Model-Driven, Component Engineering

Colin Atkinson

2nd May 2007

University of St. Andrews
Agenda

- Components, Services and Models
- Model-Driven, Component-Based Development
- Orthographic Service Engineering
Components, Services and Models
Contents

- Components
- Web services
- Models and MDA
- Design by contract
Motivation for Components

- the concept of systematic reuse in software is very attractive
  - increased reliability
    - components exercised in working systems
  - reduced process risk
    - less uncertainty in development costs
  - standards compliance
    - embed standards in reusable components
  - accelerated development
    - avoid original development and hence speed-up production

promoted in software engineering in three main ways -
  - reusing knowledge and experience
    - patterns, standards, guidelines
  - developing generic solutions
    - product lines, frameworks
  - developing and assembling parts
    - component-based development
What is a Component?

Each author has his own favorite definition

"A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

UML Specification

"A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction."

Booch 87

Frequently asked questions

- Does a component have state?
- Is a component an object?
- Is a component a module?

Most commonly accepted definition

"A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

ECOOP'96
Software versus System Components

- key is to distinguish software and system components

**Software Components**
- functional elements of a software application at development time
- units of independent deployment
- units of third-party composition
- have no (externally) observable state
- define external context dependencies
- may be instantiatable

⇒ **types, modules**

**System Components**
- functioning parts of a system in its execution environment
  - a.k.a Subsystem
- (semi)-autonomous parts of an executing system
- interact with system elements developed by third parties
- may have externally visible state
- have unique identity

⇒ **objects, functions**
Key Characteristics of Components

- although components have some similarities to traditional classes and/or modules they have some important additional properties

- they define “required” as well as “provided” interfaces
  - provided interface
    - services offered by the component
  - required interface
    - services required by the component

- they are self descriptive
  - accompanying meta-data describes relevant features of the component for potential users
Component Composition

by definition components are assembled to create larger entities

- ideally component assemblies have the same properties as primitive components and can be combined with into larger components
- contemporary component technologies do not have this property

components are assembled by using connectors

- delegation connectors
  - link the external interface of a component to its internal realization via its parts

- assembly connectors
  - indicate that one component provides the services that another component requires
Ports

- A mechanism for isolating a classifier from its environment
  - Provides a point for conducting interactions between the internals of the classifier and its environment

- Allows a component to be defined independently of its environment
  - Makes it reusable in any environment that conforms to the constraints imposed by its ports

- Required interfaces of a port describes requests which may be made from the component to its environment

- Provided interfaces of a port characterize requests to the component from its environment

Diagram:

- «component»
- OrderProcess
- OnlineServices
- Payment
- OrderEntryTracking
Logical Containment

In recursive component models, one component can be nested or contained in another component to arbitrary depths.

- The composite component can be viewed as a (logical) container of its parts.
Component Description Levels

Components can be realized at various abstraction levels

- language and platform independent
  - requires vendor-neutral interface specification language
  - tools translate vendor-neutral specifications to specific languages and/or platforms
  - main example – CORBA Component Model

- language specific, platform independent
  - requires a “write once, read everywhere” language
  - components must be written in that language
  - main example – Java component models

- language neutral, platform specific
  - language neutral binary specification
  - requires operating system to support the standard
  - main example – COM Component Models
Binary Component Models

- define how components are represented in memory
  - but not how programming languages are bound to them

- most well known is COM (Common Object Model)
  - foundation for all Microsoft component software
  - is widely available on other platforms also
  - is agnostic the use of objects to implement components

- QueryInterface Operation
  - takes a named interface and checks if the current COM object supports it
    - if so, it returns the corresponding interface reference
    - if not, it returns an error indication
  - allows a client with a reference to an interface to “get to” any other interface supported by the same COM object
Disadvantages of Component Based Development

- time and effort required for development of components
  - anecdotal evidence indicates that the effort invested in generalizing component is recovered after 5th reuse

- unclear and ambiguous requirements
  - reusable components are to be used in different applications, some of which may yet be unknown and the requirements of which cannot be predicted

- conflict between usability and reusability
  - to be widely reusable, a component must be sufficiently general, scalable and adaptable and therefore more complex

- component maintenance costs
  - while application maintenance costs can decrease, component maintenance costs can be very high
Motivation for Web Services

- distributed-object and component solutions have shortcomings
  - mainly for use within an intranet
  - a lot of interoperability problems due to their proprietary nature
  - do not scale to the Internet
  - tightly coupling services and consumers
  - server object implementations not portable

- to promote B2B interaction need an solution that
  - enables universal interoperability
  - enables widespread adoption
  - is based on ubiquitous open, extendible standards
  - requires minimal supporting infrastructure
  - focuses on messages and documents, not on APIs
What Are Web Services?

“Web services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes. …

Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service.”

IBM

- **self-contained**
  - functionality and attributes are exposed in a public interface while implementation is hidden

- **self-describing**
  - have a machine-readable description used to understand their interface

- **modular**
  - are reusable and can be composed to generate higher level functionality

- **published**
  - can be registered in electronic “yellow pages” for easy location by other applications

- **located**
  - are tied to a fixed, globally unique location identified through a URI

- **invoked**
  - can be invoked using a standard Internet protocol
Web Services Architecture

Elements in a system built from web services play one of three roles:

- Service requestor
- Service provider
- Service broker (repository)

- Service providers publish services by advertising service descriptions in the registry.
- Service requestors use find operation to retrieve service descriptions from the service registry.
- Service requestors bind to service providers using binding information found in service descriptions to locate and invoke a service.
Core Web Service Technologies

**SOAP** (Simple Object Access Protocol)
- a message layout specification defining a uniform way of passing XML-encoded data
- a way to simulate RPC over standard Web communication protocols

**WSDL** (Web Service Description Language)
- defines Web Services as collections of network endpoints or *ports*
- a port is defined by associating a network address with a binding

**UDDI** (Universal Description, Discovery and Integration)
- provides a mechanism for clients to find web services
- the basis for repository services for business applications

![Diagram of Interaction Stack, Description Stack, and Discovery Stack]
Important Dichotomies

- **Web Services versus Web Service providers**
  - the term “service” is sometimes used to refer to just the abstract interface and sometimes to an implementing object
  - the terms “service interface” and “service provider” should be used when clarity is needed

- **Web Service types versus Web Service instances**
  - strictly speaking Web Services are instances
  - WSDL specifications bind operations to specific URL’s as part of the definition of ports, and thus have a unique instance identity
  - however, SOAP specifications define an abstract interaction protocol (interface) which can be used with any conformant service provider

- **Web Services versus components**
  - Web services are not software components
  - they are instances, can have state, do not define required interfaces ..
  - but they are clearly system components

⇒ Web Services are objects!
Are Web Services Stateless or Stateful?

- the core Web Service standards allow Web Services to be stateful since one web service can export multiple methods.
- however, they are often characterized as stateless because:
  - the core standards have no mechanism for controlling concurrent access to web services in a multi-client environment.
  - the “state” of stateful web service abstractions is usually stored outside the service provider code.
Pros and Cons of the Web Service Model

- Increase development efficiency
- Increase flexibility
- Increase opportunities to generate revenue from services
- Increase reusable components/services
- Increase interoperability via standards

- Decreased IT control of software assets
- Decreased security/reliability
- Decrease trend to in-house centralized systems (more global distribution)

Increased flexibility and efficiency for developers
Decreased control for IT organizations
Motivation for Model-Driven Development

- heterogeneity hinders the development of enterprise distributed systems

- there is (and will never be) complete consensus on
  - hardware
  - operating systems
  - network protocols
  - programming languages

- middleware is intended to solve this problem, but has itself proliferated
  - CORBA, ..
  - COM / .NET, ..
  - Java / J2EE, ..
  - SOAP / WSDL, …
  - …
What is Model Driven Development?

- an approach to IT system specification that separates the specification of system functionality from the implementation of that functionality on a particular technology platform
  - “design once, build on any platform”
- an open, vendor-neutral approach to interoperability using OMG's modeling specifications
  - a software development process driven by the activity of modeling software systems
CIMs, PIMs and PSMs

- Computation Independent Models
  - describe the requirements for the system and its environment
  - the details of the structure and processing of the system are hidden or undetermined

- Platform Independent Models
  - focuses on the operation of a system while hiding the details necessary for a particular platform.
  - shows that part of the complete specification that does not change from one platform to another.

- Platform Specific Models
  - combines the platform indep. viewpoint with an additional focus on the detail of the use of a specific platform by a system
PIM and PSM Examples

**PIM**
- A “formal” specification of the structure and function of a system that abstracts away technical detail
- usually expressed using standard UML

**PSM**
- Specifies how the functionality specified in a PIM is realized on a particular platform
- expressed using UML extended with platform specific UML profiles
Key Components of MDA

- **Metalanguage**
  - extends
  - is written in

- **Transformation Definition**
  - is written in
  - is written in

- **Language**
  - is written in

- **Transformation Tool**
  - is used by

- **PIM**
  - is written in

- **PSM**
  - is written in
Design by Contract

- a software design principle derived from the legal notion of a contract
  - agreement between two parties in which both accept obligations and on which both can found their rights.

- in SE, provides a means to clearly establish the expectations and responsibilities of an object
  - an object must deliver its services (obligations) if and only if certain stipulations (the rights) are fulfilled
  - provides an exact specification of an object's interface

- an object's contract is formally defined in terms of
  - invariants
  - operation pre and post conditions
Contract Example

Example

- For the price of 4 Euros a letter with a maximum weight of 80 grams will be delivered anywhere in the country within 24 hours

<table>
<thead>
<tr>
<th>Party</th>
<th>Obligations</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Customer</strong></td>
<td>Pay 4 Euros</td>
<td>Letter delivered within 24 hours</td>
</tr>
<tr>
<td></td>
<td>Supply letter less than 80 grams</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specify delivery address within country</td>
<td></td>
</tr>
<tr>
<td><strong>Delivery Company</strong></td>
<td>Deliver letter within 24 hours</td>
<td>Delivery address is within country</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receive 4 Euros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receives letter less than 80 grams</td>
</tr>
</tbody>
</table>
Invariants

- constraints coupled to classes, types and interfaces
  - transcend any one particular operation

- define what must be true for all instances of the class
  - when one of the operations is not executing

- can be viewed as part of the pre and post condition of every operation of a class

context Stack
inv: self.noElements <= maxSize

context Stack
inv: self.noElements >= 0

context Stack
inv: self.elements->size() = self.noElements
Pre and Post Conditions

- Post conditions often refer to the value of an attribute or association at the start of an operation's execution, achieved by appending `@pre` to the attribute or association concerned.

- The keyword `result` can be used to identify the value returned by an operation.

```cpp
context Stack::push(o : Object)
pre: elements->size() < maxSize
post: elements->size() = elements@pre->size() + 1
      and elements->last() = o

context Stack::pop():Object
pre: elements->size() > 0
post: elements->size() = elements@pre->size() - 1
      and elements = elements@pre->excluding(elements@pre->last())
      and result = elements@pre->last()
```