EMERGO Toolkit 2.0

EMERGO Deliverable 5.1
Technologische keuzes

Auteurs: Hub Kurvers, Aad Slootmaker
EMERGO penvoerende instelling:
Open Universiteit Nederland
- Onderwijstechnologisch Expertisecentrum
- Faculteit Psychologie
- Faculteit Natuurwetenschappen
- Ruud de Moor Centrum

EMERGO partner instellingen:
Universiteit Maastricht, Faculteit Psychologie
Radboud Universiteit, Faculteit Managementwetenschappen
Universiteit Utrecht, Faculteit Geowetenschappen

Datum: 24 april 2006
Versie 0.9
Kenmerk: U2006/2751

Onderwijstechnologisch expertisecentrum

OpenUniversiteitNederland

1. Introductie

In dit document geven we een verantwoording van de technologische keuzes die we binnen het Emergo project maken.
Nog niet alle keuzes kunnen nu worden gemaakt, omdat de eisen nog niet helemaal bepaald zijn. Daarom zal dit document later in het project geactualiseerd worden.

Emergo wordt een complexe applicatie: complexe casussen, multimediaal, levensecht, met game elementen.
Om de student te confronteren met de gevolgen van zijn handelen, zal de reactie van Emergo op gebruikersacties sterk gestuurd worden door veelal complexe condities (regels die wel of niet af gaan en bepaalde programma acties tot gevolg hebben).
Ook moeten studenten mede-studenten kunnen inschakelen voor reflectie, weten welke andere studenten on-line zijn en in waar zij mee bezig zijn en moeten docenten kunnen volgen hoe studenten de casussen doorlopen. Dit pleit voor een centrale database voor de opslag van al dit soort gegevens.
Emergo zal een invoer-, afspeel- en beheeromgeving kennen, waarin de verschillende rollen hun eigen taken kunnen uitvoeren. Het invoeren en afspelen zal door meerdere personen op verschillende locaties gebeuren.

Emergo zal uiteindelijk worden opgeleverd als een domeinonafhankelijke toolkit, die door alle SURF-leden zal kunnen worden gebruikt en uitgebreid. De ambitie is dat de toolkit de basis zal vormen voor applicaties die de komende jaren zullen worden ontwikkeld.

Een complexe applicatie die uitbreidbaar moet zijn, door zo veel mogelijk instellingen gebruikt kan worden en langere tijd mee moet kunnen.
Dat stelt eisen aan de keuze van de applicatiearchitectuur, de keuze van het ontwikkelplatform/taal en afspeelplatform, de keuzes van applicatieframeworks (een samenhangende set componenten die de basis vormen voor een deel van de applicatie) en de keuzes van bestaande al ontwikkelde componenten.
Deze keuzes zijn onderling afhankelijk. Met name de keuze voor het ontwikkelplatform bepaalt de keuzemogelijkheden van frameworks en componenten.

Om onze keuzes te kunnen maken zijn we uit gegaan van een set criteria die verrassend genoeg veelal toepasbaar is op alle keuzes, van applicatiearchitectuur tot componenten.
Daarom beginnen we met een paragraaf waarin deze criteria worden behandeld.
Daarna volgt nog een drietal paragrafen over de (nog te maken) keuzes van resp. applicatiearchitectuur, ontwikkelplatform/afspeelplatform/taal en applicatieframeworks en componenten.
We sluiten af met de conclusies: keuzes die we hebben gemaakt en die we nog moeten maken.

2. Keuzecriteria

Om onze keuzes te kunnen maken zijn we uit gegaan van set criteria die verrassend genoeg veelal toepasbaar zijn op alle keuzes, van applicatiearchitectuur tot componenten.

Community

Het gaat hierbij zowel om de community van ontwikkelaars als van gebruikers.
Wat is de kwaliteit van de community? Hoe groot is de community?
Een grotere community is min of meer een garantie voor een grotere betrouwbaarheid (proven technology), meer bestaande componenten, een snelle respons bij problemen (oplossen van bugs) en een doorontwikkeling (nieuwe verbeterde versies en uitbreidingen). Ook is er de optie om gebruik te kunnen maken van de expertise van anderen, c.q. samen te werken met anderen.
Dit hoeft niet alleen te gaan over software, maar kan ook duiden op het aantal toepassers van een bepaalde architectuur en hoeverre die architectuur nog wordt doorontwikkeld.
Ook speelt mee wat andere ontwikkelaars, binnen het HO, hebben gekozen.

Levensduur

Het kan zowel gaan om de levensduur van een ‘leverancier’ (dit kan een bedrijf, instelling of community zijn) als van een technologie. Met Emergo willen we een technologische basis leggen waar toekomstige applicaties weer jaren op kunnen teren.

Innovatie

Is er doorontwikkeling of staat de ontwikkeling stil? Wordt meegegaan met nieuwe ontwikkelingen? Worden die geimplementeerd?
Een ‘leverancier’ hoeft niet achter de laatste hypes aan te lopen, maar moet wel bewezen technologie meenemen, zodat toekomstige applicaties hiervan kunnen profiteren.

Inwerktijd

De inwerktijd moet niet te groot zijn, dit gaat ten koste van de applicatie als geheel.
Hier speelt dan ook sterk mee wat we al kunnen, maw welke technologische keuzes we in voorgaande projecten hebben gemaakt. Aan de andere kant moeten we ook vernieuwen. Hiertussen moet een goede balans zijn.

Integratie

In hoeverre kan bijv. een framework of een component worden geintegreerd in Emergo?
In hoeverre kan Emergo worden geintegreerd in de elo’s van de SURF-instellingen? Dit laatste stelt eisen aan het te kiezen onwikkel- en afspeelplatform en mogelijk aan de architectuur.

Requirements

Deliverables 1.2 en 1.3 zullen eisen opleggen aan technisch ontwerp en bouw van Emergo en daarmee met name aan aan te schaffen componenten.
Oa: Emergo moet een client-server applicatie worden, Emergo moet voor gebruikers zo eenvoudig mogelijk zijn, Emergo moet eenvoudig uitbreidbaar zijn met andere componenten.

Kosten

Als de aanschaf van een bepaald ontwikkelplatform of bepaalde componenten niet binnen het project budget vallen, kunnen we die keuze niet maken. Ook kunnen we toekomstige gebruikers (onderwijsinstellingen) van Emergo niet opzadelen met hoge kosten.

Voorkeuren

Naast bovengenoemde criteria, hebben we ook nog persoonlijke voorkeuren die de keuzes kunnen beinvloeden.

We willen ons zelf ontwikkelen door laatste proven technology toe te passen.

We willen liefst gebruik maken van een OO omgeving/taal met klassen en overerving.

We willen ervaring opdoen met het toepassen van design patterns.

We willen ervaring opdoen met webservices en service oriented architecture.

3. Applicatiearchitectuur

Bij de ontwikkeling van software bepaalt de gekozen architectuur de structuur van de applicatie. In feite is de architectuur in het technisch ontwerp het raamwerk waarbinnen de functionele componenten worden opgehangen. Zeker bij complexe applicaties als EMERGO is het van belang een weloverwogen architectuurkeuze te maken, en die in het technisch ontwerp goed uit te werken.

Bij de keuze van een architectuur moeten we rekening houden met het volgende:

- de architectuur moet de functionele eisen zo veel mogelijk ondersteunen
- de architectuur moet leiden tot een logisch en begrijpbaar opgebouwde applicatie
- de architectuur moet de applicatie niet onnodig complex maken
- de architectuur mag een goede performance niet in de weg staan
- de architectuur moet leiden tot een goed onderhoudbare applicatie
- er moeten frameworks en ontwikkelplatforms zijn die de implementatie van de architectuur ondersteunen (in hoofdstukken 4 en 5 zullen we dit verder uitwerken)

Van de in hoofdstuk 2 genoemde keuzecriteria zijn in het bijzonder de volgende van belang:

- we willen een client-server-applicatie maken, met een gemeenschappelijke database voor alle gebruikers
- we willen eenvoudig (eventueel al bestaande) componenten aan het systeem kunnen toevoegen; een al bestaande component zou in principe in een willekeurige technologie geïmplementeerd kunnen zijn, als hij maar beschikt over een standaard interfaceprotocol (we krijgen dan een gedistribueerd, misschien zelfs GRID-achtig systeem)
- we willen dat de applicatie zoveel mogelijk platformonafhankelijk is; een gebruiker zou bijvoorbeeld zowel onder Windows als onder Linux de applicatie moeten kunnen gebruiken
- we willen eenvoudig applicatiedata kunnen uitwisselen met bestaande systemen, bijvoorbeeld een Student Informatie Systeem, of een overkoepelende ELO
- we willen dat onderdelen van de applicatie eenvoudig vervangen kunnen worden; de userinterface zou bijvoorbeeld zoveel mogelijk onafhankelijk van de rest van de applicatie aanpasbaar moeten zijn; hetzelfde geldt ook voor de gemeenschappelijke database
- we willen dat de gekozen architectuur zijn waarde in het verleden heeft bewezen; we moeten erop kunnen vertrouwen dat de architectuur geschikt is voor complexe applicaties als EMERGO
- we willen niet te veel tijd besteden aan inwerken in een ons tot dusver onbekende architectuur, of daarmee samenhangende onbekende frameworks of ontwikkelplatforms

We zullen nu eerst enkele veel gebruikte architectuursystemen bespreken.

1. <u>pipes en filters</u>

   Een aantal aan elkaar geschakelde componenten (filters) transformeert inputdata naar outputdata. De verbindingen (pipes) tussen de verschillende filters vertalen de output van een filter naar de input voor een ander filter.

   Voordelen van het gebruik van deze architectuur zijn:

   - begrijpelijkheid: de applicatie wordt opgedeeld in een aantal kleine componenten, elk met een goed gedefinieerde data-invoer en –uitvoer
   - herbruikbaarheid: componenten kunnen in principe op meerdere plaatsen in het proces worden ingezet
   - onderhoudbaarheid en uitbreidbaarheid: we kunnen filters gemakkelijk aanpassen; we kunnen nieuwe filters toevoegen; we kunnen paden verleggen
   - specialistische en gedetailleerde systeemanalyse is mogelijk: we kunnen één of enkele componenten isoleren en nader analyseren
   - parallelle verwerking: de filters kunnen in principe “gelijktijdig” hun taken verrichten

Dit soort architecturen zijn per definitie geschikt voor bulkverwerking van data. In Emergo zouden we een dergelijke architectuur misschien kunnen toepassen in het gedeelte dat logging-informatie verwerkt.

2. data-abstractie en objectgeoriënteerde organisatie

In deze architectuur is een applicatie een verzameling met elkaar communicerende objecten. Een object vertegenwoordigt een samenhangende set data plus functionaliteit om die data te bewerken. Via interfacefunctionaliteit kunnen andere objecten de data impliciet beïnvloeden.

We kunnen in deze architectuur de interne werking van een object naar believen aanpassen zonder andere objecten te hoeven wijzigen. De architectuur is zeer geschikt als we een probleem kunnen opdelen over een aantal met elkaar communicerende agents.

Als een object van identiteit verandert, of als de interfacefunctionaliteit wordt aangepast moeten we ook alle objecten die potentieel met het aangepaste object kunnen communiceren aanpassen. Een object kan beïnvloed worden door meerdere andere objecten. Hierdoor kunnen er onoverzichtelijke situaties ontstaan.

3. getriggerde, impliciete aanroep

In deze architectuur melden componenten zich bij starten van de applicatie bij het “systeem” aan als belangstellenden voor bepaalde gebeurtenissen. Het systeem zal dan, wanneer zo’n gebeurtenis optreedt, een melding sturen naar de betreffende componenten. Door het laten optreden van een gebeurtenis, of “event”, kan een component aldus “impliciet” functionaliteit in andere componenten activeren.

Een groot voordeel van de architectuur is, in tegenstelling tot de objectgeoriënteerde architectuur, het gemak waarmee we nieuwe componenten kunnen toevoegen, of bestaande componenten kunnen vervangen of aanpassen. We hoeven dan meestal alleen de eventregistratie-lijst aan te passen. Wijziging van een bestaande component kan vaak zonder andere systeemelementen te hoeven wijzigen.

Bij complexe applicaties met veel op events reagerende componenten zijn de effecten van het optreden van een event vaak onoverzichtelijk. De volgorde van reageren door de verschillende componenten is vaak willekeurig, en die componenten kunnen zelf ook weer events uitsturen die invloed hebben op andere componenten. In systemen waarin resourcedata gedeeld wordt door componenten, kan het beheer van die data complex zijn, en kunnen er performanceproblemen optreden. En tenslotte is het, vanwege de onoverzichtelijkheid, moeilijk te bewijzen dat de applicatie correct functioneert.

Een variant van deze architectuur, die men tegenwoordig veel gebruikt bij de ontwikkeling van web-applicaties, is de Service-Oriented Architecture of SOA. De componenten (meestal gebouwd volgens de Web services-technologie) zijn zelfstandige eenheden, vaak zelfs aparte (deel-)applicaties die in verschillende ontwikkelomgevingen kunnen zijn geïmplementeerd. Ze hebben wel een gemeenschappelijk standaard interfaceprotocol. Meestal zal de userinterface-component de events triggeren, het “systeem” zorgt er dan via het interfaceprotocol voor dat de juiste componenten het event afhandelen.

4. gelaagde systemen

In gelaagde systemen bestaat de applicatie uit een aantal lagen van componenten. Binnen een laag kunnen de componenten vrijelijk met elkaar communiceren. Tussen 2 lagen zijn er interfaceprotocollen: de onderliggende laag biedt diensten aan aan de bovenliggende “client”-laag. Soms is er alleen sprake van communicatie tussen opeenvolgende lagen, in andere systemen mag een laag ook communiceren met verderaf gelegen lagen.

Gelaagde systemen zijn gemakkelijk te verbeteren. Verbetering van één laag betekent meestal dat hooguit de bovenliggende en onderliggende laag nog enigszins moeten worden aangepast, maar niet alle andere lagen. Ook zijn lagen gemakkelijk herbruikbaar en vervangbaar. Een applicatie is daardoor gemakkelijk geschikt te maken voor gebruik in een andere omgeving.

Het opdelen van een applicatie in lagen is niet altijd gemakkelijk, en soms zelfs onmogelijk. Ook performance kan een zodanig probleem zijn dat layer-protocollen doorbroken moeten worden.

De bekendste van dit soort systemen is de Model-View-Controller-architectuur (MVC). De hoofdvariant hiervan bestaat uit 3 lagen: de view-laag, die de user-interface implementeert, de model-laag, die applicatielogica bevat, en vaak een gekoppeld is aan een externe database, en de controller-laag, die de functionele componenten van de applicatie verbindt aan de view-laag. De lagen dienen zoveel mogelijk technisch van elkaar gescheiden te zijn, zodat men bijvoorbeeld gemakkelijk een ander userinterface kan implementeren, of een andere database kan gebruiken, zonder dat dat invloed heeft op de andere lagen.

5. depots

Depotsystemen bestaan uit 2 verschillende componenttypen: een centrale dataopslagplaats waarin de toestand van het systeem wordt bijgehouden, en een verzameling onafhankelijke componenten die de data bewerken.

Er zijn verschillende subcategorieën van dit soort systemen. Zo kan het initiatief tot aanpassing van de toestand van het systeem bij de componentenverzameling liggen. Het depot is dan vaak een traditionele database. In andere gevallen ligt het initiatief bij de centrale opslagplaats. Het gaat dan om zogenaamde “blackboard”-modellen. De componenten zijn onafhankelijke “kennisbronnen”, het depot is een “blackboard”. Veranderingen in de toestand van het blackboard activeren een controlecomponent (vaak, maar niet noodzakelijk, onderdeel van het blackboard zelf). Deze controlecomponent stuurt de kennisbronnen aan, die op hun beurt de blackboard-data weer kunnen aanpassen.

Blackboard-architectuur wordt wel toegepast in signaalverwerkingstoepassingen, zoals spraak- en patroonherkenning, in agentsystemen, en in programmeeromgevingen.

Er zijn nog een groot aantal andere architecturen, veelal specifiek gericht op bepaalde domeinen (interpretersystemen, gedistribueerde processen, hoofdprogramma-subroutine-systemen, toestand-

transformatie-systemen, etc.). Deze zijn niet direct toepasbaar binnen Emergo, we zullen er niet nader op ingaan.

Meestal zien we meerdere architecturen binnen 1 applicatie. Het combineren vindt plaats op 3 manieren (die vaak door elkaar worden gebruikt). Architecturen kunnen hiërarchisch van elkaar verschillen: op macroniveau is er bijvoorbeeld een gelaagd systeem, maar op microniveau, dus binnen een laag, werkt men objectgeoriënteerd. Architecturen kunnen ook naast elkaar gebruikt worden. Het ene deel van de applicatie werkt volgens een lagenstructuur, het andere deel bestaat uit componenten die via impliciete aanroepen met elkaar interageren. Tenslotte kan een component nog via verschillende architecturen aan andere onderdelen gekoppeld zijn. Een component kan enerzijds gebruikmaken van een centraal depot, en tegelijkertijd anderzijds via een pipeline gekoppeld zijn aan andere componenten, en bovendien via weer een andere interface bestuurd worden door een controlecomponent.

Gezien eerder genoemde criteria ligt het bij Emergo voor de hand uit te gaan van een gelaagd systeem. Het is dan mogelijk om eenvoudig bijvoorbeeld de database-laag of de View-laag te wijzigen zonder de rest van de applicatie aan te passen. Er zijn in het verleden bovendien goede ervaringen opgedaan met de Model-View-Controller-architectuur, met name in de DU-webapplicaties BaMaS en Espace (een systeem waarin studenten kunnen zoeken naar vervolgopleidingen op de bachelor-opleiding die ze hebben afgerond, respectievelijk een systeem waarin studenten opdrachten uitwerken en elkaar feedback geven).

Als we gemakkelijk functionaliteit willen kunnen toevoegen aan Emergo, dan zullen we de architectuur zodanig moeten opzetten dat toevoeging of aanpassing van functionaliteit zo weinig mogelijk invloed zal hebben op de al aanwezige andere functionaliteit. De mate waarin dat mogelijk is hangt samen met het soort functionaliteit. De grootste groep componenten wordt gevormd door het gereedschap dat studenten gaan gebruiken bij de aanpak van een casus, maar er zullen ook componenten zijn die betrekking hebben op de besturing van het systeem, en componenten die als “hulpcomponent” door andere componenten worden aangeroepen (toners van tekstuele of andere informatie, conditie-analysatoren, scorecomponenten), en tenslotte zijn er “verzamel”- of monitorcomponenten, die logging- en statusgegevens vertalen naar zinvolle informatie. Vooral het gereedschap zal functionaliteit bevatten die voor een groot gedeelte onafhankelijk is van andere componenten. Kijken we naar de architectuur, dan willen we per component de Model-View-Controller-architectuur toepassen, en de componenten onderling via getriggerde aanroepen met elkaar laten samenwerken.

Om ook te voldoen aan de eis van eenvoudige interactie met andere systemen, en om onszelf de gelegenheid te geven kennis te verkrijgen van een voor ons nieuwe technologie, zouden we een SOA-architectuur kunnen gebruiken om componenten met elkaar te laten communiceren. We zouden dan een Web services-gedeelte kunnen toevoegen aan de Controller-laag zodat eenvoudig verschillende View-lagen mogelijk zijn.

4. Ontwikkelplatform/taal en afspeelplatform

Bekende combinaties van ontwikkelplatform/taal zijn J2EE/Java, Delphi/Pascal en .NET/C#.
Aan de keuze van een ontwikkelplatform zit vaker de keuze van een operating system (bijv. Windows of Linux) vast. Dit geldt bijv. voor Delphi en .Net, die alleen draaien op Windows. J2EE en bijv. PHP draaien zowel op Windows als Linux.

Het ontwikkelplatform bepaalt vaak ook het afspeelplatform.
Dit geldt bijv. voor Delphi en .Net voor zover clientapplicaties worden ontwikkeld. Deze draaien alleen op Windows. J2EE ondersteunt wel ontwikkeling van clientapplicaties op meerdere platforms (oa Windows en Linux), maar hierbij zijn er vaak vervelende afhankelijkheden van de JVM (Java Virtual Machine). Gebruikers moeten een bepaalde versie van de JVM geinstalleerd hebben, wil de applicatie draaien. Ook het bouwen van een complexer userinterface is geen sinecure.
Ondersteunt het ontwikkelplatform het bouwen van webapplicaties, dan kan de browser gebruikt worden als afspeelplatform en spelen deze problemen veel minder. Enkele jaren geleden waren er nog vrij grote verschillen tussen de verschillende browsers, soms zodanig dat browser-afhankelijke code moest worden geschreven. De meest gangbare browsers zijn echter steeds meer naar elkaar toe gegroeid zodat dit probleem niet meer speelt. En er zijn browsers beschikbaar voor de belangrijkste operating systemen.

Integratie/gebruik bij andere SURF-instellingen is een belangrijk criterium. We kunnen hieraan zo veel mogelijk tegemoet komen als we ontwikkelplatform/taal en afspeelplatform zodanig kiezen, dat ze zo veel mogelijk operating systemen ondersteunen.

Uitgaande van dit criterium vallen Delphi, .NET en andere OS afhankelijke platforms af.
J2EE en andere minder OS afhankelijke platforms genieten de voorkeur.
Als we kijken naar de TIOBE Programming Community Index voor april 2006 (een index voor de populariteit van programmeertalen, http://www.tiobe.com/tiobe_index/index.htm), zien we dat Java het meest populair is, ruim 21%, en dat de populariteit het afgelopen jaar met ruim 4% is gestegen. Dit lijkt dus een goede kandidaat.
J2EE/Java scoort goed op al onze criteria:

- Er is een grote ontwikkelaars community. Ook valt J2EE/Java binnen de technische randvoorwaarden genoemd in het handboek voor technische standarden van de DU (http://www.digiuni.nl): een instelling waarin zowel universiteiten als hogescholen participeren en die dus een redelijke afspiegeling is van het Nederlands hoger onderwijs. Ook binnen OUNL is J2EE/Java een breed gebruikt platform.
- Met de levensduur zit het ook wel goed, het aantal gebruikers stijgt nog volgens de TIOBE index.
- Er is nog voldoende innovatie, gezien ontwikkelingen als JSF (Java Server Faces) en JDO (Java Data Objects, de vele Apache initiatieven en SourceForge producten.
- De grote populariteit geeft de beste garantie voor integratie/gebruik bij andere instellingen.
- De inwerktijd is minimaal. We hebben al twee DU web-applicaties (BaMaS en Espace) ontwikkeld op dit platform.
- Wat betreft de requirements zijn er ook geen problemen te verwachten. Bij de twee DU projecten hebben we alle requirements kunnen realiseren.
- Er zijn geen aanschafkosten en geen kosten voor gebruik door de instellingen.
- Java past in onze voorkeuren want is een OO taal met klassen en overerving. Zo'n taal is ook noodzakelijk om design patterns te kunnen toepassen. En Apache AXIS is een laagdrempelige Java implementatie van webservices (proven technology, al uitgeprobeerd in een van de DU projecten), waarbij ook integratie met bestaande systemen tot de mogelijkheid behoort.

Kijken we naar de rest van de top vijf van Tiobe (C, C++, PHP en Basic) dan hebben deze programmeertalen in ieder geval als nadeel dat het aantal gebruikers minder is en we er geen ervaring mee hebben.
Op de tweede en derde plaats staan C en C++, maar dit zijn complexe talen om aan te leren en bovendien is C aan het dalen.
PHP heeft als nadelen dat het een scripttaal is, waar OO pas later is toegevoegd. Ook is PHP specifiek geschikt voor webapplicaties.
Basic is geen OO taal en niet geschikt om webapplicaties mee te bouwen.
De andere talen hebben een aandeel van onder de 6%. Een veelbelovende runner-up is Ruby on Rails (bijna 0.5%), maar Ruby is ook een scripttaal (wel OO).

Gezien bovenstaande kiezen we voor **J2EE/Java** als **ontwikkelplatform/taal**.

Als afspeelplatform lijkt ons de browser het meest geschikt. Voordelen zijn dat zo veel mogelijk gebruikers bereikt kunnen worden, dat door gebruikers geen software hoeft te worden geinstalleerd en dat nieuwe versies eenvoudig kunnen worden uitgeleverd (door applicatie-componenten op de server te updaten).
Een client applicatie daarentegen zal altijd geinstalleerd moeten worden, ook nieuwe versies. En de client applicatie is of OS afhankelijk of (in geval van bijv. Java) is de ondersteuning voor meerdere OS niet optimaal.
Ook wat betreft de keuzecriteria scoort de browser goed:

- Er zijn zeer grote communities van gebruikers en ontwikkelaars.
- De browser is niet meer weg te denken in zowel werk- als prive-leven, dus met de levensduur zit het goed.
- Door de grote community is er nog veel innovatie: weblogs, rss, etc. Een recente ontwikkeling is Ajax (Asynchronous JavaScript And XML), waarmee de interactiviteit, snelheid en het gebruikersgemak van webpagina's zullen toenemen.
- De inwerktijd is minimaal. We hebben al twee DU projecten ontwikkeld op dit platform (gebruikmakend van JSP: Java Server Pages).
- Integratie bij de andere Surf-instellingen is mogelijk door een Emergo page op te nemen binnen een instellingspagina. Een andere optie is gebruik van webservices, waarmee Emergo data kan worden opgehaald en getoond in de huisstijl van de instelling.
- Wat betreft de requirements verwachten we ook weinig problemen, omdat de applicatie logica uitgevoerd wordt op de server en de browser alleen data zal moeten presenteren en gebruikersinput doorsturen naar de server. Mocht de gebruikers experience rijker moeten zijn, dan verwachten we dit te kunnen realiseren met het eerder genoemde Ajax. Voor het afspelen van audio en video kunnen we of Windows Mediaplayer of Real Player gebruiken. Voor het afspelen van animaties e.d. de Flash Player. Deze players zijn veelal al geinstalleerd bij gebruikers.
- Iedereen heeft al een browser dus zijn er geen aanschafkosten. Ook boven genoemde players zijn gratis.
- Wat betreft onze voorkeuren zitten we ook goed. Zodra een van de vele innovaties proven technology is kunnen we deze desgewenst toepassen.

Gezien bovenstaande kiezen we voor de **browser** als **afspeelplatform**.

5. Applicatieframeworks

Een framework is een geheel van voorschriften dat de organisatie rond het bouwen van een applicatie en het bouwen zelf vergemakkelijkt. Vaak zie je dat men naast het begrip “framework” ook het begrip “ontwikkelplatform” gebruikt. Een ontwikkelplatform is echter de bibliotheek of de verzameling van bibliotheken waarin functionaliteit zit waarmee tijdens het ontwikkelen van een applicatie kan worden gecommuniceerd met de systeemomgeving (beeldscherm/toetsenbord/muis/internet/etc.).

Voor de realisatie van de ideeën van een framework moeten we goed nadenken over de keuze van het platform. Platforms leveren meestal ondersteuning voor bepaalde frameworks, terwijl andere frameworks moeilijk gebruikt kunnen worden. Frameworks worden meestal opgesteld door een aantal samenwerkende groeperingen, om een zo groot mogelijke ondersteuning te krijgen. Ook komt het voor dat een commercieel bedrijf een framework bedenkt en daar gelijktijdig een ontwikkelplatform bij levert (denk aan Microsoft, met .NET als framework en Windows IIS plus Visual Studio als ontwikkelplatform).

Een framework voegt dus iets toe aan een programmeeromgeving. Het zegt hoe de programmacode gestructureerd moet worden. Dit heeft een aantal voordelen:

- snellere ontwikkeling van nieuwe soortgelijke applicaties
- betere onderhoudbaarheid omdat de applicaties op een standaardmanier zijn opgebouwd
- hergebruik van code wordt gestimuleerd
- de organisatie van de applicatieontwikkeling wordt verbeterd

Een framework richt zich vaak op een bepaald type van (delen van) applicaties. Ook kan het de implementatie van een bepaald type architectuur ondersteunen.

Waaraan moet een goed framework voldoen?

- het moet eenvoudig zijn; programmeurs moeten zich snel kunnen inwerken
- het moet helder zijn; de terminologie en structuur moeten logisch in elkaar zitten
- de grenzen moeten duidelijk zijn: zowel binnen de applicatie (welk gedeelte van de applicatie moet de programmeur zelf nog implementeren) als wat betreft toepassingsgebied (welk soort applicaties komen in aanmerking bij gebruik van het framework)
- het moet goed aanpasbaar/inpasbaar zijn; het moet eenvoudig te gebruiken zijn in combinatie met andere frameworks of deelapplicaties

Een framework heeft meestal één programmeertaal als uitgangspunt. Hier, ter illustratie van de overvloed aan frameworks, enkele voorbeelden:

- Java: Apache Cocoon (gebaseerd op XML en Java; te gebruiken voor implementatie van de pipeline-architectuur, bijvoorbeeld bij tekst-transformaties), Apache Struts (zie verder), Echo, Eclipse, Grails, MARF (voor audio/spraak/natuurlijke talen), Maverick, Netbeans, RIFE, SOFIA (MVC), Stripes (geen XML maar veel eenvoudiger benadering; speciaal gericht op Java 5), Wicket (event-getriggerde architectuur, maar per component MVC-architectuur), Trails, XINS, ZK (rijke UI-webapplicaties via event-getriggerde componenten, uitgaande van Ajax-technologie; te integreren met Spring en hybernate; maakt gebruik van XML Userinterface Language (XUL))
- PHP: CakePHP en Canvas (Open Source, voor bouwen van web-applicaties), Mach-II (MVC), PRADO en RNA en ZOOP (objectgeoriënteerd, webapplicaties), Seagull, Symfony, WASP,

Molins en ZNF (voor grote webapplicaties, lijken op Struts), web.framework (MVC-webapplicaties), Zend framework (MVC, robuuste high-performance-applicaties, ondersteuning voor Web services)

- C#: .NET (verschillende typen architectuur, op Windows gebaseerd)
- Ruby: Ruby On Rails (of ROR, MVC-architectuur, ondersteuning van Ajax-technologie)
- Python: Django en Turbogears (Open Source, voor bouwen van web-applicaties), Zope
- Perl: Catalyst (Open Source, voor bouwen van web-applicaties)
- ColdFusion: Model-Glue

frameworks in EMERGO

We zagen eerder dat we een MVC-lagen-architectuur willen gebruiken, eventueel aangevuld met een SOA-webservices-architectuur. Ook vinden we Java een geschikte programmeertaal. Op basis hiervan, en op grond van een vluchtige Java-MVC-frameworks-inventarisatie op internet (kijk bijvoorbeeld eens op http://www.darwinsys.com/javawebframeworks), hebben we een “shortlist” samengesteld van potentieel te gebruiken frameworks:

- Apache Struts Action Framework(2001): is al enkele jaren oud, en uitontwikkeld; ondersteunt vooral de View- en de Controller-laag; ondersteund door vele IDE-tools; veruit grootste gebruikersgroep

  voordeel is dat we hiermee in voorgaande projecten ervaring hebben opgedaan; techniek is uitontwikkeld, dus veel standaardobjecten, voorbeelden en documentatie; HTML-tag-bibliotheek schijnt erg uitgebreid te zijn vergeleken met andere frameworks; links naar andere URLs volledig onder controle; Commons Validator voor validatie is rijpe, goed uitgewerkte methode; redirecten van POST-berichten is eenvoudig

  nadelen: wordt niet meer verder ontwikkeld (onder voorbehoud; zie de opmerkingen bij WebWorks); testen is moeilijk; de afhandeling van formulieren via ActionForms is complexer dan door andere frameworks gebruikte methoden

- Spring (2004): gebaseerd op JavaBeans; goede ondersteuning van Hibernate en Java Data Objects (JDO); links naar andere URLs volledig onder controle; Spring-IDE, geen UI-tool; gebruikersgroep neemt toe

  moderner dan Struts; schijnt eenvoudiger te zijn; kent in tegenstelling tot Struts een goede ondersteuning van de Model-laag; eenvoudig te integreren met groot aantal view-componenten; Commons Validator voor validatie is rijpe, goed uitgewerkte methode; gemakkelijk te testen met behulp van “mocks”; redirecten van POST-berichten is eenvoudig

  nadelen: zeer veel configuratie in XML-bestanden; er moet veel JSP-code geschreven worden; flexibiliteit kan leiden tot minder goed gestructureerde code

- OpenSymphony’s WebWork (2004): in versie 2 (maart 2006) initiatief tot combinatie met het Struts-framework tot Struts Action 2.0; zal leiden tot 2 subprojecten: Struts Shale ten dienste van de opkomende JSF-gemeenschap, en Struts Action voor de bestaande JSP-gemeenschap; EclipseWork-IDE

geavanceerde Ajax-ondersteuning; gemakkelijk te integreren met andere frameworks; grote functionaliteit; goede ondersteuning van internationalisatie; eenvoudige, gemakkelijk uitbreidbare architectuur; door gebruik van namespaces gemakkelijk links naar andere applicatiedelen toe te voegen; moderne OGNL-technologie voor krachtige validatie; gemakkelijk te testen met behulp van “mocks”

nadelen: nog slechts weinig documentatie; kleine gebruikersgroep, nog niet uitontwikkeld

- Tapestry (2004): op Java en XML gebaseerd, ondersteunt MVC; XML zorgt voor verbinding tussen HTML-elementen en de bijbehorende Java-code; Spindle-IDE

  voordelen: ontwikkeling van applicaties gaat snel als je de techniek beheerst; grote en actieve gebruikersgroep; vormgevers kunnen templates in HTML aanleveren die rechtstreeks gebruikt kunnen worden; robuuste validatie met duidelijke waarschuwingsberichten

  nadelen: documentatie beperkt; techniek moeilijk; weinig updates; applicatie moeilijk op te delen door middel van gebruik van verschillende URL’s; moeilijk te testen vanwege abstracte classes; onlogische manier van redirecten van POST-berichten

- Java Server Faces (JSF) (2004): streeft naar vereenvoudiging van het bouwen van de userinterface van Java EE-applicaties; JSP als technologie, maar er zijn toolkits voor het gebruik van Ajax en XUL; vele IDE’s (bijvoorbeeld Sun Java Studio Creator, nu vrij beschikbaar); toenemend aantal gebruikers, maar nog niet op het niveau van Spring en Struts

  voordelen: gemakkelijk en snel applicaties bouwen; sterk gericht op interactiviteit; gemakkelijk configureerbare validatie (maar standaard-waarschuwingsberichten zijn lelijk); gemakkelijk te testen

  nadelen: veel JSP-tags; techniek nog onvolwassen, er ontbreken nog zaken; vooral gericht op de View-laag; specifiek: sorteerroutines om tabellen te sorteren ontbreken; altijd POST-methode bij linken naar andere delen van applicatie

- Grails (“Groovy on rails”, 2006): gebaseerd op de JAVA-scripttaal Groovy; pretendeert zeer eenvoudige ontwikkeling van Java-MVC-applicaties mogelijk te maken, zoals Ruby On Rails dat pretendeert voor Ruby-MVC-applicaties (agile, RAD); naadloze integratie met AJAX, Spring, Hybernate; actieve developper- en user-mailing-list

  nadelen: versie 0.1, nog weinig ervaringen of ontwikkelde applicaties

Kijken we naar de in hoofdstuk 2 genoemde keuzecriteria, dan moeten we concluderen dat Struts het beste voldoet als het gaat om de grootte van de gebruikersgroep en de inwerktijd. Vanwege de vele al ontwikkelde applicaties en tools zal dit ook de meest geschikte kandidaat zijn in onze zoektocht naar al bestaande componenten. Struts is echter zeker niet de laatste proven technology. Wat dat betreft zouden Spring en Webwork eerder in aanmerking komen. Beide zouden minder complex zijn, waarbij Spring ook nog eens een goede ondersteuning biedt voor de Model-laag, en Webwork vanwege de Ajax-technologie interessant is. Ajax, of Asynchronous Javascript And XML, zouden we graag gebruiken om betere userinterfaces te maken. Bij gebruik van deze technologie hoeft niet meer steeds na elke actie de hele pagina in de webbrowser ververst te worden, maar kunnen steeds kleine gedeeltes van de pagina aangepast worden. Grails belooft veel, maar is nog lang niet uitontwikkeld;

versie 0.1 is recent gepubliceerd. Indien in deze omgeving op korte termijn updates ter beschikking komen, willen we hier meer gedetailleerd naar kijken.

We kunnen ook denken aan combinaties van frameworks. Denk bijvoorbeeld aan Struts met JSF om het bouwen van de View-laag te vereenvoudigen. Of aan Spring gecombineerd met ZK voor het gebruik van Ajax. Ook de toekomstige Webwork-Struts-integratie klinkt erg interessant.

De conclusie luidt dat we de laatst genoemde frameworks nog wat nader willen testen, en dat we de ontwikkelingen in Grails, een nu nog erg premature omgeving, willen blijven volgen. We moeten er daarbij rekening mee houden dat als we kiezen voor een combinatie, de uiteindelijk door ons gekozen frameworks liefst in één ontwikkelplatform moeten zijn geïmplementeerd. Ook willen we op een eenvoudige manier Web services kunnen inbouwen.

6. Conclusies

Op korte termijn (2 maanden) zullen we het technisch ontwerp van Emergo klaar moeten hebben. In principe zijn er twee manieren om Emergo te structureren: zelf bouwen, of gebruikmaken van een bestaande ontwikkelomgeving. Voordelen van gebruik van een bestaande omgeving zijn de korte tijd die nodig is om een applicatieskelet te bouwen, en het reeds beschikbaar zijn van componenten die weinig aanpassing vergen. Nadelen zijn dat er minder flexibiliteit is, dat je vastzit aan een architectuur die misschien iets minder geschikt is voor Emergo, dat bepaalde functionaliteiten niet helemaal zullen overeenkomen met wat gewenst is, dat uitzoeken of een bestaande omgeving bruikbaar is veel tijd kost, dat aanpassing of toevoeging van functionaliteit ook veel inwerktijd zal vergen en misschien moeilijk te realiseren is, en dat onze eigen “innovatiewens” minder aan bod komt. Gezien al deze nadelen, en gezien het feit dat we in voorgaande projecten al enige ervaring hebben opgedaan met de structurering van applicaties, kiezen we voor zelfbouw.

We hebben hetvolgende besloten:

- we gaan een gelaagde architectuur gebruiken; meer in het bijzonder een variant van de Model-View-Controller-architectuur; we voegen hier volgens de SOA-architectuur een Web Services-laag aan toe om de koppeling aan andere systemen te vereenvoudigen
- we kiezen J2EE/Java als ontwikkelplatform en programmeertaal; het afspeelplatform is een gangbare webbrowser

We moeten nog definitief bepalen welke frameworks we gaan gebruiken. Hierbij zullen we ons laten leiden door de volgende constateringen:

- Struts en JSF zijn goede kandidaten
- Spring, ZK, Webworks en in mindere mate Tapestry willen we nog aan een nader onderzoek onderwerpen
- de ontwikkelingen in Grails volgen we met belangstelling