Software Engineering 2004

Components: Concepts and Techniques

Martin Bravenboer

Center for Software Technology

Utrecht University

martin@cs.uu.nl

October 12, 2004

Concepts

Component-based Software Engineering

Developing a software system from prefabricated parts

- From custom-made to standard software
- Required in all industries
- No replication: different configurations

- Cross-cuts the software engineering:
  - Requirements, deployment, versioning, configuration, building, maintenance, architecture, design
- Aspects: both technical and market-related
  - Time to market, competitiveness.
  - Stimulation of marketplace

Dreaming in Components

- Maturity of software engineering
- Components of different vendors working together
  - Requires protocols between components
- Marketplace
  - Sell components
  - Provide services
  - Infrastructure
  - Service provider
- Reduce need for skilled system programmers
  - Component architects, assemblers

Has taken almost 40 years to take off!

Components according to Czarnecki

"Building blocks from which different software systems can be composed."

- Component is a natural concept
- Component is part of a production process
- What a component is, depends on the production process.

Economic pressure towards standardized solutions

Components according to Szyperski

Very strict definition of component

1. Unit of composition
2. Unit of independent deployment
3. Contractually specified interfaces
   - Deployment, instantiation and behavior
4. Explicit context dependencies only
   - Deployment environment
5. No externally observable state
6. Binary components (no source)

Aspects of Components-based Software Engineering

- **Standards**
  - Composition, deployment, introspection
- **Reuse**
  - Design, interfaces
- **Deployment**
  - Packaging, versioning, configuration
- **Reliability**
  - Verification, testing, contracts, security

Aspect: Standards enable Composition

- **Composition**

  Connection, interaction, protocols

- **Deployment**

  Packaging, customization, instantiation

- **Introspection**

  Low-level reflection
  Bean introspection
  Target Programming Tools

- **Distribution**

  Remote wiring, serialization, marshalling

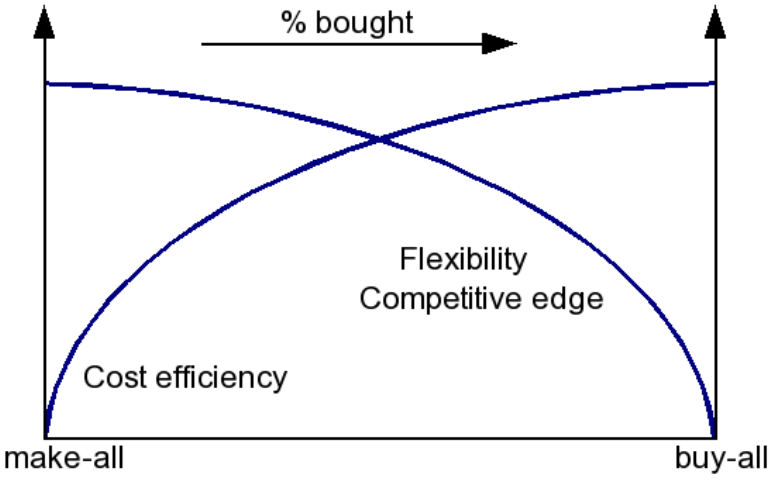
Component Model and Component Framework

Aspect: Standards

Making everything compatible does not necessarily solve a problem.

- Example: same transport and plug mechanism for gas, water, electrics.
- Example: web services of HTTP and port 80.

Aspect: Reuse



- Competitive edge
- Suboptimal fit
- Dependency on 3^{rd} party

+ Maintenence
+ Time to market
+ Interoperability

Reuse: Standard versus Custom-made Software

- Off-the shelf components
- Black box reuse
  - Interface only
  - Szyperski
- Whitebox reuse
  - Knowledge of internal implementation
  - Leaks: Upgrade more difficult
- Source versus binary (contracts?)

Reuse: Component Interfaces

- Components work together through interfaces
  - aka plugs, access points
- Specifies
  - Named operations
  - Invariants, pre and post-conditions
  - Concurrency properties
  - Extra-functional
    - Safety and progress
    - Time and space requirements
- Alternative: Interaction on the wire
  - Message formats, protocols
  - RMI/CORBA versus XML Web Services

Reuse: Interface Versioning

- Window of supported versions
  - compatibility, deprecation, removal
- Approach: immutable interfaces
  - Multiple versions $\longrightarrow$ multiple interfaces
- Approach: mutation of interfaces

Aspect: Deployment

- Components: units of release
- Packaging
  - Metadata, dependencies
  - Security
- Should support third party binding
  - Late binding
  - Introspection
- Where to find dependencies?
  - Identification, component management
- Configuration
  - Local settings

Deployment: Component Versioning

- Side-by-side installation?
  - Surprisingly ignored
- Side-by-side execution?
  - Surprisingly hard
- How to refer to dependencies?
  - Surprisingly weak
- Updates of components?

Aspect: Security

- Check dangerous operations

  e.g. array bound checks
- Restrict access (only explicit context)
- Restricted access rights

  Java ClassLoader, .NET Assemblies
- Load in isolation

  Java ClassLoader, .NET AppDomains
- Late binding: check code at load time

  e.g. Java bytecode verification
- System classes: should not be replaced.

Components: Software versus Hardware

- Basic idea taken from hardware engineering
- Different from components in classical engineering.
- Delivered as a blueprint
- Different instances can be derived from it
  - Configuration
  - Parameterization
- Late integration of components is possible

Components versus Object Orientation

Objects

- Techniques
  - Inheritance, encapsulation, polymorphism.
- No emphasis on independence
- No late composition

Components

- Concepts
- Economic values: time to market, quality, viability.
- Extreme encapsulation
- Extreme abstraction
  - Different kinds of interfaces: more monolithic

Component Techniques

Component Technology: Basics

- Pipes and Filters
  - Unix
  - XML Pipelining
  - Monad/Microsoft Shell
- Java
  - ClassLoader
  - Reflection
  - JavaBeans
- .NET CLR
  - Assemblies
  - Metadata
  - Reflection

Component Technology (Distributed)

Aspects: garbage collection, persistence, transactions, concurrency, security

- OMG CORBA: IDL/IIOP
- Sun's Java: RMI/EJB
- Microsoft .NET: Remoting
- Microsoft (D)COM
- Compound Documents: OLE, ActiveX
- XML Web Services

Subject of Distributed Object Systems course

Unix Pipes and Filters

```
find | grep '\.java' | grep -v '.svn' | xargs wc | sort -bg
```

- 1968: *"Mass Produced Software Components"*, Douglas McIlroy
- 1978: The Unix Philosophy: write programs ...
  - that do one thing and do it well.
  - to work together.
  - to handle text streams, because that is a universal interface.
- *Successful*: used users and unexperienced programmers
- *Limitation*: most filters focus on text processing.

  Text scrapping

  Applicable to specific domains?

Extension: Microsoft's Monad Shell

- **Pipeline of .NET objects**
  - Explicit structure and methods.
  - Multiple records (ProcessRecord)
- **Commandlets** implemented in .NET
  - Reflection: command-line parameters to properties
  - Attributes: mandatory, optional, prompting
  - Reflection: automated help information
- **Scripting** language

Other extensions:

- XML Pipelining
- StrategoXT transformation tool composition

Java: Basic Machinery

- ClassLoader
- Classes
- Packages
- SecurityManager
- Serialization
- Reflection
- Annotations
- Bytecode Verification

Java: Policies and Abstractions

- **Packaging**
  - Java Archives (JAR)
- **Finding** components
  - By classname on classpath
  - Standard extensions
  - System classes: bootclasspath
- **Introspection** of components
  - JavaBeans
- **Deployment**
  - JNLP: Java Network Launch Protocol
  - WAR deployment descriptors

Java: ClassLoader

Load classes in any way, from any location

- Provide bytecode to the JVM
- Namespaces: unique classes, multiple versions

  e.g. Applet in web browser
- Reload updated versions of a component

  e.g. Servlet Container
- ClassLoader delegation:

  Parent ClassLoader consulted first
- Application can still be run if a component is missing.
- Bytecode can be instrumented (e.g. NextGen)

Java: Java Archives

- **Versioning**
  - Weak: only informative.
  - No selection based on versioning requirements
- **Sealing**
  - All code loaded from same JAR
- **Naming**
  - JAR with same name do not conflict
  - Loading by class, not JAR
- **Dependencies**
  - No serious mechanism
  - Manifest can refer to dependencies

Java: Standard Class Loading

Delegation

- Bootstrap, Extension, System ClassLoader

Classpath

- No dependencies in components
- No versioning support

Rather limited: Java deployment is a mess

- Result: nasty scripts
- Maven: dependencies and repository
- Uberjar: create self-contained archive

JNLP: Java Network Launching Protocol

Specification of:

- Application, Applet
- Component

Advantages

- 'Self-contained'
- Platform variability
- Local caching and organization
- Transparent update

Application

- Unfortunately only used for Java Web Start
- Netx: Open-source, command-line JNLP client

JNLP: Java Network Launching Protocol

- Java version

```
<j2se href="http://java.sun.com/products/autodl/j2se" version="1.4+"/>
```

- Java archives

```
<jar href="..."/>
```

- Native libraries

```
<nativelib href="....jar"/>
```

- Extensions

```
<extension href="....jnlp"/>
```

- Locale or operating system specific

```
<resources os="Windows"> ...  </resources>
```

Java: JavaBeans

- Composition and configuration by tools
- More high level introspection
  - Features
  - Events
  - Properties
- Method patterns or custom BeanInfo
- Based on the basic techniques of serialization and reflection.

Java: Further Reading

- *"Component Development for the Java Platform"*
  Stuart Dabbs Halloway

- *"Component Software. Beyond Object-Oriented Programming"*
  Clemens Szyperski

.NET Common Language Runtime

.NET Assembly

- Self-contained
- Boundary of naming and security
- Collection of modules
- Code is loaded by assembly

.NET CLR Module

- Metadata: types, inheritance, method signatures, dependencies.
- Code: CIL or native
- Resources: static files
- Module types: library, (win) exe

.NET Assembly: Boundary

Visibility

- `internal` (C#) or `Friend` (VB .NET)
- `protected internal`
- `public`
- Java packages access is weaker

Identity of types

- Assembly and type name
- Assemblies often named after namespace prefix.

Comparison

- Java ClassLoader namespace and class name

.NET Assemblies: Naming

Exact identification

Goal: assemblies with same name for different organizations.

Four part name

1. Friendly name
2. Culture (default: neutral)
3. Public key of developer (optional)
4. Version (default: 0.0.0.0)

String format

```
friendlyName,Version=0.1,Culture=neutral,PublicKeyToken=...
```

.NET Assemblies: Loading and Resolving

Loading

- Implicit or explicit
- Name: `Assembly.Load(name, culture, version,token)`
- Location: `Assembly.LoadFrom(uri)`

Resolving

- AssemblyResolver: Name to location
- Apply version policies
  - Redirect and assembly
  - Order: application, publisher, machine-wide
- *"DLL Hell"* to *"Versioning Hell"*?

.NET Resolving and the Global Assembly Cache

Public key?

Yes:

1. Apply version policy
2. Global Assembly Cache
3. codeBase hints in assemblyBindings
   - Application and machine
4. Probing
   - assemblyBinding, APPBASE
   - explicitly defined subdirectories of APPBASE
   - Supports cultures

No: Probing

.NET Assemblies: Side by Side Execution

Side by side execution is a problem

- Types: assembly + name
- Distinct types, distinct statics

Common parts must not be version sensitive:

- Common state
- Version invariant types: interfaces
- Deploy in separate assembly

Attributes can disable side-by-side execution:

- Application domain
- Process
- Machine

.NET Application Domains

- Similar to processes
- Scope:
  - Execution of code, faults, security, resources
- AppDomain
  - Assemblies
  - UnLoad
  - CrossAppDomainDelegate
  - Events (Loading, Resolving, Exit, Exceptions)
- AppDomains can be unloaded
- Assemblies cannot be unloaded.
- Inter-AppDomain communication: marshalling.

Java versus .NET

Java

- No built-in notion of component
- But, basic machinery is quite good:
  - ClassLoaders
  - Reflection
  - Different policies could be implemented
- Naming not based on components, but on ClassLoaders.

.NET

- Elaborate notion of components
- Assemblies, dependencies, metadata
- Global Assembly Cache

General Questions

Should components themselves refer to dependencies?

- How exact are references?
- When should references be exact?
- Nix expressions: themselves or arguments
- Nix store: exact references

Where and when should versioning problems be solved?

- Nix: installation time?
- .NET: load-time
- Java: uh ...

.NET: Further Reading

- *"Essential .NET, Volume 1: The Common Language Runtime"*
  Don Box and Chris Sells

- *"Inside Microsoft .NET IL Assembler"*
  Serge Lidin

- *"Applied Microsoft .NET Framework Programming"*
  Jeffrey Richter