne Boswinkel	NITG TNO
	Anton van der Giessen	RIVM
	Miep van Gijsen	Alterra
	Gaele Rodenhuis	WL   Delft Hydraulics
	Theo van Stijn	RIKZ
	Frans van de Ven	RIZA
Begeleidingscommissie:	Jandirk Bulens (vz)	Alterra
	Henk Alkemade	RIZA
	Aldrik Bakema	RIVM
	Michiel Blind	RIZA
	Piet Groenendijk	Alterra
	Jan Noort	Sepra (voor STOWA)
	Ludolph Wentholt	STOWA
	Rick Wortelboer	RIVM
Concept Control Team:	Bas van Adrichem	MX.Systems
	Jan Jellema	NITG TNO
	Joost Maus	Geodan IT
	Jaco Stout	WL   Delft Hydraulics
	Tamme van der Wal	W!SL/Alterra
Projectteam:	Johan Tacke (pl)	MX.Systems
	Rob Brinkman	WL   Delft Hydraulics
	Michiel Dijkman	Geodan IT
	Edwin Frieswijk	EDS International BV
	Joris Sierman	Geodan IT
	Tonny Otjens	W!SL/Alterra

De begeleidingscommissie heeft namens de stuurgroep Realisatie Standaard Raamwerk de voortgang van het project bewaakt en (tussen)producten inhoudelijk beoordeeld.

Het Concept Control Team heeft namens het consortium van opdrachtgevers zorggedragen voor inhoudelijke aansturing van het projectteam en voor interne reviews van de (tussen)producten.

**Rapport**

**Project**

SRW2000

**Versie** 1.1

**Registratienummer**

P4118-R-3



Blanco pagina.



# INHOUD

Voorwoord .....	I
1 Inleiding .....	1
1.1 Achtergrond .....	1
1.2 Doelgroep .....	1
1.3 Leeswijzer .....	1
2 SR architectuur .....	3
2.1 Inleiding .....	3
2.2 Raamwerk en raamwerkcomponenten .....	3
2.3 Implementatie .....	5
3 SRW library .....	7
3.1 Inleiding .....	7
3.2 Opbouw van de bibliotheek .....	7
3.3 Collecties en Iteratoren .....	8
3.4 eXtensible Markup Language .....	11
3.5 Events .....	13
4 Ontwikkeling van raamwerkcomponenten .....	16
4.1 Inleiding .....	16
4.2 Waar moet elk raamwerkcomponent aan voldoen? .....	16
4.3 Tips voor ontwikkeling van raamwerkcomponenten .....	18
5 Ontwikkeling van modelapplicaties .....	20
5.1 Inleiding .....	20
5.2 Stappenplan .....	20
5.3 Realisatie van een modelapplicatie .....	21
5.3.1 Stap 1: Definitie project source .....	21
5.3.2 Stap 2: Implementatie project source .....	21
5.3.3 Stap 3: Implementatie modelementen en connectoren .....	26
5.3.4 Stap 4: Implementatie methodes ModelApplication .....	29
5.3.5 Stap 5: Koppeling met rekenkern .....	31
5.3.6 Stap 6: Testen .....	32
6 Ontwikkeling van generieke tools .....	33
6.1 Inleiding .....	33
6.2 Stappenplan .....	33
6.3 Realisatie van een presentatiecomponent .....	33
6.3.1 Stap 1: Definitie project source .....	34
6.3.2 Stap 2: Implementatie project source .....	34
6.3.3 Stap 3: Implementatie BuildingFC methodes .....	35



6.3.4	Stap 4: Implementatie GenericTool methodes.....	36
6.3.5	Stap 5: Implementatie PresentationComponent methodes.....	37
6.3.6	Stap 6: Testen.....	38
7	Gebruik van data definities.....	39
7.1	Inleiding.....	39
7.2	Lezen en schrijven van data.....	39
7.2.1	DataDefinition.....	40
7.2.2	SRWDataObject.....	41
7.2.3	Scope van een DataDefinition.....	43
7.3	Data-definities voor modelapplicaties.....	43
7.4	Data-definities voor generieke tools.....	47
8	Standaarden en naamconventies.....	48
8.1	Inleiding.....	48
8.2	Gebruik van XML.....	48
8.3	Naamconventies.....	48
9	Literatuur.....	50
	Bijlage A: Inhoud SRWLib.....	51
	Bijlage B: Delphi instellingen.....	53
	Bijlage C: Veelgestelde vragen.....	57





# 1 Inleiding

## 1.1 Achtergrond

Dit document maakt onderdeel uit van de rapportage behorend bij de realisatiefase van het project 'Standaard Raamwerk – Specificatie en Realisatie versie 1.0'. Globaal bestaat deze fase uit de onderstaande onderdelen:

- Bouw raamwerk en raamwerkcomponenten;
- Implementatie/testen landelijke en regionale case;
- Aanpassing Technisch Ontwerp;
- Aanvulling Richtlijnen voor componentenbouwers.

Dit document is de aangepaste versie van de Richtlijnen voor componenten-bouwers. Deze aanpassingen en aanvullingen zijn op basis van de ervaringen tijdens de bouw van het raamwerk en de raamwerkcomponenten verzameld.

## 1.2 Doelgroep

In dit document worden stappenplannen gegeven die als leidraad gebruikt kunnen worden bij de ontwikkeling van modelapplicaties en generieke tools. Dit document is dus in eerste instantie gericht op systeemontwikkelaars.

De implementatiekeuzes die tijdens de realisatie van het Standaard Raamwerk (SR<sup>1</sup>) gemaakt zijn hebben invloed op deze richtlijnen en daardoor op de vereiste voorkennis van systeemontwikkelaars die SR-componenten gaan bouwen. Er wordt vanuit gegaan dat de lezer kennis heeft van object oriëntatie, de Unified Modeling Language (UML), de Delphi syntax (Object Pascal) en de eXtensible Markup Language (XML).

## 1.3 Leeswijzer

In dit document is niet alle informatie opgenomen om SR-componenten te kunnen bouwen. Het SR-Technisch Ontwerp en in mindere mate het SR-Functioneel Ontwerp zijn hiernaast vereiste naslagwerken.

Ter introductie wordt in hoofdstuk 2 een overzicht gegeven van de belangrijkste SR architectuur-concepten. Tijdens de realisatie van het raamwerk en de raamwerkcomponenten is een bibliotheek ontwikkeld met hierin functionaliteit die het raamwerk en elk raamwerkcomponent gebruikt kan worden. Deze bibliotheek – SRWLib – is beschikbaar als een set van Delphi units. In hoofdstuk 3 worden de onderdelen van SRWLib beschreven.

---

<sup>1</sup> De naam Standaard Raamwerk Water (SRW) is in de loop der tijd gewijzigd in Standaard Raamwerk (SR). In dit document wordt in het algemeen de term SR gebezigd, tenzij het om "eigennamen" gaat zoals "Kernteam SRW", "Projectteam SRW", "SRW2000" en "SRW-Editor".



Elk raamwerkcomponent moet aan een aantal algemene specificaties voldoen. Door deze specificaties te volgen bij ontwikkeling van een component wordt de toepasbaarheid van deze component in het raamwerk gegarandeerd. De specificaties met als aanvulling een aantal tips zijn in hoofdstuk 4 uitgewerkt.

In hoofdstuk 5 worden de raamwerkcomponent-specificaties verder uitgebreid ten behoeve van de ontwikkeling van modelapplicaties. In de vorm van een stappenplan wordt de ontwikkeling van een modelapplicatie uitgewerkt. Analooq hieraan wordt voor de ontwikkeling van generieke tools een stappenplan beschreven in hoofdstuk 6.

Het raamwerk kan gebruik makend van de DataEngine flexibel omgaan met verschillende data-structuren en -formaten. De werking hiervan is in hoofdstuk 7 uitgewerkt.

Naast alle specificaties om een correcte werking van het raamwerk met componenten te kunnen garanderen kan het ontwikkelen van raamwerkcomponenten vereenvoudigd worden door een aantal standaarden en conventies te volgen. De toegepaste standaarden en conventies worden in hoofdstuk 8 beschreven.



## 2 SR architectuur

### 2.1 Inleiding

Op basis van de Architectuur Standaard Raamwerk [SR-A] en de functionele analyse SR [SR-FO] is een ontwerp gemaakt voor het Standaard Raamwerk - versie 1. Dit ontwerp beschrijft de diensten die het raamwerk en de verschillende soorten raamwerkcomponenten moeten kunnen leveren en de manier waarop deze diensten aangeboden worden.

### 2.2 Raamwerk en raamwerkcomponenten

Het raamwerk fungeert als basis voor een SR applicatie. Hiervoor bezit het raamwerk algemene functionaliteit:

- Beheer van geregistreeerde raamwerkcomponenten;
- Starten van raamwerkcomponenten;
- Opvangen en versturen van zogenaamde 'events' van raamwerkcomponenten.

Applicaties op basis van dit raamwerk bezitten een aantal raamwerkcomponenten. Een aantal componenten moet beschikbaar zijn om cases samen te kunnen stellen en uit te kunnen voeren. Deze componenten zijn:

- ProcessManager

Deze component is verantwoordelijk voor het beschikbaar stellen van een 'Composer' waarmee een case samengesteld kan worden en voor het coördineren (starten, pauzeren, stoppen, rekenvolgorde bepalen) van een simulatie.

- DataEngine

Deze component is verantwoordelijk voor het uitwisselen van gegevens tussen modelapplicaties (onderling) en generieke tools en voor het beheer van data van deze modelapplicaties en generieke tools. De DataEngine kan de in- en uitvoer van modelapplicaties geheel beheren of alleen verwijzingen naar data-locaties beheren. De DataEngine is dus geen centrale database voor alle modelapplicaties.

- SRW-Editor

De SRW-Editor maakt verbindingen tussen de aansluitpunten (schematisaties) van modelapplicaties op basis van selecties van de gebruiker. De SRW-Editor legt dus vast welke gegevens van een modelapplicatie naar een andere modelapplicatie of generieke tool doorgeven moeten worden.

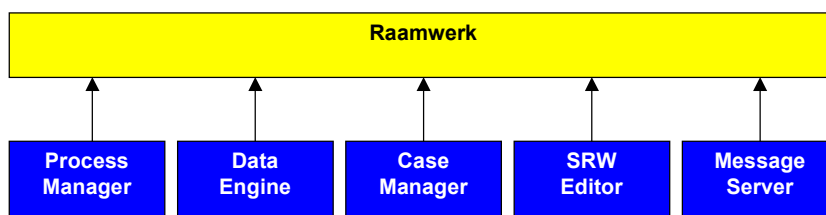
- CaseManager

Met behulp van deze component kan een lijst van eerder gemaakte cases beschikbaar gesteld worden. Hierdoor kan bij opstarten van een SR-applicatie de gebruiker de keuze maken uit een bestaande cases of besluiten een nieuwe case samen te stellen.

- (Eventueel) MessageServer

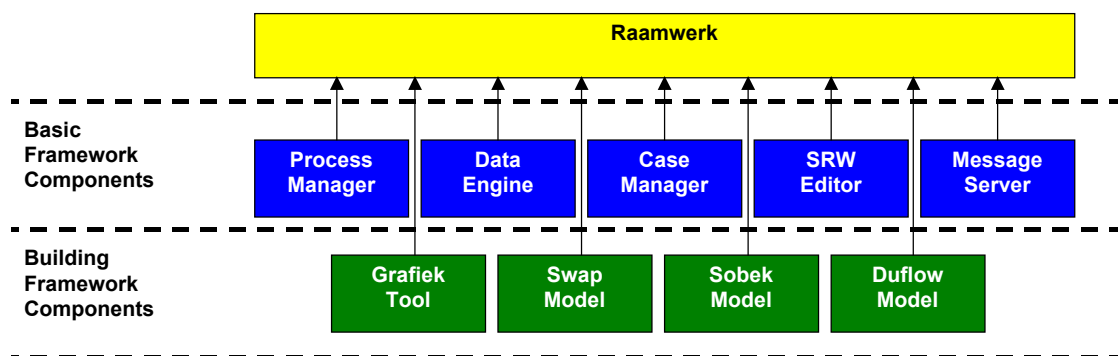
Voor het sturen van berichten naar de gebruiker in de vorm van een dialoogschermbaan kan gebruik gemaakt worden van een MessageServer. Deze server vangt automatisch alle berichten op die door raamwerkcomponenten verstuurd worden.

Van elk van de bovenstaande componenten kan slechts één instantie gebruikt worden (indien bijvoorbeeld een tweede ProcessManager geregistreerd wordt, negeert het raamwerk deze tweede versie). Deze componenten worden binnen de SR architectuur aangeduid als BasicFrameworkComponents (kortweg BasicFC's).



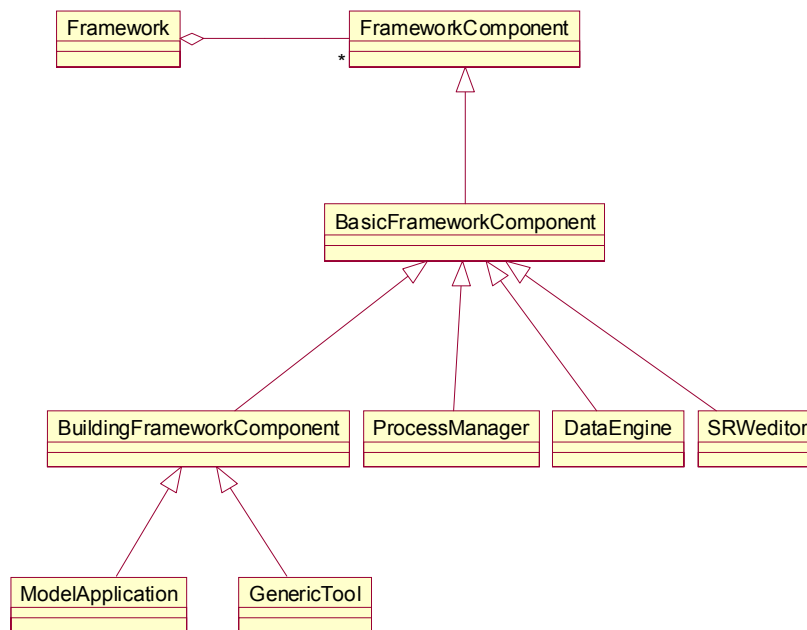
**Figuur 2.1 – Basis configuratie voor een SR applicatie**

Van de flexibiliteit die het SR biedt met betrekking tot het toevoegen en verwijderen van raamwerkcomponenten zal vooral gebruik gemaakt worden doordat nieuwe modelapplicaties ontwikkeld worden of bijvoorbeeld nieuwe presentatiecomponenten. Dit zijn ook de raamwerkcomponenten waarvan meerdere 'soorten' naast elkaar gebruikt kunnen worden. De componenten worden binnen de SR architectuur aangeduid als BuildingFrameworkComponents (kortweg BuildingFC's). De BuildingFC's zijn de componenten die gebruikt worden als onderdeel van een case.



**Figuur 2.2 – Basis configuratie voor een SR applicatie, aangevuld met Building FrameworkComponents**

De bovenstaande structuur is afgeleid van het onderstaande klassediagram, waarin de relatie tussen het raamwerk, BasicFC's en BuildingFC's is weergegeven.



**Figuur 2.3 – Klassediagram raamwerk en raamwerkcomponenten**

## 2.3 Implementatie

Voor de ontwikkeling van een SR component is het volgende vereist:

- Ontwikkelomgeving Borland Delphi 5, inclusief ServicePack 1.

Borland levert drie versies van Delphi 5: Standard, Professional en Enterprise. Omdat binnen het raamwerk of de BasicFC's geen gebruik gemaakt wordt van Internet-, Database- en COM-componenten zijn alle versies toepasbaar.

De vereiste ServicePack is te downloaden bij Borland:

<http://www.borland.com/devsupport/delphi/downloads/index.html>

- XML parser

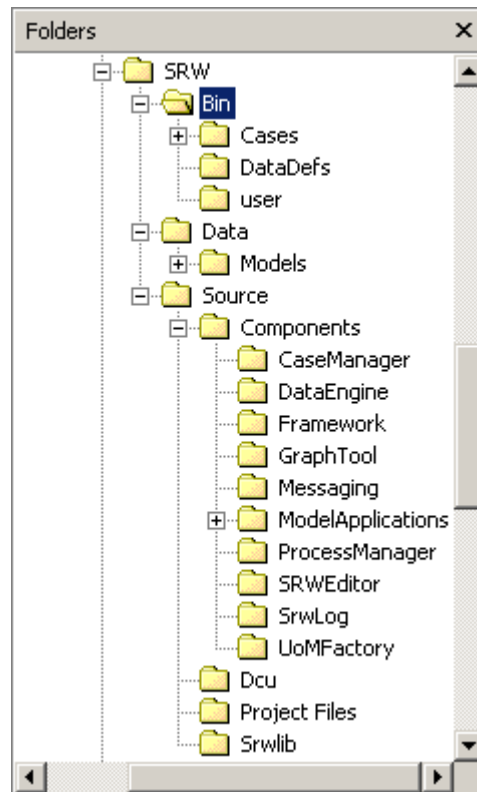
Voor het lezen en schrijven van XML (eXtensible Markup Language) is een parser vereist. Microsoft® levert dergelijke parsers in de vorm van een COM DLL onder de naam MSXML. Bij ontwikkeling van SR componenten dient versie 2.5 van deze parser aanwezig te zijn. Indien een PC over MS Internet Explorer 5.0 beschikt is deze parser automatisch beschikbaar. Installatie van de parser afzonderlijk is ook mogelijk, zie: <http://msdn.microsoft.com/xml/default.asp>

- Delphi sourcecode of gecompileerde units van raamwerk en BasicFC's

Om een nieuw raamwerkcomponent te kunnen ontwikkelen is de basis-sourcecode vereist. Met behulp van deze basiscode kan de eigen code gecompileerd en gedebugged worden.

In de onderstaande afbeelding is de directory-structuur weergegeven op basis waarvan de ontwikkeling van het raamwerk en de raamwerkcomponenten voor versie 1 heeft plaatsgevonden. Voor het ontwikkelen (compileren) van eigen raamwerkcomponenten is het van

belang dat deze structuur aangehouden wordt zodat (relatieve) zoekpaden naar deze code correct zijn.



**Figuur 2.4 – Directory-structuur SR sourcecode**

Naast de beschreven tools en sourcecode is kennis van een aantal technologieën vereist om SR componenten te kunnen ontwikkelen:

- Delphi ontwikkelomgeving en Object Pascal syntax  
Naast kennis van de syntax is ervaring binnen de Delphi omgeving (compiler, debugger, etc) vereist.
- XML  
Het XML Document Object Model waarin COM interfaces als DOMDocument, XMLDOMNode, etc worden toegepast worden als bekend verondersteld. Ook hiervoor is informatie te verkrijgen op: <http://www.msdn.microsoft.com/xml/default.asp>
- Ontwikkeling van Delphi DLL's  
Een SR component wordt ontwikkeld als een Dynamic Link Library (DLL). Kennis over de ontwikkeling van DLL's en zogenaamde class-DLL's (DLL's die in plaats van een set functies een klasse ter beschikking stellen) is vereist.



## 3 SRW library

### 3.1 Inleiding

Bij de ontwikkeling van het raamwerk en de raamwerkcomponenten zijn een aantal afspraken gemaakt over hoe veelgebruikte programma-onderdelen geïmplementeerd worden. Voor deze programma-onderdelen zijn daarom een aantal standaardoplossingen gemaakt, die binnen elk raamwerkcomponent toegepast kunnen worden. Deze standaardoplossingen zijn in een bibliotheek opgenomen, de SRW Library (SRWLib).

Een aantal klassen die onderdeel uitmaken van SRWLib zijn SR interfaces die in het technisch ontwerp beschreven zijn. Deze interfaces zijn in SRWLib opgenomen omdat deze binnen verschillende raamwerkcomponenten worden toegepast. Gebruik van deze interfaces is dus belangrijk voor een correcte werking van de SR applicatie. Naast deze 'verplichte' onderdelen zijn een aantal standaardoplossingen opgenomen die geen onderdeel uitmaken van de SR specificaties, maar slechts als hulpmiddel worden geboden. Er kan dus wel afgeweken worden van deze standaardoplossingen, hoewel het uiteraard niet logisch is een eigen XML parser te gebruiken indien SRWLib over een uitvoerig geteste versie beschikt.

### 3.2 Opbouw van de bibliotheek

SRWLib is in de vorm van een Delphi project beschikbaar. Binnen dit project zijn een zestigtal sourcefiles (pascal files) opgenomen waarin de standaard oplossingen zijn geïmplementeerd. Hoewel het SRWLib project gecompileerd kan worden tot een DLL is dit niet de beoogde gebruikswijze. Hiervoor zou deze DLL een onoverzichtelijk en complex interface moeten bezitten. Gebruik van SRWLib is daarom alleen mogelijk door te verwijzen naar de betreffende sourcefiles binnen deze bibliotheek. Binnen Delphi betekent dit dat de zogenaamde 'uses clause' wordt uitgebreid met units uit SRWLib.

Globaal bestaat SRWLib uit de volgende onderdelen:

1. SR klassen

Alle interfaces die in het SR Technisch Ontwerp zijn beschreven en binnen meerdere raamwerkcomponenten toegepast (kunnen) worden zijn in SRWLib opgenomen.

2. Collection/Iterator design pattern

Voor het gebruik van verzamelingen van objecten en bewerken van deze verzamelingen is het Iterator Design Pattern [GAMMA, 1995] uitgewerkt.

3. Event-model

Voor het versturen van berichten tussen raamwerkcomponenten en het raamwerk (en andersom), zoals statusveranderingen of foutmeldingen worden zogenaamde events gebruikt. Events worden altijd naar het raamwerk (de Publisher) verstuurd. Het raamwerk is op de hoogte van



welke raamwerkcomponenten 'geïnteresseerd zijn' in deze events en stuurt deze events op basis van deze informatie door. Voor het gebruik van dit concept worden een aantal standaardklassen in SRWLib geboden.

#### 4. Basis constanten en types

Naast standaard klassen/interfaces zijn standaard types en constanten in SR opgenomen. Deze basis- en enumeratietypes zorgen voor een minimale afhankelijkheid van specifieke Delphi (Object Pascal) taalconcepten. Voorbeelden hiervan zijn eigen types voor strings en datum/tijd.

#### 5. XML en GML klassen

(Meta-)informatie van raamwerkcomponenten en cases wordt in XML formaat uitgewisseld. Modelschematisaties worden in een specifieke implementatie hiervan, GML, gegeven. Voor het gebruik van dit formaat zijn Delphi klassen ontwikkeld die rechtstreeks gebruik van de MSXML parser voorkomen.

In Bijlage A is een totaaloverzicht opgenomen van de units in SRWLib. Het zoeken van een unit in SRWLib wordt vereenvoudigd doordat de naam van een unit altijd gelijk is aan de klasse, voorafgegaan door de prefix 'u' (dus unit uPosition bevat de klasse Position).

### 3.3 Collecties en Iteratoren

Op diverse plaatsen in een component worden verzamelingen van objecten bijgehouden. Een verzameling objecten is zelf ook een object met operaties voor het manipuleren van deze verzameling. Hiervoor biedt SRWLib de interface Collection. Dit interface biedt de onderstaande operaties:

- `Add(aObject: TRootObject)`  
Voegt het object aObject toe aan de collectie.
- `Remove(aObject: TRootObject)`  
Verwijdert het object aObject uit de collectie.
- `Size()`  
Geeft de grootte van de lijst.
- `Includes(aObject: TRootObject)`  
Geeft als resultaat true als object aObject element van de collectie is, anders false.
- `CreateIterator()`  
Maakt een object waarmee de verzameling objecten één voor één afgelopen kan worden.
- `ValidateElement(aObject: TRootObject)`  
Controleert of object aObject aan de lijst toegevoegd mag worden.
- `SetElementConstraint(aPredicate: TPredicate)`  
Zet het predikaat aPredicate als randvoorwaarde voor objecten in de verzameling.

De klassen die gebruik maken van dit interface (dit interface implementeren) zijn:





- **IndexedCollection** – Deze klasse houdt de objecten gegarandeerd in een vaste, geïndexeerde volgorde. Daarvoor worden ook operaties als `getFirstIndex`, `getLastIndex` en `getItem` geboden. Deze klasse is abstract, er mogen dus geen instanties hiervan gemaakt worden.
- **AssociationList** – Deze subklasse van **IndexedCollection** kan een verzameling objecten bijhouden, waarbij de lijst geen ‘eigenaar’ is van de objecten in deze lijst. Verwijderen van de lijst betekent dat de objecten in deze lijst blijven bestaan.
- **FastAssociationList** – Deze klasse is identiek aan **AssociationList**, maar is speciaal voor erg grote verzamelingen bedoeld (meer dan enkele tientallen objecten).
- **TempAssociationList** – Deze specialisatie van **AssociationList** is identiek aan **AssociationList**, met het verschil dat de **Iterator** die hierbij opgevraagd kan worden ook zorgt voor verwijdering van deze lijst.
- **AggregationList** – Deze klasse is identiek aan **AssociationList**, maar nu is de lijst wel eigenaar van de objecten in de lijst. Verwijderen van de lijst betekent dat ook de objecten in de lijst verwijderd worden.
- **FastAggregationList** – Deze klasse is identiek aan **AggregationList**, maar is speciaal voor erg grote verzamelingen bedoeld (meer dan enkele tientallen objecten).
- **ManagedAggregationList** – Bij deze specialisatie van **AggregationList** kan een maximale lijstgrootte ingesteld worden. Zodra deze grootte overschreden wordt, worden de ‘oudste’ objecten verwijderd.

Bij gebruik van een collectie wordt of de verzameling bewerkt (objecten toevoegen/verwijderen) of de lijst wordt afgelopen (om bijvoorbeeld een object te zoeken). Voor het aflopen van objecten in een verzameling biedt een collectie een **Iterator**. Deze **Iterator** biedt operaties om de lijst te doorlopen. De fysieke structuur van de collectie wordt hierdoor verborgen. De interface **Iterator** biedt de onderstaande operaties:

- `NextElement()`  
Geeft het volgende element van de verzameling terug.
- `Done()`  
Geeft `true` als resultaat indien het einde van de lijst is bereikt. Deze methode moet altijd aangeroepen worden, voordat `NextElement()` wordt aangeroepen.
- `First()`  
Zet de **Iterator** terug aan het begin van de verzameling.
- `Last()`  
Zet de **Iterator** aan het eind van de verzameling.
- `Count()`  
Geeft het aantal elementen dat in de bijbehorende verzameling is opgenomen.

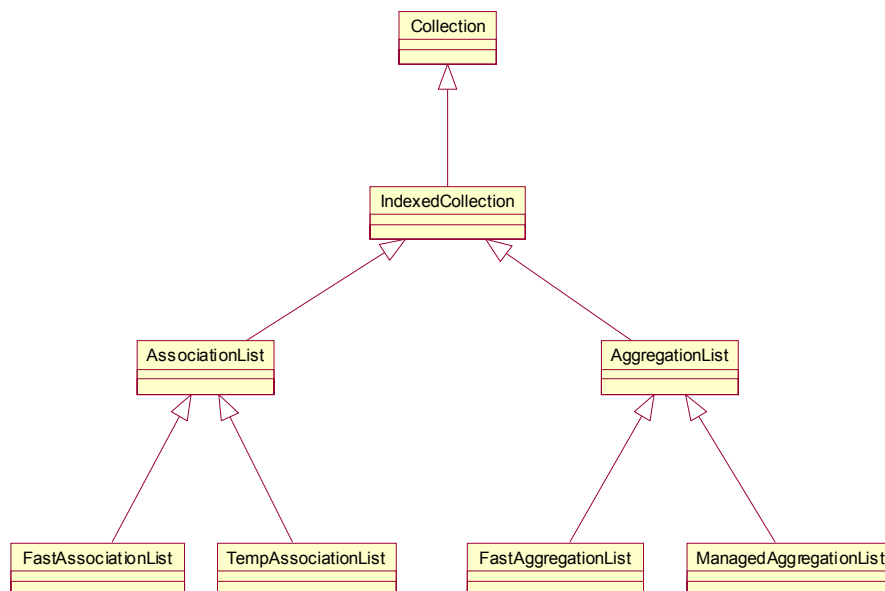
De onderstaande klassen implementeren de **Iterator** interface:

- **EmptyIterator** – Indien een iterator van een lege/niet bestaande lijst wordt gevraagd, kan een instantie van **EmptyIterator** gegeven worden.

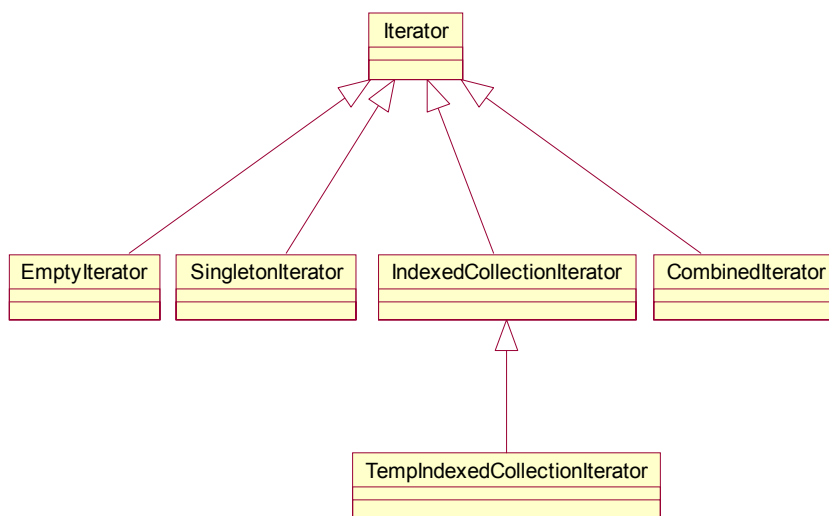


- SingletonIterator – Indien een verzameling uit één element bestaat, kan een instantie van SingletonIterator gegeven worden.
- CombinedIterator – Indien twee verzamelingen na elkaar afgelopen moeten worden, kan dit gecombineerd worden in een CombinedIterator.
- IndexedCollectionIterator – Deze iterator is bedoeld voor het aflopen van IndexedCollections.
- TempIndexedCollectionIterator – Deze iterator zorgt ervoor dat zodra de iterator wordt verwijderd, ook de bijbehorende verzameling wordt verwijderd.

Het ontwerp van de combinatie van collecties en iterators is weergegeven in de onderstaande klassediagrammen:



**Figuur 3.1 – Collections**



**Figuur 3.2 – Iterator design pattern**

### 3.4 eXtensible Markup Language

XML is een taal voor het beschrijven van data. Hierbij wordt gebruik gemaakt van zogenaamde XML Elements. Deze elementen bestaan uit data, voorafgegaan en afgesloten met een 'tag', vergelijkbaar met HTML. Een voorbeeld van een XML element is

```
<raamwerk>SRW</raamwerk>
```

Het bovenstaande element "Raamwerk" heeft als waarde "SRW". Het element Raamwerk beschrijft wat de waarde SRW betekent.

Voor verzamelingen van deze elementen worden zogenaamde XML Documents toegepast. Een XML Document is een XML Element dat sub-elementen kan bevatten. Een voorbeeld hiervan is:

```
<applicatie>
  <raamwerk>SRW</raamwerk>
  <component>ProcessManager</component>
</applicatie>
```

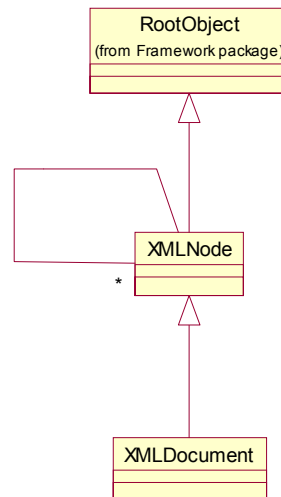
Het bovenstaande voorbeeld toont een XML Document "Applicatie" met hierin de elementen "raamwerk" en "component".

Voor het gebruik van XML Elements en Documents is een object model gemaakt. Dit object model is een verzameling objecten waarmee XML elements/documents gemanipuleerd kunnen worden. In dit object model worden de elementen in een XML Document beschouwd als een 'Node' en beschrijft wat de eigenschappen van deze Nodes zijn en welke bewerkingen op deze Nodes uitgevoerd kunnen worden.

Microsoft® stelt een parser beschikbaar waarmee XML kan worden gelezen/geschreven. Het XML object model (kortweg XMLDOM) is uitgewerkt in een aantal COM interfaces. SRWLib heeft de belangrijkste onderdelen 'ingepakt' (gewrapped) in een tweetal Delphi klassen. Hierdoor wordt het gebruik van COM-interfaces binnen Delphi voor ontwikkelaars van SR componenten verborgen. De Delphi klassen zijn:

- XMLDocument – de wrapper rond de COM interface IXMLDOMDocument;
- XMLNode – de wrapper rond de COM interface IXMLDOMNode.

De structuur is rechtstreeks afgeleid van het XML object model.



**Figuur 3.3 – Klassediagram XML wrappers voor Delphi**

De Delphi klassen bieden globaal de onderstaande mogelijkheden:

- Lezen van een XML bestand

Een bestand XML bestand kan geopend worden door:

```
XMLDocument.Load(aURL)
```

waarbij aURL de naam van het XML bestand is.

Het opslaan van (wijzigingen in) een XML bestand wordt uitgevoerd door:

```
XMLDocument.Save(aURL)
```

- Schrijven in een XML bestand

Bij het schrijven in een XML bestand kunnen nodes worden toegevoegd. Hierbij moet worden aangegeven wat de 'ParentNode' moet zijn, dus waar de nieuwe node onder toegevoegd moet worden. Een node toevoegen gaat als volgt:

```
XMLNode.CreateChild(aTag)
```

waarbij aTag de naam van de nieuwe node is.

Het schrijven van waarden in een node kan uitgevoerd worden door:

```
XMLNode.SetValue(aString)
```

waarbij aString de waarde representeert.

- Zoeken van nodes en bijbehorende waarden in een XML bestand

In een XML bestand zijn de nodes opgenomen in een boomstructuur. Aan een node kan dus een 'parent' gevraagd worden en om een verzameling van sub-nodes ('Childs').

```
XMLNode.GetParentNode
```

```
XMLNode.GetChildNodes
```

Het zoeken van nodes op naam kan uitgevoerd worden door:

```
XMLNode.SelectNode(aPattern)
```

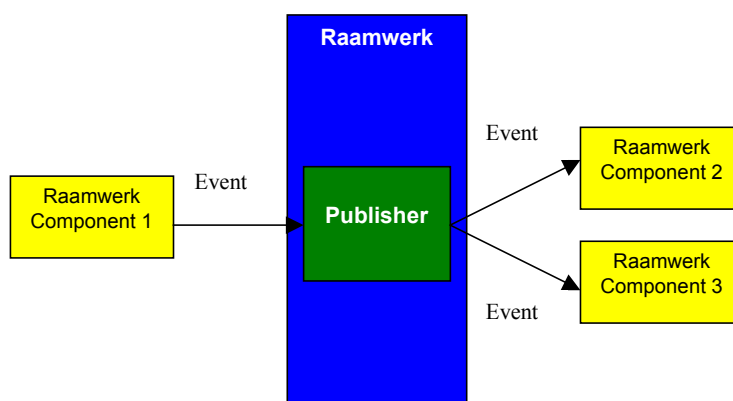
waarbij *aPattern* de node-naam (of namen) bevat waaronder gezocht moet worden.

Binnen SR wordt slechts een klein deel van de mogelijkheden van XML toegepast en hier beschreven. Het grote aantal aanvullende mogelijkheden zoals StyleSheets, DTD's, etc zijn voor versie 1 van het Standaard Raamwerk Water niet van toepassing. Er zijn veel help-bestanden beschikbaar voor het XML object model. Het helpprogramma XMLSDK25.CHM wat door Microsoft® ter beschikking wordt gesteld biedt een compleet overzicht van alle mogelijkheden.

### 3.5 Events

Tussen raamwerkcomponenten (onderling) en het raamwerk is uitwisseling van informatie vereist. Door informatie als statusovergangen, fouten, veranderingen in een case als zogenaamd 'event' te versturen in plaats van rechtstreekse communicatie tussen componenten worden afhankelijkheden tussen deze componenten sterk verminderd. Dit concept is afgeleid van het Observer Design Pattern, ook wel Publish-Subscribe mechanisme genoemd.

Raamwerkcomponenten kunnen zowel events versturen als ontvangen. Een event dat verstuurd wordt door een raamwerkcomponent wordt altijd door de Publisher in het raamwerk zelf opgevangen. Een raamwerkcomponent kan zich bij het raamwerk 'aanmelden' voor bepaalde event-typen. Een raamwerkcomponent kan zich bijvoorbeeld aanmelden voor events die een verandering in de case aangeven. De Publisher weet welke raamwerkcomponenten zich voor welke soorten events hebben aangemeld. De Publisher stuurt de ontvangen events dus door naar de abonnees.



**Figuur 3.4 – Publish – Subscribe mechanisme**

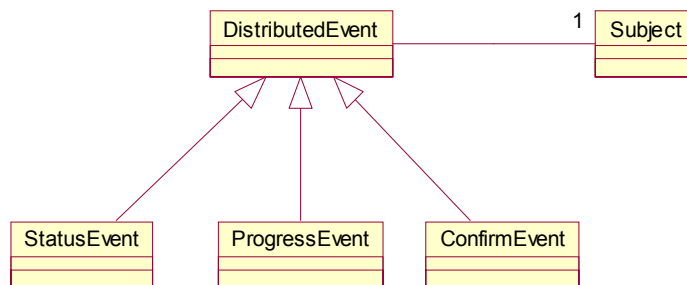
Elk raamwerkcomponent heeft altijd een verwijzing naar het raamwerk. Daardoor kan elk raamwerkcomponent events versturen naar de Publisher (de Publisher is onderdeel van het raamwerk).

Elk event-object in SR is afgeleid van de klasse DistributedEvent. Dit object heeft standaard informatie over de verzender van het event en een

beschrijving van het event. Deze beschrijving is opgenomen in een object van het type Subject. Afhankelijk van het soort event kan een subklasse gemaakt worden van DistributedEvent. In de eigen subklasse kan specifieke informatie of gedrag opgenomen worden. Standaard zijn in SRWLib de volgende eventtypes opgenomen:

- ConfirmEvent – Event dat kan worden gebruikt voor het bevestigen van een vraag.
- StatusEvent – Event dat kan worden gebruikt voor het doorgeven van statusveranderingen in een case.
- ProgressEvent – Event dat kan worden gebruikt voor het melden van de voortgang van een simulatie.

In het onderstaande klassediagram is de structuur van events en subjects weergegeven.



**Figuur 3.5 – Events in SR**

Voor het abonneren van raamwerkcomponenten voor een bepaald eventtype kan de onderstaande code toegepast worden.

```
FFramework.Subscribe(aSubscriber, aEventType, aNotifyProcedure)
```

met hierin:

- *aSubscriber*: een object van het type Subscriber dat het voor de abonnee mogelijke maakt om events daadwerkelijk te ontvangen.
- *aEventType*: het type event waarop het abonnement van toepassing is, in dit geval het CaseChanged eventtype.
- *aNotifyProcedure*: een verwijzing naar de procedure die aangeroepen moet worden door de publisher bij het versturen van het event. Deze procedure moet voldoen aan de onderstaande specificatie:

```
TDistributedEventProc = procedure(aDistributedEvent : TDistributedEvent)
of object
```

Deze aanmelding kan bijvoorbeeld uitgevoerd worden bij het instantiëren van een raamwerkcomponent. Het is belangrijk dat voordat een raamwerkcomponent weer verwijderd wordt, deze aanmelding weer ongedaan gemaakt wordt (anders stuurt de Publisher events naar niet bestaande componenten). Hiervoor biedt het raamwerk de operatie UnSubscribe.

Bij het versturen van events naar de Publisher kan de onderstaande code toegepast worden:



```
FFramework.Notify(aSubjectname, aSubjectValue)
```

met hierin:

- *aSubjectName*: de naam het subject dat verstuurd wordt.
- *aSubjectValue*: de informatie (bijvoorbeeld tekst of een waarde) die meegestuurd wordt met het event.

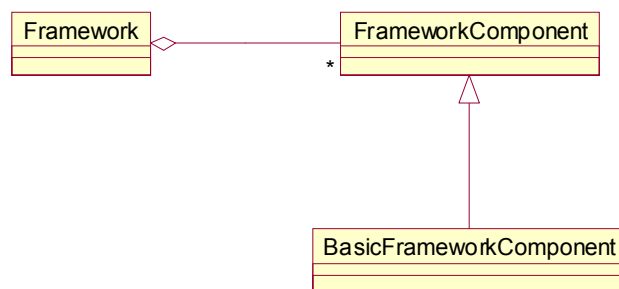
## 4 Ontwikkeling van raamwerkcomponenten

### 4.1 Inleiding

Bij de ontwikkeling van een raamwerkcomponent zal veelal sprake zijn van de ontwikkeling van een modelapplicatie of generieke tool, de zogenaamde BuildingFCs. Een ProcessManager of DataEngine zal veel minder frequent verwisseld worden dan dat een modelapplicatie wordt toegevoegd aan de raamwerkapplicatie.

### 4.2 Waar moet elk raamwerkcomponent aan voldoen?

De standaard functionaliteit die elke raamwerkcomponent moet bezitten kan afgeleid worden van de klasse-hiërarchie die hieraan ten grondslag ligt. In de onderstaande figuur is deze hiërarchie weergegeven.



**Figuur 4.1 – Klassediagram raamwerkcomponenten**

Uit het klassediagram is af te leiden dat het raamwerk (Framework) een verzameling raamwerkcomponenten (FrameworkComponents) bezit. Elk raamwerkcomponent is een subklasse van BasicFrameworkComponent. Dit betekent dat een raamwerkcomponent altijd twee interfaces moet implementeren, de interface FrameworkComponent en de interface BasicFrameworkComponent. Een raamwerkcomponent biedt dus een minimale set van operaties. De ontwikkelaar moet deze operaties implementeren, indien de default implementatie niet toereikend is. Het betreft de onderstaande operaties:

```

FrameworkComponent

function GetComponentDescription(): TComponentDescriptor;

    Geeft een verwijzing naar het ComponentDescriptor-object dat een
    beschrijving geeft van de raamwerkcomponent. De standaard
    implementatie hiervan hoeft niet overschreven te worden.

function GetComponentReference(): TComponentReference;

    Geeft een verwijzing naar het ComponentReference-object wat bij
    deze raamwerkcomponent hoort. De standaard implementatie hiervan
    hoeft niet overschreven te worden.

function GetProvidedServices: TIterator;

    Geeft een verwijzing naar de verzameling diensten die de
    raamwerkcomponent levert. De verzameling wordt uit de
    ComponentDescriptor gelezen, de standaard implementatie hoeft
    niet gewijzigd te worden.
  
```





```
function GetRequiredServices: TIterator;

  Geeft een verwijzing naar de verzameling diensten die de
  raamwerkcomponent vereist. De verzameling wordt uit de
  ComponentDescriptor gelezen, de standaard implementatie hoeft
  niet gewijzigd te worden.

function GetSubscriber : TSubscriber;

  Geeft een verwijzing naar het Subscriber-object dat bij dit
  raamwerkcomponent hoort. De standaard implementatie hoeft niet
  overschreven te worden.

function MultipleInstance : Boolean;

  Het resultaat van deze functie geeft aan of er meerdere
  instanties van dit raamwerkcomponent aanwezig mogen zijn.
  Standaard is dit mogelijk. Voor componenten waar slechts één
  instantie van aanwezig mag zijn moet de implementatie
  overschreven worden.

BasicFrameworkComponent

function GetFramework: TIFramework;

  Geeft een verwijzing naar het raamwerk. De implementatie kan
  niet overschreven worden in eigen raamwerkcomponenten.

function GetComponentType: TComponentType; override;

  Geeft het type aan van de component. Standaard wordt
  BasicFrameworkComponent als type gegeven (ctBasic). Deze
  implementatie dient overschreven te worden in de eigen
  raamwerkcomponent als het een modelapplicatie of generieke tool
  betreft (ctBuilding).
```

Elk raamwerkcomponent wordt als Delphi DLL gecompileerd. Deze DLL's bezitten een eenvoudig interface, bestaande uit twee operaties. Naast de eventuele overschrijving van standaard implementaties van de raamwerk-component-methodes moeten de standaard DLL functies ingevuld worden:

- **InitFrameworkComponent**

Deze functie geeft als resultaat de betreffende raamwerkcomponent. De DLL is dus de ingang om een raamwerkcomponent te instantiëren. De functie kent de onderstaande signatuur:

```
function(aFramework: TIFramework; aComponentReference:
ComponentReference):
  TFrameworkComponent;
```

- **GetXMLString**

Deze functie geeft de ComponentDescriptor van de raamwerkcomponent in de vorm van XML. De functie kent de onderstaande signatuur:

```
function: String
```

Als voorbeeld worden de DLL functies gegeven zoals deze in de grafiektool zijn opgenomen. Aanpassing voor eigen raamwerkcomponenten is eenvoudig doordat alleen klassenamen en de ComponentDescriptor aangepast moeten worden.

```
function InitGraphTool(aFramework: TIFramework; aComponentReference:
  TComponentReference): TPresentationComponent; export; stdcall;
begin
```



```

try
    // create frameworkcomponent
    Result := TGraphTool.Create(aFramework, aComponentReference);
except
    Result := nil;
end;
end;

function GetGraphToolXML: String;
begin
    Result :=
        '<SRWXML>' +
        '<HEADER>' +
            '<VERSION>0.1</VERSION>' +
            '<ACCESS>atReadOnly</ACCESS>' +
        '</HEADER>' +
        '<CONTENTS>' + '
            <COMPONENTDESCRIPTOR>' +
                '<NAME>' +
                    '<TDESCRIPTION>GraphTool</TDESCRIPTION>' +
                '</NAME>' +
                '<VERSION>1.0</VERSION>' +
                '<SIZE>50</SIZE>' +
                ...
            '</COMPONENTDESCRIPTOR>' +
            '<SERVICES>' +
            ...
            '</SERVICES>'
        '</CONTENTS>' +
        '</SRWXML>';
end;

```

Voor de exacte invulling van de DLL functies wordt verwezen naar het stappenplan voor ontwikkeling van modelapplicaties (hoofdstuk 5) en voor generieke tools (hoofdstuk 6)

### 4.3 Tips voor ontwikkeling van raamwerkcomponenten

Naast alle standaard specificaties zijn een aantal tips te geven die tijd kunnen besparen bij ontwikkeling van een raamwerkcomponent:

- Gebruik geen hints bij buttons en andere visual controls.
- Verstuur geen events naar de Publisher in de destructor van een raamwerkcomponent.



- Let op dat Delphi geen Garbage Collection functionaliteit bezit. Elk object moet expliciet weer vrijgegeven worden. Bij elke 'create' moet dus een 'destroy' opdracht opgenomen zijn.
- Een object mag alleen door een raamwerkcomponent gemaakt worden als dit raamwerkcomponent het object zelf weggooit en geen andere raamwerkcomponenten referenties hebben naar dit object.
- Voorkom dat in een try-except constructie een exception uit een ander raamwerkcomponent opgevangen wordt.
- Stuur exceptions altijd naar de Publisher.
- Toon zelf geen dialoogschermen maar gebruik de Messaging component.
- Bij de aanpassing van SRWLib moeten alle componenten opnieuw gecompileerd worden.
- Gebruik nooit tekst in sourcecode, maar verwijst naar een resourcestring die in een unit met constanten is opgenomen.
- In het zoekpad van een Delphi project mag alleen SRWLib opgenomen zijn en geen directories van andere raamwerkcomponenten.
- Indien een raamwerkcomponent andere bestanden gebruikt (data, dll, etc) moeten deze in de 'Additional files' sectie van een ComponentDescriptor opgenomen te zijn.
- Gebruik van **as** voor typecasting is niet mogelijk als een object in een ander raamwerkcomponent gecreëerd is. Gebruik in dat geval C-style typecasting (harde typecasting, bijvoorbeeld `ConcreteClass(AbstractClass).getName()`).
- Maak bij elke klasse en elke operatie een documentatie header.
- Gebruik de standaard naamconventies (zie hoofdstuk 8).
- Gebruik alleen externe (Delphi of ActiveX) componenten als hiervan de Sourcecode beschikbaar is. Door componenten die geen onderdeel van de Delphi VCL vormen altijd runtime te creëren is installatie van die component bij alle andere ontwikkelaars niet vereist.



## 5 Ontwikkeling van modelapplicaties

### 5.1 Inleiding

Een modelapplicatie valt binnen de SR architectuur onder de BuildingFrameworkComponents (BuildingFCs). Een modelapplicatie kan dus als bouwsteen in een case worden toegepast. Een modelapplicatie onderscheidt zich van andere BuildingFCs omdat een verzameling van aansluitpunten kan worden aangeboden. Elk aansluitpunt kan eigenschappen vragen en/of leveren.

Binnen de modelapplicatie is een rekenkern ingepakt (gewrapped). De modelapplicatie moet het mogelijk maken dat deze rekenkern op tijdstapbasis kan worden aangeroepen. Voor mogelijke uitwisseling van gegevens met andere modelapplicaties of generieke tools moet de eigen modelschematisatie in de wrapper vertaald worden naar de generieke vorm van aansluitpunten (modelcomponenten) en eigenschappen (modelattributen).

### 5.2 Stappenplan

Ondanks de grote verscheidenheid aan modellen die in SR kunnen worden opgenomen bestaat het migratietraject van deze modellen naar SR uit een groot aantal standaard activiteiten

Bij de ontwikkeling van een SR modelapplicatie kan ontwikkeld worden op basis van een aantal units sourcecode (in Delphi) waar vanuit de te ontwikkelen modelapplicatie verder ingericht kan worden. Deze sourcecode kan gezien worden als een set van functies (gegroepeerd in klassen) waarbinnen default implementaties gegeven zijn. Een aantal default implementaties moet verplicht overschreven worden om specifieke functionaliteit te kunnen bieden, een groot aantal functies is echter algemeen en vereist daarom geen aanpassing.

De implementatie van een modelapplicatie kan verdeeld worden in de onderstaande stappen (per stap worden in de hiernavolgende paragraaf de details uitgewerkt):

1. Definitie project source
2. Implementatie project source
3. Implementatie modelelementen en connectoren
4. Implementatie ModelApplication methodes
  - a) Wijziging klassenaam in interface en implementatie
  - b) Ontwikkeling SchematisationCreator
  - c) Ontwikkeling PropertyEditor
  - d) Implementatie methode Init
  - e) Implementatie methode Start
  - f) Implementatie methode Finalize
  - g) Implementatie methode Check
  - h) Implementatie methode Load
  - i) Implementatie methode Save



5. Koppeling met rekenkern
  - a) Communicatie
  - b) Foutafhandeling
  - c) Geheugenbeheer
6. Testen
  - a) Registratie in het raamwerk
  - b) Samenstellen van een case
  - c) Property editor
  - d) Simulatie
  - e) Geheugengebruik

### 5.3 Realisatie van een modelapplicatie

Aan de hand van het stappenplan dat in de voorgaande paragraaf is weergegeven kan een modelapplicatie gerealiseerd worden. Er is een default Delphi project beschikbaar waar vanuit de ontwikkeling gestart kan worden. Dit project, DefaultModelApplication.dpr kan in Delphi 5 ingelezen worden.

#### 5.3.1 Stap 1: Definitie project source

Na het inlezen van het project DefaultModelApplication.dpr in Delphi dienen zowel het project, als de daarbinnen opgenomen units onder een andere naam bewaard te worden. In een standaard situatie zal de term 'default' vervangen worden door een meer toepasselijke term (zoals Modflow).

Na deze stap zijn de bestanden DefaultModelApplication.dpr, fDefaultPropertyEditor.pas, fDefaultPropertyEditor.dfm, uDefaultModelApplication.pas en uDefaultSchematisation.pas hernoemd tot bijvoorbeeld ModflowModelApplication.dpr, fModflowPropertyEditor, etc.

Standaard staan in het project alle zoekpaden goed ingesteld om bijvoorbeeld alle units uit SRWLib te kunnen vinden. Dit kan eventueel gecontroleerd worden door het scherm dat getoond wordt na de menu-optie Project | Options te controleren met de schermafdrucken van bijlage B.

#### 5.3.2 Stap 2: Implementatie project source

In de sourcecode van het projectbestand DefaultModelapplication.dpr worden standaard implementaties gegeven van de methodes:

```
InitModelApplication
GetModelApplicationXML
```

Deze methodes moeten aangepast worden om eigen gedrag toe te kunnen kennen aan de modelapplicatie<sup>2</sup>.

- a) InitModelApplication

---

<sup>2</sup> Tip: zoek in het projectbestand naar de term 'default' en vervang dit door de eigen modelnaam, zoals SOBEK of DUFLOW.



Deze methode wordt aangeroepen bij het instantiëren van een modelapplicatie. In de default implementatie wordt een object van het type `TDefaultModelApplication` gecreëerd. Dit type moet vervangen worden door de klasse waarin de eigen modelapplicatie is geïmplementeerd (bijvoorbeeld `TSwapModelApplication`).

```
function InitModelApplication(aFramework: TIFramework;
    aComponentReference: TComponentReference):
    TModelApplication; export;stdcall;
begin
    try
        // create frameworkcomponent
        Result := TDefaultModelApplication.Create(aFramework,
            aComponentReference);
    except
        Result := nil;
    end;
end;
```

#### b) GetModelApplicationXML

In deze methode wordt alle vereiste meta-informatie van de modelapplicatie opgenomen in de vorm van XML. Deze informatie wordt gebruikt tijdens registratie van een component. De XML string moet aangepast worden zodat het raamwerk kan controleren welke component het betreft. De XML structuur bestaat uit twee onderdelen: <header> en <contents>. Het onderdeel <header> hoeft niet te worden aangepast. Het <contents> deel dient op een aantal plaatsen aangepast te worden. XML bestaat uit een aantal zogenaamde tags (zoals ook binnen HTML wordt gebruikt), waartussen een waarde of een nieuwe tag kan worden opgenomen. SR verwacht tags op een bepaalde plaats (volgens een bepaalde hiërarchie). Het is daarom belangrijk dat deze hiërarchie niet wordt gewijzigd.

Standaard bestaat de XML string uit de onderstaande tags en values (er zijn regelnummers voor geplaatst zodat in de tekst hiernaar gerefereerd kan worden)

```
01 <SRWXML>
02   <HEADER>
03     <VERSION>1.0</VERSION>
04     <ACCESS>atReadOnly</ACCESS>
05   </HEADER>
06   <CONTENTS>
07     <COMPONENTDESCRIPTOR>
08       <NAME>
09         <TDESCRIPTION>DefaultModelapplication</TDESCRIPTION>
10       </NAME>
11       <VERSION>1.0</VERSION>
12       <SIZE>50</SIZE>
13       <CREATOR>
14         <TOWNER>
15           <COMPANY>
16             <TDESCRIPTION>Default Company</TDESCRIPTION>
17           </COMPANY>
```



```

18      <AUTHOR>
19      <TDESCRIPTION>Default SoftwareEngineer</TDESCRIPTION>
20      </AUTHOR>
21      </TOWNER>
22      </CREATOR>
23      <DATE>01/01/2000</DATE>
24      <TYPE>ctBuilding</TYPE>
25      <ACCES>atNormal</ACCES>
26      <ICON>
27      <TURL>Default.ico</TURL>
28      </ICON>
29      <ADDITIONALFILES>
30      </ADDITIONALFILES>
31      <CREATEOPTION>coManual</CREATEOPTION>
32      <COMPONENTMODEL>cmDLL</COMPONENTMODEL>
33      </COMPONENTDESCRIPTOR>
34      <SERVICES>
35      <SERVICEDESCRIPTOR>
36      <NAME>
37      <TDESCRIPTION>snModelApplication</TDESCRIPTION>
38      </NAME>
39      <TYPE>stData</TYPE>
40      <PART>spProvide</PART>
41      </SERVICEDESCRIPTOR>
42      <SERVICEDESCRIPTOR>
43      <NAME>
44      <TDESCRIPTION>snProcessManager</TDESCRIPTION>
45      </NAME>'
46      <TYPE>stEvent</TYPE>
47      <PART>spNeed</PART>
48      </SERVICEDESCRIPTOR>
49      </SERVICES>
50      </CONTENTS>
51      </SRWXML>

```

In de XML string dient het volgende worden ingevuld:

- Naam van de component

Op regel 9 in de XML structuur kan binnen de TDescription-tag de naam "DefaultModelApplication" vervangen worden door een nieuwe naam. Hierin mogen spaties toegepast worden, echter niet de tekens "<" en ">".

- Versie

Op regel 11 in de XML structuur kan het versienummer ingevuld worden tussen de Version-tag.

- Bestands grootte

Op regel 12 in de XML structuur kan de omvang van de DLL ingevuld worden. Deze tag wordt in deze versie niet gebruikt. Invullen/wijzigen is daarom niet vereist.

- Beheerder/eigenaar/contactpersoon

Van de beheerders van een component kan de bedrijfsnaam en de naam van de contactpersoon ingevuld worden.

```

<TOWNER>
  <COMPANY>
    <TDESCRIPTION>Default Company</TDESCRIPTION>

```



```

</COMPANY>
<AUTHOR>
  <TDESCRIPTION>Default Software Engineer</TDESCRIPTION>
</AUTHOR>
</TOWNER>

```

Deze namen kunnen in de TDescription tags ingevuld worden (regel 16 en 19 in de XML structuur).

- Datum

De datum van compileren van de DLL kan op regel 23 in de XML structuur ingevuld worden. Het is daarbij vereist dat het reeds ingevulde datum formaat (dd/mm/yyyy) wordt aangehouden. SR werkt namelijk intern altijd volgens dit format, ongeacht specifieke PC-instellingen.

- Component type

Het type component geeft aan welke subklassen van FrameworkComponent is geïmplementeerd in de DLL: BasicFrameworkComponent (BasicFC) of BuildingFrameworkComponent (BuildingFC). Een modelapplicatie is altijd afgeleid van een BuildingFC. Daarom mag de standaard invulling hiervan niet overschreven worden.

- Toegangstype

Een toegangstype legt vast of een component fysiek op een plaats moet blijven staan of dat kopiëren toegestaan is. In versie 1.0 wordt alleen het type "atNormal" toegepast en dient daarom niet gewijzigd te worden.

- Naam icoon

Het is mogelijk bij de modelapplicatie een icoon van 32x32 pixels op te nemen, zodat dit gebruikt kan worden in bijvoorbeeld het raamwerk en de ProcessManager. De naam van dit icoon (\*.ico) kan in de Icon-tag ingevuld worden (regel 27).

```

<ICON>
  <TURL>Default.ico</TURL>
</ICON>

```

Binnen de XML structuur wordt gewerkt met relatieve paden, ten opzichte van de locatie van de DLL. Indien geen pad wordt opgegeven, zoals in de default icoon-naam wordt verondersteld dat de icoon onder dezelfde directory als de DLL staat.

- Namen van eventueel extra vereiste bestanden

Binnen de DLL kan gebruik gemaakt worden van aanvullende bestanden (bijvoorbeeld een DLL die als rekenkern fungeert). Deze afhankelijkheden dienen in de XML structuur opgenomen te worden (regel 29/30).

```

'<ADDITIONALFILES>'
'</ADDITIONALFILES>'

```

Standaard zijn er geen additionele bestanden opgenomen. Indien dit wel vereist is moet in de AdditionalFiles-tag deze bestandsnaam (of namen) ingevuld worden. Een bestandsnaam wordt in SR als TURL-type beschouwd. Indien de bestanden "Rekenkern.DLL" en "Database.DBF" zijn vereist ziet de AdditionalFiles-tag er als volgt uit:



```

<ADDITIONALFILES>
  <TURL>Rekenkern.DLL</TURL>
</ADDITIONALFILES>
<ADDITIONALFILES>
  <TURL>Database.DBF</TURL>
</ADDITIONALFILES>

```

- **CreateOption**

Op regel 31 in de XML structuur kan een zogenaamde CreateOption worden gedefinieerd. Hiervoor zijn de types "coManual" en "coAuto" beschikbaar. Een component volgens type coAuto wordt bij het activeren van de "Design-mode" van het raamwerk (de mode waarin een case samengesteld kan worden) automatisch geïntanceerd. Een modelapplicatie mag pas tijdens het initialiseren van een simulatie worden geïntanceerd. Daarom staat hiervoor het type coManual vast.

- **Component model**

Op dit moment worden alleen DLL's binnen het raamwerk geregistreerd. Dit wordt aangegeven met het componentmodel-type cmDLL. Dit type staat vast voor versie 1.0 van SR en dient daarom niet gewijzigd te worden.

- **Vereiste en te leveren diensten**

Elk raamwerkcomponent levert minimaal één dienst (service). Een modelapplicatie biedt de service "modelapplicatie". Intern wordt hiervoor het type "snModelApplication" gebruikt. Daarnaast is het mogelijk dat een component alleen kan functioneren als een bepaalde dienst door een ander component geleverd kan worden. Een modelapplicatie kan bijvoorbeeld niet zonder de service "ProcessManager". Intern wordt deze dienst getypeerd als "snProcessManager". Deze afhankelijkheden dienen in de XML structuur vastgelegd te zijn. De default afhankelijkheden van een modelapplicatie zijn in de XML structuur weergegeven (regels 34 t/m 49). Hieraan kunnen extra vereiste of te leveren diensten worden toegevoegd. Alle onderscheiden services zijn opgenomen in de code-unit uConsts.pas in SRWlib.

```

<SERVICES>
  <SERVICEDESCRIPTOR>
    <NAME>
      <TDESCRIPTION>snModelApplication</TDESCRIPTION>
    </NAME>
    <TYPE>stData</TYPE>
    <PART>spProvide</PART>
  </SERVICEDESCRIPTOR>
  <SERVICEDESCRIPTOR>
    <NAME>
      <TDESCRIPTION>snProcessManager</TDESCRIPTION>
    </NAME>'
    <TYPE>stEvent</TYPE>
    <PART>spNeed</PART>
  </SERVICEDESCRIPTOR>
</SERVICES>

```



### 5.3.3 Stap 3: Implementatie modelelementen en connectoren

Binnen SR wordt een schematisatie van een modelapplicatie gezien als een netwerk van verbonden *modelcomponenten*. Er wordt hierbij allereerst een onderscheid gemaakt tussen knooppunten en verbindingen tussen knooppunten: *Modelelementen* en *Connectoren*. Daarnaast wordt een onderscheid gemaakt in de mogelijkheid tot aggregaties. Een modelcomponent dat zelf modelcomponenten kan bevatten is een samengesteld modelcomponent. Van zowel een Modelelement als een connector bestaat een dergelijk samengestelde variant: het *CompositeModelelement* en de *CompositeConnector*. Indien een modelcomponent geen elementen kan bevatten wordt dat component als enkelvoudig beschouwd en zijn het *SingleModelelement* en de *SingleConnector* de mogelijke opties.

Bij de ontwikkeling van een modelapplicatie moeten eigen modelelementen en connectoren ontworpen worden. Dit betekent dat verschillende types onderscheiden moeten worden. Elk type is een subklasse van *CompositeModelelement*, *CompositeConnector*, *SingleModelelement* of *SingleConnector*. Voorbeelden van dergelijke element zijn een pomp, een bodemlaag, de atmosfeer en een opp.water-sectie. Per onderscheiden type moet het volgende worden vastgelegd:

- Welke superklasse geldt voor het type?
- Indien het een samengesteld element betreft: welke typen elementen mogen onderdeel uitmaken van het type?
- Welke attributen heeft het type?

Elk modelcomponent kan attributen bezitten. Een opp.water-sectie kan bijvoorbeeld waterhoogte, lengte en de hellingshoek als attribuut bezitten. Elk attribuut wordt geïmplementeerd als subklasse van het type *ModelAttribute*.

Na het ontwerp van alle typen modelcomponenten met bijbehorende attribuut-typen dienen deze types geïmplementeerd te worden in Delphi als klassen. In het DefaultModelApplication project is standaard de unit *uDefaultSchematisation* opgenomen. In deze unit zijn een aantal voorbeeld modelcomponenten en attributen opgenomen. Deze code heeft uiteraard geen functie bij de implementatie van een specifieke modelapplicatie. Deze code kan daarom verwijderd worden of gewijzigd worden in commentaar, zodat het als hulp kan dienen bij de implementatie van eigen klassen.

De implementatie van de modelcomponenten en attributen kan volgens onderstaande stappen worden uitgevoerd:

#### a) Specificatie ModelComponent

Bij de specificatie van een modelcomponent dient allereerst vastgelegd te worden welke superklasse toegepast wordt. In het onderstaande voorbeeld is een modelcomponent "TTestCompositeModelElement" afgeleid van de klasse *CompositeModelElement*.

```
TTestCompositeModelElement = class(TCompositeModelElement)
private
    FSinus,
    FConstant: TModelAttribute;
public
```

```

    constructor Create(aID: LongInt; aGeometry: TGeometry;
        aBuildingFC : TBuildingFrameworkComponent);
    destructor Destroy; override;
    function GetDescription: TDescription; override;
    function GetAttributes: TIterator; override;

```

```
end;
```

Vervolgens kan een lijst met attribuutnamen gespecificeerd worden in het “private-part” van de klasse-definitie. In het bovenstaande voorbeeld zijn de attributen “FSinus” en “FConstant” gespecificeerd.

Na de specificatie van de klasse dient uiteraard de implementatie hiervan gemaakt te worden. Bij de hierboven gegeven klasse “TestCompositeModelElement” is de onderstaande implementatie van toepassing (er zijn regelnummers toegevoegd om te kunnen refereren naar specifieke regels code):

```

01  { TTestCompositeModelElement }
02
03  constructor TTestCompositeModelElement.Create(aID:LongInt;
04      aGeometry:TGeometry; aBuildingFC :
05      TBuildingFrameworkComponent);
06  begin
07      inherited Create(aID, aGeometry, aBuildingFC);
08      FSinus:= TSinusAttribute.Create(self, 'SinusAttribute');
09      FConstant := TConstantAttribute.Create(self,
10      'ConstantAttribute');
11  end;
12
13  destructor TTestCompositeModelElement.Destroy;
14  begin
15      FSinus.Destroy;
16      FConstant.Destroy;
17      inherited;
18  end;
19
20  function TTestCompositeModelElement.GetAttributes: TIterator;
21  begin
22      Result := TSingletonIterator.Create(FSinus);
23      Result := TCombinedIterator.Create(Result,
24      TSingletonIterator.Create(FConstant));
25  end;
26
27  function TTestComposModelElement.GetDescription: TDescription;
28  begin
29      Result := 'TestCompositeModelElement' + IntToStr(GetID);
30  end;

```

In de constructor moeten de attribuut-objecten gemaakt worden. In regel 7 en 8 worden de voorbeeld-attributen gemaakt. Als argument wordt de naam van het attribuut opgegeven.

Objecten die in de constructor worden gemaakt moeten in de destructor weer verwijderd worden. Dit geldt dus voor de attribuut-objecten. In het bovenstaande code voorbeeld worden in regel 13 en 14 de attributen verwijderd.

Aan elk modelcomponent kan een lijstje met alle attributen gevraagd worden. Hiervoor wordt de methode GetAttributes toegepast. Als resultaat van deze functie wordt een *Iterator* object gegeven. De lijst met attributen wordt dus niet gegeven, maar een object waarmee deze



lijst afgelopen kan worden. In de implementatie van deze methode moet dit Iterator-object compleet gemaakt worden. In regel 20 en 21 van de voorbeeld implementatie is dit uitgewerkt. Indien een modelcomponent geen attributen bezit is de implementatie van deze methode:

```
Result := EmptyIterator.Create;
```

Van een modelcomponent kan de naam opgevraagd worden met de methode `GetDescription`. Regel 27 van bovenstaande implementatie geeft het resultaat hiervan weer. De naam "TestCompositeModelcomponent" dient vervangen te worden door een eigen naam.

Er worden twee soorten attributen onderscheiden binnen het raamwerk. Een attribuut-object kan waarden leveren of een attribuut-object kan waarden vereisen/gebruiken. De lijsten van 'leverende' en 'ontvangende' attributen zijn dus subsets van de totale lijst attributen. De methoden `GetProvidedAttributes` en `GetRequiredAttributes` dienen als resultaat deze subsets te leveren, in de vorm van een Iterator-object. Voor deze methoden zijn default implementaties in de klasse `ModelComponent` opgenomen. Deze implementaties geven een lijst met leverende of vragende attributen als resultaat.

### Specificatie ModelAttribute

Elk attribuut dat aan een `ModelComponent`-type wordt toegekend moet als subklasse van `ModelAttribute` geïmplementeerd worden. In het onderstaande code fragment is een voorbeeld attribuut gespecificeerd:

```
TSinusAttribute = class(TModelAttribute)
public
    function CanProvide: Boolean; override;

    function GetValue(aTime: TSrwTime): Double; override;
    function SetValue(aTime: TSrwTime;
        aValue: Double): TOperationResult; override;

    function GetType(): TAttributeType; override;
    function GetSource(): TSourceType; override;
    function GetUnitOfMeasurement : TunitOfMeasurement; override;
end;
```

De algemene klasse `ModelAttribute` biedt voor een aantal functies een default implementatie. Deze functies dienen dus alleen overschreven te worden indien voor dit attribuut een andere implementatie geldt. In het bovenstaande voorbeeld wordt voor de onderstaande functies de default implementatie gebruikt (en zijn daarom niet in de interface opgenomen):

```
function GetName: TDescription;
function IsRequired: Boolean; virtual;
function CanUse: Boolean; virtual;
function HasExchangeItems: Boolean;
function AddExchangeItem(aExchangeItem: TExchangeItem):
    TOperationResult;
function GetProvidingExchangeItems: TIterator;
```

In de unit "uModelAttribute.pas" in `SRWlib` zijn de default implementaties opgenomen. De functies die in deze klasse geen implementatie hebben (abstract methods) of een ander implementatie vereisen dienen in de eigen subklasse opgenomen te worden. In het voorbeeld van `TSinusAttribute` is dit voor een 7-tal functies toegepast.



### 5.3.4 Stap 4: Implementatie methodes ModelApplication

Het object dat gebruik maakt van de aansluitpunten met bijbehorende attributen en de aansturing van de rekenkern verzorgt is een instantie van (een subklasse) van ModelApplication. Deze klasse is in het default project opgenomen en is in een voorgaande stap hernoemd van DefaultModelApplication naar een nieuwe naam.

```

01 TDefaultModelApplication = class(TModelApplication)
02     private
03
04         FEndTime, FStartTime: TSrwTime;
05     protected
06         procedure CreateSchematisation;
07         function CheckAttributes(aModelComp:TModelComponent):Boolean;
08     public
09         constructor Create(aFramework: TIFramework;
10             aComponentReference: TComponentReference);
11         destructor Destroy; override;
12         { BuildingFrameworkComponent }
13         function HasPropertyEditor: Boolean; override;
14         function Edit: TOperationResult; override;
15
16         function Save(aSRWDataObject : TISRWDataObject) :
TISRWDataObject; override;
17         function Load(aSRWDataObject : TISRWDataObject) :
TOperationResult; override;
18         function Check: Boolean; override;
19         function Init: TOperationResult; override;
20         function Start(aStartTime,aEndTime:TSrwTime;aTraceID:LongInt) :
TOperationResult; override;
21         function GetStartTime : TSrwTime; override;
22         function GetEndTime : TSrwTime; override;
23         { ModelApplication }
24
25         function GetNextTime: TSRWTime; override;
26         function Finalize: TOperationResult; override;
27     end;

```

In het bovenstaande voorbeeld is de default klasse weergegeven. Deze default klasse dient achtereenvolgens op de volgende plaatsen aangepast te worden:

#### a) SchematisationCreator

De methode CreateSchematisation (regel 6 in bovenstaande code fragment) zorgt ervoor dat alle aansluitpunten met bijbehorende attributen worden gemaakt. De aansluitpunten moeten na creatie van de modelapplicatie beschikbaar zijn dus de methode CreateSchematisation wordt in de constructor van deze klasse aangeroepen. Het opbouwen van de schematisatie bestaat uit de volgende onderdelen:

- Creëer de aansluitpunten en connectoren;
- Indien logisch, verbind de aansluitpunten met behulp van de connectoren;
- Voeg alle aansluitpunten en connectoren toe aan de verzameling "FComponents". Deze verzameling is onderdeel van elke modelapplicatie.



Bij de creatie van aansluitpunten en connectoren dient de geometrie van deze componenten bekend te zijn. Deze geometrie is als argument vereist in de constructor van zowel modelcomponenten als connectoren. Voor de geometrie wordt gebruik gemaakt van objecten van het type TGeometry. Deze objecten voldoen aan de OpenGis specificaties en bezitten daarnaast de functionaliteit om zichzelf als GML-string weer te geven.

#### b) Property Editor

Indien een editor beschikbaar is moet de methode HasPropertyEditor overschreven worden (regel 13) met de implementatie:

```
Result := True;
```

De editor kan opgevraagd worden met de methode Edit. In deze methode wordt het bijbehorende form gemaakt en getoond. Ook binnen deze methode moet het form weer verwijderd worden. Hiervoor kan de onderstaande code toegepast worden:

```
var
  lPropertyEditor: TfrmDefaultPropertyEditor;
try
  lPropertyEditor := TfrmPropertyEditor.Create(Application);

  try
    lPropertyEditor.ShowModal;
    if lPropertyEditor.ModalResult = mrOK then
    begin
      { only result success if the user didn't cancel }
      Result := orSuccessful;
    end;
  finally
    lPropertyEditor.Free;
  end; //finally
except
  Result := orFailed;
end;
```

#### c) Init functie

De initialisatie van de rekenkern wordt met behulp van deze methode geïnitieerd (regel 17 in het voorbeeld code fragment). Hier wordt dus een koppeling met de rekenkern gemaakt. Het is uiteraard afhankelijk van de rekenkern wat hierbij exact wordt uitgevoerd. Na deze methode dient de modelapplicatie de toestand "Initialized" te bezitten. Binnen SR wordt hiervoor het type "csIntialized" toegepast. Deze toestand wordt standaard toegekend in de default implementatie van de methode Init. Hiervan kan gebruik gemaakt worden in eigen modelapplicaties door het onderstaande statement toe te voegen aan de eigen implementatie van de methode Init:

```
Inherited Init;
```

#### d) Start functie

Het uitvoeren van rekentijdstappen wordt met behulp van deze functie gestart (regel 18 in het voorbeeld code fragment). In de implementatie vindt uiteraard delegatie naar de betreffende rekenkern plaats. Naast de aansturing van de rekenkern is het belangrijk dat ook hier de juiste status aan de modelapplicatie wordt toegekend. Analooq aan de Init methode wordt dit gegarandeerd als in de implementatie de gelijknamige methode van de superklasse wordt aangeroepen volgens:

```
Inherited Start(aStartTime, aEndTime, aTraceID);
```



e) Finalize functie

Om de actuele toestand van de modelapplicatie en bijbehorende rekenkern weg te schrijven zodat een nieuwe initialisatie uitgevoerd kan worden wordt de methode finalize toegepast (regel 25 in het voorbeeld code fragment). Een equivalent hiervan dient dus in de rekenkern beschikbaar te zijn. Naast deze functionaliteit is het ook hier belangrijk dat de nieuwe toestand "NotInitialized" uiteindelijk wordt toegekend. Dit is beschikbaar in de default implementatie van ModelApplication.

f) Controle van huidige instellingen

Voordat een initialisatie van een modelapplicatie wordt uitgevoerd zorgt de ProcessManager er voor dat alle modelapplicaties en connecties binnen de case gecontroleerd worden. Dit betekent dat bijvoorbeeld per modelapplicatie nagegaan wordt of alle attributen die elke tijdstap een waarde vereisen ook daadwerkelijk een leverancier (een ander attribuut) hiervoor hebben. De controle wordt geïmplementeerd in de methode Check (regel 16 in voorafgaande code fragment). De controle van attributen wordt in de default implementatie uitgevoerd waarbij gebruik gemaakt wordt van de "protected" methode CheckAttributes.

g) Opslaan van huidige instellingen

Een case kan worden bewaard met behulp van de DataEngine. Een case is een set van modelapplicaties, generieke tools en verbindingen. Bij het opslaan van een case wordt de gehele configuratie bewaard. Elke modelapplicatie is er zelf verantwoordelijk voor dat eigen instellingen (bijvoorbeeld ingesteld in de property editor) bewaard blijven. De methode Save (regel 15 in het voorafgaande code fragment) biedt deze functionaliteit. De implementatie hiervan dient per modelapplicatie ingevuld te worden, er is dus geen default implementatie beschikbaar.

### 5.3.5 Stap 5: Koppeling met rekenkern

De koppeling tussen een modelapplicatie, in de vorm van een DLL, en de rekenkern kan op verschillende manieren gerealiseerd worden. De mogelijkheden zijn direct afhankelijk van de vorm waarin de rekenkern beschikbaar is. Voor de hand liggende vormen zijn een executable, een DLL of een COM server. Een aantal aspecten zijn hierbij van belang en kunnen of moeten geregeld worden:

a) Communicatie

De modelapplicatie zal in de methode INIT( ) de rekenkern moeten starten en in de FINALIZE( ) methode de rekenkern moeten kunnen afsluiten.

b) Foutafhandeling

Als het starten of afsluiten van een rekenkern niet correct verloopt is een goede foutafhandeling van belang. De rekenkern zal dus aan de modelapplicatie aan moeten kunnen geven in hoeverre een opdracht succesvol is uitgevoerd. Een belangrijk aspect hierbij is dat de rekenkern of de modelapplicatie niet zelf foutmeldingen naar het scherm stuurt of exceptions genereert. Het raamwerk moet op de hoogte gesteld worden van eventuele fouten en daarom dient hiervoor de standaard event-oplossing gekozen te worden (zie hoofdstuk 3.5).



c) Geheugenbeheer

Geheugenruimte dat door een rekenkern wordt gealloceerd kan niet door het raamwerk vrijgegeven worden. Het is dus van belang dat in de FINALIZE methode geladen DLL's en ruimte voor tussenresultaten vrijgegeven worden.

### 5.3.6 Stap 6: Testen

Het testen van een ontwikkelde modelapplicatie start uiteraard met het succesvol compileren van het Delphi project zonder 'hints' en 'warnings'. Het testen van de gecompileerde DLL bestaat uit:

a) Registratie DLL in raamwerk

De DLL moet zonder fouten of waarschuwingen geregistreerd kunnen worden in de raamwerkapplicatie.

b) Toevoegen van de modelapplicatie aan een case

De DLL moet zonder fouten of waarschuwingen aan de Composer van de ProcessManager toegevoegd kunnen worden.

c) Gebruik van de Property-editor

Indien een eigen editor is gemaakt, moet deze via de optie 'Component | Properties' en vervolgens 'Internal editor' gestart kunnen worden.

d) Uitvoeren van een simulatie

Als een controle van de case (check optie) geen foutmeldingen geeft moet de modelapplicatie over het gespecificeerde interval foutloos de rekenstappen uit kunnen voeren.

e) Geheugengebruik door de modelapplicatie

Geheugen dat tijdens een simulatie door een modelapplicatie wordt gealloceerd voor het laden van libraries of het bewaren van tussenresultaten dient in de finalize-methode weer vrijgegeven te worden. Het veelvuldig na elkaar laten uitvoeren van dezelfde case mag geen toename in het geheugengebruik door de applicatie veroorzaken.





## 6 Ontwikkeling van generieke tools

### 6.1 Inleiding

Naast een modelapplicatie is een generieke tool een specialisatie van BuildingFrameworkComponent (BuildingFC). Een generieke tool onderscheidt zich van een modelapplicatie omdat geen aansluitpunten met attributen worden gegeven (een generieke tool heeft namelijk geen modelschematisatie). Voorbeelden van generieke tools zijn een grafiektool, een GIS-viewer en een analysetool. Een generieke tool vervult één of meerdere algemene taken binnen een case.

### 6.2 Stappenplan

In vergelijking met een modelapplicatie is voor een generieke tool een minder eenduidig stappenplan te geven. Het weergeven van resultaten in een grafiek vereist bijvoorbeeld een andere implementatie dan het valideren van resultaten op basis van metingen. Dit komt in het onderstaande stappenplan tot uiting omdat een mindere mate van detail is aangebracht in vergelijking met het stappenplan voor modelapplicaties.

- 1) Definitie project source
- 2) Implementatie project source
- 3) Implementatie BuildingFrameworkComponent methodes
  - a) Maken subklasse GenericTool
  - b) Ontwikkelen Property editor
  - c) Implementatie methode Check
  - d) Implementatie methode Init
  - e) Implementatie methode Start
  - f) Implementatie methode Finalize
  - g) Implementatie methode Load
  - h) Implementatie methode Save
- 4) Implementatie GenericTool methodes
  - a) Implementatie methodes Get/Set Starttime
  - b) Implementatie methodes Get/Set Endtime
  - c) Implementatie methode GetModelComponentByID
- 5) Indien relevant: Implementatie PresentationComponent methodes
  - a) Implementatie methode AddAttribute
  - b) Implementatie methode RemoveAttribute
  - c) Implementatie methode GetAttributes
  - d) Implementatie methode GetModelComponents
- 6) Testen

### 6.3 Realisatie van een presentatiecomponent

Aan de hand van het stappenplan dat in de voorgaande paragraaf is weergegeven kan een generieke tool gerealiseerd worden. Er is net als voor de modelapplicatie een default Delphi project beschikbaar waar vanuit



de ontwikkeling gestart kan worden. Bij de ontwikkeling van een presentatiecomponent kan tevens het project GraphTool als voorbeeld gebruikt worden.

### 6.3.1 Stap 1: Definitie project source

Deze stap is identiek aan de eerste stap bij de ontwikkeling van een modelapplicatie. De vereiste Delphi projectbestanden moeten aangemaakt worden en bewaard worden onder een unieke naam.

De meest eenvoudige mogelijkheid is het aanpassen van het projectbestand van de grafiektol: Graphtool.dpr. Indien geen presentatiecomponent ontwikkeld wordt kan het project DefaultModelapplication toegepast worden, waarbij de units fDefaultPropertyEditor.pas, fDefaultPropertyEditor.dfm, uDefaultModelApplication.pas en uDefaultSchematisation.pas verwijderd kunnen worden.

Standaard staan in het project alle zoekpaden goed ingesteld om bijvoorbeeld alle units uit SRWLib te kunnen vinden. Dit kan eventueel gecontroleerd worden door het scherm dat getoond wordt na de menu-optie Project | Options te controleren met de schermafdrucken van bijlage B.

### 6.3.2 Stap 2: Implementatie project source

In de sourcecode van het projectbestand Graphtool.dpr worden standaard implementaties gegeven van de methodes:

```
InitGraphtool
GetGraphtoolXML
```

Deze methodes moeten aangepast worden om eigen gedrag toe te kunnen kennen aan de modelapplicatie<sup>3</sup>. De methodenamen kunnen hierbij aangepast worden met een meer toepasselijke naamgeving. Bij de verdere beschrijving zal uitgegaan worden van de namen InitGenericTool en GetGenericToolXML

#### a) InitGenericTool

Deze methode wordt aangeroepen bij het instantiëren van een generiek tool. In de implementatie van Graphtool wordt een object van het type TGraphtool gecreëerd. Dit type moet vervangen worden door de klasse waarin de eigen generieke tool is geïmplementeerd (bijvoorbeeld TGISViewer).

```
function InitGenericTool(aFramework: TIFramework;
  aComponentReference: TComponentReference):
  TGenericTool; export; stdcall;
begin
  try
    // create frameworkcomponent
```

<sup>3</sup> Tip: zoek in het projectbestand naar de term 'default' en vervang dit door de eigen modelnaam, zoals SOBEK of DUFLOW.



```

    Result := TGenericTool.Create(aFramework,
        aComponentReference);
except
    Result := nil;
end;
end;

```

#### b) GetGenericToolXML

In deze methode wordt alle vereiste meta-informatie van de generieke tool opgenomen in de vorm van XML. Deze informatie wordt gebruikt tijdens registratie van een component. De XML string moet aangepast worden zodat het raamwerk kan controleren welke component het betreft. De XML structuur bestaat uit twee onderdelen: <header> en <contents>. Het onderdeel <header> hoeft niet te worden aangepast. Het <contents> deel dient op een aantal plaatsen aangepast te worden. De onderdelen zijn identiek aan de XML beschrijving van een modelapplicatie. De beschrijving bij stap 2 van de ontwikkeling van een modelapplicatie kan hierbij dus toegepast worden.

### 6.3.3 Stap 3: Implementatie BuildingFC methodes

De generieke tool die ontwikkeld wordt dient een subklasse te zijn van TGenericTool of TPresentationComponent. Door deze hiërarchie moeten een aantal standaard methodes ingevuld worden om specifieke functionaliteit te kunnen bieden. De eerste set methodes die van toepassing zijn worden bepaald door de interface van BuildingFC.

#### a) Maken subklasse GenericTool

De ontwikkeling start met het vastleggen van een nieuwe subklasse. De naam van deze klasse moet gelijk zijn aan de naam die is ingevuld in stap 1a in het Delphi projectbestand. Indien een presentatiecomponent ontwikkeld wordt is het mogelijk de unit uGraphTool aan te passen voor de eigen component. Indien geen presentatiecomponent gemaakt wordt moet een nieuwe unit aan het project toegevoegd worden waarin de klasse wordt vastgelegd.

#### b) Ontwikkelen Property editor

Als de generieke tool gebruik maakt van een eigen scherm, wordt dit als zogenaamde Property editor aan de component toegevoegd. De methode hasPropertyEditor moet aan de eigen klasse toegevoegd worden met de volgende implementatie:

```

    Result := True;

```

Het eigen scherm kan als Delphi TForm aan het project worden toegevoegd.

#### c) Implementatie methode Check

De methode Check dient in de eigen klasse ingevuld te worden. Deze methode wordt door de ProcessManager aangeroepen, voordat een case wordt gestart. Alle instellingen die een gebruiker



ingevoerd moet hebben, moeten in deze methode gecontroleerd worden.

d) Implementatie methode Init

De start van een simulatie begint met de aanroep van de methode Init van elke modelapplicatie en generieke tool binnen de case. Het starten van een hulpprogramma of het inlezen van een database of bestand zijn typische onderdelen die in de Init methode ingevuld worden.

e) Implementatie methode Start

De methode Start wordt door de ProcessManager aangeroepen tijdens een simulatie, zodra de component een rekentijdstap kan uitvoeren. De gewenste functionaliteit tijdens de simulatie dient in deze methode dus ingevuld te worden. In de grafiektool worden de resultaten van gekoppelde modellen ingelezen en aan lijnseries toegevoegd.

f) Implementatie methode Finalize

Na het afsluiten van een case is het van belang dat geladen bestanden, gealloceerd geheugen en gestarte hulpbestanden vrijgegeven worden. In de methode Finalize dient dit uitgevoerd te worden. In een grafiektool worden bijvoorbeeld alle ingelezen resultaten verwijderd uit de grafiek.

g) Implementatie methode Load

Alle instellingen die specifiek zijn voor de component, maar tijdens het inlezen van een eerder gemaakte case weer gebruikt moeten worden, moeten door de component zelf bewaard en ingelezen worden. Hiervoor kan gebruik gemaakt worden van de functionaliteit van de DataEngine. Het aanpassen van de Load-methode van de grafiektool is de meest eenvoudige oplossing hiervoor. In hoofdstuk 7 wordt het gebruik van de DataEngine hiervoor in detail beschreven.

h) Implementatie methode Save

Vergelijkbaar met het inlezen van specifieke component-instellingen moeten deze instellingen bewaard worden door aanroep van de methode Save. Gebruik van de implementatie hiervoor van de grafiektool als voorbeeld wordt aangeraden.

### 6.3.4 Stap 4: Implementatie GenericTool methodes

Op basis van de klassehierarchie dient na het invullen van de BuildingFC methodes de methodes van de interface GenericTool ingevuld te worden.

a) Implementatie methodes Get/Set Starttime

Met behulp van deze methodes kan de starttijd opgevraagd en gespecificeerd worden. Standaard wordt elke gespecificeerde tijd zonder controle ingesteld. Indien de eigen component wel controles vereist (omdat bijvoorbeeld alleen binnen een bepaald tijdsinterval gebruik gemaakt kan worden van de component) kan deze controle in de set-methode geïmplementeerd worden.

b) Implementatie methodes Get/Set Endtime



Met behulp van deze methodes kan de eindtijd opgevraagd en gespecificeerd worden. Standaard wordt elke gespecificeerde tijd zonder controle ingesteld. Indien de eigen component wel controles vereist (omdat bijvoorbeeld alleen binnen een bepaald tijdsinterval gebruik gemaakt kan worden van de component) kan deze controle in de set-methode geïmplementeerd worden.

c) Implementatie methode `GetModelComponentByID`

Bij een modelapplicatie kunnen aansluitpunten op basis van een ID opgevraagd worden. Bij een generieke tool is geen sprake van aansluitpunten van een schematisatie. Deze methode dient in de eigen klasse toch geïmplementeerd te worden. Het is toegestaan hier 'nil' als resultaat te geven. Voor een presentatiecomponent is het mogelijk een speciaal aansluitpunt te geven als resultaat: een `PresentationModelComponent`.

### 6.3.5 Stap 5: Implementatie `PresentationComponent` methodes

Indien een presentatiecomponent ontwikkeld wordt, wordt de eigen klasse niet direct afgeleid van `GenericTool`, maar van de subklasse van `GenericTool`, `PresentationComponent`. Deze subklasse maakt het mogelijk om attributen van een modelapplicatie aan- en af te melden (om bijvoorbeeld aan een grafiek toe te voegen). De onderstaande methodes moeten in dit geval ingevuld worden:

a) Implementatie methode `AddAttribute`

Met deze methode wordt een eigenschap van een aansluitpunt aangemeld aan een presentatiecomponent. Een grafiektool zal in dit geval een lijnserie aanmaken voor het attribuut.

b) Implementatie methode `RemoveAttribute`

Met deze methode wordt een aanmelding van een attribuut ongedaan gemaakt. In een grafiektool zal in dit geval de overeenkomstige lijnserie weer verwijderd worden.

c) Implementatie methode `GetAttributes`

Een complete lijst van alle aangemeld attributen aan de presentatiecomponent moet in deze methode als resultaat gegeven worden. Dit resultaat dient als `Iterator`-object gegeven te worden.

d) Implementatie methode `GetModelComponents`

Een presentatiecomponent heeft uiteraard geen modelschematisatie. Er wordt door dit type component echter een speciaal aansluitpunt gegeven, waaraan de aangemeld attributen gekoppeld worden: een `PresentationModelComponent`. Dit presentatie-aansluitpunt maakt het mogelijk dat met behulp van de SRW-Editor koppelingen worden gemaakt tussen een modelapplicatie en een presentatiecomponent. In de implementatie van deze methode dient de presentatie-modelcomponent als resultaat gegeven te worden (als `Singleton-iterator` object, zie `SRWLib`).



### 6.3.6 Stap 6: Testen

Het testen van een ontwikkelde generieke tool start uiteraard met het succesvol compileren van het Delphi project zonder 'hints' en 'warnings'. Het testen van de gecompileerde DLL bestaat uit:

a) Registratie DLL in raamwerk

De DLL moet zonder fouten of waarschuwingen geregistreerd kunnen worden in de raamwerkapplicatie.

b) Toevoegen van de modelapplicatie aan een case

De DLL moet zonder fouten of waarschuwingen aan de Composer van de ProcessManager toegevoegd kunnen worden.

c) Gebruik van de Property-editor

Indien een eigen editor is gemaakt, moet deze via de optie 'Component | Properties' en vervolgens 'Internal editor' gestart kunnen worden.

d) Uitvoeren van een simulatie

Als een controle van de case (check optie) geen foutmeldingen geeft moet de generieke tool over het gespecificeerde interval foutloos de functionaliteit uit kunnen voeren.

e) Geheugengebruik door de generieke tool

Geheugen dat tijdens een simulatie door een generiek tool wordt gealloceerd voor het laden van libraries of het bewaren van tussenresultaten dient in de finalize-methode weer vrijgegeven te worden. Het veelvuldig na elkaar laten uitvoeren van dezelfde case mag geen toename in het geheugengebruik door de applicatie veroorzaken.

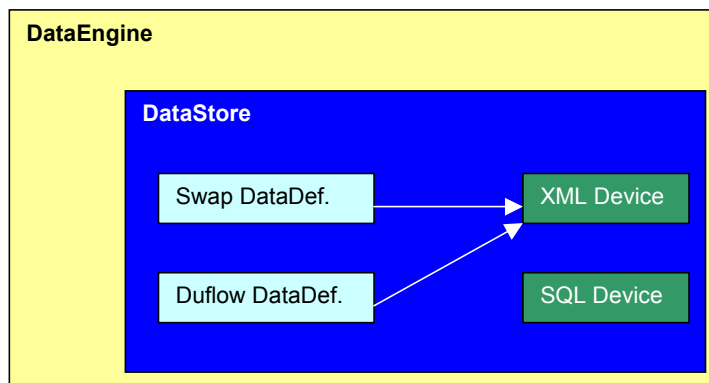
## 7 Gebruik van data definities

### 7.1 Inleiding

De DataEngine kan gebruikt worden voor het bewaren van (locaties van) data van raamwerkcomponenten. Het deel van de DataEngine dat hiervoor verantwoordelijk is, is de DataStore. Voor het bewaren van data heeft de DataStore informatie nodig over:

- Opslagmedium – Dit bepaalt waarin data geschreven wordt, bijvoorbeeld een bestand of relationele database. Binnen de SR architectuur wordt een opslagmedium aangeduid als Device. In versie 1 van SR is een XML Device beschikbaar. Het schrijven van data in XML bestanden wordt dus ondersteund.
- Opslagstructuur – Voordat in een Device geschreven kan worden moet bekend zijn wat hierin geschreven kan worden en op welke plaats. Deze beschrijving wordt binnen SR aangeduid als een DataDefinition. Deze DataDefinitions worden in XML beschreven.

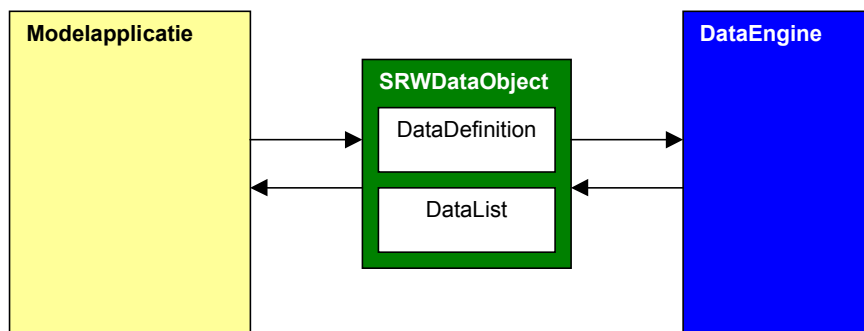
In onderstaande figuur is structuur van Devices en DataDefinitions binnen de DataStore weergegeven.



**Figuur 7.1 – Devices en DataDefinitions**

### 7.2 Lezen en schrijven van data

Het lezen en schrijven van gegevens van en naar de DataEngine wordt volgens een standaard procedure uitgevoerd. Bij het schrijven van data wordt een object 'gevuld' met gegevens en aan de DataEngine aangeboden, bij het lezen van data wordt een 'met data gevuld' object door de vrager ontvangen. Dit object is een instantie van de klasse SRWDataObject. Dit object bevat de te lezen of schrijven data inclusief de datadefinitie.



**Figuur 7.2 – Lezen en schrijven van data met een SRWDataObject**

### 7.2.1 DataDefinition

Een DataDefinition is uit een header- en definitiedeel opgebouwd. Onderstaande XML structuur toont de basis van elke DataDefinition.

```

<SRWXML>
  <HEADER>
    <NAMEID>unieke naam van de datadefinitie</NAMEID>
    <TDESCRIPTION>beschrijving </TDESCRIPTION >
    <VERSION>versie van de datadefinitie</VERSION>
    <DEVICE>devicenaam</DEVICE>
    <SCOPE>geldigheid van de datadefinitie</SCOPE>
    <OWNER>
      <COMPANY>bedrijfsnaam</COMPANY>
      <AUTHOR>verantwoordelijke</AUTHOR>
    </OWNER>
  </HEADER>
  <SRWDEFINITION>
    ...
  </SRWDEFINITION>
</SRWXML>
  
```

Binnen de SRWDEFINITION-tag wordt de structuur van de data beschreven. Deze structuur kan op twee manieren zijn gedefinieerd:

- Definitie als data-structuur

Een data-structuur, aangegeven als SRWSTRUCTURE kan enkelvoudige data bevatten, bijvoorbeeld een string, integer of floating point waarde. Om meerdere strings of getallen op te slaan kunnen wel meerdere SRWStructures in één DataDefinition worden opgenomen. Een SRWStructure is als volgt opgebouwd:

```

<SRWSTRUCTURE>
  <NAMEID>unieke naam</NAMEID>
  <TYPE>datatype (string, integer, double, datetime of
    boolean)</TYPE>
</SRWSTRUCTURE>
  
```





Optioneel kunnen de tags LENGTH en PRECISION toegevoegd worden. Voor bijvoorbeeld toepassing van een SQL-device kan dit van belang zijn.

- Definitie als data-lijst

Naast enkelvoudige waarden kunnen lijsten met waarden opgenomen worden. Hiervoor worden zogenaamde DataLists toegepast. Een DataList in een DataDefinition is een verwijzing naar een andere DataDefinition. In de DataDefinition waar deze verwijzing voor is kan een lijst van objecten opgeslagen worden. De onderstaande XML structuur wordt hiervoor toegepast.

```
<SRWLIST>
  <NAMEID>unieke naam van de datadefinitie waarnaar verwezen
  wordt</NAMEID>
</SRWLIST>
```

## 7.2.2 SRWDataObject

Uitwisseling van data met behulp van de DataEngine vindt plaats via een generiek object: een SRWDataObject. Om data op te kunnen halen moet een raamwerk-component een 'leeg' SRWDataObject aan de DataEngine aanbieden. De DataEngine 'vult' het object vervolgens met de gewenste gegevens. Bij het bewaren van gegevens verloopt deze constructie omgekeerd.

Bij het benaderen van de DataEngine door een raamwerkcomponent wordt dit nooit rechtstreeks gedaan. Een raamwerkcomponent heeft namelijk nooit een verwijzing naar de DataEngine. Een raamwerkcomponent vraagt daarom aan het raamwerk om een dienst, in dit geval de dienst 'DataDefinition'. De DataEngine zal voor deze dienst door het raamwerk gevonden worden, zodat de vrager van de dienst alsnog de DataEngine kan gebruiken. In onderstaand codefragment is het benaderen van de DataEngine uitgewerkt.

```
try
  // maak een ServiceDescriptor
  lServiceDescriptor := TserviceDescriptor.Create(snDataDefinition,
  stData, spNeed);
  try
    Zoek met de ServiceDescriptor de DataEngine
    lDataEngine :=
    TIDataEngine(Fframework.CreateServiceProvider(lServiceDescriptor
    ));
    if not (lDataEngine = nil) then
      begin
        // do something with the DataEngine
      end;
  finally
    lServiceDescriptor.Free;
  end;
except
```



```
Fframework.Notify(cRaiseError, cDoSomethingWhenFailed);
end;
```

Bij elke SRWDataObject waarmee gegevens van en naar de DataEngine gestuurd worden, dient aangegeven te worden welke DataDefinition gebruikt wordt. Een DataDefinition kan als volgt aangevraagd worden:

```
lDataDefinition := lDataEngine.GetDataDefinition(
  cNameIdOfDataDefinition );
```

Zodra de DataEngine en de DataDefinition bekend zijn voor een raamwerkcomponent kan een SRWDataObject gemaakt worden. Een SRWDataObject bezit de onderstaande interface:

- SetParent(aSRWDataObject: TISRWDataObject)
 

Zet een verwijzing naar een SRWDataObject waar dit object onderdeel van uitmaakt.
- GetParent()
 

Geeft een verwijzing naar het SRWDataObject waar dit object onderdeel van uitmaakt.
- GetCaseID()
 

Geeft de unieke identificatie van de case waarbinnen het SRWDataObject gebruikt wordt.
- GetDataDefinition()
 

Geeft de DataDefinition op basis waarvan het SRWDataObject gegevens uitwisselt.
- GetDataValues()
 

Geeft een verwijzing naar de verzameling DataValue-objecten.
- GetDataLists()
 

Geeft een verwijzing naar de verzameling DataListDefinition-objecten.
- GetChildInstance(aDataDefinition: TIDataDefinition)
 

Maakt een nieuw sub-SRWDataObject. Het object wordt zelf de parent van het nieuwe sub-object.
- ValueByName(aNameID: TNameID)
 

Geeft een verwijzing naar het DataValue-object dat hoort bij de naam aNameID.

Een SRWDataObject beheert twee lijsten:

- een lijst met enkelvoudige waarden (DataStructures)
- een lijst met samengestelde waarden (DataLists)

Een enkelvoudige waarde wordt bewaard in een DataValue-object. Dit object bevat de waarde en het type van de waarde (integer, string, boolean of double).

Voor samengestelde waarden worden DataLists toegepast. Een DataList bevat een verwijzing naar een andere DataDefinition. De verzameling van dergelijke DataLists worden in een SRWDataObject in een lijst bewaard, in deze lijst worden DataListDefinition-objecten gebruikt. Een DataListDefinition object bevat een nieuwe DataDefinition en een lijst van bijbehorende SRWData objecten.



### 7.2.3 Scope van een DataDefinition

Elke DataDefinition heeft een bepaalde scope. De volgende worden onderscheiden:

- SRW – De DataDefinition geldt voor elke case (bijvoorbeeld een verwijzing naar een executable binnen een modelapplicatie).
- Case – De DataDefinition geldt voor een bepaalde case (bijvoorbeeld de start- en eindtijd van een modelapplicatie).
- Object – De DataDefinition geldt voor één component binnen een bepaalde case (bijvoorbeeld de naam van een uitvoerbestand van een modelapplicatie).

## 7.3 Data-definities voor modelapplicaties

Een modelapplicatie heeft een DataDefinition nodig, zodat bij het inlezen van een eerder gemaakte case alle instellingen van de modelapplicatie opnieuw gebruikt kunnen worden. Voor de hand liggende instellingen zijn:

- Starttijd van het model binnen de case;
- Eindtijd van het model binnen de case;
- Rekentijdstap van het model;
- Naam van een invoerbestand;
- Naam van een uitvoerbestand;

Indien de bovenstaande gegevens in een DataDefinition opgenomen worden kan deze definitie er als volgt uitzien:

```
<SRWXML>
  <HEADER>
    <NAMEID>MODELAPPLICATIONINSTANCE</NAMEID>
    <TDESCRIPTION>ModelApplication properties</TDESCRIPTION>
    <VERSION>0.1</VERSION>
    <DEVICE>XML</DEVICE>
    <SCOPE>OBJECT</SCOPE>
    <OWNER>
      <COMPANY>SRW B.V.</COMPANY>
      <AUTHOR>SRW Employee</AUTHOR>
    </OWNER>
  </HEADER>
  <SRWDEFINITION>
    <SRWSTRUCTURE>
      <NAMEID>STARTTIME</NAMEID>
      <TYPE>Double</TYPE>
    </SRWSTRUCTURE>
  </SRWSTRUCTURE>
```



```

        <NAMEID>ENDTIME</NAMEID>
        <TYPE>Double</TYPE>
    </SRWSTRUCTURE>
    <SRWSTRUCTURE>
        <NAMEID>TIMESTEP</NAMEID>
        <TYPE>Double</TYPE>
    </SRWSTRUCTURE>
    <SRWSTRUCTURE>
        <NAMEID>INPUTFILE</NAMEID>
        <TYPE>String</TYPE>
    </SRWSTRUCTURE>
    <SRWSTRUCTURE>
        <NAMEID>OUTPUTFILE</NAMEID>
        <TYPE>String</TYPE>
    </SRWSTRUCTURE>
</SRWDEFINITION>
</SRWXML>

```

De modelapplicatie is er zelf verantwoordelijk voor dat deze gegevens bij opstarten ingelezen worden en bij beëindigen weer weggeschreven. Hiervoor bezit elke modelapplicatie de operaties Load() en Save(). Voor het bovenstaande voorbeeld kunnen deze methoden als volgt ingevuld worden:

```

Load:
// execute default implementation
Result := inherited Load(aSRWDataObject);
lDataDefinition := nil;

// find DataEngine
lServiceDescriptor :=
TServiceDescriptor.Create(snDataDefinition,stData,spProvide);
try
    lDataEngine :=
        TIDataEngine(FFramework.CreateServiceProvider(lServiceDescriptor));
    if lDataEngine <> nil then
        begin
            lDataDefinition :=
                DataEngine.getDataDefinition(cModelDataDefinitionNameID);
        end;
finally
    lServiceDescriptor.Free;
end; //finally
if lDataDefinition = nil then exit;
// load settings
lSRWDataObject := lSRWDataObject.Create(lDataDefinition, FCaseID);
lSRWDataObject.setObjectID(self.getObjectID);
try
    lSRWDataObject := lDataEngine.ReadSRWData(lSRWDataObject);
    if lSRWDataObject <> nil then
        begin
            lIterator := lSRWDataObject.GetDataValues;
            try
                while not lIterator.Done do
                    begin
                        lDataValue := TDataValue(lIterator.NextElement);

```



```
lNameID := lDataValue.getDataStructure.getNameId;
if lNameID = cStarttime then
begin
    FSettings.Starttime :=
        DoubleDataValue(lDataValue).GetValue;
    continue;
end;
if lNameID = cEndtime then
begin
    FSettings.Endtime := TDoubleDataValue(lDataValue).GetValue;
    continue;
end;
if lNameID = cTimestep then
begin
    FSettings.MinTimestep :=
        TDoubleDataValue(lDataValue).GetValue;
    continue;
end;
if lNameID = cInputFileName then
begin
    FSettings.InputFileName :=
        TStringDataValue(lDataValue).GetValue;
    continue;
end;
if lNameID = cOutputFileName then
begin
    FSettings.OutputFileName :=
        TStringDataValue(lDataValue).GetValue;
    continue;
end;
end; //while Iterator
finally
    lIterator.Free;
end; //finally
end; // <> nil
finally
    lSRWDataObject.Free;
end; //finally
Result := orSuccessful;
```

**Save:**

```
// default implementation
Result := inherited Save(aSRWDataObject);

// find DataEngine
lServiceDescriptor :=
  TServiceDescriptor.Create(snDataDefinition, stData, spProvide);
try
  lDataEngine :=
    TIDataEngine(FFramework.CreateServiceProvider(lServiceDescriptor
    ));
  lDataDefinition :=
    lDataEngine.GetDataDefinition(cSwapInstanceDataDefinitionNameID
    );
  if lDataDefinition = nil then exit;
finally
  lServiceDescriptor.Free;
end; //finally
if (lDataEngine = nil) then exit;

// Save settings
if (Result <> nil) then
begin
  RetrieveCaseID;
  lSRWDataObject := TSRWDataObject.Create(lDataDefinition, FCaseID);
  lSRWDataObject.SetObjectID(self.GetObjectID);
  lIterator := lSRWDataObject.GetDataValues;
  try
    while not lIterator.Done do
    begin
      lDataValue := TDataValue(lIterator.NextElement);
      lNameID := lDataValue.GetDataStructure.GetNameID;
      if lNameID = cStarttime then
      begin
        TDoubleDataValue(lDataValue).SetValue(FSettings.Starttime);
        continue;
      end;
      if lNameID = cEndtime then
      begin
        TDoubleDataValue(lDataValue).SetValue(FSettings.Endtime);
        continue;
      end;
      if lNameID = cTimestep then
      begin
        TDoubleDataValue(lDataValue).SetValue(FSettings.Timestep);
        continue;
      end;
      if lNameID = cInputFileName then
      begin
        TStringDataValue(lDataValue).SetValue(FSettings.InputFileNa
        me);
        continue;
      end;
      if lNameID = cOutputFileName then
      begin
        TStringDataValue(lDataValue).SetValue(FSettings.OutputFileN
        ame);
        continue;
      end;
    end; //while Iterator
  finally
    lIterator.Free;
  end; //finally
end;
```



```
lDataEngine.WriteSRWData(lSRWDataObject);  
lSRWDataObject.Free;  
end;
```

## 7.4 Data-definities voor generieke tools

Een generieke tool heeft een DataDefinition nodig, zodat bij het inlezen van een eerder gemaakte case alle instellingen van de tool opnieuw gebruikt kunnen worden. Voor de hand liggende instellingen zijn:

- Starttijd van de generieke tool binnen de case;
- Eindtijd van de generieke tool binnen de case;
- Bijvoorbeeld gebruikte kleuren in een grafiek of kaart.

Ook hierbij moet een DataDefinition in XML vorm gemaakt worden. Deze structuur is analoog aan een DataDefinition van een modelapplicatie.

De Load- en Save operaties van een generieke tool moeten ingevuld worden, zodat de DataDefinition daadwerkelijk toegepast wordt bij het inlezen en afsluiten van een case. De implementatie hiervan is identiek aan de Load- en Save operaties van een modelapplicatie.



## 8 Standaarden en naamconventies

### 8.1 Inleiding

Standaardisatie zorgt ervoor dat producten van verschillende ‘leveranciers’ uitgewisseld kunnen worden. Met het oog hierop is het Standaard Raamwerk Water ook ontwikkeld. De standaardisatie van interfaces van raamwerkcomponenten zorgt ervoor dat deze uitwisseling mogelijk is.

Naast standaardisatie van deze interfaces kunnen aanvullende afspraken er voor zorgen dat de structuur en werking van raamwerkcomponenten sneller duidelijk worden. De standaarden die tijdens implementatie van SR versie 1 gemaakt zijn hebben betrekking op XML en naamconventies binnen Delphi.

### 8.2 Gebruik van XML

Het gebruik van XML is beschreven in paragraaf 3.4. XML is uitermate geschikt voor het uitwisselen van gegevens en vooral het beschrijven van gegevens. De snelle toename van het gebruik van XML en de prominente plaats van XML binnen recente ontwikkelingen als .NET technologie onderschrijven de kracht van deze taal.

Toepassing van XML is voor componentenbouwers vooral van belang wanneer een DataDefinition gemaakt wordt. Deze DataDefinition beschrijft welke gegevens van een raamwerkcomponent bewaard worden voor toekomstig gebruik.

Binnen versie 1 van SR wordt de MSXML parser gebruikt (versie 2.5). Deze parser biedt voldoende mogelijkheden voor SR. Updates van deze parser worden onder een nieuwe naam uitgebracht (doordat het versienummer in de DLL-naam is opgenomen) zodat parallel gebruik van andere XML parsers mogelijk is.

### 8.3 Naamconventies

Voor het programmeren van raamwerkcomponenten moet minimaal de interface in Delphi geschreven worden. De opbouw van deze code is voor een groot deel identiek voor alle raamwerkcomponenten. Het is dus verstandig om gebruik te maken van deze code van andere raamwerkcomponenten. De leesbaarheid van code wordt vergroot door gebruik van een aantal naamconventies. De binnen SR toegepaste naamconventies zijn in onderstaande tabel weergegeven.





Table 1 – Naamconventies Delphi code SR

Code onderdeel	Prefix (case sensitive)	Voorbeeld
Constance	c	cFileName
Type (klasse)	T	TMyObject
Enumeratie (set)	T	TMyOption
Lokale variabele	l	lString
Unit variabele	u	uStatus
Globale variabele	g	glnifile
Argument	a	aOwner
Property	F	FMyObject
Read/Write methode	Get/Set	GetName/SetName
Form	f	fMain
Datamodule	d	dCustomers
Unit	u	uMyClass



## 9 Literatuur

[GAMMA, 1995]

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Re-usable Object-Oriented Software, Addison-Wesley, Reading, MA 1995

[SR-PVA]

Noort, J. (red.), 2000, Plan van Aanpak Standaard Raamwerk Specificatie en Bouw versie 1, in opdracht van STOWA, Utrecht, 15p.

[SR-A]

van der Wal, T. (red.), 1999, “Architectuur Standaard Raamwerk Water”, ISBN 90.5773.065.0, Stichting Toegepast Onderzoek Waterbeheer (STOWA), Utrecht, rapport 99.16, Rijkswaterstaat, Rijksinstituut voor Integraal Zoetwaterbeheer en Afvalwaterbehandeling (RIZA), Lelystad, rapport 99.063, Alterra, Wageningen, rapport 72, 116 p.

[SR-FO]

Tacke, J. (MX.Systems B.V.), Brinkman, R. (WL | Delft Hydraulics), Frieswijk, E. (EDS International BV), Levelt, D. (WL | Delft Hydraulics), Otjens, T. (Alterra), 2000, SRW2000 Standaard Raamwerk Water – versie 1.0 Functioneel Ontwerp, ISBN 90-5773-140-1, Stichting Toegepast Onderzoek Waterbeheer (STOWA), Utrecht, rapport 2001-28, Rijkswaterstaat, Rijksinstituut voor Integraal Zoetwaterbeheer en Afvalwaterbehandeling (RIZA), Lelystad, rapport 2001.038, MX.systems, Rijswijk, rapport P4118-R-1, 35p.

[SR-TO]

Tacke, J.H.P.M., (MX.Systems BV), Brinkman, R. (WL | Delft Hydraulics), Frieswijk, E. (EDS International BV), Levelt, D. (WL | Delft Hydraulics), Otjens, A.J. (WISL-Alterra), 2001, Standaard Raamwerk – versie 1.0, Technisch Ontwerp, ISBN 90-5773-142-8, Stichting Toegepast Onderzoek Waterbeheer (STOWA), Utrecht, werkdocument 2001-W-06, Rijkswaterstaat, Rijksinstituut voor Integraal Zoetwaterbeheer en Afvalwaterbehandeling (RIZA), Lelystad, werkdocument 2001.125X, 54 p.



## Bijlage A: Inhoud SRWLib

Onderdeel	Unit naam	Opgenomen klassen
SRW interface	uBasicFrameworkComponent	TBasisFrameworkComponent
	uBuildingFrameworkComponent	TBuildingFrameworkComponent
	uCase	TCase
	uCaseVisualisation	TCaseVisualisation
	uComponentDescriptor	TComponentDescriptor TFileObject
	uCondition	TCondition
	uConnection	TConnection
	uConverter	TConverter
	uDataStructure	TDataStructure TDataStructureList TDataValue TDataValueList Subklassen van TDataValue voor het definiëren van een DataDefinition (zie ook uIDataDefinition)
	uEmptyComponentReference	TEmptyComponentReference
	uFrameworkComponent	TFrameworkComponent TComponentReference
	uGenericTool	TGenericTool
	uGeometry	TGeometry TWKSPoint TWKSLineString
	uICaseManager	TICaseManager
	uIDataDefinition	TIDataDefinition TIDataDefinitionList
	uIFramework	TIFramework
	uIMessages	TISRWMessages
	uIProcessManager	TIProcessManager
	uISrwData	TISrwData
	uModelApplication	TModelApplication
	uModelApplicationInterface	TIModelApplication
	uModelAttribute	TModelAttribute TExchangeItem
	uModelComponent	TModelComponent
	uModelElement	TModelElement TConnector TCompositeConnector TSingleConnector TCompositeModelElement TSingleModelElement
	uObserver	TPublisher TInternalSubject TSubscriber TSubscription TSubscriptionList TSubjectList TInternalSubjectList TSubscriberList
	uPosition	TPosition

Onderdeel	Unit naam	Opgenomen klassen
	uPredicate	TPredicate
	uPresentationComponent	TPresentationComponent TPresentationModelComponent TPresentationModelAttribute
	uRootObject	TRootObject
	uServiceDescriptor	TServiceDescriptor
	uSrwData	TSRWDataObject TSRWDataList TSRWDataListDefinition
	uSrwEditInterface	TISRWEditor
	uUnitOfMeasurement	TUnitOfMeasurement TConversion TConversionCollection
Constanten en basistypes	uConsts	Alle gebruikte resource strings
	uDLLIntf	Standaard functies voor laden en vrijgeven van een SRW component in DLL vorm
	uMainMenuButton	Specifieke toolbutton, indien een raamwerkcomponent een menu-item toe wil voegen aan de raamwerkapplicatie
	uTypeDefinitions	Alle gebruikte typen en enumeraties
Collecties en iteratoren	uAggregationList	TAggregationList
	uArrayIterator	TArrayIterator
	uAssociationList	TAssociationList
	uCollection	TCollection
	uCombinedIterator	TCombinedIterator
	uEmptyIterator	TEmptyIterator
	uFastAggregationList	TFastAggregationList
	uFastAssociationList	TFastAssociationList
	uIndexedCollection	TIndexedCollection
	uIndexedCollectionIterator	TIndexedCollectionIterator
	uIterator	TIterator
	uManagedAggregationList	TManagedAggregationList
	uSingletonIterator	TSingletonIterator
	uTempAssociationList	TTempAssociationList
	uTempIndexedCollectionIterator	TTempIndexedCollectionIterator
XML en GML	MSXML_TLB	Typelibrary voor MSXML.DLL (XML parser)
	uEmptyXMLDocument	TEmptyXMLDocument
	uXMLDefinitions	TXMLDefinition TDescriptorDefinition TRepositoryDefinition
	uXMLDOM	TXMLNode TXMLDocument
Events	uAddBuildingFCEvent	TAddBuildingFCEvent
	uConfirmEvent	TConfirmEvent
	uEvent	TDistributedEvent TSubject
	uProgressEvent	TProgressEvent
	uStatusEvent	TStatusChangedEvent



## Bijlage B: Delphi instellingen

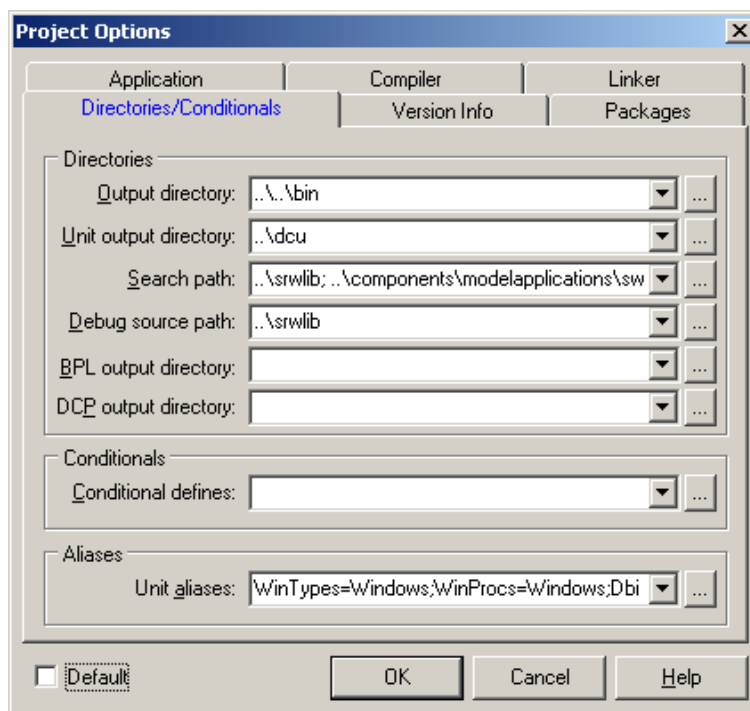
Bij het ontwikkelen van een modelapplicatie of generieke tool voor SR wordt een DLL ontwikkeld met behulp van de ontwikkelomgeving Delphi. Om een DLL te kunnen compileren wordt in Delphi een Project gemaakt. Binnen dit project wordt de sourcecode voor de DLL geïmplementeerd en gecompileerd. Voor de ontwikkeling van een modelapplicatie kan gebruik gemaakt worden van een standaardproject: DefaultModelapplication.

Bij elk Project in Delphi kunnen een aantal instellingen ingevoerd worden, die meer of minder invloed hebben op de werking en het gedrag van de uiteindelijk gecompileerde DLL. In de onderstaande schermafdrukken zijn de instellingen weergegeven die geadviseerd worden voor SR componenten. Er wordt hierbij onderscheid gemaakt in een Debug-versie van een DLL en een Release-versie van een DLL.

De Project-instellingen kunnen binnen Delphi opgevraagd worden met de menu-optie: Project – Options (of shift-ctrl-F11).

### Instellingen voor een Debug versie van een SR DLL

(voor niet weergegeven tabbladen geldt dat de default Delphi instellingen gebruikt worden)





**Project Options**

Directories/Conditionals    Version Info    Packages

Application    Compiler    Linker

**Code generation**

- ☐ Optimization
- ☒ Aligned record fields
- ☐ Stack frames
- ☐ Pentium-safe FDIV

**Syntax options**

- ☒ Strict var-strings
- ☐ Complete boolean eval
- ☒ Extended syntax
- ☐ Typed @ operator
- ☒ Open parameters
- ☒ Huge strings
- ☒ Assignable typed constants

**Runtime errors**

- ☐ Range checking
- ☒ I/O checking
- ☐ Overflow checking (Q)

**Debugging**

- ☒ Debug information
- ☒ Local symbols
- ☒ Reference info (Y)
- ☒ Definitions only
- ☒ Assertions (C)
- ☐ Use Debug DCUs

**Messages**

- ☒ Show hints
- ☒ Show warnings

☐ Default    OK    Cancel    Help

**Project Options**

Directories/Conditionals    Version Info    Packages

Application    Compiler    Linker

**Map file**

- ☒ Off
- ☐ Segments
- ☐ Publics
- ☐ Detailed

**Linker output**

- ☒ Generate DCUs
- ☐ Generate C object files
- ☐ Generate C++ object files
- ☐ Include namespaces
- ☐ Export all symbols

**EXE and DLL options**

- ☐ Generate console application
- ☐ Include ID32 debug info
- ☒ Include remote debug symbols

**Memory sizes**

Min stack size: \$00004000

Max stack size: \$00100000

Image base: \$00400000

**Description**

EXE Description:

☐ Default    OK    Cancel    Help



## Instellingen voor een Release versie van een SR DLL

(voor niet weergegeven tabbladen geldt dat de default Delphi instellingen gebruikt worden)

The 'Project Options' dialog box is shown with the 'Directories/Conditionals' tab selected. The 'Directories' section contains the following fields:

- Output directory: ..\bin
- Unit output directory: ..\dcu
- Search path: ..\srwlib; ..\components\modelapplications\sw
- Debug source path: ..\srwlib
- BPL output directory: (empty)
- DCP output directory: (empty)

The 'Conditionals' section contains:

- Conditional defines: (empty)

The 'Aliases' section contains:

- Unit aliases: WinTypes=Windows;WinProcs=Windows;Dbi

At the bottom, there is a 'Default' checkbox (unchecked) and buttons for 'OK', 'Cancel', and 'Help'.

The 'Project Options' dialog box is shown with the 'Version Info' tab selected. The 'Code generation' section contains the following options:

- ☐ Optimization
- ☒ Aligned record fields
- ☐ Stack frames
- ☐ Pentium-safe FDIV

The 'Syntax options' section contains the following options:

- ☒ Strict var-strings
- ☐ Complete boolean eval
- ☒ Extended syntax
- ☐ Typed @ operator
- ☒ Open parameters
- ☒ Huge strings
- ☒ Assignable typed constants

The 'Runtime errors' section contains the following options:

- ☐ Range checking
- ☒ I/O checking
- ☐ Overflow checking (Q)

The 'Debugging' section contains the following options:

- ☐ Debug information
- ☒ Local symbols
- ☒ Reference info (Y)
- ☒ Definitions only
- ☒ Assertions (C)
- ☐ Use Debug DCUs

The 'Messages' section contains the following options:

- ☒ Show hints
- ☒ Show warnings

At the bottom, there is a 'Default' checkbox (unchecked) and buttons for 'OK', 'Cancel', and 'Help'.



**Project Options** [X]

Directories/Conditionals	Version Info	Packages
Application	Compiler	Linker

**Map file**

☒ Off  
☐ Segments  
☐ Publics  
☐ Detailed

**Linker output**

☒ Generate DCUs  
☐ Generate C object files  
☐ Generate C++ object files  
☐ Include namespaces  
☐ Export all symbols

**EXE and DLL options**

☐ Generate console application  
☐ Include ID32 debug info  
☐ Include remote debug symbols

**Memory sizes**

Min stack size: \$00004000  
 Max stacksize: \$00100000  
 Image base: \$00400000

**Description**

EXE Description:

☐ Default



## Bijlage C: Veelgestelde vragen

- Kan ik een raamwerkcomponent ontwikkelen in een andere taal dan Delphi?

*Een raamwerkcomponent dient als Dynamic Link Library (DLL) beschikbaar gesteld te worden. In principe is een DLL taalafhankelijk. Er zijn wel zogenaamde 'calling conventions' voor functies die een DLL beschikbaar stelt. Deze calling conventions kunnen wel taalspecifiek zijn. Een calling convention legt bijvoorbeeld vast hoe argumenten van een functie gelezen moeten worden. Binnen SR is gekozen voor de calling convention 'STDCALL'.*

*Specifiek voor SR raamwerkcomponenten geldt dat voor één van de functies die de DLL biedt (de interface van een SR DLL bestaat slechts uit 2 functies) een object als resultaat gegeven wordt. Het raamwerk verwacht een object van een bepaald type, en moet hierop kunnen controleren. Omdat het raamwerk in Delphi is geprogrammeerd herkent het raamwerk alleen Delphi objecttypen. Dit is de reden dat de DLL in Delphi gecompileerd moet zijn.*

*Het is mogelijk om binnen de DLL gebruik te maken van een andere DLL of executable, die in een andere taal is ontwikkeld. Zo kan een rekenhart van een modelapplicatie in Fortran of C ontwikkeld worden, alleen de interface naar het raamwerk is dan in Delphi geprogrammeerd.*

- Ondersteund SR het gebruik van gedistribueerde modellen?

*Omdat een raamwerkcomponent als DLL beschikbaar moet worden gesteld, moeten raamwerkcomponenten op dezelfde machine geplaatst worden als waar de raamwerkapplicatie geïnstalleerd is. De DLL kan echter zelf via een zelf te kiezen communicatieprotocol verbinding maken met bijvoorbeeld een rekenkern op een andere locatie. Op deze manier kan op een andere locatie (zoals een server) zwaar rekenwerk uitgevoerd worden. Het raamwerk heeft in dit geval alleen contact met de locale DLL, deze DLL fungeert daarom als een zogenaamde 'Proxy'.*

- Waaraan moet mijn model voldoen om het te kunnen migreren naar een SR modelapplicatie?

*SR stelt zeer weinig eisen aan een model. Doordat elk model 'gewrapped' wordt in een SR DLL kunnen vereiste vertaalslagen binnen deze DLL uitgevoerd worden. Een voorbeeld hiervan is de beschikbaarstelling van de aansluitpunten van een modelschematisatie. Een model kan met een eigen modelschematisatie werken, in de 'wrapper' wordt dan een vertaalslag gemaakt naar SR aansluitpunten, de zogenaamde 'ModelComponents'.*

*Een vereiste aanpassing aan een model zelf zit vaak in de tijdscontrole die in bestaande modelapplicaties is opgenomen. Tijdsbesturing wordt door SR verzorgd (de ProcessManager), SR zal dus rekenopdrachten*



*met een bepaald tijdsinterval moeten kunnen geven. Na elke rekentijdstap (bv een dag) moet SR resultaten op kunnen vragen van eigenschappen van de aansluitpunten van dat model.*

*Het gebruik van een model kan verder gefaciliteerd worden door het ontwikkelen van een zogenaamde 'Property Editor'. Hierdoor kunnen modelspecifieke instellingen via SR ingesteld worden.*

- Wat moet ik doen om mijn model aan te passen aan de SR specificaties?

*Voor het aanpassen van een bestaand model aan de SR-specificaties bestaat een stappenplan, wat is uitgewerkt in hoofdstuk 5 van dit document. Globaal bestaat de ontwikkeling uit:*

- 1. Invullen standaard DLL functies en compileren tot DLL;*
- 2. Invullen specifieke implementaties van de operaties van de interfaces FrameworkComponent, BuildingFrameworkComponent en ModelApplication;*
- 3. Koppeling met bestaande model implementeren.*

- Moet ik mijn specifieke formaat van in- en uitvoerbestanden aanpassen voordat ik mijn model binnen SR kan gebruiken?

*Binnen SR zijn geen standaarden voor opslag van data gedefinieerd. Het is mogelijk een standaard als Adventus toe te passen, een eigen formaat is eveneens toepasbaar. Om ondanks de verscheidenheid aan dataformaten toch een generieke DataEngine (verantwoordelijk voor opslag (tussen)resultaten) toe te kunnen passen zijn echter wel afspraken nodig. De oplossing hiervoor is gezocht in het beschrijven van data-formaten op een standaard manier, in plaats van het beschrijven van een standaard data-formaat. Dit betekent dat elk formaat gehanteerd kan worden, als het maar op de 'SR-manier' beschreven is. Hier speelt de taal eXtensible Markup Language (XML) een belangrijke rol. Met behulp van deze taal kan een dataformaat beschreven worden. De structuur hiervoor is in hoofdstuk 7 van dit document beschreven.*

- Waar kan ik mijn wensen ten aanzien van een nieuwe SR versie kenbaar maken?

*Op de website <http://www.mx-groep.nl/cgi-bin/SR/YaBB.pl> kunnen wensen kenbaar gemaakt worden voor een volgende versie van het Standaard Raamwerk.*