



SRW2000

**Standaard Raamwerk Water -
versie 1.0**

Technisch Ontwerp versie 1.1

Blanco pagina.



SRW2000

Standaard Raamwerk Water - versie 1.0

Technisch Ontwerp versie 1.1

Kernteam SRW 2000

In opdracht van STOWA, ALTErrA, 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, Technisch Ontwerp, ISBN 90-5773-142-8, Stichting Toegepast Onderzoek Waterbeheer (STOWA), Utrecht, werkdokument 2001-W-06, Rijkswaterstaat, Rijksinstituut voor Integraal Zoetwaterbeheer en Afvalwaterbehandeling (RIZA), Lelystad, werkdokument 2001.125X, MX.Systems BV document P4118-R-2, 54 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	Septra (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
	Edwin Frieswijk	EDS International BV
	David Levelt	WL Delft Hydraulics
	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.

Blanco pagina.

INHOUD

Voorwoord	I
1 Inleiding	1
1.1 Achtergrond en uitgangspunten	1
1.2 Leeswijzer	1
2 Globale architectuur	3
2.1 Inleiding	3
2.2 Raamwerk en raamwerkcomponenten	3
3 Raamwerk	7
3.1 Inleiding	7
3.2 Interface	7
4 Basic Frameworkcomponents	12
4.1 Inleiding	12
4.2 Procesketen beheertool	12
4.3 DataEngine	18
4.4 Converter	21
4.5 SRW-Editor	22
4.6 Attribuut editor	25
5 Building Frameworkcomponents	26
5.1 Inleiding	26
5.2 Modelapplicaties	26
5.2.1 Interface	27
5.2.2 Schematisaties van aansluitpunten	27
5.2.3 Geometrie	30
5.3 Generieke tools	31
5.3.1 Algemeen	31
5.3.2 Presentatietool	32
6 Dynamiek	33
6.1 Inleiding	33
6.2 Registratie van componenten	33
6.2.1 Toevoegen van raamwerkcomponenten	33
6.2.2 Verwijderen van raamwerkcomponenten	34
6.3 Samenstellen van een case	35
6.4 Uitvoeren van simulaties	41
6.4.1 Controleren case	41
6.4.2 Bepalen rekenvolgorde	41
6.4.3 Uitvoering simulatie	44
7 Infrastructuur	46
7.1 Inleiding	46
7.2 Communicatiestandaard en distributie	46
7.2.1 Het gedistribueerde object model	46
7.2.2 Ondersteunende technologieën: RMI, CORBA and DCOM	47
7.2.3 Vergelijking tussen de drie gedistribueerde object technologieën	48
7.2.4 Vergelijkingscriteria	48
8 Conclusies	49
9 Literatuur	51
Bijlage A: Gedistribueerd rekenen	52

Blanco pagina.



1 Inleiding

1.1 Achtergrond en uitgangspunten

Dit document maakt onderdeel uit van de rapportage behorend bij de definitiefase van het project 'Standaard Raamwerk Water – Specificatie en Realisatie versie 1'. Deze fase bestaat uit de volgende onderdelen:

- Onderzoek modellen;
- Beschrijving case(s);
- Functioneel ontwerp;
- Technisch ontwerp;
- Richtlijnen voor SR- componenten;
- Werkplan fase 2.

In dit document is het technisch ontwerp van het Standaard Raamwerk (SR¹) uitgewerkt. Er wordt beschreven hoe de in het functioneel ontwerp vastgelegde functionaliteit wordt gerealiseerd door het raamwerk en de componenten binnen dit raamwerk. Samen met het functioneel ontwerp en de richtlijnen voor componentenbouwers biedt dit rapport de vereiste specificaties voor realisatie van SR versie 1 en de daarbij gedefinieerde modelapplicaties en tools.

De basis van dit technisch ontwerp wordt gevormd door de Architectuur Standaard Raamwerk [SR-A] en de DelftWISE specificaties [Gijsbers *et al*]. Deze architecturen zijn samengevoegd en aangepast tot het fundament van dit technisch ontwerp. Waar afgeweken wordt van ontwerpbeslissingen uit de Architectuur SR wordt dit expliciet vermeld en verklaard.

Het raamwerk en de raamwerkcomponenten worden beschreven met behulp van een aantal standaard notatiewijzen. Voor de beschrijving van interfaces van componenten is gebruik gemaakt van de Interface Definition Language (IDL). De diagrammen die de werking van en de samenhang tussen componenten weergeven zijn op basis van de Unified Modeling Language (UML) gemaakt. Er wordt vanuit gegaan dat de lezer bekend is met deze notatiestandaarden.

1.2 Leeswijzer

In hoofdstuk 2 wordt de globale architectuur, ontwikkeld tijdens de uitvoering van het functioneel ontwerp, behandeld. Dit biedt het vereiste kader om de belangrijkste onderdelen van het raamwerk en bijbehorende componenten te kunnen plaatsen. De interface en interne structuur van het raamwerk wordt in hoofdstuk 3 beschreven.

In hoofdstuk 4 worden de zogenaamde BasicFrameworkcomponents beschreven. In de eerste versie van het raamwerk zijn de procesketen beheertool en de DataEngine de belangrijkste specialisaties van dit type component.

¹ 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".



In hoofdstuk 5 worden de BuildingFrameworkcomponents behandeld qua interface en opbouw. Zowel modelapplicaties als generieke tools behoren tot dit type component.

Naast de interfaces en interne structuur van componenten is veel aandacht besteed aan de beschrijving van de samenwerking tussen componenten. In hoofdstuk 6 wordt deze dynamiek uitgewerkt.

Voordat tot implementatie van het raamwerk en de raamwerkcomponenten kan worden overgegaan dienen een aantal technische keuzes gemaakt te worden. Veel belangrijker dan het kiezen van een programmeertaal is de keuze voor een communicatiestandaard tussen componenten. De standaarden op dit gebied met bijbehorende voor- en nadelen worden in hoofdstuk 7 beschreven. Naast een inventarisatie van voor- en nadelen wordt volstaan met een overzicht van keuzecriteria voor SR. In deze fase is dus nog geen expliciete keuze gemaakt voor een bepaalde communicatiestandaard. Dit hoofdstuk faciliteert slechts het maken van deze keuze.

De belangrijkste ontwerpbeslissingen worden in de conclusies van dit rapport weergegeven. Hoofdstuk 8 bevat deze conclusies.



2 Globale architectuur

2.1 Inleiding

Onder een architectuur van een softwaresysteem wordt verstaan [SR-A]:

Een beschrijving van een systeem in abstractie, uitgedrukt in de onderdelen en de samenhang ertussen.

Het ontwerp van het Standaard Raamwerk Water is een component gebaseerd ontwerp. De architectuurbeschrijving bestaat dus uit de definitie van componenten en de beschrijving van de samenhang tussen deze componenten. Onder een component wordt verstaan [SR-A]:

Een samenhangend pakket van herbruikbare software dat onafhankelijk als eenheid kan worden ontwikkeld en geleverd en dat ongewijzigd kan worden samengevoegd met andere componenten om een groter geheel te maken.

De globale architectuur wordt gevormd door een combinatie van de Architectuur Standaard Raamwerk Water en de DelftWISE specificaties. In het functioneel ontwerp [SR-FO] is het resultaat van deze combinatie reeds beschreven.

Dit hoofdstuk heeft als doel om de belangrijkste begrippen (dus de belangrijkste componenten) met betrekking tot het Standaard Raamwerk te kunnen plaatsen. Dit maakt het mogelijk om in de architectuurbeschrijving in de volgende hoofdstukken te refereren naar bepaalde componenten die pas in een later hoofdstuk in detail worden beschreven.

2.2 Raamwerk en raamwerkcomponenten

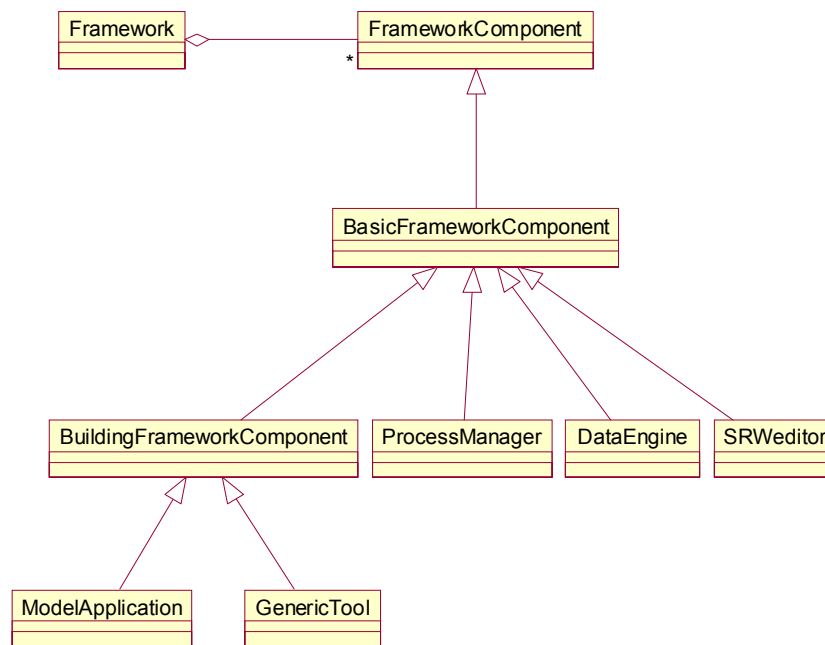
In het functioneel ontwerp is het raamwerk aangeduid als een container voor raamwerkcomponenten. Hiermee wordt aangegeven dat het raamwerk zelf een zeer beperkte functionaliteit bezit. Het raamwerk bezit als functionaliteit alleen het beheren van geregistreerde raamwerkcomponenten. Het raamwerk heeft wat dat betreft overeenkomsten met een besturingssysteem. Een gebruiker maakt nauwelijks direct gebruik van functionaliteiten van bijvoorbeeld Windows. Een PC biedt voor een gebruiker de belangrijkste functionaliteit door middel applicaties die geïnstalleerd zijn op dit besturingssysteem. Analooq hieraan zal een gebruiker van SR voornamelijk gebruik maken van functionaliteiten van geregistreerde raamwerkcomponenten.

Er worden twee soorten raamwerkcomponenten onderscheiden binnen SR:

- BasicFrameworkComponents (BasicFC's) – Raamwerk ondersteunende componenten voor het samenstellen en uitvoeren van cases. Voor de eerste versie van SR worden onder andere de procesketen beheertool en de DataEngine als specialisaties van BasicFC onderscheiden. Een procesketen beheertool (PKBT) maakt het mogelijk om een case samen te stellen en te kunnen starten. De DataEngine biedt functionaliteiten voor het opslaan en ontsluiten van (al dan niet persistente) gegevens (of referenties naar gegevens).
- BuildingFrameworkComponents (BuildingFC's) – De bouwstenen voor een case, dus de onderdelen waaruit een modellen/tools configuratie is opgebouwd. Een verdere specialisatie van BuildingFC's is gemaakt naar modelapplicaties en generieke tools.

Deze specialisatie van raamwerkcomponenten is een modificatie van de architectuurbeschrijving in [SR-A]. Door deze modificatie is het mogelijk specifiek gedrag toe te kennen aan de raamwerkcomponenten die in een case kunnen worden opgenomen.

In het onderstaande klassediagram (Figuur 2-1) wordt de relatie tussen het raamwerk en de specifieke raamwerkcomponenten weergegeven.



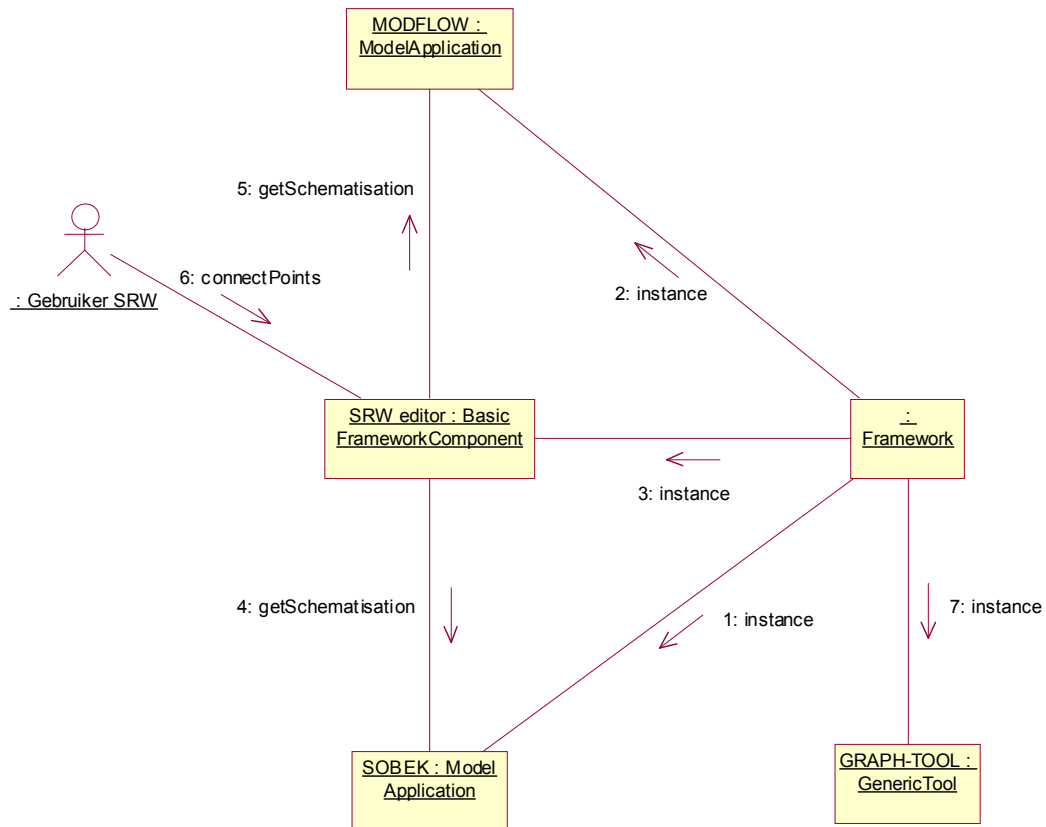
Figuur 2-1: Klassediagram raamwerk en raamwerkcomponenten

De te realiseren componenten, beschreven in het projectplan [SR-P], kunnen binnen deze structuur geplaatst worden:

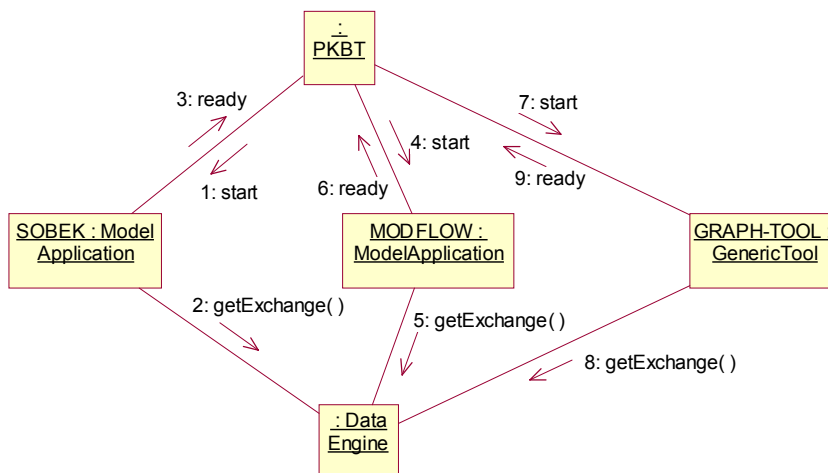
- Raamwerk en DataEngine – Het raamwerk is een instantie van de component Framework. Dit component kan voorgesteld worden als een zelfstandig draaiende applicatie. De DataEngine is niet expliciet in het projectplan opgenomen. Binnen deze architectuur maakt het echter geen integraal onderdeel uit van het raamwerk en wordt dus als zelfstandig component onderscheiden. De DataEngine is een te registreren BasicFrameworkcomponent.
- PKBT – De PKBT is een BasicFrameworkcomponent. Zonder PKBT kunnen geen cases worden gemaakt, gevisualiseerd of gestart. Door de fysieke scheiding van het raamwerk is het technisch mogelijk een PKBT te vervangen in het raamwerk.
- Rekenkernen Sobek, DufLOW, Modflow en SWAP – De rekenkernen waarmee cases kunnen worden gemaakt in versie 1 van SR zijn specialisaties van ModelApplication. Elke rekenkern dient dus de interface van FrameworkComponent, BasicFC, BuildingFC en ModelApplication te bezitten.
- SRW-Editor – De editor voor het koppelen van schematisaties is een BasicFC. Deze editor wordt gebruikt om een case te definiëren, het kan echter geen onderdeel van een case uitmaken.

- Grafiekentool – De grafiekentool is een specialisatie van GenericTool. Deze tool kan opgenomen worden in een case om attribuuwaarden te visualiseren.

In de onderstaande collaboration diagrams wordt de samenwerking tussen deze componenten tijdens het samenstellen en uitvoeren van een case weergegeven:



Figuur 2-2: Collaboration diagram samenstellen case



Figuur 2-3: Collaboration diagram simulatie met behulp van SR



In het hoofdstuk 'Dynamiek' wordt de samenwerking tussen componenten in meer detail uitgewerkt.



3 Raamwerk

3.1 Inleiding

Het raamwerk faciliteert het registreren van raamwerkcomponenten, het instantiëren van raamwerkcomponenten en het opvangen en doorsturen van berichten tussen componenten. Het raamwerk heeft dus een voornamelijk beherende rol in een SR applicatie.

3.2 Interface

De in het functioneel ontwerp beschreven diensten van het raamwerk worden door middel van de onderstaande interface aangeboden:

```

Interface Framework {
    RegisterResult register(in ComponentDescriptor cd, in RegistrationMethod rm);
        /* registreert de raamwerkcomponent met ComponentDescriptor cd volgens
        registratiemethode rm. */
    RegisterResult check(in ComponentDescriptor cd);
        /* controleert of de raamwerkcomponent met ComponentDescriptor cd voldoet aan de SR
        specificaties en of dit component reeds geïnstalleerd is. */
    RegisterResult unregister(in ComponentReference cr);
        /* verwijdert de raamwerkcomponent met ComponentReference cr uit de
        ComponentRepository. De ComponentReference cr wordt in het meest typische geval door de
        gebruiker geselecteerd in de (grafische weergave van de) ComponentRepository. */
    RegisterResult unregisterAll();
        /* verwijdert referenties naar alle raamwerkcomponenten uit het raamwerk. */
    RegisterResult unregisterAllBuildingFrameworkComponents();
        /* verwijdert referenties naar alle BuildingFrameworkComponents uit het raamwerk. */

    Sequence<ComponentReference> getRegisteredComponents();
        /* geeft een lijst van alle geregistreerde raamwerkcomponent-referenties. */
    Sequence<ComponentReference> getRegisteredBuildingFrameworkComponents();
        /* geeft een lijst van alle geregistreerde BuildingFrameworkcomponent-referenties. */
    Sequence<ComponentReference> getRegisteredBasicFrameworkComponents();
        /* geeft een lijst van alle geregistreerde BasicFrameworkComponent-referenties. */
    Sequence<ComponentReference> getFrameworkComponents(in Predicate p);
        // geeft de verzameling ComponentReferences die voldoen aan predicate p.

    OperationResult addNewComponent()
        /* deze methode start een wizard die de gebruiker assisteert bij de installatie van een
        raamwerkcomponent. */
    Boolean instantiateFrameworkComponent(in ComponentDescriptor cd);
        /* deze methode maakt een instantie van een raamwerkcomponent, gegeven een
        ComponentDescriptor cd. Het boolean-resultaat geeft aan of de instantiatie succesvol is. */
    ComponentReference findFrameworkComponent(in ServiceDescriptor sd);
        /* deze methode geeft de eerstgevonden ComponentReference die hoort bij een Framework-
        Component die de service sd aanbiedt. */
    ComponentReference findComponentReference(in Description Name, in Double Version);
        /* deze methode geeft de ComponentReference als resultaat die dezelfde naam en
        versienummer heeft als in de argumentenlijst wordt doorgegeven. */
    FrameworkComponent createServiceProvider(in ServiceDescriptor sd);
        /* deze methode maakt een instantie van een raamwerkcomponent, gegeven een
        ServiceDescriptor sd. */
    Boolean FrameworkComponentCreated(in ComponentReference cr);
        /* deze methode controleert of een instantie van de raamwerkcomponent met
        ComponentReference cr is gemaakt. */
    Boolean serviceProviderCreated(in ServiceDescriptor sd);
        /* deze methode controleert of een instantie van een raamwerkcomponent beschikbaar is die
        de service sd aanbiedt. */

    OperationResult notify(in DistributedEvent de);
        /* deze methode stelt de broker op de hoogte van event (gebeurtenis) de. */
    OperationResult subscribe(in Subscriber sc, in Subject sj, in DistributedEventProcedure ep);
        /* deze methode abonneert Subscriber sc op events met subjeet si. De subscriber geeft een

```



```

        verwijzing naar de procedure die door de publisher aangeroepen moet worden indien een
        event met subject sj wordt ontvangen. */
    OperationResult unsubscribe(in Subscriber sc);
    /* deze methode verwijdert subscriber sc van de lijst van 'abonnees' van alle typen events. */
    OperationResult unsubscribe(in Subscriber sc, in Subject sj);
    /* deze methode verwijdert subscriber sc van de lijst van 'abonnees' van events met het
    subject sj. */
};

enum RegisterResult {
    rrOK, // registratie/verwijdering succesvol
    rrComponentNotAllowed, // de componentbeschrijving voldoet niet aan SR specificaties
    rrComponentAlreadyRegistered, // de component-referentie is reeds geregistreerd
    rrComponentNotRemoved // de component-referentie is niet verwijderd
};

interface Predicate {
    OperationResult evaluate(in FrameworkComponent f);
    /* evalueert de raamwerkcomponent f aan de hand van de voorwaarde die in het predikaat is
    opgenomen. */
};

enum RegistrationMethod {
    rmReference, // er wordt een referentie naar een component gemaakt
    rmCopyAndReference // er wordt een kopie en vervolgens een referentie gemaakt
};

interface ComponentReference {
    Path getLocation();
    /* geeft de locatie (drive + directory) van de component waarnaar wordt gerefereerd. */
    FrameworkComponent instance();
    /* maakt een instantie van de FrameworkComponent waarnaar wordt gerefereerd. */
    ComponentDescriptor GetComponentDescription();
    /* geeft een beschrijving van de component waarnaar wordt gerefereerd. */
};

typedef string Description;
typedef long Time;
typedef string Path;
typedef string FileName;

struct ComponentDescriptor {
    Description name; // de naam van de component
    Double version; // het versienummer van de component2
    Double size; // de omvang van een component in Kb
    Owner creator; // naam van de maker/beheerder/contactpersoon
    Time date; // datum/tijd van compileren van de component
    ComponentType type; // het type component: Basic of Building
    AccesType acces; // het type toegang: normaal of read-only
    Path location; // locatie van de component
    FileName icon; // bestandsnaam van het icoon van de component
};

struct Owner {
    Description company;
    Description author;
};

enum accesType {
    atNormal, // component kan gekopieerd worden naar een willekeurige locatie
    atReadOnly // er kan alleen een referentie naar de component gemaakt worden
};

Interface FrameworkComponent {
    ComponentDescriptor GetComponentDescription();
    /* geeft de beschrijving van de component. */
    ComponentType GetComponentType();
};

```

² De combinatie 'naam' en 'versie' van een component is altijd uniek.



```

        /* geeft aan of het een Basic- of BuildingFrameworkComponent betreft. */
        Sequence<ServiceDescriptor> getProvidedServices();
        /* geeft een lijst van de diensten die de component aanbiedt. */
        Sequence<ServiceDescriptor> getRequiredServices();
        /* geeft een lijst van de diensten die de component nodig heeft. */
    };

    enum ComponentType {
        ctBasic,
        ctBuilding
    };

    struct ServiceDescriptor {
        Description getName();
        ServiceType getType();
    };

    enum ServiceType {
        stData,
        stEvent
    };

    enum OperationResult {
        opSuccessful,
        opFailed
    };

    Interface DistributedEvent {
        Time getTime();
        // geeft het tijdstip (simulatietijd) waarop het event gegenereerd is.
        Subject getSubject();
        // geeft het type van het event.
        Variant getValue();
        // geeft de inhoud van het event als resultaat (kan tekst, getal, boolean of object zijn)
    };

    Interface Subscriber {
    };

    Interface Subject {
        Description getName();
        // geeft de naam van het subject.
        Object getOwner();
        // geeft de eigenaar van het subject.
    };

```

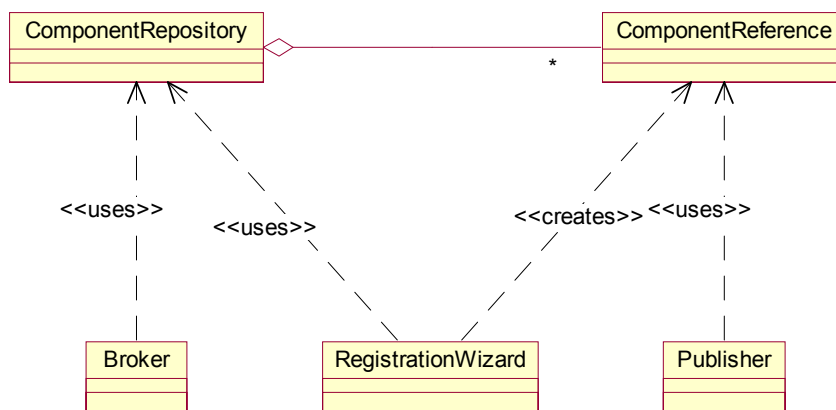
Het raamwerk bestaat intern uit een aantal nauw samenwerkende objecten:

- **RegistrationWizard** – Vanuit het hoofdscherm van SR kan een wizard worden gestart om raamwerkcomponenten toe te kunnen voegen. De wizard ondersteunt het localiseren van een component (bladeren of browsen), het opvragen van meta-informatie van een component, het kiezen van een installatietype (alleen een referentie maken of zowel een kopie als een referentie maken) en het uitvoeren van de daadwerkelijke registratie.
- **ComponentRepository** – De ComponentRepository is een archief voor referenties naar geregistreerde raamwerkcomponenten. Door de RegistrationWizard kan deze lijst met referenties worden aangevuld. Indien een referentie naar een component verwijderd moet worden biedt de ComponentRepository hiervoor een operatie.
- **Broker** – Het leveren van instanties van raamwerkcomponenten wordt door de Broker gerealiseerd. Dit betreffen zowel de instanties van BuildingFC's die door de gebruiker worden geselecteerd als voor BasicFC's die tijdens het samenstellen of uitvoeren van een case vereist zijn. Voorbeelden hiervan zijn de SRW-Editor en een attribuut

editor. De Broker maakt hierbij gebruik van de ComponentRepository van het raamwerk zelf.

- *Publisher* – Alle BuildingFC's binnen een case zullen tijdens de simulatie kenbaar maken wanneer een statusovergang plaatsvindt, bijvoorbeeld wanneer de gevraagde rekentijd is bereikt. BuildingFC's sturen hiervoor events het raamwerk; berichten met informatie over de huidige toestand. Deze events worden door de Publisher opgevangen. Dit object weet hoe deze events geïnterpreteerd moeten worden en wie van deze events op de hoogte gesteld moeten worden. Dit mechanisme is afgeleid van het Publish-Subscribe ontwerppatroon (ook wel Observer pattern genaamd). De Publisher kan zich als Observer/EventListener ('luisteren' naar events) gedragen, en zich daarnaast als Observable/EventSource ('generator' van events) gedragen. Deze structuur is overgenomen uit de Architectuur SR [SR-A].

De structuur van deze samenwerkende objecten wordt in het onderstaande klassediagram weergegeven (Figuur 3-1).



Figuur 3-1: Klassediagram Raamwerk

De objecten waaruit het raamwerk als component is opgebouwd bieden de onderstaande interfaces:

```

Interface RegistrationWizard {
    OperationResult start();
    // start de wizard.
    ComponentDescriptor readComponentDescription(in URL u);
    // leest de ComponentDescriptor van een te registreren component op de locatie met URL u.
};

typedef string URL;

Interface ComponentRepository {
    Sequence<ComponentReference> getComponentReferences();
    /* geeft een lijst van alle geregistreerde BuildingFrameworkComponents. */
    RegisterResult addComponentReference(in ComponentReference cr);
    /* voegt de ComponentReference cr toe aan de repository. */
    RegisterResult removeComponentReference(in ComponentReference cr);
    /* verwijdert de ComponentReference cr uit de repository. */
};

Interface Broker {
    ComponentReference findFrameworkComponent(in ServiceDescriptor sd);
    /* zoekt een geregistreerd raamwerkcomponent dat de aewenste dienst in ServiceDescriptor
  
```



```
sd aanbiedt. */
OperationResult createServiceProvider(in ComponentReference cr);
/* creëert een raamwerkcomponent dat ComponentReference cr bezit. ComponentReference cr
kan het resultaat zijn van methode findFrameworkComponent (maar kan ook van buiten de
Broker opgegeven worden). */

};

Interface Publisher {
    OperationResult notify(in DistributedEvent de);
    /* (methode behoort tot het EventListener-interface)
    mbv deze methode wordt de Publisher op de hoogte gebracht van events, gegenereerd door
    bijvoorbeeld BuildingFC's wanneer een statusovergang heeft plaatsgevonden. */
    OperationResult subscribe(in Subscriber sc, in Subject sj);
    // de Subscriber sc abonneert zich op events met subject sj.
    OperationResult unsubscribe(in Subscriber sc, in Subject sj);
    // de Subscriber sc annuleert het abonnement op events met subject sj.
};
```



4 Basic Frameworkcomponents

4.1 Inleiding

Een BasicFrameworkComponent (BasicFC) is een raamwerkcomponent dat *geen* bouwsteen vormt binnen een case. Een BasicFC vervult een ondersteunende rol bij het samenstellen en/of uitvoeren van een case (uitvoeren van een simulatie). De procesketen beheertool en DataEngine zijn de belangrijkste specialisaties van BasicFC. Deze specialisaties moeten binnen het raamwerk geregistreerd en geïntanceerd zijn, voordat het mogelijk is om een case samen te stellen. De procesketen beheertool en DataEngine zijn daarnaast de enige componenten die direct door het raamwerk worden geïntanceerd. Alle overige specialisaties van BasicFC worden via de PKBT gecreëerd. Voorbeelden van deze overige specialisaties zijn de attribuut editor en de schematisatie editor (binnen SR versie 1 de SRW-Editor).

4.2 Procesketen beheertool

De procesketen beheertool speelt een essentiële rol bij het samenstellen van een case (configureren van modelapplicaties en generieke tools) en het uitvoeren van een simulatie. Alleen een PKBT kan BuildingFrameworkComponents aansturen tijdens een simulatie. De PKBT biedt zijn functionaliteit door middel van de onderstaande interface:

```

Interface ProcessManager: BasicFrameworkComponent {
    OperationResult run();
        // start de huidige case, dus de aanroep van BuildingFC's in de juiste volgorde.
    OperationResult pause();
        // pauseert het aanroepen van BuildingFC's.
    OperationResult resetRun();
        // deze methode geeft aan elke BuildingFC in de huidige case de opdracht de laatste toestand
        // weg te schrijven.
};

typedef Long caseID;

interface Position {
    Int getX();
    Int getY();
    Int getZ();
};

enum ComponentStatus {
    csNotInitialized,
    csInitialized,
    csRunning,
    csReady,
    csFinalized
};

```

Voor het samenstellen van een case maakt de PKBT gebruik van een case-object. Dit object bevat alle informatie om een configuratie van modelapplicaties en tools te tekenen en geeft tijdens een simulatie de DataEngine de benodigde informatie om invoer voor elke component binnen een case ter beschikking te stellen. Een case-object is automatisch aanwezig bij een geïntanceerde PKBT. Omdat in de eerste versie van SR geen case management wordt gerealiseerd wordt het case-object altijd



overschreven bij wijzigingen. Indien case management in een later stadium wel wordt gerealiseerd is alleen een interne wijziging (aanvulling) van de PKBT vereist. De PKBT is de enige component die wijzigingen of aanvullingen in een case kan aanbrengen.

```
Interface CaseManager: FrameWorkComponent {
    sequence <case> getCases();
        // geeft de beschikbare cases als lijst weer terug.
};
```

```
Interface Case {
    OperationResult add(in BuildingFrameworkComponent c);
        // voegt BuildingFC c toe aan de case.
    OperationResult remove(in BuildingFrameworkComponent c);
        // verwijdert BuildingFC c uit de case.
    OperationResult move(in BuildingFrameworkComponent c, in Position p);
        // verplaatst BuildingFC c op het canvas naar nieuwe positie p.
    OperationResult connect(in BuildingFrameworkComponent start, in BuildingFrameworkComponent end);
        // voegt een connectie toe tussen BuildingFC's start en end.
    OperationResult disconnect(in Connection cn);
        // verwijdert de verbinding cn uit de case.
    OperationResult check();
        /* controleert voor alle BuildingFC's en verbindingen in de case of alle vereiste instellingen
        ingevoerd zijn. */
    OperationResult save();
        // deze methode initieert de opslag van de case.
    OperationResult read(in caseID id);
        // leest de eigenschappen van de case met caseID id in de huidige case.
    Time getStartTime();
        // geeft de starttijd van de case.
    Time getEndTime();
        // geeft de eindtijd van de case.
    OperationResult setStartTime(in Time t);
        // stelt de starttijd t in.
    OperationResult setEndTime(in Time t);
        // stelt de eindtijd t in, er vindt controle plaats of t groter is dan startTime.
    Sequence <BuildingFC> getBuildingFC's();
        // Geeft een lijst met referenties naar alle buildingFC's binnen de case.
    Sequence <Connection> getConnections();
        // Geeft een lijst met referenties naar alle connections binnen de case.
    BuildingFC getNext(in Connection cn);
        // geeft een verwijzing naar de BuildingFC waarnaar de verbinding verwijst.
    BuildingFC getPrevious(in Connection cn);
        // geeft een verwijzing naar de BuildingFC waar vanuit de verbinding start.
    Sequence <Connection> getEnteringConnections(in BuildingFC c);
        // geeft de lijst verbindingen die naar BuildingFC c verwijzen.
    Sequence <Connection> getLeavingConnections(in BuildingFC c);
        // geeft de lijst verbindingen die vanuit BuildingFC c starten.
    Sequence <BuildingFC> getStartBuildingFCs();
        // geeft de lijst BuildingFC's zonder ingaande verbindingen
    Sequence <Connection> getConnections(in Predicate p);
        // geeft de lijst van verbindingen binnen de case die voldoen aan het opgegeven predikaat p.
    Sequence <BuildingFC> getBuildingFCs(in Predicate p);
        // geeft de lijst van BuildingFC's binnen de case die voldoen aan predikaat p.
};
```

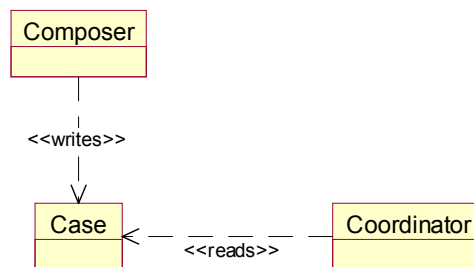
De PKBT voert de geboden functionaliteit uit met behulp van een aantal objecten die in de PKBT zijn opgenomen:

- *Composer* – Dit object wordt gebruikt voor het samenstellen, controleren en visualiseren van een case. De composer levert alle benodigde informatie om de case op bijvoorbeeld een canvas te kunnen visualiseren. Alle instellingen die gepleegd worden kunnen door de Composer in het case-object geschreven worden. Voordat een

simulatie gestart kan worden zorgt de Composer voor een controle van de case.

- **Coördinator** – Bij het uitvoeren van een simulatie moet op basis van de gegevens die in het case-object zijn opgeslagen bepaald worden welke modelapplicatie(s) of generieke tool(s) gestart kan (kunnen) worden. Deze functionaliteit biedt de Coördinator. Hierbij maakt de Coördinator gebruik van de Composer. De Coördinator kan op basis van de events die de Publisher ontvangt bepalen welke volgende taak uitgevoerd moet worden.

Het onderstaande klassediagram toont de onderdelen van de PKBT met de samenhang tussen deze onderdelen.



Figuur 4-1: Klassediagram PKBT

De onderdelen van de PKBT bezitten de onderstaande interfaces:

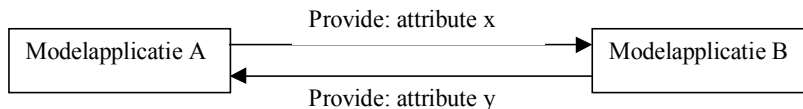
```

Interface Composer {
    OperationResult add (in ComponentReference cr);
        // creëert een BuildingFC op basis van ComponentReference cr en tekent dit op het canvas.
    OperationResult remove (in BuildingFrameworkComponent c);
        // verwijdert BuildingFC c uit de case en van het canvas.
    OperationResult move(in BuildingFrameworkComponent c, in Position p);
        // verplaatst BuildingFC c naar positie p.
    Boolean frameworkComponentCreated(in ComponentReference cr);
        // geeft true als resultaat als een component geïnstantieerd is met ComponentReference cr.
    Boolean serviceProviderCreated(in ServiceDescriptor sd);
        // geeft true als resultaat als een component geïnstantieerd is met ServiceDescriptor sd.
    OperationResult connect(in BuildingFrameworkComponent start, in BuildingFrameworkComponent end);
        // maakt een Connection-object en associeert dit object met BuildingFC start en end.
    OperationResult disconnect(in connection cn);
        // verwijdert de connection cn.
    CaseID openCase();
        // opent een dialoog waarna de gegevens van de geselecteerd case ingelezen worden.
    Boolean checkCase();
        // controleert van alle BuildingFC's en verbindingen of de vereiste instellingen ingevoerd zijn.
    OperationResult setStartTime(in Time t);
        // stelt de starttijd van de case in.
    OperationResult setEndTime(in Time t);
        // stelt de eindtijd van de case in.
};

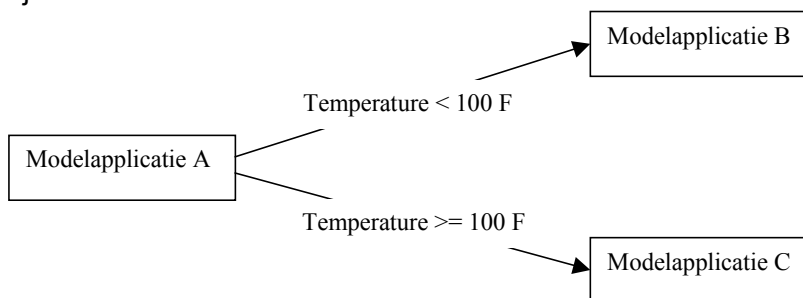
Interface Coördinator {
    OperationResult start();
    OperationResult pause();
    OperationResult reset();
    OperationResult statusChanged(in BuildingFrameworkComponent c);
    ModelApplication getNextInSimulation();
        // geeft de modelapplicatie die als eerst volgende opgestart wordt
    ModelApplication selectLowestCurrentTime();
        // geeft de modelapplicatie met de laagste current time
};
  
```

Een onderdeel van een case wordt gevormd door de verbindingen tussen de BuildingFC's. In veel gevallen wordt door de data-uitwisselingspunten (koppelingen van aansluitpunten) impliciet een rekenvolgorde vastgelegd. Indien modelapplicatie A gegevens nodig heeft van modelapplicatie B is het duidelijk dat B eerst moet rekenen voordat A gestart kan worden. In een aantal gevallen is deze volgorde niet af te leiden:

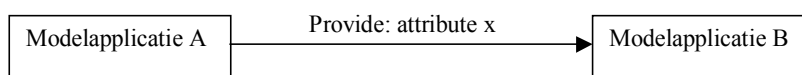
- Twee modelapplicaties hebben gegevens van elkaar nodig om te kunnen rekenen: welke modelapplicatie 'mag' nu eerst?



- Een modelapplicatie mag alleen resultaten van een andere modelapplicatie gebruiken indien aan een bepaalde conditie wordt voldaan: de volgorde van starten van modelapplicaties kan in dat geval tijdens een simulatie veranderen.



Naast de uitwisseling van gegevens en de eventuele condities die hiervoor gelden kan ook de gebruiker specifieke wensen hebben ten aanzien van de rekenvolgorde. Indien in de onderstaande situatie ModelApplicatie A elke tijdstap een resultaat voor ModelApplicatie B oplevert zou op basis van de gegevensuitwisseling bepaald worden dat ModelApplicatie A en B elke tijdstap een periode rekenen. Een gebruiker wil echter de mogelijkheid hebben om ModelApplicatie A tot de eindtijd van de case te laten rekenen voordat B wordt gestart.

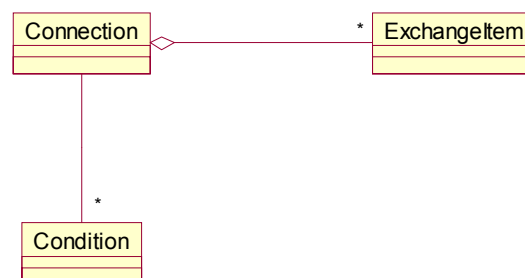


Figuur 4-2: Voorbeeld case: een modelapplicatie heeft als invoer resultaten van een andere modelapplicatie nodig.

Bovenstaande situaties illustreren het belang van de sturende rol van de PKBT tijdens een simulatie. Gegevensuitwisseling kan min of meer impliciet plaatsvinden omdat leverende en ontvangende attributen aan elkaar gekoppeld worden, de PKBT zorgt ervoor dat rekening wordt gehouden met een eventuele opgelegde volgorde of geldende condities tussen verbindingen. Ten opzichte van de Architectuur SR [SR-A] is dit een wijziging van het strikte pull-mechanisme (gevraagde attribuutwaarden impliceren het starten van rekenkernen in de juiste volgorde) naar een hybridevorm van zowel het pull- als het push-mechanisme (expliciet modelapplicaties starten). De modelapplicaties worden volgens deze

architectuur expliciet gestart (push), de uitwisseling van attribuutwaarden vindt op impliciete wijze plaats (pull) omdat met behulp van de DataEngine modelapplicaties kunnen lezen en schrijven naar stukken gedeeld (werk)geheugen.

Bij het definiëren van verbindingen tussen BuildingFC's worden objecten van het type *Connection* gemaakt. Een Connection-object representeert een verbinding tussen twee modelapplicaties of tussen een modelapplicatie en een generieke tool. Het onderstaande klassediagram toont de klasse Connection met bijbehorende afhankelijkheden.



Figuur 4-3: Klassediagram Connection

Een Connection object biedt een interface voor het nader specificeren van de verbinding in de vorm van de SRW-Editor (paragraaf **Fout!**

Verwijzingsbron niet gevonden.). Deze editor toont voor de verbonden modelapplicaties de aansluitpunten. Voor elk aansluitpunt kan per 'vragend' attribuut gedefinieerd worden welk attribuut van een ander aansluitpunt resultaten gaat leveren. Voor elk van deze koppelingen tussen attributen wordt een *ExchangeItem* object gecreëerd. In het ExchangeItem object wordt de informatie over de koppeling bewaard. Dit betreft informatie over de wijze waarop resultaten worden doorgegeven. De onderstaande typen worden onderscheiden:

- *Normaal*: Een attribuut van een modelcomponent levert waarden die direct door een attribuut van één of meerdere andere modelcomponenten gebruikt worden;
- *Verdeeld*: Een attribuut van een modelcomponent levert waarden die verdeeld worden over attributen van meerdere andere modelcomponenten volgens een procentuele verdeling, instelbaar door de gebruiker;
- *Afstands-verdeeld*: Een attribuut van een modelcomponent levert waarden die verdeeld worden over attributen van meerdere andere modelcomponenten volgens een procentuele verdeling, op basis van de afstand tussen de modelcomponenten.

Van elk van bovenstaande uitwisselingstypes wordt de omgekeerde variant ondersteund (dus meerdere attributen die samen invoer genereren voor één ander attribuut op basis van optellen of middelen van waarden).

De klassen Connection en ExchangeItem bieden de onderstaande interfaces:

```

Interface Connection {
    BuildingFrameworkComponent getStart();
    // geeft een referentie naar de BuildingFC waar vanuit de verbinding start.
  
```



```

BuildingFrameworkComponent getEnd();
    // geeft een referentie naar de BuildingFC waarnaar de verbinding wijst.
Sequence<ExchangeItem> getExchangeItems();
    // geeft de verzameling exchangeItems die in de verbinding zijn opgenomen.
OperationResult check();
    // controleert of alle vereiste instellingen in de verbinding zijn ingesteld.
OperationResult edit();
    // start de SRW-Editor waarmee exchangeItems gemaakt kunnen worden.
Boolean hasConditions();
    // geeft true als resultaat als één of meerdere condities in de verbinding zijn opgenomen.
OperationResult addCondition(in Condition c);
    // start een editor waarmee condities samengesteld kunnen worden.
OperationResult removeCondition(in Condition c);
    // verwijdert de conditie c die in de verbinding is opgenomen.
OperationResult addExchangeItem(in ExchangeItem ex)
    // voegt exchangeitem ex toe.
Boolean evaluate(in Time t);
    // controleert of op het gegeven tijdstip t aan alle ingestelde condities en specificaties wordt
    voldaan.
OperationResult save();
    /* schrijft alle instellingen van de verbinding, behorend bij de huidige case, weg m.b.v. de
    DataEngine. */
};

Interface Condition {
    Boolean evaluate(in Time t);
        // geeft true als resultaat als op het gegeven tijdstip t aan de conditie wordt voldaan.
}

enum Operator {
    smaller,
    smallerOrEqual,
    equal,
    greaterOrEqual,
    equal,
    notEqual
}

Interface ExchangeItem {
    Sequence<ModelAttribute> getAttributeProviders();
        /* geeft de verzameling attributen die samen een attribuutwaarde voor een ander aansluitpunt
        vormen (door optelling of een gemiddelde te bepalen). Als de verzameling uit slechts één
        attribuut bestaat mag de verzameling attribute-acceptors uit meerdere attributen bestaan.
        (alleen 1-n koppelingen worden ondersteund). */
    Sequence<ModelAttribute> getAttributeAcceptors();
        /* geeft de verzameling attributen die door een attribuut van een ander aansluitpunt van
        invoerwaarden worden voorzien. Als de verzameling uit slechts één attribuut bestaat mag de
        verzameling attribute-providers uit meerdere attributen bestaan (alleen 1-n koppelingen
        worden ondersteund). */
    Double getData(in ModelAttribute receiver);
        // met deze methode kan een attribute (receiver) uitwisselings-data opvragen.
    ExchangeType getExchangeType();
        // geeft het type koppeling aan.
    Boolean requiresConverter();
        /* geeft true als resultaat indien een converter vereist is om de attribuutwaarden uit te
        wisselen. */
    OperationResult setConverter(in Converter c);
        /* geeft het exchangeItem de verwijzing naar de converter die vereist is voor het uitwisselen
        van attribuutwaarden. Zie paragraaf 4.4 voor een interface beschrijving van Converter*/
};

enum ExchangeType {
    etNormal,
    etDividedValues,
    etDividedValuesByDistance,
    etSumProvidingValues,
    etAverageProvidingValues
};

```



4.3 DataEngine

De DataEngine is een onderdeel van de SR architectuur die het mogelijk maakt om gegevens ter beschikking te stellen op een zeer globaal niveau (bijvoorbeeld het beheren van bestands- of directorynamen van data) tot een zeer gedetailleerd niveau (bijvoorbeeld door gegevens in een geregistreerd formaat als Adventus te laten beheren door de DataEngine). Via subcomponenten biedt de dataEngine standaard de functionaliteit om cases persistent op te slaan en te ontsluiten en om cachers (stukjes werkgeheugen, dus niet persistent) ter beschikking te stellen voor attribuutwaarden die uitgewisseld moeten worden tussen BuildingFC's. Naast deze standaard functionaliteit biedt de DataEngine een component om datastructuren te registreren. Op deze manier kan functionaliteit worden toegevoegd. Na registratie van bijvoorbeeld de Adventus datastructuur kunnen modelapplicaties in SR die (geheel of gedeeltelijk) gebruik maken van deze structuur het databaseer (geheel of gedeeltelijk) door de DataEngine uit laten voeren. Vergelijkbaar met de generieke Adventus structuur kunnen ook specifieke datastructuren geregistreerd worden, bijvoorbeeld de datastructuur behorend bij een Modflow modelapplicatie. De bovengenoemde door de DataEngine te leveren functionaliteiten worden ondergebracht in subcomponenten welke ieder een eenduidig deel van de gevraagde functionaliteiten bieden.

De DataEngine biedt de beschreven functionaliteit door middel van de onderstaande interface:

```
Interface DataEngine {
    RegisterResult RegisterDataDefinition (in DataDefinition dd);
    // Registreert datadefinitie met bijbehorende datastructuren.
    RegisterResult UnregisterDataDefinition(in DataDefinition dd);
    // Verwijdert registratie van datastructure met bijbehorende datastructuren.
    Sequence<Description> ListCases();
    // Geeft lijst van alle bestaande cases.
};
```

De verschillende functionaliteiten van de DataEngine worden door een aantal subcomponenten gerealiseerd:

- *ExchangeServer* – Voor de uitwisseling van data tussen BuildingFC's wordt gebruik gemaakt van de ExchangeServer. Voor elke verbinding die gedefinieerd wordt tussen twee BuildingFC's wordt in de DataEngine een cacher gemaakt. Deze cacher kan resultaten van een modelapplicatie in het werkgeheugen bewaren, zolang er andere BuildingFC's kunnen zijn die van deze resultaten gebruik willen maken;
- *DataStore* – Alle persistente gegevens van een case of onderdelen daarvan (de BuildingFC's en verbindingen) worden door de DataStore verwerkt. De gegevens dienen hiervoor aangeboden te worden in een geregistreerde datastructuur. Voor modelapplicaties die zelf het databaseer uitvoeren zal deze datastructuur zeer eenvoudig zijn (bijvoorbeeld een structuur waarin gedefinieerd staat dat alle bestanden in een directory met de naam van de case staan). Het is mogelijk aan de DataStore nieuwe datastructuren aan te melden of bestaande structuren te verwijderen;
- *DataDefinition* – Elk attribuut van een modelcomponent kan aangeven volgens welke datastructuur de 'eigenaar' (de modelapplicatie) dit attribuut leest en schrijft. Gegeven deze structuur wordt in het DataDefinition-object het bijbehorende *Device*, de fysieke tabellen- of

bestandenstructuur, gezocht. Binnen een *DataDefinition* wordt de definitie van gegevens in *DataStructures* vastgelegd. Deze structures bestaan uit de naam, het type, de maximale lengte en (indien relevant) de nauwkeurigheid (decimalen) van de data. Het onderstaande voorbeeld dient ter verduidelijking van de relatie tussen *DataDefinition* en *DataStructure*, maar impliceert nog geen fysieke opslagvorm. De *DataStructures* kunnen geaggregeerd worden, in dat geval bestaat een datastructure uit sub-datastructures. In de onderstaande figuur is een voorbeeld gegeven voor de notatie van een datadefinitie. De daadwerkelijke definitie van de notatie dient in de vervolgfase vastgelegd te worden.

```

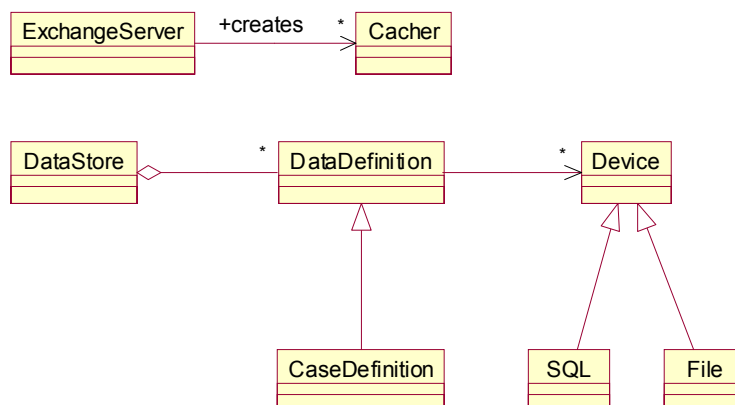
DataDefinition: Test
{
    DataStructure1:
    {
        name = groundwaterlevel
        type = double
        length = 8
        precision = 2
        device = sql
    }
    DataStructure2:
    {
        name = inputfilename
        type = string
        length = 100
        precision = 0
        device = inifile
    }
}

```

Figuur 4-4: Voorbeeld datadefinitie met datastructuren

- *Device* – Een device representeert een bepaald opslagmedium. Een SQL-Device kan SQL statements uitvoeren, een File-Device kan bestanden openen, lezen/schrijven en sluiten.

In het onderstaande klassediagram worden de onderdelen van de *DataEngine* met de onderlinge relaties weergegeven.



Figuur 4-5: Klassediagram DataEngine

De in Figuur 4-5 gepresenteerde klassen bieden de onderstaande interfaces:



```

Interface ExchangeServer {
    OperationResult createCacher(in BuildingFC c);
        // maakt een object dat intern geheugen beschikbaar stelt voor tijdelijke opslag
    OperationResult flushCache(in BuildingFC c);
        // maakt alle tijdelijke opslag leeg behorende bij BuildingFC c.
    OperationResult flushPeriod(in BuildingFC c, in Time start, in Time end);
        /* maakt alle tijdelijke opslag leeg behorende bij BuildingFC c, waarbij alleen attributwaarden
        die gelden binnen tijdsinterval 'start' en 'end' worden verwijderd. */
    OperationResult removeCacher(in BuildingFC c);
        // verwijdert de objecten die tijdelijke opslag verzorgen voor BuildingFC c.
    OperationResult flushAllCachers();
        // verwijdert alle tijdelijk opgeslagen resultaten.
    OperationResult removeAllCachers();
        // verwijdert alle objecten die tijdelijke opslag verzorgen.
    OperationResult getExchange(in BuildingFC c, in CaseID id, in Time t);
        // haalt voor tijdstip t de benodigde uitwisselingsdata voor BuildingFC c uit het werkgeheugen
    OperationResult putExchange(in BuildingFC c, in CaseID id, in Time t);
        /* schrijft de geleverde uitwisselingsdata behorend bij tijdstip t en BuildingFC c naar het
        werkgeheugen. */
};

Interface Cacher {
    BuildingFC getProvider();
        // geeft een verwijzing naar de BuildingFC die uitwisselingsdata schrijft in deze cacher.
    OperationResult setProvider(in BuildingFC c);
        // zet een verwijzing naar de BuildingFC die uitwisselingsdata schrijft in deze cacher.
    OperationResult putExchangeData(in ModelAttribute attr, in Time t);
        // schrijft de attributwaarde van attr in de cacher, geldend op tijdstip t.
    OperationResult getExchangeData(in ModelAttribute attr, in Time t);
        // leest de attributwaarde van attr uit de cacher, geldend op tijdstip t.
    OperationResult flush();
        // maakt de cacher leeg.
    OperationResult flushPeriod(in Time start, in Time end);
        // verwijdert alle attributwaarden met een geldend tijdstip tussen 'start' en 'end' uit de cacher.
};

Interface DataStore {
    Sequence<Description> getCases();
        // geeft de verzameling opgenomen cases.
    OperationResult getCase(in Case c; in CaseID id);
        // leest de case-eigenschappen behorende bij caseID id in de huidige case c.
    OperationResult putCase(in Case c);
        // schrijft alle eigenschappen van de huidige case weg.
    OperationResult getBuildingFCdata(in BuildingFC c, in CaseID id, in Time t);
        // haalt alle gegevens op om BuildingFC c te kunnen initialiseren.
    OperationResult putBuildingFCdata(in BuildingFC c, in CaseID id, in Time t);
        // schrijft alle gegevens weg nadat BuildingFC c klaar is met berekeningen (finalization)
    Sequence<DataDefinition> getDataDefinitions();
        // geeft de verzameling geregistreerde data-definities.
    Boolean dataDefinitionExists(in DataDefinition dd);
        // geeft true als resultaat als DataDefinition dd reeds geregistreerd is in de DataEngine.
    RegisterResult registerDataDefinition(in DataDefinition dd);
        // registreert de nieuwe datadefinitie dd
    RegisterResult unregisterDataDefinition(in DataDefinition dd);
        // verwijdert de datadefinitie dd
    Boolean readData(in BuildingFC c, in CaseID id, in Time t);
        // leest initialisatie gegevens uit van BuildingFC c behorende bij de case met caseID id.
    Boolean writeData(in BuildingFC c, in CaseID id, in Time t);
        // schrijft de eindtoestand van BuildingFC weg.
};

Interface DataDefinition {
    Description getDescription();
        // geeft een omschrijving van de datadefinitie.
    Double getVersion();
        // geeft het versienummer
    Device getDevice();
        // geeft een verwijzing naar de bijbehorende device
};

Interface Device {

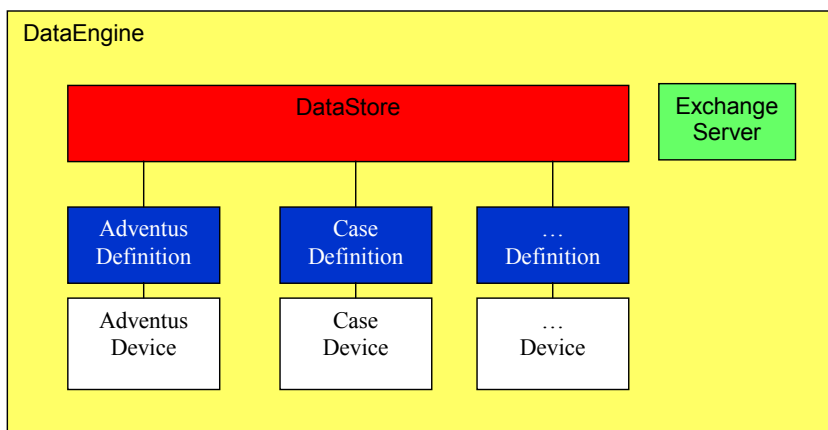
```

```

Description getDescription();
    // geeft een omschrijving van de device.
OperationResult prepare();
    // maakt een connectie met het opslagmedium om data te kunnen lezen of schrijven.
OperationResult read(in BuildingFC c, in CaseID id, in Time t);
    // leest initialisatie gegevens van BuildingFC c.
OperationResult write(in BuildingFC c, in CaseID id, in Time t);
    // schrijft eindtoestand van BuildingFC c.
OperationResult close();
    // verbreekt de verbinding met het opslagmedium (bestand, database, ...)
};

```

Elk Device kan in een bepaald type opslagmedium lezen en schrijven. Het ontwerp staat zowel het gebruik van zeer generieke Devices toe (bijvoorbeeld een device dat elke relationele database kan benaderen met behulp van SQL) als zeer specifieke Devices voor één bepaalde database (b.v. Oracle) en één bepaalde tabellenstructuur (b.v. een SWAP database). De eerste versie van de DataEngine beschikt over DataDefinition- en Device-componenten voor de Adventus structuur en een structuur voor het opslaan van cases. De daarnaast te registreren datastructuren zijn afhankelijk van de modelapplicaties die in deze versie opgenomen worden. De eerste versie van de DataEngine bestaat uit de volgende onderdelen:



Figuur 4-6: Subcomponenten in de DataEngine

4.4 Converter

Bij het maken van koppelingen tussen aansluitpunten van schematisaties wordt automatisch gecontroleerd of de te koppelen attributen overeenkomen qua meta-informatie. In de eerste versie van SR bestaat deze meta-informatie uit de beschrijving (string) en de eenheid waarin het attribuut waarden levert of wenst. Indien de meta-informatie van te koppelen attributen niet overeenkomt zorgt de Broker binnen de PKBT ervoor dat binnen SR wordt gezocht naar een converter die deze conversie naar de gewenste eenheid uit kan voeren. Indien de beschrijving van de attributen niet overeenkomt, wordt alleen een melding gegeven, als de eenheden niet overeenkomen wordt de mogelijkheid tot koppeling bepaald door de aanwezigheid van een converter.

Een converter kan net als elk FrameworkComponent aangeven welke diensten hij vereist en kan leveren. Een converter kan dus bijvoorbeeld aangeven dat inches naar meters omgezet kunnen worden. Voor de conversie biedt de converter de methode *convert()*.

```

Interface Converter : BasicFrameWorkComponent {

```



```

Double Convert(in Time t, in ModelAttribute Provider, in ModelAttribute Receiver);
/* Gegeven het tijdstip levert de converter de waarde die de Provider kan leveren in de
   eenheid die Receiver verwacht. De converter vraagt dus zelf de te converteren waarde aan
   Provider en de eenheden aan zowel Provider als Receiver. */

```

```

}
```

4.5 SRW-Editor

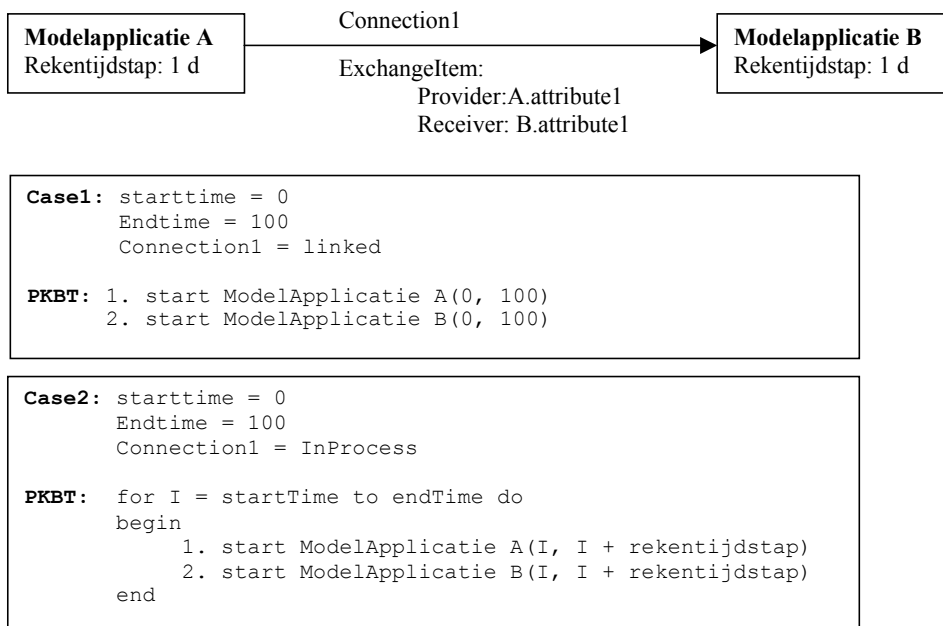
De SRW-Editor dient ervoor om het gedrag van een connectie in de modellentrein te specificeren. De volgende eigenschappen kunnen aan een connectie worden toegekend:

- *Data uitwisseling* – Er zijn twee mogelijkheden bij het definiëren van data uitwisseling:
 - Data uitwisseling tussen modelapplicaties:
Hierbij worden koppelingen gelegd tussen aansluitpunten van de twee betrokken schematisaties. De SRW-Editor kan beide schematisaties weergeven en een user-interface aanbieden om de koppelingen te leggen.
 - Data uitwisseling tussen een modelapplicatie en een generieke tool:
In dit geval zal één schematisatie getoond worden en zal de gebruiker hieruit ModelComponenten en bijbehorende attributen kunnen selecteren, met als doel de waarden hiervan te gebruiken in een GenericTool (bijvoorbeeld om een grafiek te maken)
- *Koppelingstype* – In de Architectuur SR [SR-A] zijn twee specialisaties van GenericTool gedefinieerd: LinkedTool en InProcessTool. Volgens de Architectuur SR maakt een LinkedTool wel onderdeel uit van de case, maar levert geen resultaten die als invoer van andere componenten kunnen dienen. Een InProcessTool kan wel resultaten als invoer voor andere componenten beschikbaar stellen. Binnen de onderhavige architectuur worden deze specialisaties van GenericTool niet toegepast omdat deze specialisaties niet afhankelijk zijn van het type tool of modelapplicatie, maar van de plaats in een modellen/tools-configuratie. In het SR wordt een BuildingFC wordt als *'linked'* beschouwd als in de verbinding met de voorgaande BuildingFC is vastgelegd dat de BuildingFC zo 'laat mogelijk' wordt gestart (dus de voorgaande BuildingFC zo lang mogelijk doorrekend)³. Zo laat mogelijk starten betekend dat een BuildingFC wordt gestart als er geen andere berekeningen meer uitgevoerd dienen te worden (of kunnen worden). Een BuildingFC wordt als *'In-Process'* beschouwd als in de verbinding met de voorgaande BuildingFC is vastgelegd dat de BuildingFC altijd gestart wordt zodra alle invoergegevens beschikbaar gesteld zijn.

In de SRW-Editor kan aangegeven worden of de verbinding Linked of In-Process is.

In onderstaande figuur wordt het verschil tussen deze twee soorten verbindingen verduidelijkt aan de hand van stukjes (pseudo)code, behorende bij de PKBT die de modelapplicaties A en B in de juiste volgorde moet starten.

³ Deze definitie wijkt af van de definitie in het architectuurrapport. Binnen het SRW wordt het onderscheid tussen 'in-process' en 'linked' bepaald door de manier van aansturing welke door de gebruiker aangegeven wordt. In het architectuurrapport wordt het onderscheid bepaald door het wel of niet in-process kunnen zijn.



Figuur 4-7: Linked en InProcess koppeling van BuildingFrameworkComponents

- **Conditie** – Per verbinding kunnen één of meerdere condities worden gedefinieerd. Er kan bijvoorbeeld aangegeven worden dat een BuildingFC alleen gestart mag worden als een bepaald resultaat (een attribuutwaarde) van de voorgaande BuildingFC een bepaalde waarde heeft, overschrijdt of onderschrijdt. De SRW-Editor biedt de mogelijkheid deze condities te definiëren.

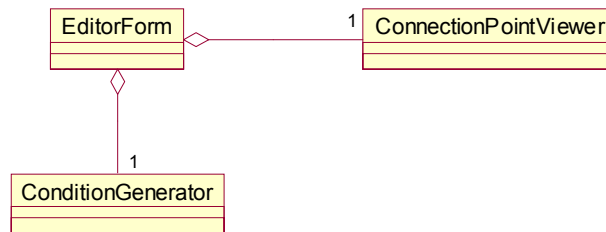
Binnen de SRW-Editor dient onderscheid gemaakt te worden tussen de grafische interface (GUI) en de onderliggende logische functionaliteit. Veel van de functionaliteit van de SRW-Editor wordt door de gebruiker via de user interface geïnitieerd. Deze functionaliteit kan tevens aangestuurd worden via de interface van deze component. Hierbij moet gedacht worden aan het leggen van de koppelingen, het tonen van een achtergrond en het opslaan en inlezen van gegevens.

Via de interface van de SRW-Editor wordt toegang verkregen tot verschillende onderdelen van de SRW-Editor. De interface is als volgt gespecificeerd:

```

Interface SRWEditor {
    SRWEditor create (in BuildingFC start, in BuildingFC end);
    //creert een instantie van een SRW-Editor waarme de verbinding tussen opgegeven
    BuildingFC's geedit kan worden.
    EditorForm EditorForm()
    //geeft andere componenten toegang tot de diensten van de Editor
    ConnectionPointViewer ConnectionPointViewer()
    // geeft andere componenten toegang tot de diensten van de ConnectionPointViewer
    ConditionGenerator ConditionGenerator()
    // geeft andere componenten toegang tot de diensten van de ConditionGenerator
};
  
```

Intern realiseert de SRW-Editor zijn functionaliteit met behulp van een aantal subcomponenten (objecten). De structuur hiervan is in Figuur 4-8 weergegeven.



Figuur 4-8: Klassediagram SRW-Editor

In de bovenstaande figuur zijn de drie onderdelen van de SRW-Editor weergegeven. Elk onderdeel bezit een deel van de functionaliteit van de SRW-Editor.

- *EditorForm* – Een SRW-Editor bezit een scherm (form) waar vanuit alle functionaliteit benaderd kan worden. Dit scherm, het EditorForm, wordt automatisch gecreëerd bij creatie van de SRW-Editor;
- *ConnectionPointViewer* – Een onderdeel van het EditorForm wordt gevormd door de ConnectionPointViewer, de schematisatie-viewer voor aansluitpunten van schematisaties. Na instantiatie van de SRW-Editor benadert de ConnectionPointViewer de verbonden ModelApplicatie(s) waarbij de lijst van aansluitpunten (ModelComponents) wordt gevraagd. Op basis van deze lijst Modelcomponenten, die coördinaten volgens OpenGis bevatten, visualiseert de viewer de schematisatie. De gebruiker kan een aansluitpunt selecteren, waarna (met behulp van de broker van de PKBT) een attribuut-editor (zie paragraaf 4.6) wordt gestart. Deze attribuut-editor toont een overzicht van alle attributen van het aansluitpunt. Door analoog hieraan een of meerdere andere aansluitpunten en attributen te selecteren kunnen koppelingen worden gedefinieerd. De ConnectionPointViewer creëert voor elke gemaakt koppeling een Exchangeltem-object;
- *ConditionGenerator* – Deze generator biedt een interface voor het definiëren van condities op een verbinding. De ConditionGenerator creëert deze Condition-objecten. De gedefinieerde Condition-objecten behoren bij een verbinding en worden aldus vastgelegd in de definitie van de case.

Binnen de SRW-Editor communiceren de subcomponenten op basis van de onderstaande interfaces.

```

Interface EditorForm {
    EditorForm create(in Connection cn);
    /* instantieert en toont het editorform waarna van het start- en eindpunt van Connection cn de
    schematisaties worden ingelezen (in geval van een GenericTool wordt uiteraard geen
    schematisatie gegeven). */
    Show(in ModelApplication ma);
    // geeft de schematisatie van de opgegeven modelapplicatie weer.
};

Interface ConnectionPointViewer {
    OperationResult showAttributes();
    // Toont een verzameling attributen behorende bij de geselecteerde ModelComponent.
    OperationResult editExchangeItem(in ExchangeItem e);
    // Toont de eigenschappen van exchangeItem e.
    Image setBackground();
    // geeft een dialoog waarmee een achtergrondplaatje geselecteerd kan worden.
  
```



```
};
```

```
Interface ConditionGenerator {  
    OperationResult createCondition();  
    // start een editor voor het definiëren van condities.
```

```
Struct Image {  
    URL name;  
};
```

4.6 Attribuut editor

Een attribuut-editor is een user interface waarmee de attributen van een modelcomponent weergegeven kunnen worden. De interface van een modelcomponent is gedefinieerd in paragraaf 5.2.2. Een modelcomponent is een aansluitpunt uit de schematisatie van een Model Applicatie zoals gedefinieerd in het Functioneel Ontwerp [SR-FO]. Een modelcomponent bevat een lijst met modelattributen welke door de attribuut-editor weergegeven worden. De interface bestaat uit het creëren van een attribuut-editor en het doorgeven van het desbetreffende modelcomponent. De user interface bevat methoden om de gewijzigde gegevens op te slaan en de editor af te sluiten. Voor het ophalen van de modelattributen en de waarden worden de methoden van andere componenten aangeroepen door de attribuut-editor.

```
Interface AttributeEditor : GenericTool {  
    OperationResult create(in ModelComponent mc);  
    // start de attribuuteditor en presenteert alle attributen behorende bij ModelComponent mc.  
  
};
```



5 Building Frameworkcomponents

5.1 Inleiding

BuildingFrameworkComponents (BuildingFC's) zijn de bouwstenen voor een case, dus de onderdelen waaruit een modellen/tools-configuratie is opgebouwd. Een BuildingFC is een specialisatie van FrameworkComponent. Elke BuildingFC die in het raamwerk geregistreerd wordt is een specialisatie (een verbijzondering) van ModelApplication of van GenericTool. Een BuildingFC biedt daarom altijd de interfaces van FrameworkComponent en van BuildingFC, daarnaast de interface van ModelApplication of GenericTool.

De interface van BuildingFC biedt de functionaliteit om componenten van dit type te tekenen op een canvas en specifieke instellingen met behulp van een eigen editor te plegen.

```

Interface BuildingFrameworkComponent : FrameworkComponent {
    Boolean canDraw();
        // Geeft aan of dit FrameWorkComponent visueel weergeven moet worden op het canvas
    CaseVisualisation getCaseVisalisation();
        // Geeft aan hoe dit FrameWorkComponent visueel weergegeven moet worden op het canvas
    Boolean multipleInstances();
        /* Geeft aan of van dit FrameworkComponent meerdere instanties binnen een case aanwezig
        mogen zijn. */
    Boolean hasPropertyEditor();
        /*Geeft aan of dit FrameworkComponent een build-in editor heeft waarmee de waarden van de
        properties ingesteld kunnen worden. */
    OperationResult edit();
        // toont de component specifieke property-editor indien de component hierover beschikt.
    OperationResult save();
        // schijft alle instellingen, behorend bij de huidige case, weg mbv de DataEngine
    OperationResult check();
        // controleert of alle vereiste instellingen voor de component ingesteld zijn.
    OperationResult Init ();
        // Initialiseert de modelapplicatie (waaronder het rekenhart) of generieke tool.
    OperationResult start(in Time starttime, in Time endtime);
        /* geeft opdracht om voor de opgegeven periode te rekenen (ModelApplication) of de
        generieke taak (GenericTool) uit te voeren
};

Struct CaseVisualisation {
    Description name;
    URL image;
        //locatie van de afbeelding welke op het canvas wordt weergegeven als representatie van dit
    component
};

```

5.2 Modelapplicaties

De modelapplicatie is de schil (wrapper) rondom een rekenkern en een schematisatie. De schematisatie is in principe vast opgenomen in de modelapplicatie. Het is echter ook mogelijk binnen een modelapplicatie meerdere schematisaties aan te bieden, waarbij de gebruiker een keuze kan maken. In het eerste geval is in SR bijvoorbeeld de modelapplicatie 'ModFlow-Groningen' geregistreerd. In het tweede geval is de ModelApplicatie 'ModFlow' binnen SR geregistreerd en wordt tijdens het vastleggen van de eigenschappen van de modelapplicatie in de modellentrein een schematisatie van een bepaald gebied geselecteerd.



Informatie omtrend de schematisatie van een modelapplicatie is in het functioneel ontwerp [FO] gedefinieerd als een verzameling aansluitpunten. Dit komt overeen met de in het Architectuur Rapport [SR-A] volgens het Composite Design Pattern opgezette *ModelComponenten*. De schematisatie van de modelapplicatie wordt gerepresenteerd door een verzameling ModelComponenten.

Indien gegevens tussen BuildingFrameworkComponents uitgewisseld worden houdt dit concreet in dat ModelComponenten gegevens van attributen (attribuutwaarden) als input nodig hebben of attribuutwaarden als output moeten kunnen leveren.

Een modelapplicatie heeft geen kennis van verbindingen met andere BuildingFC's. De ModelComponenten kunnen echter met behulp van de gedefinieerde Exchangeltems achterhalen waar de uitwisselingsdata gelezen of geschreven moet worden. Het uiteindelijke schrijven en lezen van data wordt uitgevoerd door de DataEngine, deze faciliteert hierbij het in het werkgeheugen houden van resultaten (attribuutwaarden).

5.2.1 Interface

ModelApplication is een verdere specialisatie van BuildingFrameworkComponent. De in het Functioneel Ontwerp beschreven diensten van de ModelApplicatie worden aangeboden via de onderstaande interface:

```

Interface ModelApplication : BuildingFrameworkComponent {
    OperationResult editProperties();
        /* Roept de Build-in property editor aan waarmee alle properties van de ModelApplicatie
        weergegeven en aangepast kunnen worden. De property-Editor bewaard zijn gegevens via de
        Case in de DataEngine. */
    Time getCurrentTime();
        // Geeft de actuele tijd van de modelapplicatie.
    Time getNextTime();
        /* Geeft aan wat de tijd zo zijn na het uitvoeren van een volgende tijdstap. De tijdstaplengthte
        is daardoor af te leiden. */

    OperationResult Flush();
        // Geeft expliciet de opdracht om alle resultaten op te slaan.
    OperationResult Finalize();
        /* Geeft de opdracht om de alle interne gegevens op te ruimen en eventueel de instantie van
        het rekenhart te vernietigen. */
    Status GetStatus();
        // Geeft de huidige status.

    Sequence<Geometry> getSchematisationGeometry();
        /* Geeft een lijst van geometrische referenties naar de modelcomponenten (aansluitpunten)
        volgens de geometry specificatie. */
    ModelComponent getModelComponent(in Geometry gm);
        // Geeft instantie van een modelcomponent gegeven een geometrische referentie.
};

enum Status {
    stNotInitialized,    // De modelapplicatie is toestandloos, de rekenkern is niet geïnitieerd
    stInitialized,       // De modelapplicatie is gereed voor het uitvoeren van een rekenstap
    stRunning,           // De modelapplicatie is bezig met het uitvoeren van een rekenstap
    stReady,             // De modelapplicatie is klaar met het uitvoeren van een rekentijdstap
    stFinished           // De modelapplicatie is klaar met de gehele simulatie
};

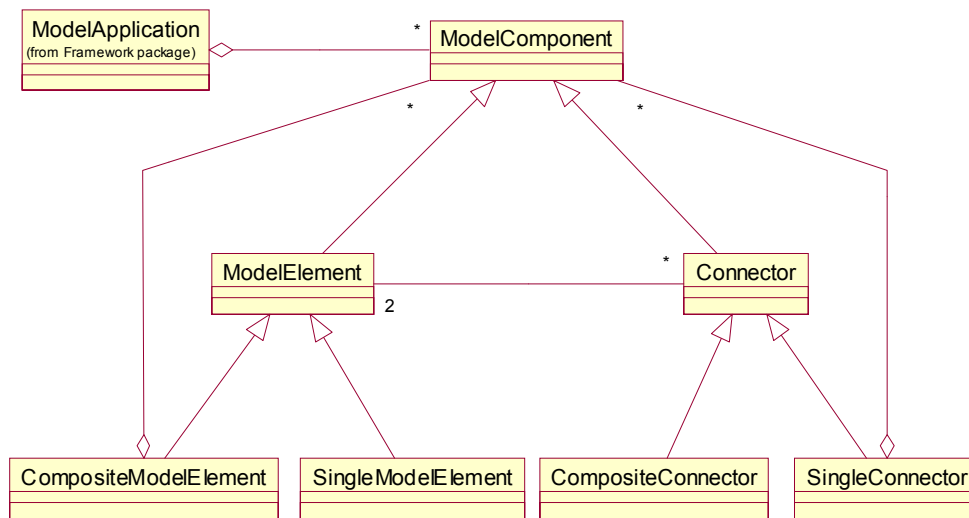
```

5.2.2 Schematisaties van aansluitpunten

Binnen het raamwerk wordt een schematisatie van een modelapplicatie gezien als een netwerk van verbonden ModelComponenten. Bij deze ModelComponenten wordt een onderscheid gemaakt tussen de

knooppunten die systeemvariabelen en stuurvariabelen als attributen bezitten, de *ModelElementen*, en *ModelComponenten* die systeemparameters als attributen bezitten, de *Connectoren*. Een *ModelElement* kan verbonden zijn met een ander *ModelElement* middels een *Connector*. Een *Connector* verbindt altijd twee *ModelElementen*. Een aansluitpunt van een schematisatie is een *ModelComponent*, dit kan dus zowel een *ModelElement* als een *Connector* zijn.

In het onderstaande klassediagram is deze generieke structuur voor schematisaties weergegeven:



Figuur 5-1: Klassediagram ModelComponenten met bijbehorende specialisaties

De mogelijkheid wordt geboden om subcomponenten binnen *ModelComponenten* te onderscheiden. In dat geval bestaat een *ModelComponent* zelf uit *ModelElementen* en *Connectoren*. Deze structuur wordt gerealiseerd met behulp van het Composite Design Pattern. Deze structuur maakt het mogelijk om attribuutwaarden af te leiden uit de attribuutwaarden van de onderdelen van een *ModelComponent*.

Een *ModelComponent* heeft kennis van zijn eigen geometrie. Een modelapplicatie kan gegeven een bepaalde geometrie de bijbehorende *ModelComponent* zoeken. Elk *ModelComponent* kan attributen bezitten. Deze attributen kunnen door de modelapplicatie berekend worden, vereist worden om te kunnen rekenen of beide rollen vervullen. Een *ModelComponent* biedt deze informatie middels de onderstaande interface:

```

Interface ModelComponent {
    Geometry getGeometry();
        // Geeft de geometrische informatie van het modelcomponent.
    string Description();
        // optioneel, extra info omtrend het modelcomponent, bijvoorbeeld voor weergave in SRW-
        Editor
    Long ID();
        //Verplichte id indien Geometry niet uniek is binnen de schematisatie
    Sequence<ModelAttribute> getProvidedAttributes();
        /* Geeft een lijst met alle ModelAttributen die door de ModelApplicatie op dit schematisatie-
        element geleverd worden. */
    Sequence<ModelAttribute> getRequiredAttributes();
        /* Geeft een lijst met alle ModelAttributen die door de ModelApplicatie op dit schematisatie-
  
```

```
element gevraagd worden. */
```

```
};
```

De subclasses van ModelComponent, ModelElement en Connector bezitten naast het interface van ModelComponent een eigen interface waarmee de relaties tussen ModelElementen en Connectoren achterhaald kunnen worden. Een ModelElement geeft aan welke connectoren aan hem gekoppeld zijn, een Connector kan aangeven tussen welke twee modelelementen een verbinding gevormd wordt.

```
Interface ModelElement : ModelComponent {
    Sequence <Connector> getConnectors();
    // Geeft een lijst met connectoren die verbonden zijn met dit ModelElement .
};

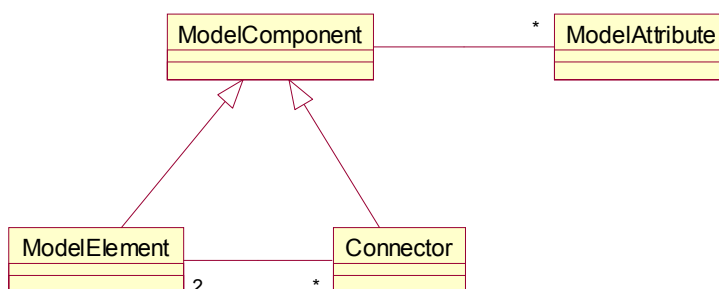
Interface Connector : ModelComponent {
    ConnectionElements GetConnectedElements();
    // Geeft aan welke modelelementen door de connector verbonden worden.
};

struct ConnectionElements{
    ModelElement from_element;
    ModelElement to_element
};
```

Met behulp van samengestelde ModelElementen is het mogelijk aggregaties van ModelComponenten in een schematisatie op te nemen. Uiteraard vereist dit een specifiek interface wat in subclasses van ModelElement en Connector wordt geboden.

```
Interface CompositeModelElement/ CompositeConnector : ModelElement/ Connector {
    Boolean canAddModelComponent(in ModelComponent mc);
    /* Controleer of het modelcomponent aan het samengestelde element of connector
    toegevoegd kan worden. */
    OperationResult addModelComponent(in ModelComponent mc);
    // Voegt een modelcomponent toe aan de samengestelde element of connector.
    OperationResult removeModelComponent(in ModelComponent mc);
    // Verwijdt een modelcomponent uit de samengestelde element of connector.
    Sequence <ModelComponent> contains();
    /* Geeft een lijst met modelcomponenten die in de verzameling van het samengestelde
    element of connector zijn opgenomen zodat deze objecten benaderd kunnen worden. */
};
```

Een ModelComponent bevat een verzameling modelattributen. Modelattributen representeren de eigenschappen (variabelen en parameters) van een modelcomponent. Gegeven een tijdstip kan een attribuut-object een waarde bepalen.



Figuur 5-2: ModelComponent en ModelAttributen



```

Interface ModelAttribute {
    String getName();
        // Geeft de naam van het attribuut.
    Boolean isRequired();
        // Geeft aan of het leveren van invoer voor dit attribuut vereist is. Als het resultaat true is, is
        het resultaat van canUse() ook true.
    Boolean canUse();
        // geeft aan of het attribuut invoerwaarden accepteert voor berekeningen.
    Boolean canProvide();
        // geeft aan of het attribuut een resultaatwaarden kan leveren.
    Boolean hasExchangeItems();
    OperationResult addExchangeItem(in ExchangeItem ex);
    Sequence <ExchangeItem> GetProvidingExchangeItems();
        /* Geeft een lijst met referenties naar ExchangeItems via welke dit ModelAttribute zijn waarde
        aangeleverd krijgt. Via het ExchangeItem zijn de providing ModelAttributes op te vragen. */
    Double getValue(in Time t, in Geometry gm);
        /* Geeft de waarde van het attribuut op een gegeven tijdstip. Optioneel kan extra
        geometrische informatie meegegeven worden voor de exacte plaatsbepaling indien afwijkend
        van de knoop zelf. */
    setValue(in Time t, in Geometry gm, Value)
        /* Geeft het attribuut een waarde op een gegeven tijdstip. Optioneel kan extra geometrische
        informatie meegegeven worden voor de exacte plaatsbepaling indien afwijkend van de knoop
        zelf. */
    DoubleSeq getValueSequence(in Time t1, in Time t2, in Geometry gm);
        // Geeft een reeks waarden van het attribuut liggende in het tijdsinterval [t1,t2].
    AttributeType getType();
        // Geeft het resultaattype.
    SourceType getSource();
        // Geeft de bron van de attribuutwaarde.
    DataDefinition getDataDefintion();
        // Geeft aan volgens welke datadefinitie de ModelApplicatie het attribuut verwacht of oplevert.
};

typedef sequence<Double> DoubleSeq;

enum AttributeType {
    atSystemVariable,
    atExternalVariable,
    atSystemParameter
};

enum SourceType {
    stDataEngine,
    stFormula,
    stConstant
};

```

5.2.3 Geometrie

De uitwisseling van geometrische gegevens vindt plaats volgens de Well Known Structures uit de Open-GIS specificaties. Hieronder wordt het deel van definitie van een geometrie voor WKS-structures zoals deze ook is opgenomen in het Architectuur rapport [SR-A] weergegeven. De gehele IDL staat gespecificeerd in het document te vinden op [http://www.opengis.org/public/sfr1/sfcorba_rev_1_0.pdf].

```

Struct WKSPoint {
    Double x;
    Double y;
};

Typedef sequence <WKSPoint> WKSPointSeq;
Typedef sequence <WKSPoint> WKSLineString;
Typedef sequence <WKSLineString> WKSLineStringSeq;
Typedef sequence <WKSPoint> WKSLinearRing;
Typedef sequence <WKSLinearRing> WKSLinearRingSeq;

```

```

Struct WKSLinearPolygon {
    WKSLinearRing externalBoundary;
    WKSLinearRingSeq internalBoundaries;
};

Typedef sequence <WKSLinearPolygon> WKSLinearPolygonSeq;

Struct Envelope {
    WKSPoint minm;
    WKSPoint maxm;
};

enum WKSType {
    WKSPointType, WKSMultiPointType, WKSLineStringType, WKSMultiLineStringType, WKSLinearRingType,
    WKSLinearPolygonType, WKSMultiLinearPolygonType, WKSCollectionType
};

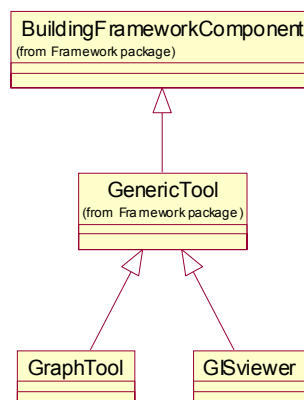
union Geometry
    switch(WKSType) {
        case WKSPointType: WKSPoint point;
        case WKSMultiPointType: WKSPointSeq multi_point;
        case WKSLineStringType: WKSLineString line_string;
        case WKSMultiLineStringType: WKSLineStringSeq multi_line_string;
        case WKSLinearRingType: WKSLinearRing linear_ring;
        case WKSLinearPolygonType: WKSLinearPolygon linear_polygon;
        case WKSMultiLinearPolygonType: WKSLinearPolygonSeq multi_linear_polygon;
        case WKSCollectionType: sequence <WKSGeometry> collection;
    };

```

5.3 Generieke tools

5.3.1 Algemeen

Generieke tools zijn BuildingFrameworkcomponents die een algemene taak binnen een case uit kunnen voeren zoals visualisatie, calibratie of analyse. Een generieke tool bezit in tegenstelling tot een modelapplicatie geen rekenkern en geen schematisatie. Binnen een case kunnen generieke tools vergelijkbaar met modelapplicaties opgenomen worden in een modellen/tools configuratie door een referentie naar een generieke tool te selecteren en op het canvas van de PKBT te slepen. Na deze instantiatie kan met behulp van de SRW-Editor een verbinding tussen een modelapplicatie en een generieke tool gedefinieerd worden.



Figuur 5-3: Klassediagram Generieke tools

Een generieke tool erft het interface van BuildingFC, daarnaast biedt dit component een eigen interface om user-interface functionaliteit te kunnen bieden.



```

Interface GenericTool : BuildingFrameWorkComponent {
    Boolean HasDialog();
    // Generieke Tools die voorzien zijn van een scherm maken dat kenbaar via deze methode.
    SetWindowState(in WindowState ws);
    // Past de weergave van de generieke tool ten opzicht van het standaardraamwerk aan.
    WindowState getWindowState();
    // Geeft de huidige windowstate
};

enum WindowState {
    wsNone,
    wsNormal,
    wsMinimized,
    wsMaximized
};

```

5.3.2 Presentatietool

Met behulp van een presentatietool worden de resultaten van modelapplicaties gevisualiseerd. In de eerste versie van het SR betreft de presentatietool een grafiekentool. De interface van PresentationComponent is dermate generiek dat deze ook voor bijvoorbeeld een GISviewer toegepast kan worden. Via de interface van PresentationComponent worden de te presenteren attributen van de verschillende modelcomponenten aan de presentatietool doorgegeven. De interface van een presentatiecomponent is een verdere specialisatie van GenericTool en ziet er als volgt uit:

```

Interface PresentationComponent : GenericTool {
    Boolean addAttribute(in ModelComponent mc, in ModelAttribute attr);
    /* Voegt een modelattribuut van een modelcomponent toe aan de presentatietool om grafisch
    te worden weergegeven. Resulteert True indien het attribuut daadwerkelijk toegevoegd is. */
    Boolean removeAttribute(in ModelComponent mc, in ModelAttribute attr);
    /* Verwijdert een modelattribuut van een modelcomponent uit de lijst met weer te geven
    attributen. Resulteert True indien het attribuut daadwerkelijk verwijderd is. */
    Sequence<ModelComponent, ModelAttribute> getAttributes();
    // Geeft een lijst met alle attributen die aangemeld zijn om gepresenteerd te worden.
};

```



6 Dynamiek

6.1 Inleiding

In dit hoofdstuk wordt de dynamische view op het Standaard Raamwerk Water met de daarin opgenomen componenten uitgewerkt. Door deze uitwerking wordt een beeld gegeven van de wijze waarop de beschreven componenten samenwerken om de functionaliteiten uit het functioneel ontwerp [SR-FO] te kunnen realiseren. Bij de uitwerking van deze dynamiek wordt uitgegaan van het onderstaande voorbeeld:

Uitgangssituatie:

Er wordt vanuit gegaan dat de eerste versie van SR beschikbaar is met daarin opgenomen een PKBT en een DataEngine. In de component-repository zijn reeds de modelapplicatie SOBEK, de SRW-editor en een converter (voor de conversies meter naar centimeter en omgekeerd) geregistreerd. Daarnaast is een test-modelapplicatie geregistreerd die tijdens implementatie van SR is gebruikt.

Registraties:

Voordat een case wordt samengesteld dienen de modelapplicatie SWAP en een grafiekentool geregistreerd te worden. Daarnaast is de test-modelapplicatie overbodig geworden en kan worden verwijderd.

Cases:



Case A: de configuratie SOBEK – SWAP – Grafiekentool waarbij SOBEK waterstanden van het oppervlaktewater systeem doorgeeft aan SWAP. SOBEK levert waterstanden in meters tov NAP, SWAP verwacht waterstanden in centimeters tov NAP.

De grafiekentool toont grondwaterstanden die berekend worden door SWAP. Alle verbindingen worden 'linked' gemaakt, dus alle modelapplicaties rekenen van begin- tot eindtijd voordat de volgende component wordt gestart. Sobek heeft een rekentijdstap van 1 uur, SWAP van 1 dag.

Case B: identieke configuratie als CASE A, nu geeft SWAP daarnaast drainagefluxen door aan SOBEK en alle verbindingen zijn in-process, dus de modelapplicaties wisselen op tijdstapbasis resultaten uit en in de grafiekentool worden grondwaterstanden tijdens het draaien van SWAP gevisualiseerd. Rekentijdstappen zijn identiek aan Case A.

Simulaties:

Zowel Case A als B worden uitgevoerd.

6.2 Registratie van componenten

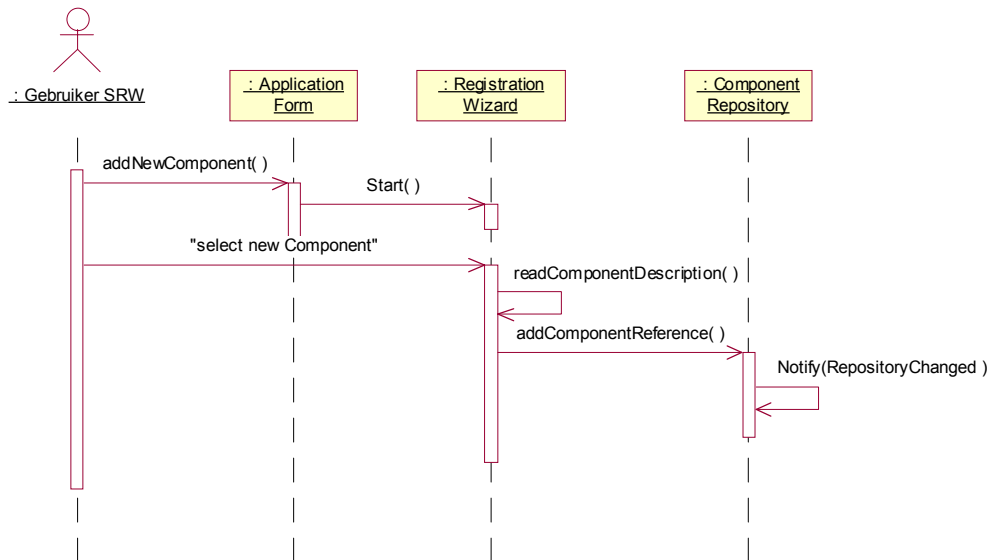
Voor het uitvoeren van de beschreven Cases A en B dienen een tweetal raamwerkcomponenten geregistreerd te worden. Daarnaast kan een overbodige test-modelapplicatie verwijderd worden.

6.2.1 Toevoegen van raamwerkcomponenten

De gebruiker geeft via een menu-optie aan dat hij een nieuw component wil toevoegen. Het raamwerk zal deze vraag doorgeven aan de Registratie-Wizard. In de getoonde dialoog dient de gebruiker de te registreren frameworkcomponent aan te geven, door het selecteren van een bestand. De wizard kan van de component de eigenschappen opvragen in de vorm van een ComponentDescription. Indien in deze ComponentDescription voldoende informatie is opgenomen en deze informatie correct is zal van de geselecteerde component daarna een ComponentReference gemaakt worden. Indien de registratieprocedure succesvol is verlopen, wordt in het geval dat het geregistreerde component een PKBT of een DataEngine is direct een instantie van het geregistreerde component gemaakt door het

Framework. Dit omdat er binnen het SR altijd een instantie van een PKBT en een DataEngine moeten zijn om een case te kunnen samenstellen of uitvoeren.

Het registratieproces van de modelapplicatie SWAP is schematisch weergegeven in het onderstaande sequence diagram. De registratie van de grafiekentool verloopt analoog hieraan.



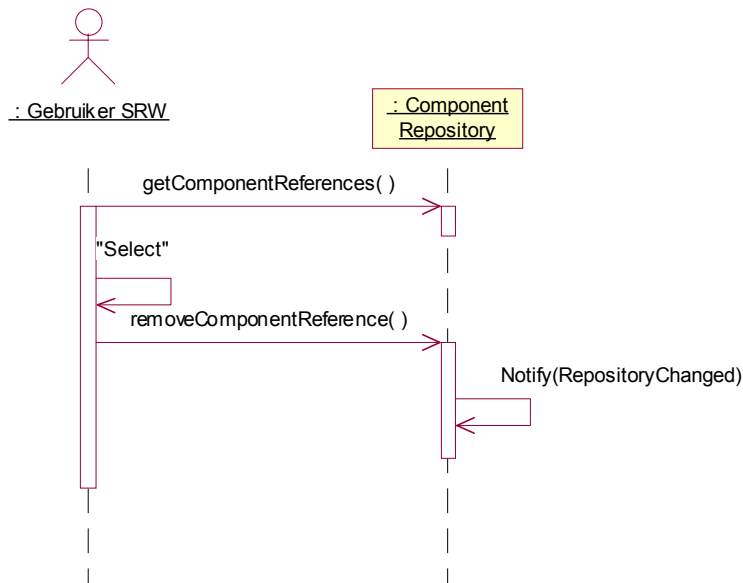
Figuur 6-1: Registratie van modelapplicatie SWAP in de SR applicatie

6.2.2 Verwijderen van raamwerkcomponenten

Een component dat eenmaal geregistreerd is in de SR-applicatie kan ook weer uit het SR verwijderd worden. Het raamwerk kan een lijst met geregistreerde componenten geven waarna de gebruiker het te verwijderen component selecteert. Het raamwerk zal de component repository de opdracht geven dit component te verwijderen. Indien nog instanties van dit component actief zijn wordt hiervan melding gemaakt en wordt de procedure afgebroken.

Bij het verwijderen van BuildingFC's zoals modelapplicaties en generieke tools dient de gebruiker er rekening mee te houden dat cases waarin het te verwijderen BuildingFC is opgenomen niet meer geopend kunnen worden nadat de component verwijderd is en niet vervangen wordt door een compatible component met een overeenkomende service descriptor. Een raamwerkcomponent wordt uniek geïdentificeerd door de naam en het versienummer dat in de ComponentDescriptor is opgenomen. Binnen de case wordt deze identificatie gebruikt.

In het onderstaande voorbeeld wordt de referentie naar de test-modelapplicatie uit het voorbeeld verwijderd uit het raamwerk.



Figuur 6-2: Verwijderen referentie naar test-modelapplicatie

6.3 Samenstellen van een case

Het samenstellen van een case bestaat uit de volgende activiteiten:

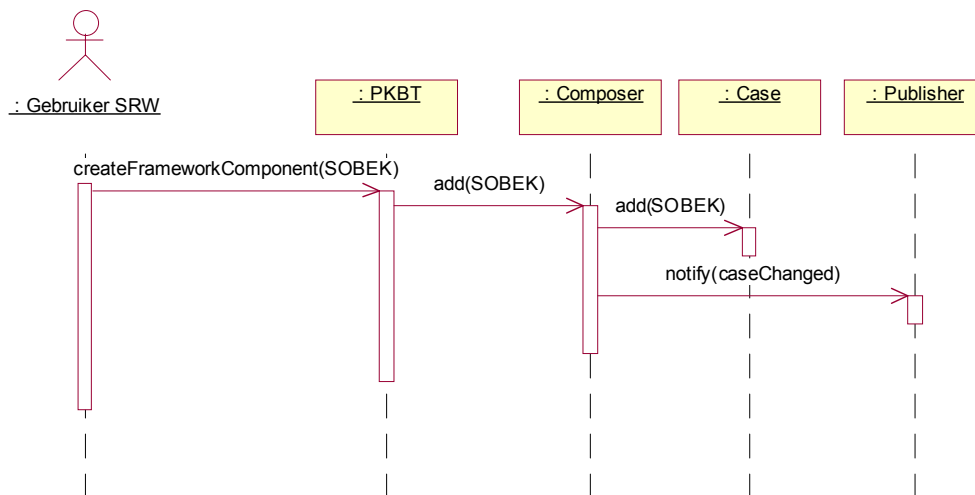
1. Instantiëren BuildingFrameworkComponents:
Een gebruiker instantieert een BuildingFC door een referentie van de ComponentRepository naar het canvas van de PKBT te slepen.
2. Verbinden BuildingFrameworkComponents:
Op het canvas kunnen twee BuildingFC's verbonden worden door een Connection-object hiertussen te plaatsen. De pijlrichting geeft de volgorde van initialiseren aan en de volgorde van starten indien dit niet uit data-uitwisseling afgeleid kan worden.
3. Invoeren van component-specifieke instellingen:
Indien een BuildingFC eigen instellingen vereist voordat gestart kan worden dient de BuildingFC hiervoor een editor te bieden. Deze editor wordt gestart na dubbelklikken op een component op het canvas. Mogelijke instellingen hierbij zijn een rekentijdstap, locaties naar bestanden en handmatig in te voeren parameters.
4. Definiëren eigenschappen van verbindingen:
Analoog aan het starten van editors van BuildingFC's kan door dubbelklikken een editor voor verbindingen (connections) gestart worden. De PKBT (de Broker) zoekt en instantieert hiervoor de SRW-Editor. Specifieke Connection instellingen zijn het type verbinding ('linked' of 'in-process') en eventuele condities geldend voor de verbinding.
5. Definiëren data-uitwisseling tussen BuildingFrameworkComponents:
Bij het vormgeven van een verbinding tussen BuildingFC's speelt het vastleggen van data-uitwisselpunten een belangrijke rol. Op deze manier kunnen de achterliggende rekenkernen immers gegevens uitwisselen. De SRW-Editor kan met behulp van het Connection-object de aansluitpunten van de te koppelen schematisaties presenteren. Door deze aansluitpunten van de schematisaties te koppelen worden Exchangeltms gecreëerd.
6. Invoeren case-specifieke instellingen:

In de eerste versie van SR zullen als specifieke instellingen binnen de case de begin- en eindtijd van de simulatie ingevoerd kunnen worden. De case kan bewaard worden waarbij een unieke naam ingevoerd moet worden. Hierbij worden zowel de case-specifieke instellingen bewaard als alle instellingen behorende bij de BuildingFC's en de verbindingen.

Aan de hand van de voorbeeld cases uit paragraaf 6.1 worden deze activiteiten uitgewerkt in de onderstaande sequence diagrammen.

1. Toevoegen BuildingFrameworkComponents.

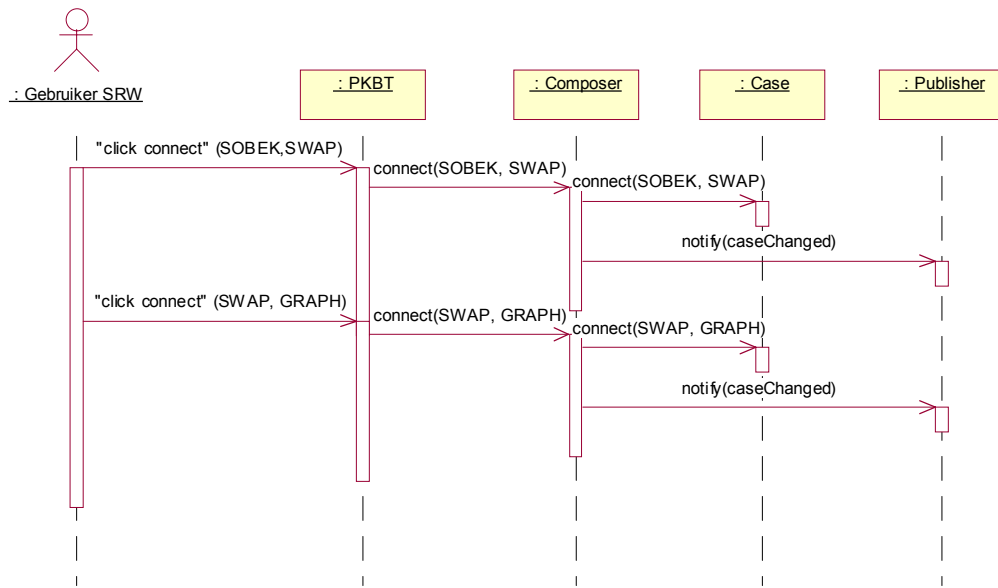
In Figuur 6-3 is het toevoegen van SOBEK aan het canvas van de PKBT uitgewerkt. Deze operaties worden geïnitieerd als de gebruiker een component uit de ComponentRepository op het canvas van de PKBT sleept. Het toevoegen van SWAP en een grafiekentool verloopt hetzelfde, alleen parameterwaarden zijn dan afwijkend.



Figuur 6-3: Toevoegen van SOBEK aan het canvas van de PKBT

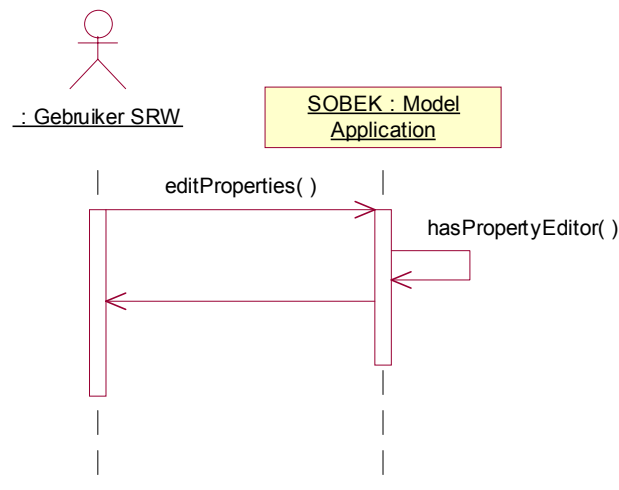
2. Verbinden van BuildingFrameworkComponents.

De geïntanceerde componenten SOBEK, SWAP en de grafiekentool moeten verbonden worden om de PKBT voldoende informatie te geven over de volgorde van starten en het kunnen vastleggen van data-uitwisseling tussen de componenten. In het onderstaande sequence diagram is uitgewerkt hoe de betrokken componenten samenwerken bij het vastleggen van een verbinding.



Figuur 6-4: Verbinden van modelapplicaties SOBEK, SWAP en de grafiekentool

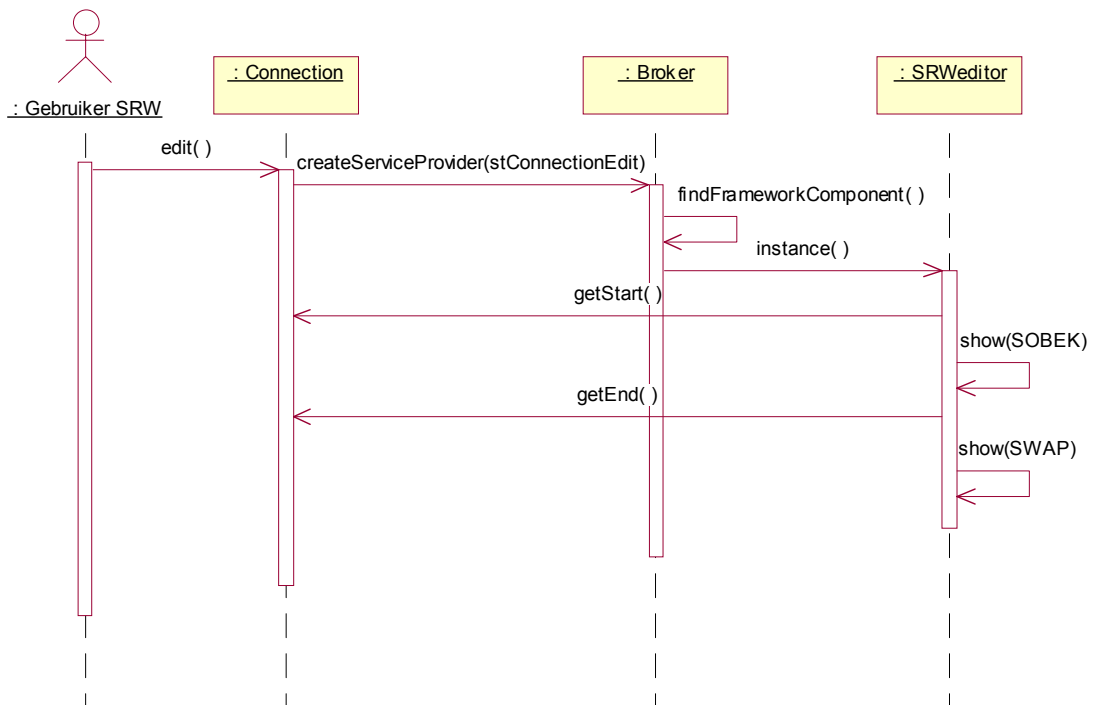
3. Invoeren component specifieke instellingen.
Voor zowel de afzonderlijke BuildingFC's als de verbindingen daartussen kunnen specifieke instellingen vereist zijn. Een BuildingFC kan hiervoor een eigen editor aanbieden die gestart wordt mbv de methode *edit()*. Een voor de hand liggende instelling is de rekentijdstap voor een modelapplicatie. In het onderstaande diagram is het starten van een specifieke editor voor SOBEK weergegeven. Een editor wordt gestart door dubbelklikken op een component op het canvas van de PKBT.



Figuur 6-5: Starten editor voor SOBEK

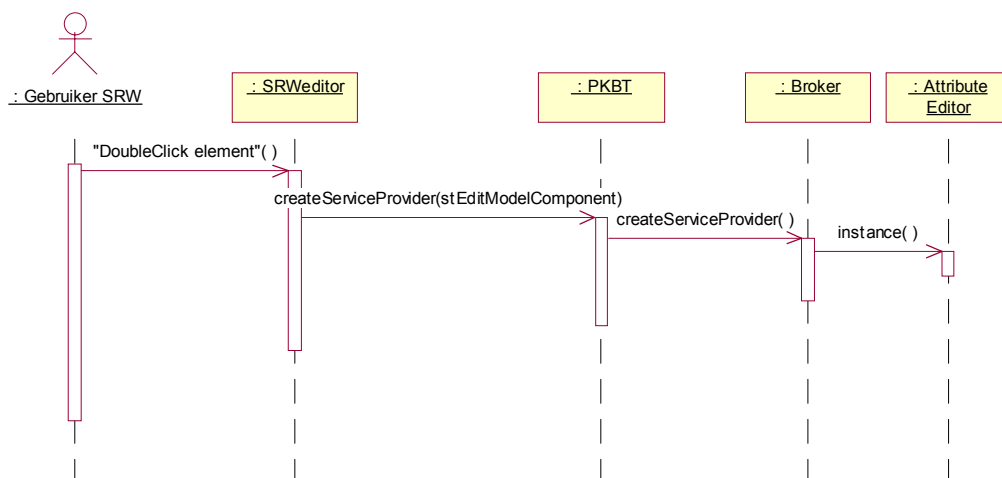
4. Definiëren eigenschappen van verbindingen.
Voor het vastleggen van eigenschappen van verbindingen wordt een generieke editor ter beschikking gesteld in de vorm van de SRW-Editor.

De Broker binnen de PKBT zoekt en instantieert deze editor als de gebruiker op een verbinding op het canvas dubbelklikt.

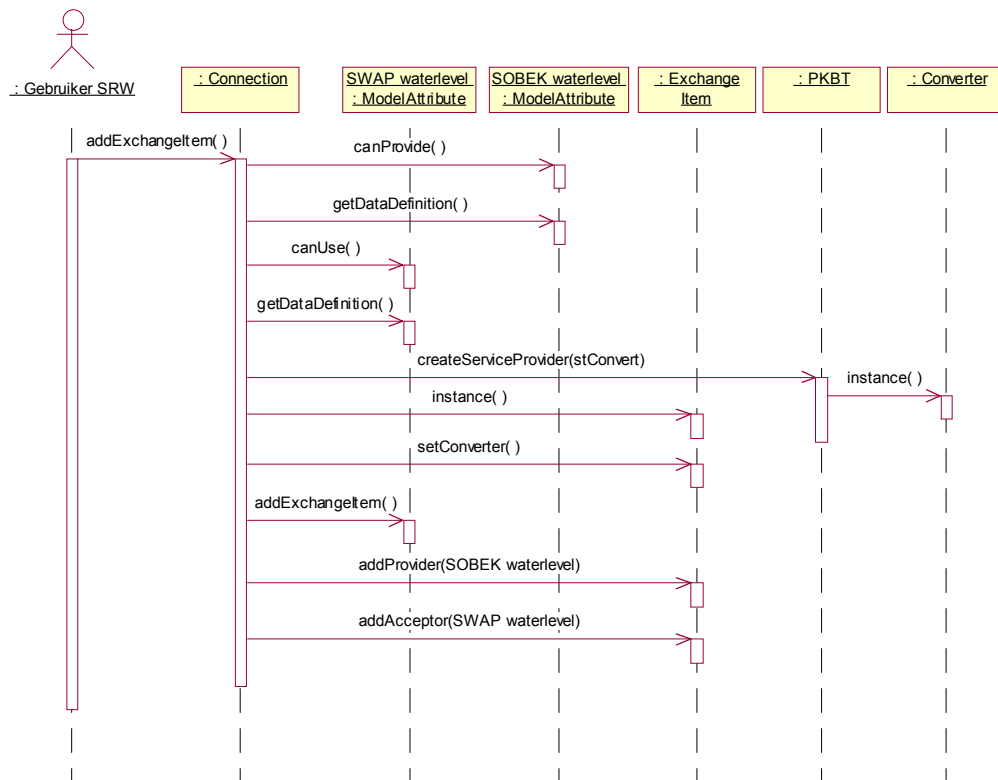


Figuur 6-6: Starten SRW-Editor voor specificatie van verbinding

- Definiëren data-uitwisseling tussen componenten.
Zodra de SRW-Editor gestart is worden de schematisaties getoond van de verbonden modelapplicaties (indien een modelapplicatie en een generieke tool verbonden worden wordt uiteraard slechts één schematisatie gepresenteerd). Het vastleggen van Exchangeltems tussen de aansluitpunten van SOBEK en SWAP wordt in onderstaande sequence diagrammen weergegeven. Hierbij wordt de uitwisseling van waterstanden uit SOBEK naar SWAP vastgelegd.



Figuur 6-7: Selectie van Modelcomponent en attribuut binnen de SRW-Editor

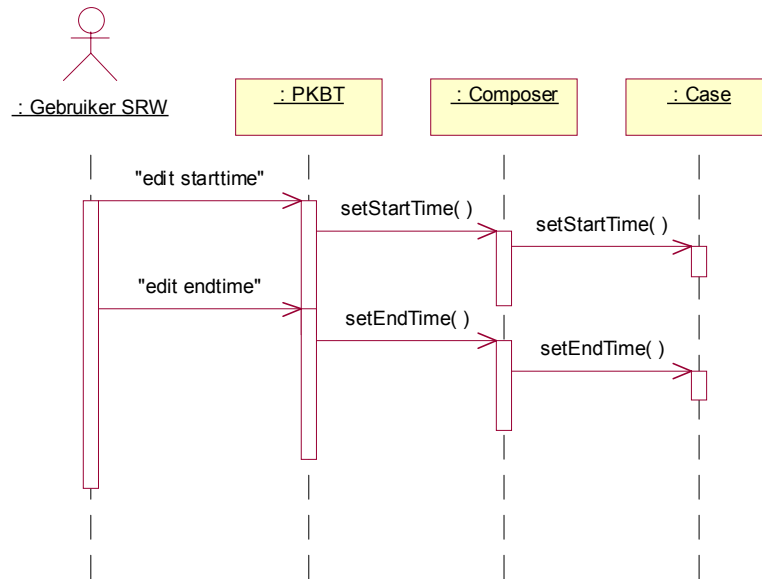


**Figuur 6-8: Koppeling van SOBEK en SWAP aansluitpunt
(uitwisseling waterstanden)**

Het definiëren van exchangelitems, zoals in Figuur 6-8 is uitgewerkt wordt afhankelijk van het aantal gewenste koppelingen herhaald. Indien een 1-n koppeling gewenst is worden eerst meerdere modelcomponenten geselecteerd. Per exchangelitem zijn daarnaast nog instellingen mogelijk die de wijze van uitwisselen verder specificeren.

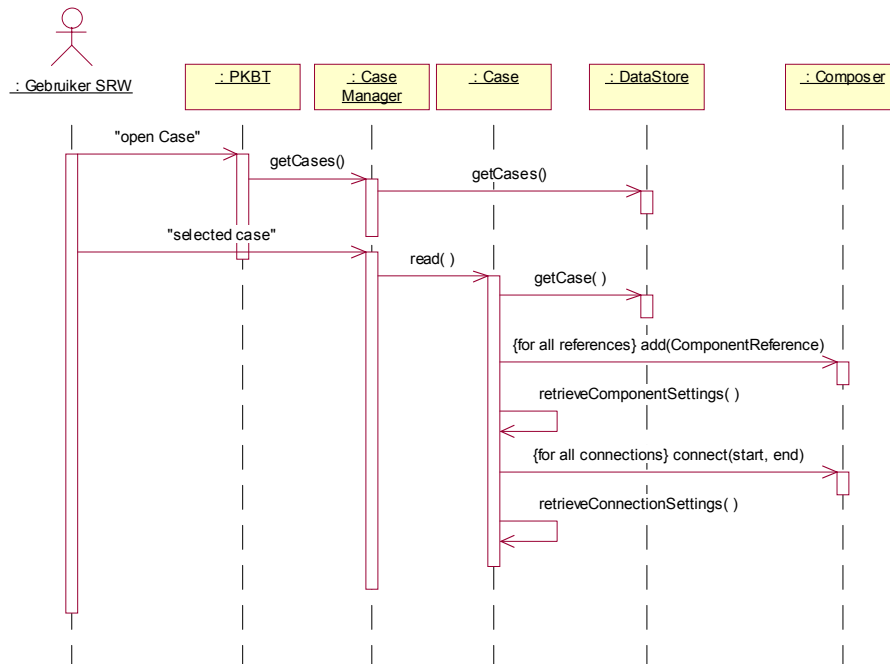
6. Invoeren case-specifieke instellingen.

Voordat de case gestart kan worden zijn nog enkele instellingen vereist. In de eerste versie betreft het de specificatie van de start- en eindtijd van de simulatie. Via de PKBT kan deze instelling ingevoerd worden. De PKBT zorgt ervoor dat dit in de case beschrijving terecht komt (een gebruiker schrijft nooit rechtstreeks in de case definitie).



Figuur 6-9: Instelling start- en eindtijd van de simulatie.

Alle voorgaande beschreven stappen worden grotendeels ‘overgeslagen’ danwel achter de schermen uitgevoerd indien de gebruiker een bestaande case wilt inlezen. De eerste versie van SR beschikt over een zeer eenvoudige CaseManager die in de PKBT is opgenomen. Deze CaseManager kan een overzicht van alle eerder gemaakte cases presenteren. Een selectie door de gebruiker zorgt voor het ophalen van de case met behulp van de DataEngine.



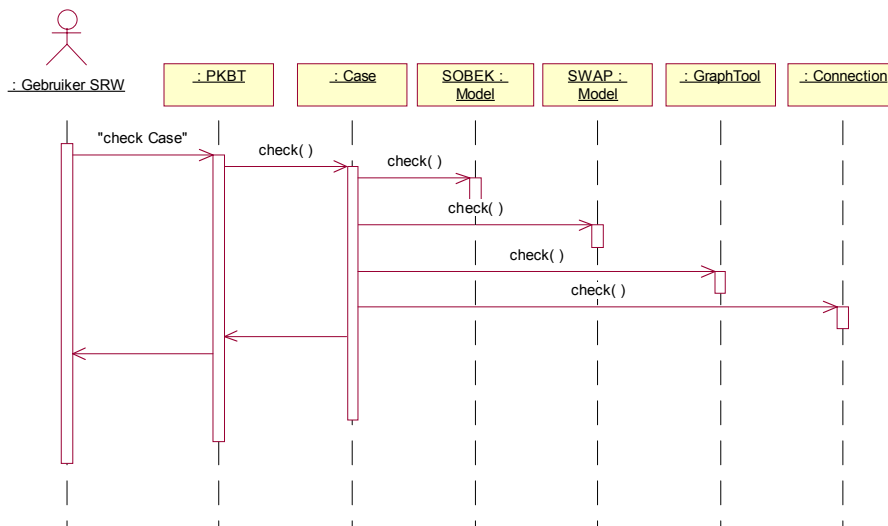
Figuur 6-10: Inlezen bestaande case.

6.4 Uitvoeren van simulaties

Het uitvoeren van simulaties bestaat allereerst uit het controleren van de uitvoerbaarheid van de case. Indien deze controle met positief resultaat afgesloten wordt kan de case gestart worden. Daarvoor dient de volgorde van aansturen van de verschillende BuildingFC's door de PKBT bepaald te worden. De status van de verschillende BuildingFC's wordt tijdens de simulatie visueel weergegeven op het canvas van de PKBT, de Composer wordt daarom tijdens de simulatie door de Publisher op de hoogte gehouden.

6.4.1 Controleren case

Zodra de gebruiker een case wil uitvoeren wordt deze eerst door de PKBT gecontroleerd. De PKBT roept hiervoor de methode *check()* van het case-object aan. Het case-object delegeert deze methode aan alle BuildingFC's en verbindingen. De controles kunnen ook tijdens het samenstellen van de case per BuildingFC en verbinding individueel aangeroepen worden. In onderstaande figuur is de controle van voorbeeld-case A uitgewerkt.



Figuur 6-11: Controle van een case.

6.4.2 Bepalen rekenvolgorde

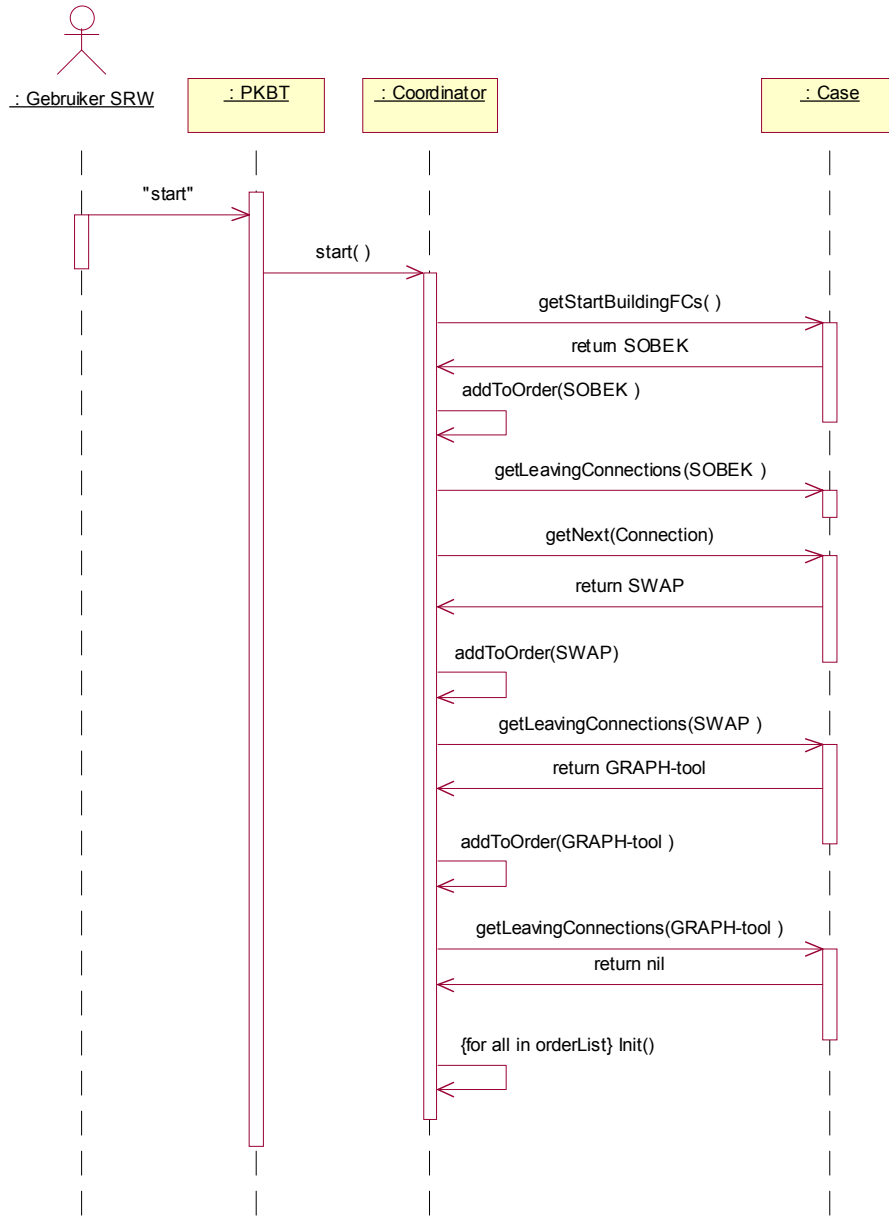
Het bepalen van de rekenvolgorde binnen de case is een taak van de Coördinator van de PKBT. De rekenvolgorde wordt bepaald door de samenstelling van de modellentrein en eventueel geldende condities bij verbindingen. De Coördinator maakt dus gebruik van de gegevens die in het case-object zijn opgenomen. Het case-object weet van elke BuildingFC wat de ingaande en uitgaande verbindingen zijn. Gegeven een verbinding kan het case-object de start- en eind-BuildingFC geven (dit houdt het Connection-object immers zelf bij).

Het bepalen van de rekenvolgorde is verdeeld in twee onderdelen:

1. Volgorde van initialiseren:

De coördinator gaat er vanuit dat BuildingFC's zonder ingaande verbindingen altijd geïnitieerd kunnen worden. In de voorbeeld cases (zie paragraaf 6.1) is dat modelapplicatie SOBEK. Nadat deze BuildingFC('s) geïnitieerd is, wordt met behulp van alle uitgaande verbindingen bepaald welke volgend component gestart kan worden.

In de voorbeeld cases is dat modelapplicatie SWAP. Bij een vertakte modellentrein zijn uiteraard meerdere uitgaande verbindingen aanwezig.



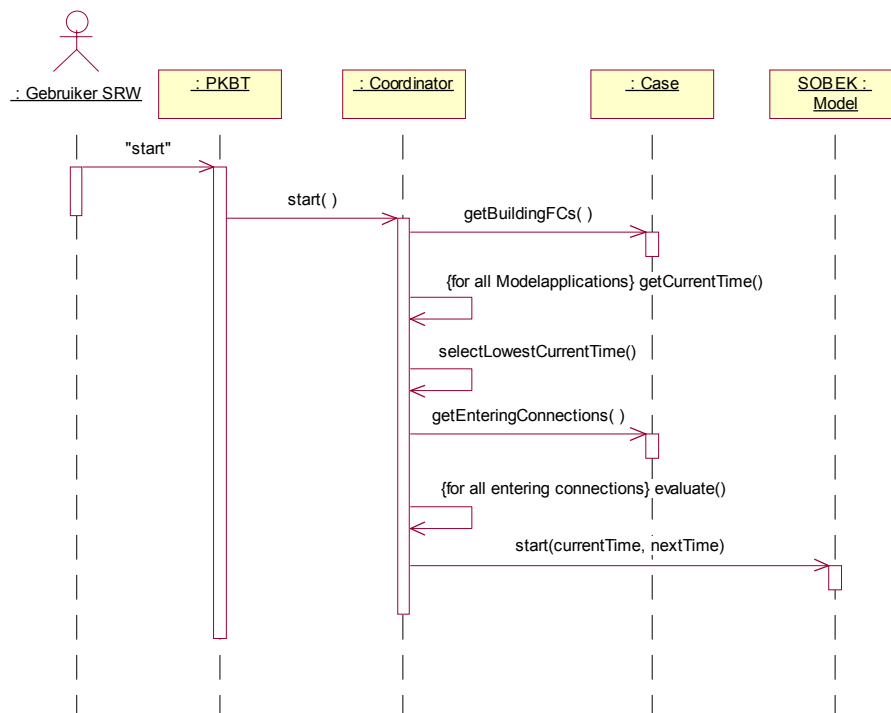
Figuur 6-12: Volgorde voor initialisatie.

2. Volgorde van uitvoeren volgende rekentijdstappen:
 Na initialisatie wordt de volgorde van aanroepen van BuildingFC's bepaald door de rekentijdstap van elke BuildingFC en de verbindingen tussen de BuildingFC's.
 Er worden door de Coördinator een tweetal regels geëvalueerd:
 - *Welke modelapplicatie heeft de laagste currenttime?*
 Aan elke modelapplicatie kan gevraagd worden wat zijn interne tijd is, hiervoor biedt BuildingFC de methode *getCurrentTime()*. De

modelapplicatie waarbij deze currenttime het laagst is, kan altijd worden gestart, omdat alle uitwisselingsdata beschikbaar is. Bij de generieke tools in de case verloopt dit anders. Een 'InProgress' tool wordt niet expliciet gestart door de Coördinator. In dit geval reageert de tool op events van de Publisher. Indien de tool 'linked' is opgenomen wordt de tool wel gestart door de Publisher, maar pas nadat alle voorgaande modelapplicaties de totale case hebben afgerond.

- *Wordt aan de condities en eigenschappen van de ingaande verbindingen voldaan?*

Op basis van condities of specifieke instellingen in verbindingen kan bepaald worden dat de start van een BuildingFC niet plaats mag vinden (als aan conditie niet wordt voldaan) of dat de start moet worden uitgesteld (de ingaande verbinding is 'linked'). In voorbeeld case A wilt de gebruiker dat SOBEK rekent van tijdstip 0 tot 100 en vervolgens SWAP van 0 tot 100, terwijl na 24 tijdstappen van SOBEK op basis van de eerste regel (laagste nexttime) SWAP zou kunnen rekenen.



Figuur 6-13: Rekenvolgorde bepalen voor de voorbeeld cases

Volgens bovenstaand mechanisme kan voor de voorbeeld cases de volgorde van starten worden bepaald:

- Case A:
(SOBEK-SWAP-GRAPH gelinkte koppeling, alleen uitwisseling van waterhoogten)
 1. Start SOBEK (2400 keer)
 2. Start SWAP (100 keer)
 3. Start GRAPH-Tool (1 keer)
- Case B:



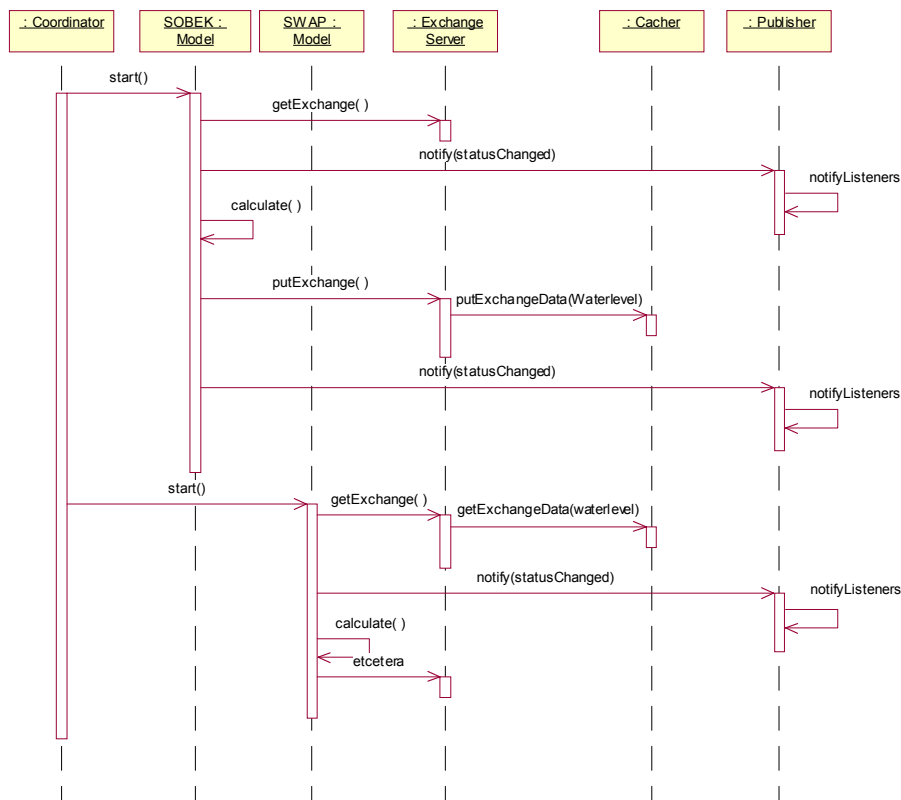
(SOBEK-SWAP-GRAPH InProcess koppeling, uitwisseling van waterhoogten en drainagefluxen)

1. Start SOBEK (24 keer);
2. Start SWAP (1 keer)
3. Automatische start (op basis van events) van GRAPH-Tool

Herhaling van drie bovenstaande stappen 100 keer.

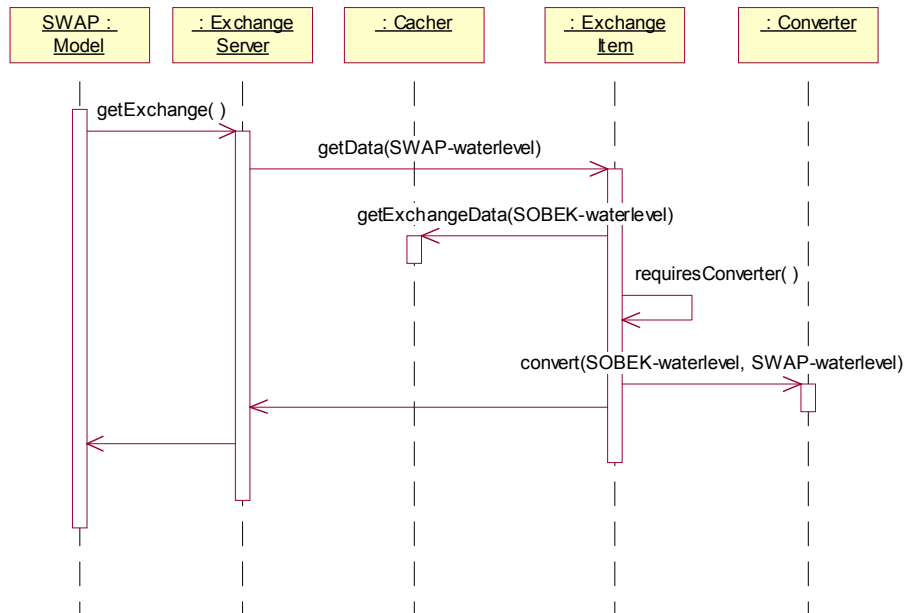
6.4.3 Uitvoering simulatie

Bij het uitvoeren van de simulatie, het starten van de modelapplicaties en generieke tools in de juiste volgorde, speelt het uitwisselen van data tussen de componenten een belangrijke rol. De modelapplicaties zorgen er zelf voor dat de uitwisselingsdata aan de ExchangeServer van de DataEngine worden aangeboden. De modelapplicatie die deze data als invoer nodig heeft leest dit uit de ExchangeServer. De leverende en vragende attributen kunnen hiervoor van een gedeeld stukje werkgeheugen gebruik maken. Voor de uitwisseling en de daarbij eventueel vereiste conversie wordt het ExchangeItem-object gebruikt wat met de attributen geassocieerd is. De coordinator dient op de hoogte gehouden te worden van de voortgang van berekeningen, zodat op het juiste moment een volgende component gestart kan worden. BuildingFrameworkcomponents maken daarvoor gebruik van events, om aan de PKBT duidelijk te maken dat een statusovergang heeft plaatsgevonden. De PKBT kan deze events opvangen en de juiste vervolgacties initiëren met behulp van de Publisher. Een standaard vervolgactie is het op de hoogte stellen van de Composer (binnen de PKBT), zodat de nieuwe status van de componenten gevisualiseerd kan worden.



Figuur 6-14: Uitwisseling van waterhoogten tussen SOBEK en SWAP.

Binnen de ExchangeServer wordt gebruik gemaakt van de instellingen die in het ExchangeItem-object zijn ingevoerd. Zowel de omrekening van attribuutwaarden indien het totaal verdeeld moet worden over meerder ontvangers als het aanroepen van een eventueel vereiste converter vindt hier plaats.



Figuur 6-15: Ophalen van SOBEK waterhoogten voor gebruik binnen SWAP.



7 Infrastructuur

7.1 Inleiding

Bij de implementatie van een component gebaseerde architectuur van een softwaresysteem kan elke component in een willekeurige programmeertaal ontwikkeld worden. Componenten communiceren op basis van interfaces, alleen het kunnen aanspreken van deze interfaces is dus van belang voor het functioneren van het gehele systeem. Om er voor te zorgen dat een component de interface van een ander component kan benaderen dient echter een afspraak gemaakt te worden op basis van welk protocol deze communicatie plaats gaat vinden. Van deze communicatiestandaarden voor componenten, ook wel Middleware genaamd, bestaan een aantal varianten. De meest gebruikte standaarden zijn DCOM, CORBA en RMI. Dit hoofdstuk geeft een kort overzicht van deze standaarden met de belangrijkste voor- en nadelen. Door criteria met betrekking tot het Standaard Raamwerk te vergelijken met deze voor- en nadelen kan een keuze gemaakt worden voor de meest geschikte variant.

7.2 Communicatiestandaard en distributie

Beschrijvingen en vergelijkingen van communicatiestandaarden zijn in het verleden al veelvuldig gemaakt. Daarom is in literatuur en op internet voldoende informatie hierover te verkrijgen. Binnen dit technisch ontwerp wordt gebruik gemaakt van (een vertaling van) een artikel, te vinden op [http://developer.earthweb.com/news/techfocus/022398_dist1.html].

7.2.1 Het gedistribueerde object model

Object georiënteerde technologie richtte zich in het verleden met name op single-user omgevingen. Doordat applicaties steeds complexer en client/server technologie de standaard werd, is er een behoefte ontstaan om componenten (objecten) te kunnen gebruiken in een gedeelde multi-user omgeving. Initieel is dit probleem aangepakt door het gebruik van databases. Objecten kunnen in principe opgeslagen worden in een database, opgehaald en gebruikt door de ene gebruiker, weer opgeslagen in de database en daarna door een andere gebruiker weer gebruikt worden. Een object model is echter niet simpel om te zetten (mappen) naar een conventionele (relationele) database. Het alternatief zijn gedistribueerde objecten, waarbij de verschillende objecten draaien op meerdere computers die via een netwerk met elkaar in verbinding staan. Tezamen vormen deze objecten de applicatie.

In vergelijking met grote mainframe applicaties bieden gedistribueerde object technologieën een aantal voordelen. Gedistribueerde object applicatie (ontwikkeling) maakt het mogelijk om:

- Functionaliteit van bestaande systemen te hergebruiken.
- Onafhankelijke ontwikkeling en implementatie van componenten zonder dat dit effect heeft op andere componenten. De gehele applicatie wordt onderverdeeld in zelf-voorzienende componenten die onafhankelijk van elkaar kunnen werken. De componenten onderling zijn in staat om met andere componenten samen te werken. Om deze samenwerking mogelijk te maken communiceren alle componenten via een gedeeld protocol of interface.



- Effectief onderhoud van de code en systematische verspreiding van updates. Het aanpassen van een component heeft geen effect op de werking van andere componenten. Hierdoor is de kans op potentiële fouten geringer.
- Lichte client interfaces die in verbinding staan met server programma's en databases op meerdere locaties. Hierdoor is het mogelijk een applicatie te creëren bestaande uit een groot aantal componenten welke tezamen slechts een geringe aanspraak maken op de disk-space en het geheugen van de client

7.2.2 Ondersteunende technologieën: RMI, CORBA and DCOM

Er bestaan in hoofdlijnen drie verschillende technologieën die de gedistribueerde object architectuur faciliteren. Dit zijn *RMI*, *CORBA* en *DCOM*. Er is geen simpel antwoord op de vraag welke technologie superieur is, elke technologie heeft zijn eigen unieke eigenschappen die van invloed zijn op de te nemen beslissing over de keuze van de ondersteunende technologie.

- *RMI*
RMI is de Java implementatie van de gedistribueerde object architectuur. RMI staat voor Remote Method Invocation en is opgenomen in de Java Developers Kit (vanaf JDK1.1). RMI is een methode om voor zowel client als server applicaties methoden aan te roepen via een gedistribueerd netwerk van clients en servers. Deze gebruiken hiervoor de Java Virtual Machine waardoor RMI een platform onafhankelijke architectuur is. RMI wordt in het algemeen gezien als een lichtgewicht en minder krachtige technologie als CORBA en DCOM. RMI biedt echter een aantal unieke mogelijkheden zoals het automatische management van gedistribueerde objecten en de mogelijkheid om objecten zelf van machine naar machine door te geven.
- *CORBA*
CORBA is een acroniem voor 'Common Object Request Broker Architecture' en reeds in 1990 ontwikkeld door de Object Management Group (OMG). Via CORBA kunnen methoden van objecten die zich waar dan ook op het netwerk bevinden aangeroepen worden net alsof het lijkt dat deze objecten zich op de lokale machine zouden bevinden. De schakel in de gedistribueerde object architectuur van CORBA zijn de Object Request Brokers (ORBs), die de communicatie tussen de client en de server opzetten en onderhouden. De ORBs bevinden zich op zowel de client als de server machines. Hierdoor kunnen objecten van de client diensten vragen van objecten aan de server kant, zonder dat de client objecten enige kennis hebben waar de server objecten zich bevinden, in welke taal deze geschreven zijn of op welk besturingssysteem deze draaien.
- *DCOM*
DCOM (Distributed Component Object Model) is sinds 1996 Microsofts architectuur voor gedistribueerde objecten. In de nieuwere versies van Microsoft producten wordt DCOM als DNA aangeduidt. DCOM is de opvolger van Network OLE. DCOM gebruikt zijn eigen netwerk protocol, de Object Remote Procedure Call (ORPC). DCOM is architecturaal zodanig opgezet dat het de sterke kanten van CORBA via een dit protocol binnen de Microsoft omgeving kan toepassen. DCOM is ook taalafhankelijk, gebruikt de beveiliging van Windows NT en kan via



verschillende transport protocollen (CP/IP, UDP/IP, IPX/SPX, AppleTalk en HTTP) functioneren.

7.2.3 Vergelijking tussen de drie gedistribueerde object technologieën

- **DCOM**
DCOM is gebouwd rondom een bewezen desktop componenten architectuur. Op COM gebaseerde applicaties zijn robuust en hebben een goede performance. Door de integratie met programmeertalen en -tools wordt de ontwikkel inspanning positief beïnvloed. DCOM is daardoor een voor de hand liggende keuze voor een Microsoft geïntendeerde omgeving. Indien een besturingssysteem anders dan Microsoft NT of WIN9x gebruikt wordt in de architectuur van de applicatie dan is DCOM hoogstwaarschijnlijk niet de eerste keuze. Ondanks pogingen van Microsoft om DCOM cross-platform compatible te maken komt DCOM het beste tot zijn recht in omgevingen die uitsluitend op Microsoft producten draaien.
- **RMI**
RMI is de meest simpele en snelste methode om een gedistribueerde object architectuur op te zetten. Het is een goede keuze voor RAD-prototypes en kleine applicaties die volledig in Java geïmplementeerd zijn. Het nadeel van RMI ten opzichte van DCOM en CORBA is dat het minder robuust en schaalbaar is. Bijvoorbeeld RMI kan alleen met ander Java RMI componenten communiceren, waardoor het niet mogelijk is bijvoorbeeld legacy applicaties te integreren.
- **CORBA**
CORBA heeft een zeer complete en goed opgezette architectuur. CORBA lijkt de meest effectieve tool voor de ontwikkeling van grote gedistribueerde applicaties met een open architectuur. CORBA en DCOM tonen grote overeenkomsten in geboden functionaliteit, echter DCOM mist een belangrijke element, multi-platform ondersteuning.

7.2.4 Vergelijkingscriteria

Bij de vergelijking van de gedistribueerde object technologieën zijn een aantal selectiecriteria als kenmerken van de technologieën geïdentificeerd. Elke technologie voldoet in meer of mindere mate aan deze criteria. In onderstaande tabel wordt hiervan een overzicht gegeven. De uiteindelijke technologie-keuze kan gemaakt worden door het toekennen van wegingsfactoren aan de criteria voor SR.

Criterium	RMI	DCOM	CORBA
Robuustheid	+	+	++
Schaalbaarheid	-	+	+
Performance	0	+	+
Compleetheid	0	++	+
Platform onafhankelijkheid	++	--	++
Programmeertaal onafhankelijkheid	-	+	++
Licentiekosten server	++	++	-
Licentiekosten client	0	0	0

Tabel 7-1: Criteria voor gedistribueerde object technologie



8 Conclusies

Het technisch ontwerp voor de eerste versie van het Standaard Raamwerk Water is een beschrijving van de componenten binnen de eerste SR-applicatie. Deze beschrijvingen bestaan uit:

- Interfaces van alle componenten:
De interface-beschrijvingen zijn het meest belangrijk voor een goed functionerende SR-applicatie. Door bij alle componenten de beschreven interfaces te implementeren wordt de gewenste samenwerking tussen componenten gegarandeerd.
- Interne objectstructuur van de complexe componenten:
Bij de belangrijkste en meest complexe componenten is naast de interface-beschrijving ook een beschrijving gemaakt van de interne structuur. Dit biedt de programmeur van een dergelijke component ondersteuning bij het realiseren van de gewenste functionaliteit.
- Samenhang tussen componenten:
De samenhang die tussen componenten is beschreven biedt allereerst bij de implementatie van SR-applicaties benodigde informatie om componenten op de juiste manier samen te laten werken. Daarnaast bieden deze dynamische views een mogelijkheid de opgestelde interfaces te verifiëren. Ontbrekende operaties in een interface worden onder andere opgespoord bij deze uitwerking.

Naast dit document bestaat het eindproduct van het technisch ontwerp uit een model binnen de CASE-tool Rational Rose. Dit model bevat alle componenten en diagrammen en is in eerste instantie bedoeld voor de programmeurs van componenten. Het is onder andere mogelijk vanuit deze tool code te genereren op basis van de gedefinieerde interfaces.

De architectuur is zodanig opgezet dat het raamwerk zelf slechts als container voor raamwerkcomponenten dient. De raamwerkcomponenten worden verdeeld in Basic FrameworkComponents (BasicFC's) en Building FrameworkComponents (BuildingFC's). BasicFC's bieden ondersteunende functionaliteit bij het samenstellen en/of uitvoeren van een case. De belangrijkste specialisaties zijn de procesketen beheertool, de DataEngine en de SRW-Editor. BuildingFC's zijn de bouwstenen voor een case, waarbij onderscheid gemaakt wordt tussen modelapplicaties en generieke tools. Modelapplicaties zijn de schillen (zogenaamde wrappers) rond een rekenkern en een schematisatie (of meerdere schematisaties waaruit gekozen kan worden), generieke tools bieden functionaliteit zoals visualisatie en calibratie.

De procesketen beheertool (PKBT) faciliteert het samenstellen van cases en het aansturen van modelapplicaties en generieke tools. Deze expliciete aansturing van componenten wordt aangeduid als het push-mechanisme. Dit is een modificatie ten opzichte van de Architectuur SR [SR-A]. De uitwisseling van gegevens tussen componenten wordt niet expliciet geïnitieerd, dit wordt impliciet bij de start van modelapplicaties en generieke tools uitgevoerd. Deze impliciete overdracht (ook wel conversie of projectie



genoemd) kan gezien worden als een pull-mechanisme. Dit mechanisme komt overeen met de Architectuur SR.

Modelapplicaties kunnen tot een zelf te kiezen detailniveau in- en uitvoergegevens laten beheren door de DataEngine. Hiervoor moet gebruik gemaakt worden van een geregistreeerde datastructuur binnen deze DataEngine. Het is zowel mogelijk om elke attribuutwaarde door de DataEngine te laten beheren als het laten beheren van bijvoorbeeld alleen bestandsnamen van modelinvoer of –uitvoer. Naast deze functionaliteit zorgt de DataEngine voor persistente opslag van cases en voor het ter beschikking stellen van stukjes werkgeheugen (cachiers) voor uitwisselingsdata middels een zogenaamde ExchangeServer.

Bij het definiëren van cases speelt het projecteren van schematisaties van modelapplicaties een belangrijke rol. Hiervoor biedt de eerste versie van SR de SRW-Editor. Deze editor maakt het mogelijk de aansluitpunten van schematisaties weer te geven en de uit te wisselen attributen vast te leggen. Deze functionaliteit leunt sterk op de in de Architectuur SR gedefinieerde generieke schematisatiestructuur op basis van ModelElementen en Connectoren.

Om componenten te laten communiceren is het van belang dat er een communicatiestandaard word afgesproken. Voor de meest toegepaste standaarden CORBA, DCOM en RMI gelden een aantal voor- en nadelen. De standaarden voldoen dus in een bepaalde mate aan belangrijke criteria op het gebied van gedistribueerde technologie. Afhankelijk van het belang dat aan bepaalde criteria wordt gesteld binnen SR kan een keuze gemaakt worden.

Op basis van dit technisch ontwerp kunnen richtlijnen worden opgesteld voor het ontwikkelen van raamwerkcomponenten en voor het gebruik van een raamwerkapplicatie. Het ontwikkelen van modelapplicaties en het doordacht opzetten van cases binnen een raamwerkapplicatie spelen hierbij de belangrijkste rol.



9 Literatuur

[SR-PVE]

van Adrichem, B. (EDS), van Geer, F., Minnema, B. (NITG-TNO), Bakema, A. (RIVM), Ubbels, A., Terveer, R., van Baalen, S. (RIZA), Bulens, J. (Staring Centrum-DLO), Noort, J. (STOWA) en J. Stout (WL-Delft Hydraulics), 1998, Programma van Eisen Standaard Raamwerk Water, ISN 90-5773-136-3, Stichting Toegepast Onderzoek Waterbeheer (STOWA), Utrecht, werkdocument 2001-W-03, Rijkswaterstaat, Rijksinstituut voor Integraal Zoetwaterbeheer en Afvalwaterbehandeling (RIZA), Lelystad, werkdocument 2001.124X, 16p.

[SR-PVA]

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

[SR-P]

s.n., 2000, Standaard Raamwerk Water, Projectplan, 28 april 2000, EDS, Alterra, Geodan IT, TNO, WL|Delft Hydraulics, 19p.

[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.

[Gijsbers et al]

Gijsbers, P., R. Brinkman, D. Levelt en M. van Elswijk, 2000, DelftWISE Specificaties, versie 0.903, WL | Delft Hydraulics, s.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.

[BP]

Best practices in distributed object application development, Tom Albertson,
http://developer.earthweb.com/news/techfocus/022398_dist1.html

Bijlage A: Gedistribueerd rekenen

Als vervolg op het huidige ontwerp kan gedacht worden aan een systeem, waarin gedistribueerd wordt gerekend. In het huidige ontwerp is dit concept niet uitgewerkt, maar bij de genomen ontwerpbeslissingen is er wel rekening gehouden met deze toekomstige uitbreiding. In deze bijlage wordt geschetst waar uitbreidingen aan het huidige ontwerp vereist zijn en wat in deze uitbreidingen dient te worden uitgewerkt.

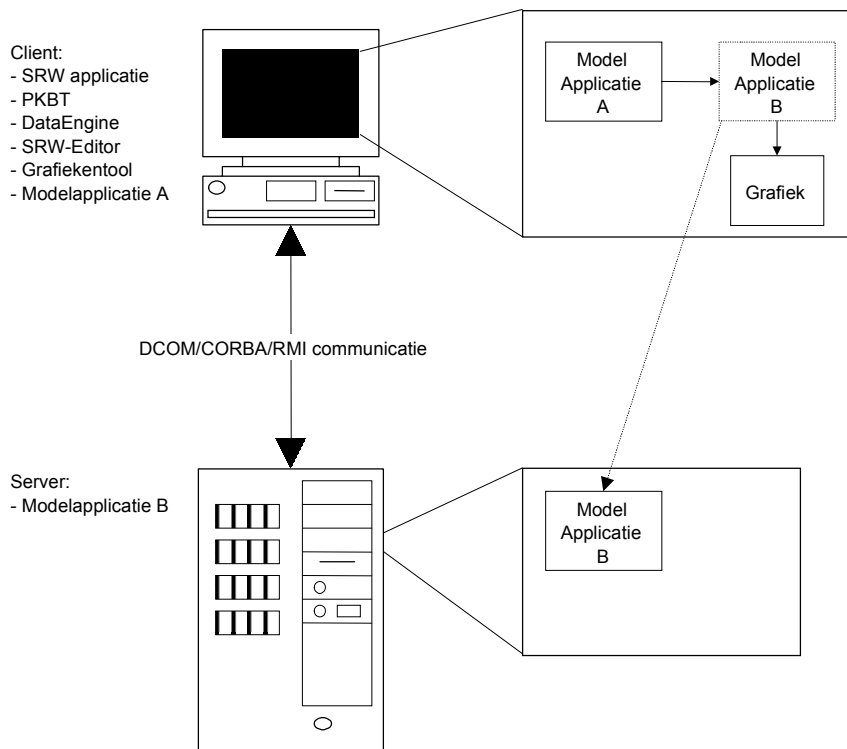
Er zijn globaal een tweetal configuraties denkbaar bij een gedistribueerde SR-applicatie:

1. Rekenkern (en eventueel de gehele modelapplicatie) draaien op een server.

Indien alleen een rekenkern, die opgenomen is in een modelapplicatie, draait op een server is in principe geen aanpassing van het huidige ontwerp vereist. De modelapplicatie, die op een client machine draait, kan in dat geval zelf de communicatie met de rekenkern op de server regelen.

Indien de gehele modelapplicatie op een server draait, bepaalt het gekozen communicatie-protocol tussen componenten (zie hoofdstuk 7) of een aanpassing aan de huidige architectuur vereist is.

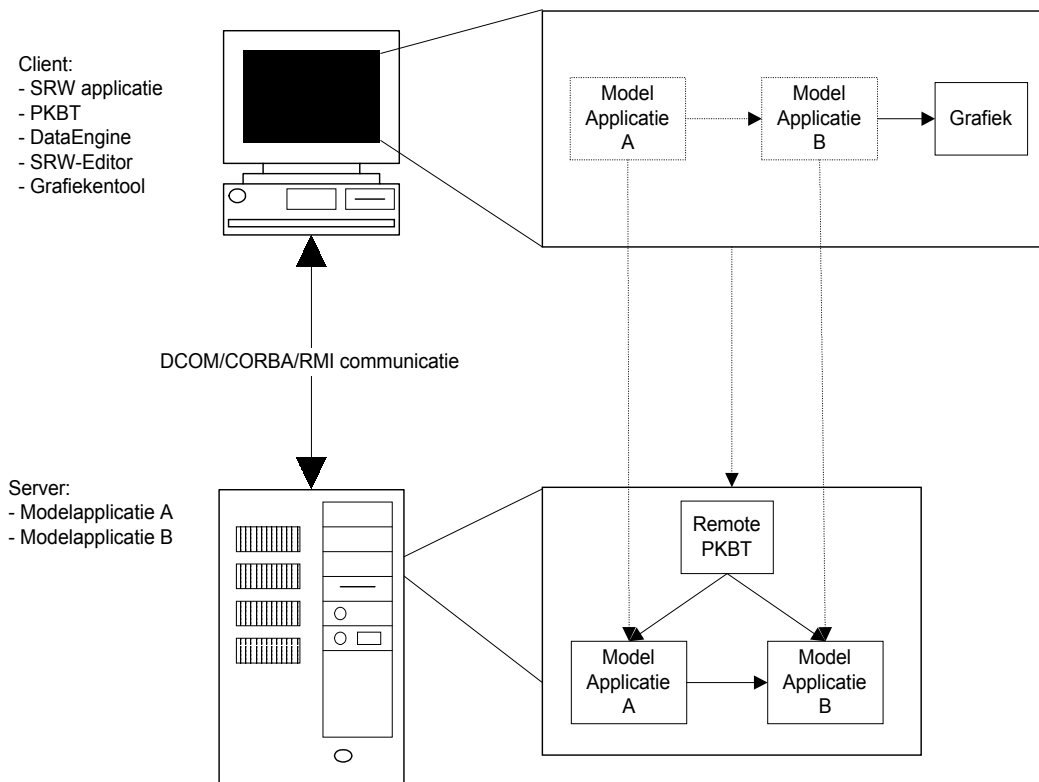
Communicatiestandaarden als DCOM, COBRA en RMI bieden voldoende functionaliteit om de vereiste communicatie te realiseren.



Figuur A-1: Gedistribueerde configuratie waarbij ModelApplicatie B op een server staat maar aangestuurd wordt door de PKBT op de client, geheel op basis van de huidige architectuur.

2. Meerdere modelapplicaties draaien op een server en wisselen onderling gegevens uit.

Indien meerdere modelapplicaties op een server geplaatst worden is het qua performance interessant om de vereiste uitwisseling van data tussen deze modelapplicaties geheel op de server af te handelen. Als elke rekentijdstep events naar de PKBT op de client verstuurt worden en uitwisselingsdata via de DataEngine op de client doorgegeven worden, wordt een intensief netwerkverkeer gecreëerd. De voordelen voor het draaien van zware modelapplicaties op snelle servers worden hiermee in grote mate teniet gedaan.



Figuur A-2: Gedistribueerde configuratie waarbij meerdere modelapplicaties op een server draaien waardoor een server-side PKBT en evt. DataEngine vereist is.

Het voorkomen van veelvuldig netwerkverkeer tussen de client en server vereist coördinatie van de modelapplicaties A en B op de server. Hierdoor wordt dus een sub- of remote PKBT denkbaar die op de server draait, maar aangestuurd wordt door de hoofd-PKBT op de client. Hierdoor wordt het grote aantal events, dat tussen client en server verstuurd wordt, voorkomen.

Naast het minimaliseren van netwerkverkeer, veroorzaakt door events die naar de PKBT gestuurd worden, dient de uitwisseling van data tussen de modelapplicaties op de server geheel op de server uitgevoerd te worden, anders wordt alle uitwisselingsdata via de client van modelapplicatie A naar B verstuurd. Dit vereist dus een sub- of remote DataEngine op de server.

Zowel de toepassing van een sub-PKBT als een sub-DataEngine is niet in het huidige ontwerp opgenomen. Deze uitbreidingen zijn te realiseren



door bijvoorbeeld een sub-PKBT als BuildingFrameworkComponent te ontwerpen en in een case op te nemen. Voor een sub-DataEngine zijn een aantal mogelijkheden zoals twee gesynchroniseerde DataEngine's op client en server of een DataServer met een remote exchange-server. De voor- en nadelen hiervan vereisen aanvullende studie.