De softwarebouwers bakken er nog niets van

Een beschouwing over de gangbare manier van softwaregeneratie.

Door Ir J.A.J. van Leunen

Asten, 12 Maart, 2002

Abstract

Software wordt via een andere werkwijze en veel minder effectief en betrouwbaar ontworpen en gebouwd dan hardware. Er zijn geen fundamentele redenen aan te wijzen, waarom dit zo moet zijn. Er zijn wel historische redenen te vinden. Er begint nu een kentering op te komen, waardoor software op meer betrouwbare en aanmerkelijk efficiëntere wijze gebouwd kan worden. Dit artikel gaat in op de historische aspecten en schetst de toekomstige mogelijkheden.

Contents

De softwarebouwers bakken er nog niets van

Een beschouwing over de gangbare manier van softwaregeneratie.

Inleiding

Sinds de opkomst van de computer hebben softwarebouwers geworsteld met het vinden van de juiste wijze van bouwen van software. In het begin was de aanpak vooral gericht op het gemakkelijker kunnen aansturen van de instructies van de computer. Dit was de tijd van de assemblagetalen. Later ontstonden richtingen, waarin meer aandacht geschonken werd aan het doel van het programma. Er werden deeltaken van de programmeur geautomatiseerd, zodat zijn taak minder routinematig en meer doelgericht werd. Dat was de periode, waarin de derde-generatie-talen, zoals fortran, cobol, algol, pascal en C tot bloei kwamen. Deze verbeteringen kwamen vooral neer op het vereenvoudigen van het genereren van

veelvoorkomende functionaliteit. Er ontstonden allerlei functiebibliotheken die dit aspect nog verder ondersteunden. Langzaam maar zeker werden de programma's groter en complexer. Doordat de programmeurs het overzicht dreigden te verliezen begon men na te denken over ontwerptechnologieën. Hierbij ontstonden verschillende stromingen, waarbij er één al snel de overhand kreeg. Deze stroming wordt gebruikelijk aangeduid met de term functioneel gestructureerd ontwerpen. Deze wijze van werken houdt in, dat de functies worden ondergebracht in processen en de gegevens, waarop de functies werken, worden ondergebracht in opslagplaatsen. Deze werkwijze laat zich op leuke wijze grafisch etaleren met plaatjes waarin bolletjes de processen voorstellen en balken in de vorm van strepen de opslagplaatsen verbeelden. Pijlen tussen deze elementen geven aan hoe gegevens tussen processen en opslagplaatsen uitgewisseld worden. In deze periode ontstond het begrip softwarearchitect. Dit waren indertijd individuen, die er hun werk van maakten om met de eerder beschreven gegevensstroomdiagrammen complexe programma's op een meer inzichtelijke wijze weer te geven en nieuwe programma's volgens hetzelfde stramien te ontwerpen. Dat deze werkwijze in feite erg onnatuurlijk is en daardoor tot een verkeerde optimalisatie kan leiden ontdekte men pas later toen de programma's zo groot en complex werden, dat men steeds grotere moeite moest doen om de relatie tussen de functies en datgene waar de functies op werkte op een efficiënte wijze te beheren. De op functionele structuur gerichte ontwerptechniek past dus eigenlijk alleen op relatief kleine en niet al te complexe software stukken.

De op functionele structuur gerichte aanpak heeft desondanks wel enkele zeer ingrijpende gevolgen gehad. Zo zijn op deze wijze ongetwijfeld de relationele gegevensbanken ontstaan. Deze passen namelijk precies in de functioneel gerichte aanpak. Deze gegevensbanken bestaan voor een groot gedeelte uit lange tabellen. In deze tabellen zijn onderwerpen opgeslagen die allen dezelfde gegevenstructuur hebben. Per tabel worden gebruikelijk alleen gegevens opgeslagen die tot een zelfde categorie behoren. Een reeks van sleuteltabellen verzorgen de relaties tussen de opgeslagen gegevens. De sleutels bestaan telkens uit een verwijzingsterm en de bijbehorende verwijzing. Een centraal opgesteld gegevenslexicon beschrijft de samenstelling van het geheel. Als de relaties erg ingewikkeld worden, dan wordt het beheer en de toepassing van de gegevensbank onevenredig veel moeilijker. Dit komt onder andere doordat de op een functionele structuur gebaseerde ontwerptechnologie de functies en datgene waarop zij werken op onnatuurlijke wijze uiteenhaalt. Dit moet weer goedgemaakt worden door een ingewikkelde vorm van relatiebeheer tussen functies en de gegevens waarop zij werken. De databasegoeroes hebben indertijd de hiërarchisch gestructureerde gegevensbestanden als minder goed beheersbaar bestempeld. Dat is wellicht waar, als alleen uit het oogpunt van gegevensbeheer geredeneerd wordt. Als er ook gekeken wordt naar wat er met die gegevens gedaan wordt, dan valt de balans in zeer omvangrijke en complexe systemen eerder uit in het voordeel van een hiërarchische bestandstructuur. Dit uit zich momenteel in de snel

toenemende belangstelling voor XML. De XML bestanden hebben allen een puur hiërarchische structuur.

In de hardware wereld heeft men de functioneel gerichte ontwerpwijze al snel verlaten. Alleen relatief eenvoudige apparaten of onderdelen van apparaten worden op deze wijze ontworpen en gebouwd. Ingewikkelder apparaten worden uit onderdelen samengesteld. Het ontwerp van de onderdelen is gebruikelijk ook nog bruikbaar voor de bouw van andere apparaten. Het benutten van herbruikbaarheid heeft in de hardwarewereld tot een grote toename van de efficiëntie geleid. Het meest spectaculair komt dit tot uiting in de halfgeleiderindustrie. Elke achttien maanden verdubbelt zowel de snelheid als het aantal functionele elementen van de geïntegreerde schakelingen. Complexe apparaten bestaan dus grotendeels uit onderdelen die via goed gedefinieerde interfaces en eventueel via communicatiebussen met elkaar verbonden zijn. Het ontwerp van de interfaces en van de communicatiebussen is net zoals het ontwerp van de meeste onderdelen ook in andere apparaten opnieuw te gebruiken.

De in de hardware zo intensief gebezigde modulaire bouwtechniek heeft in de software pas laat een wat ruimere ingang gevonden. Dit ligt niet zozeer aan het feit dat er niet al vroeg pogingen ondernomen zijn om op eendere wijze modulair te bouwen. Omstreeks 1970 hebben Parnas en enkele anderen bekendheid verworven vanwege hun promotie van het ontwerpen van software in de vorm van onderling samenwerkende modules. Deze adviezen zijn indertijd nauwelijks opgevolgd. De in de laatste decennia opgekomen objectgeoriënteerde ontwerpwijze heeft maar een gedeeltelijke verbetering gebracht. In plaats van de harde inkapseling van functionaliteit en eigendommen, waar de hardware modules hun herbruikbaarheid voor een flink deel aan ontlenen, is de inkapseling van objectgeoriënteerde objecten gebruikelijk verre van volmaakt. Het is alsof de softwareobjecten voor een deel met hun ingewanden bloot liggen, zodat vreemden vrijelijk aan hun eigenschappen kunnen sleutelen. Op die manier kan niemand een gegarandeerd product afleveren. Bovendien zijn in de softwarewereld de afspraken over herbruikbare interfaces en bussen slecht geregeld. Pas met de nu opkomende componentgeoriënteerde ontwerpaanpak begint er een kentering in te zetten. Er begint zelfs een groeiende markt voor softwarecomponenten te ontstaan. Het geheel staat echter nog steeds in zijn kinderschoenen.

Markt situatie

Daar waar voor hardwaremodules al een eeuw lang een levendige open markt bestaat, bestaat er voor softwaremodules nog slechts een kleine en dan nog zeer specifieke markt. Er bestaan inmiddels bedrijven die softwarecomponenten leveren voor desktoptoepassingen en voor bedrijfstoepassingen. Het betreft hier dan modules die gebaseerd zijn op het ActiveX componentenmodel van Microsoft of het JavaBeans componentenmodel van Sun. Softwarecomponenten kunnen evenmin als de meeste hardwarecomponenten zonder een passende infrastructuur functioneren. Bij software maakt deze infrastructuur onderdeel uit van een systeem dat zelf verre van componentgeoriënteerd is. Alleen in een

nagenoeg geheel componentgeoriënteerde omgeving komen de grote voordelen van componenten volledig tot hun recht. Zowel het huidige relatief kleine toepassingsgebied als het verzuipen van de softwarecomponenten in een anders gerichte omgeving zorgen ervoor dat softwarecomponenten niet de waardering krijgen die zij eigenlijk verdienen. Pas als er een werkelijke open markt voor softwarecomponenten ontstaat die vergelijkbaar is met de markt voor hardwarecomponenten en wanneer er infrastructuren komen die nagenoeg geheel componentgeoriënteerd zijn, dan zullen de systeembouwers met een eendere efficiëntie en betrouwbaarheid als in de hardwarewereld de softwaresubsystemen kunnen samenstellen.

Het geheim van het succes van componenten.

De reden achter het succes van componententechnologie kent vele aspecten. De voornaamste reden ligt in de drastische vermindering van potentiële relaties. Deze relaties bepalen voor een groot gedeelte de beheersbaarheid van het systeemontwerp en van het bouwproces. Een nieuwkomer moet alle potentiële relaties naar hun waarde inschatten voordat hij deze op verstandige wijze kan behandelen. Ook een vakexpert heeft ooit die weg moeten behandelen. Automaten worden door hun geringere intelligentie vaak gedwongen om alle potentiële relaties af te lopen en de oninteressante relaties vervolgens te elimineren. Dit vergt tijd en inzet. In grote en complexe systemen kan dit zover oplopen dat een volledige overdekking niet meer menselijkerwijs haalbaar is. Zowel het aantal als de soort en de complexiteit van de relaties zijn van belang. Monolieten zijn ongestructureerde systemen. In monolieten is aantal potentiële relaties nagenoeg gelijk aan het kwadraat van het aantal deelnemende onderwerpen. Als er tien dingen meedoen, dan zijn er 99 potentiële relaties. Doen er duizend mee, dan zijn het er 999.000. In een monoliet gaat de beheersbaarheid dus snel achteruit als het aantal functionele elementen toeneemt.

Binnen een gelaagd systeemontwerp mag de bovenlaag elke onderliggende laag aansturen en bevragen. Opdrachten en berichten in omgekeerde richting komen zelden voor en zijn aan strikte regels onderworpen. Op deze wijze wordt het aantal potentiële relaties met ongeveer een kwart verminderd. Dit heeft voor een flink gedeelte bijgedragen aan het succes van gelaagde systemen boven monolieten. De lagenstructuur heeft nog een ander voordeel. De interfaces tussen de lagen laten zich behandelen als een contract tussen de ontwerpers en bouwers van de onderlaag tegenover de ontwerpers en bouwers van de bovenliggende laag. Op deze wijze kan men werk en verantwoordelijkheden verdelen tussen verschillende ontwerp- en bouwgroepen.

Componenten gaan daarin een heel eind verder. In een op componenten gebaseerd systeem wordt de functionaliteit verdeeld over hard ingekapselde eenheden die hun diensten laten controleren via duidelijk gedefinieerde interfaces. Als elk interface ongeveer tien functies aanstuurt, dan is het aantal potentiële relaties in een op componenten gebaseerd systeem, twee ordegroottes lager dan bij een gelijkwaardige monoliet. De

beheersbaarheid wordt daarmee omgekeerd evenredig hoger. Dat is dus heel wat beter dan bij systemen met een gelaagde structuur. Omdat de complexiteit binnen een component relatief klein is en ook het gedrag van de component op relatief eenvoudige wijze omschreven kan worden, is het mogelijk om de component volledig te testen en er een gegarandeerd product van te maken. Door de onbeheersbare relaties lukt dat niet meer bij delen van grote en complexe monolieten. Het aantal relaties tussen componenten is meestal relatief laag. Dat maakt dat complexe, op componenten gebaseerde systemen juist wel op een robuuste wijze gebouwd kunnen worden. Het feit dat componenten op een goed gedefinieerde wijze aangestuurd en gekoppeld kunnen worden levert tevens de mogelijkheid om deze in verschillende systemen toe te passen. Het betekent dat men van componenten producten kan maken die men op een daarvoor geschikte markt kan aanbieden.

Een uiterst gunstige eigenschap van componenten is, dat zij expertkennis kunnen bevatten die zodanig verpakt is dat de eigenaar ervan zijn intelligentie eigendom veilig weet, terwijl de systeembouwer dit bouwblok kan benutten zonder dat hij zelf over de betreffende expertkennis behoeft te beschikken. Dit heeft op softwaregebied geleid tot een markt voor componenten die helpen om bestaande databases te koppelen aan moderne voorgrondtoepassingen.

Het hier geschetste beeld geldt niet alleen voor hardware en software. Het gaat ook op in bestuurszaken. Een hiërarchische structuur maakt een organisatie beter bestuurbaar. Wordt de organisatie te complex, dan heeft een gedistribueerde bestuursvorm grotere kansen. Dit wordt bewezen door het feit, dat centraal geleide regeringen het moeten afleggen tegen meer democratisch opgezette regeringsvormen. Het verklaart ook het succes van guerrillagroepen tegen de gevestigde orde. Kleine groepen met een duidelijke missie maken een goede kans centraal geleide systemen flink onderuit te halen. Hierbij wordt niets gezegd over de morele of politieke rechtvaardiging van deze acties. Wel over de kracht van de wetmatigheden die dit mogelijk maken. Het zegt ook iets over de haalbaarheid van grote bedrijfsfusies.

Componenten maken dus kennelijk gebruik van een algemeen geldig principe, een natuurlijke wetmatigheid. Deze wetmatigheid wordt door mensen, vanwege hun geloof in eigen kunnen, maar al te vaak veronachtzaamd. In tegenstelling daartoe weet de natuur deze wetmatigheid op uitzonderlijke wijze te benutten. Vrijwel alle complexe levensvormen zijn uit cellen opgebouwd.

Nieuwe ontwikkelingen

Er is een trendbreuk in de goede richting waar te nemen. Microsoft heeft door de nood gedwongen de architectuur van zijn softwaresystemen drastische aan moeten pakken. De nieuwe aanpak, welke de naam dotNet heeft meegekregen, is in principe vrijwel volledig componentgeoriënteerd. Er wordt echter nog steeds de hand gelicht met dit principe. Om bestaande software niet meteen waardeloos te maken heeft men toch een flinke

opening opengelaten voor reeds bestaande software. Heel typisch benoemt men deze oplossing ‘unmanaged’. Het maakt duidelijk dat men de puur componentgeoriënteerde delen veel beter beheersbaar vindt. Er is echter meer aan de hand. Om relatief snel van de grond te komen is het nieuwe platform niet als een ‘native’ systeem neergezet, maar in plaats daarvan als een virtuele machine. Deze virtuele machine draait boven op een bestaand systeem dat zelf nog geen componentgeoriënteerde architectuur heeft. Deze concessies doen de robuustheid van het geheel geen goed. Ter compensatie heeft Microsoft een zeer effectief extra beheersingssysteem toegevoegd dat intensief gebruik maakt van meta-informatie. Deze meta-informatie beschrijft de eigenschappen van de diverse componenten en andere systeemonderdelen. Van deze beschrijving kan ook worden afgeleid hoe de diverse onderdelen met elkaar kunnen communiceren en hoe zij onderling moeten samenwerken. In het ‘managed’ deel van het systeem maken deze meta-bestanden een integraal onderdeel van de ‘runtime’, de draaiende code, uit. Om dit geheel goed te kunnen ondersteunen zijn zelfs een aantal nieuwe programmeertalen aan het gereedschap van de programmeur toegevoegd. De meest belangrijke taal is de Microsoft Intermediate Language (MSIL). Deze taal wordt, zoals de naam al zegt, niet rechtstreeks door de programmeur gebruikt, maar vormt de basis van alle andere dotNet talen. MSIL heeft een ingebouwde ‘garbage collector’ voor vrijkomende geheugenruimte. Dit heeft tot gevolg, dat in tegenstelling tot de meeste oudere talen, de programmeurs zich geen zorgen meer behoeven te maken over het tijdig opruimen van deze afvalgeheugenruimte. Java programmeurs kennen deze voorziening als sinds de introductie van Java. Deze aanpak heeft echter tegelijk wat hinderlijke nadelen. De ‘garbage collection’ kost extra computercapaciteit en zonder extra maatregelen wordt het reactievermogen van de computer er nadelig door beïnvloed. MSIL is een objectgeoriënteerde en tegelijk componentgeoriënteerde taal. Deze en andere eigenschappen worden door de afgeleide dotNet talen overgenomen. De meeste dotNet talen zijn bedoeld om de overstap voor programmeurs die aan een bepaalde taal gewend zijn zo gemakkelijk mogelijk te maken. Zo bestaat er inmiddels een C++.Net, een VisualBasic.Net en een J#. Uit politiek oogpunt heeft deze laatste taal niet de naam Java.Net, maar de naam J-sharp meegekregen. Het is dan ook fout om deze taal te zien als een nieuwe versie van de Java taal. Het vormt net zoals de andere dotNet talen een flinke verbetering op de bestaande talen, inclusief Java, zoals indertijd Java een flinke verbetering vormde op C++. De taal die waarschijnlijk het dichtst bij de MSIL taal staat is de taal C#. Als enige taal heeft deze taal niet de functie om een bepaalde categorie programmeurs te verleiden om naar dotNet over te stappen. Het is wel de taal waarin dotNet functionaliteit op de meest flexibele wijze neergezet kan worden. Naast de genoemde talen worden nog een flink aantal andere bestaande programmeertalen omgetoverd tot dotNet talen. Vaak heeft dat ook weer ten doel om het de programmeurs makkelijker te maken om de overstap naar dotNet te maken. Soms is het doel om het programmeren van bepaalde zaken in dotNet omgeving te vereenvoudigen. De dotNet talen worden allen ondersteund door een zeer krachtige ontwikkelomgeving, welke de efficiency en de betrouwbaarheid van het programmeren nog

eens extra verhoogt. Ook deze ontwikkelomgeving maakt daarvoor intensief gebruik van de meta-informatie. Hieruit blijkt nogmaals dat de meta-informatie een zeer essentieel onderdeel vormt van de dotNet aanpak.

Microsoft heeft ervoor gekozen om een oplossing te bieden via het integreren van meta-informatie in de ‘runtime’ en voor het verbeteren van de programmeerbetrouwbaarheid en programmeerefficiëntie via een nieuwe reeks van programmeertalen. Deze keuze levert een behoorlijke overhead. De kracht van de huidige desktop computers en bedrijfscomputers is echter ruimschoots voldoende om deze overhead te dragen. Dat zal minder gelden voor de minder in middelen bedeelde computers die in de meeste elektronische apparaten zitten. Er bestaat echter een alternatief voor de aanpak van Microsoft. Het is mogelijk om de meta-informatie niet in de ‘runtime’ te integreren maar los daarvan beschikbaar te houden. Ook behoeft men de betere beheersbaarheid van de software niet af te dwingen met de introductie van nieuwe talen. Als alternatief kan men de intelligentie leggen in de softwareontwikkelomgevingen en in een strikte modularisering van de geproduceerde software. Deze benadering vormt naar alle waarschijnlijkheid een beter aanpak voor de in elektronische apparaten ingebedde computers. Het programmeren van ingebedde computers vraagt om een minder voorkomend specialisme, dan het programmeren van de meer ‘normale’ computers. Dat neemt niet weg dat ook in de hoek van de ingebedded toepassingen de complexiteit en de omvang van de programma’s in snel tempo groeit. Daarbij moet men bedenken dat er honderden malen meer ingebedde computers bestaan dan er desktopcomputers en bedrijfscomputers bestaan. Met andere woorden, er is grote kans dat er binnenkort een snel groeiend tekort zal ontstaan aan specialisten die in staat zijn om de ingebedde computers te programmeren. Het is dus van wezenlijk belang om naast een met dotNet vergelijkbare aanpak ook de alternatieve aanpak te kunnen bieden.

XML

Behalve zijn bruikbaarheid als gegevensopslagmedium heeft XML nog een aantal zeer aantrekkelijke extra eigenschappen. Door zijn hiërarchische structuur past XML uitstekend bij de componentgeoriënteerde aanpak. Bovendien berust XML volledig op tekst. Dat maakt dat XML geheel platformonafhankelijk is. Daar komt nog bij dat XML door de meeste moderne webbrowsers op een leesbare wijze weergegeven kan worden. Bovendien bestaan er transformatiehulpmiddelen die XML bestanden in een appetijtelijke vorm kunnen omvormen. Al deze eigenschappen dragen ertoe bij dat leveranciers van commerciële programma’s overwegen om de technologie in deze hulpmiddelen te vervangen door een technologie die meer gebruikmaakt van de mogelijkheden van XML. Maar behalve verbetering van de bestaande technologie brengt XML ook volstrekt nieuwe mogelijkheden.

XML blijkt een uitstekend middel te zijn om gegevens tussen systemen uit te wisselen. Een protocol met de naam SOAP maakt hier intensief gebruik van. De naam SOAP staat voor Simple Object Access Protocol. Op zijn beurt wordt SOAP weer gebruikt om andere functies te ondersteunen. Zo maakt de snel opkomende webservicestechnologie intensief gebruik van het SOAP gegevensuitwisselingsmechanisme. Door zijn combinatie van flexibiliteit, kracht en eenvoud maakt SOAP grote kans om CORBA van zijn huidige positie te verdringen.

Daar waar bij relationele databases het gegevenslexicon meestal voor de gebruiker verborgen blijft, ligt het gegevenslexicon van XML bestanden als het ware op straat. Het gegevenslexicon van een XML bestand heet schema. Het schema is zelf een XML bestand. De fundamentele structuur van een XML bestand is relatief eenvoudig en is door iedereen gemakkelijk te begrijpen. Elk XML element begint met een label en eindigt met datzelfde label. Binnen een element kunnen andere elementen voorkomen. Binnen het label is nog plaats voor met name genoemde attributen, welke het betreffende element kenmerken. Dat is zowat alles wat er over de fundamentele structuur van XML te vertellen valt. Het schema legt vast, welke labels er gebruikt mogen worden, welke elementen er binnen een bepaald element voor mogen of moeten komen en welke attributen het label van het element mag of moet bevatten. Met andere woorden, als het XML bestand als een document wordt opgevat, dan bepaalt het schema het betreffende documenttype. Omdat het schema bestand openbaar is, opent dit onvermoede mogelijkheden. Men kan een bestaand schema analyseren en op grond van deze informatie een nieuw document aanmaken dat tot ditzelfde documenttype behoort. Ook kan men overwegen om vergaarplaatsen aan te leggen, waar documenten die tot hetzelfde type behoren, worden opgeslagen. Het betreffende schema kan dan als kenmerk voor deze vergaarplaats dienen. Dit idee laat zich nog verder uitwerken tot een plek waar vergaarplaatsen voor allerlei typen documenten bijeenstaan. Voegt men nog wat navigatie- en categorisatiehulpmiddelen toe, dan heeft men een doorzoekbare openbare bibliotheek met een grote diversiteit van documenten. XML documenten zijn uitstekend geschikt om er formele specificaties in vast te leggen. Met andere woorden XML is een uitstekend middel om openbare bibliotheken met daarin formele specificatiedocumenten neer te zetten. Op dit moment verschijnen formele specificaties nog grotendeels in de vorm van handboeken en referentieboeken. Gezien de huidige ontwikkelingen heeft dat zijn langste tijd gehad.

De uitweg

De huidige trends geven aan, dat er heel wat betere wijzen van softwareproductie mogelijk zijn dan nu door de softwarebouwers gebezigd worden. Op de desktop en op bedrijfscomputers zal de huidige wijze van werken nog wel een tijd gehandhaafd blijven. De nieuwe architectuur en hun krachtige ontwikkelomgeving geven Microsoft de kans om nog een tijd op het ingeslagen pad voort te gaan. In de wereld van de ingebedde computers loopt het verhaal heel anders. Daar bestaat nog geen infrastructuur, waar softwarecomponenten in kunnen samenwerken. Om

deze reden bestaat er voor dat doelgebied ook geen open markt voor softwarecomponenten. De kans is klein, dat er in dit werkgebied op korte termijn een algemeen geaccepteerde infrastructuur opkomt die wel softwarecomponenten kan ondersteunen en tegelijk schaalbaar genoeg is om vrijwel alle relevante ingebedde computers te kunnen bedienen. Deze situatie lijkt dus onoplosbaar, maar dat is niet zo. Het is mogelijk om door volledig automatisch werkende hulpmiddelen een infrastructuur te laten genereren die volledig toegesneden is op de behoeften van een gekozen verzameling van componenten. Ook deze infrastructuur kan uit componenten worden opgebouwd.

Hardwarecomponenten worden gepromoot door hun gegevens te publiceren in handboeken. Op de meest moderne wijze zou dit gebeuren via XML documenten, welke in een vergaarplaats op het web gepubliceerd worden. Dit kan voor softwarecomponenten op eendere wijze. Daarbij is het mogelijk om softwarecomponenten zo te specificeren dat er geheel automatisch skeletten van de draaiende componenten uit afgeleid kunnen worden, zonder dat daarbij essentiële informatie over de interne samenstelling van de component vrijgegeven wordt. Een passende toolkit kan dan vervolgens zien of de gekozen componenten met elkaar kunnen samenwerken. Dit gaat zover, dat de toolkit volledig automatisch een passende infrastructuur toevoegt, zodat er een werkend prototype ontstaat. Dit is de droom van elke moderne software architect. Hij kan in zeer korte tijd een werkend en testbaar prototype formeren en zien of dit globaal aan de eisen voldoet. Vervolgens kan dit prototype stapsgewijs worden omgezet in een steeds verder vervolmaakt eindproduct. De skeletten kunnen daarbij worden vervangen door op de vergaarplaats aangeboden actieve componenten of de skeletvormige ontwerpen van de architect kunnen door domeinexperts verder worden uitontwikkeld.

Op deze wijze kan de time to market van het softwaredeel van een systeem met ordegroottes worden teruggebracht. Wat belangrijker is, op deze wijze kunnen de in dit gebied gespecialiseerde programmeurs uit de hele wereld worden ingezet om gezamenlijk de enorme hoeveelheid software te bouwen die ongetwijfeld voor de snel toenemende omvang en complexiteit van ingebedde computersystemen benodigd is. Zonder deze toename van de inzetbaarheid en efficiency van de softwarebouwers zullen de hoge investeringen in de fabrieken voor ingebedde computersystemen verwateren. Dat zou betekenen dat de markt voor high-tech IC’s zou instorten. Dit kan, als alles tegenzit al over enkele jaren plaatsvinden. Om dit gevaar te keren moet het hierboven geschetste beeld verwerkelijkt worden. Dat het geschetste beeld geen droombeeld is, hebben proeven op laboratoriumschaal reeds bewezen. Het is mogelijk om een vergaarplaats voor formele softwarespecificatiedocumenten op het web te plaatsen en de daar gepubliceerde specificaties voor softwarecomponenten door geïnteresseerde klanten binnen te laten halen. De binnengehaalde documenten kunnen door passende toolkits benut worden om er snel en doelmatig een werkend prototype van de op deze componenten gebaseerd systeem mee te construeren. Om de stap naar een open markt voor softwarecomponenten mogelijk te maken, moet deze technologie verder

worden uitgewerkt en door meerdere tool vendors ondersteund worden. Bovendien moeten essentiële onderdelen van de technologie middels internationale standaards geüniformeerd worden. Zodra de belanghebbende partijen er toe komen om de koppen bij elkaar te steken, kan dit proces een aanvang nemen. Het is te hopen dat men daarmee niet wacht tot er echt ongelukken in deze sector gebeurd zijn.