Towards Software Manufacturing

Wojtek Kozaczynski
Jack Greenfield

Session MSP11
Speakers

- **Jack Greenfield**
  - Chief Architect, Practitioner Desktop Group
  - Leads architecture practice for the organization developing the Rational Rose family of products
  - Editor for Wiley Application Development Series
  - Active contributor to related J2EE and OMG standards

- **Wojtek (Voytek) Kozaczynski**
  - Director, Architectures & Application Frameworks
  - Leads development of reusable software assets on the Rational e-development Accelerators initiative
  - Active contributor to related J2EE and OMG standards
Economic importance of enterprise applications is growing
Demand for high quality enterprise applications is growing
  Requires quicker, less costly, more reliable provisioning
Software development industry is changing to meet the demand
  Natural consequence of industry maturation
  Transitioning from craftsmanship to manufacturing
Rational is driving and supporting the transition
  A process-centric, standards-based approach to automating enterprise application development


- **Enterprise Applications**
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
Enterprise Application Characteristics

- Implement all or part of a business process
  - Contain a model of a “business reality”
- Large amounts of highly structured information
  - May have 1000s of data types and associations
  - May have 1000000s of instances with complex relationships
- Small amounts of complex computation
  - User interaction
  - Constraint enforcement
  - Process automation, …
- Often complex, distributed deployment infrastructure
  - Multiple processors
  - Complex interfaces and communication
Enterprise Application Characteristics (Cont.)

- Stringent quality of service requirements
  - Security of critical data and processes with widely exposed interfaces
  - Scalability to large numbers of clients and transactions
- Overwhelming logistical complexity
  - Rapid response to constantly changing business conditions
  - Multiple stakeholders
  - Different technologies and development methods
  - Concurrent projects at different stages of development
By 2004, the worldwide IT services industry will grow at a compound annual growth rate (CAGR) of 11% to reach almost $584.6 billion


- Systems integration will grow at 13% CAGR; Rising from $59 billion in 1999 to $109 by 2004
- Packaged software support and integration will grow at 15% CAGR; Rising from $35 billion in 1999 to $71 billion in 2004
- Network infrastructure management will grow at 16% CAGR (interoperability is the major issue)
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
Enterprise Applications Are Handcrafted

- Ad-hoc development
- Development at low levels of abstraction
- Developers must cope with broad abstraction gap between requirements and designs
- Reliance on labor-intensive activities
- No economically-significant reuse
- High levels of discovery
- One-off implementations
Evidence Of The Problem

According to the Standish Group's Chaos study more than half of the IT development projects undertaken in the United States come in late and over budget.

<table>
<thead>
<tr>
<th>Company Size</th>
<th>Average Project Cost</th>
<th>Project Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>$1.2M</td>
<td>24%</td>
</tr>
<tr>
<td>Medium</td>
<td>$1.1</td>
<td>28%</td>
</tr>
<tr>
<td>Small</td>
<td>$0.6</td>
<td>32%</td>
</tr>
</tbody>
</table>

Standish Group's Chaos Report, 1999
Example Of The Abstraction Gap

- A feature (requirement) maps to multiple logical constructs…
  - Interfaces, methods, data members, message headers, persistence bindings, schema elements, configuration elements
- …distributed across multiple physical artifacts
  - Source code, deployment descriptors, external connectors, schemas
- …expressed in multiple implementation technologies
  - Java, JavaScript, EJB, SQL, JCX, CICS, DTP/XA, JMS, JNDI, LDAP, NDS, JAAS, Kerberos, JSP, Servlet, XML, HTML, SOAP, WSDL, …
- …developed and managed by multiple participants
  - Component provider, application developer, application deployer, platform provider, database administrator, system administrator
This Situation Is Not Sustainable

- Low quality, performance, hard to maintain
- High failure rate
- Long time to market
- Requires highly skilled resources

- Tools tied to changing implementation technology
- Platforms provide proprietary solutions
- Few reusable architectures or frameworks
- Application developers still build infrastructures
Industry In Transition

- Market Consolidation
  - Number of key deployment infrastructure providers reduced to four or five
  - Number of key development infrastructure providers reduced to three or four
- ... leads to Comoditization
  - Enterprise class component architectures
  - Software development infrastructure and IDEs in particular
- Standardization
  - MOF, UML, XML, XMI, JMI
  - UML profiles for specific purposes
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
Fabrication: the process by which individual structural [or behavioral] components of a system are manufactured [and assembled]

Academic Press Dictionary of Science and Technology

Fabrication: 1) to make or create 2) to construct by combining or assembly 3) the act or process of manufacturing

The American Heritage Dictionary

Fabricate; to construct from diverse and usually standardized parts

Webster’s
Guiding principles

- Facilitate use of standard parts
- Raise level of abstraction for designers
- Generate implementations directly from designs
- Automate frequently recurring tasks
- Reduce reliance on labor intensive activities
- Promote reuse of designs and products

“ilities” of mature fabrication process

- Repeatability
- Predictability
- Replaceability
What Is Software Fabrication?

- Rapid enterprise application provisioning paradigm
- Promoting development from a base of reusable, standardized software assets
- Based on pattern, model and component technologies
- Replacing manual development practices with process automation and standardized development infrastructure
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component-Based Development</td>
<td>Use of pre-fabricated, self-contained software constructs (components) built to be composed to collaborate with other components and to be managed by component container. Key types of a <strong>Standard Parts</strong> providing a foundation for <strong>Automated Design and Assembly</strong>.</td>
</tr>
<tr>
<td>Model Driven Development</td>
<td>Use of a common, unambiguous graphically-oriented (user-friendly) language to represent software artifacts. Provides a semantic foundation for <strong>Standard Parts</strong>, <strong>Tools and Automated Design and Assembly</strong>.</td>
</tr>
<tr>
<td>Pattern-Oriented Development</td>
<td>Use of known solutions to common design problems. Provides a way of representing and communicating best practices and promotes <strong>Automated Design and Automated Assembly</strong>.</td>
</tr>
<tr>
<td>Asset-Based Development</td>
<td>Development based on reuse of pre-packaged software assets. It is the software realization of the <strong>Standard Parts</strong> fabrication principle. The assets include both deployment infrastructure and domain parts.</td>
</tr>
<tr>
<td>Automation</td>
<td>Tool support providing automation of common design, code development, assembly and deployment operations. It is the software realization of <strong>Standardized Tools</strong> and <strong>Automated Design and Assembly</strong>.</td>
</tr>
<tr>
<td>Standard Process</td>
<td>The underpinning and the &quot;backdrop&quot; of modern software development. Assures predictability and repeatability through describing recommended development workflows, activities, artifacts, skill and roles.</td>
</tr>
</tbody>
</table>
Putting It All Together

- Component-Based Development
  - Standardized reusable parts
- Model-Driven Development
  - Direct specification of concepts
- Pattern-Oriented Development
  - Solutions to known problems
- Asset-Based Development
  - Comprehensive approach to deriving solutions from a base of pre-packaged software assets
- Supported by
  - Software Development Process
  - Tools Automation
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
What’s A Component?

- A component is a self-contained software construct that has a defined use, has a run-time interface, can be autonomously deployed, and is built with foreknowledge of a specific component socket.
- A component socket is a well-defined and well-known run-time interface to a supporting infrastructure into which the component will fit.
- A component is built for composition and collaboration with other components.
- A component socket and the corresponding components are designed for use by a person with a defined set of skills and tools.

*Herzum & Sims:*
Example – Java 2 Platform, Enterprise Edition

- Separation of concerns
  - Application developers focus on components
  - Platform developers focus on containers
  - Resource manager developers focus on connectors

- Containers mediate clients and components
  - Transactions, resource pooling, persistence
  - Policies specified through configuration not code

- Connectors mediate containers and resource managers
  - Portable service APIs to external resources
  - Resolve M-to-N problem across multiple vendors
Component Types

- **Client Components**
  - Applets, applications
  - Execute on client virtual machine
  - All other types execute on server virtual machine

- **Web Components**
  - Servlets, JavaServer Pages
  - Respond to requests from HTTP and other protocols
  - Generate web-based application user interface
  - Deployed, managed, and executed by web server

- **Application Server Components**
  - Enterprise Beans (Session & Entity)
  - Deployed, managed, and executed by application server
  - Maintain conversational or persistent state in instance variables between method invocations
  - Participate in distributed transactions that span multiple resources
  - Provide services concurrently to large numbers of clients
  - Perform client authentication and authorization to access protected services
Example Container Architecture
Connectors

- Driver based interface to external resource managers
- Application contract from component to driver
  - Common Client Interface
  - J2EE standard service APIs
    - JDBC, JMS
  - EIS specific interface
    - JBAPI
- System contracts from container to driver
  - Transaction management, connection management, security
- Standard driver packaging format
  - JAR with XML based deployment descriptor
Example Connector Configurations

Single application server connected to multiple enterprise systems

Single enterprise system connected to multiple application servers
Packaging Architecture

Diagram showing J2EE Application components, including EJB, Web, Application Client, and Deployment Tool, with deployment descriptors (DD) indicated for each component.
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
Model Driven Development

- System described by a “family” of models
  - A model describes the system from a specific perspective
  - A model describes the system at a specific level of abstraction
    - Requirements ➔ Analysis ➔ Design ➔ Implementation ➔ Deployment

- Mappings between concepts in different models
  - Refinement of systems concepts/elements
  - Transformation of higher level elements to one or more lower-level elements
  - Traceability between elements in different models

- Modeling conventions for specifying a system
- Rules of validating model consistency and completeness
- Capability of transforming design, implementation and deployment elements into code
  - Developers add code to specialize operations, rules, constraints
Commonly Used Models

- Business Model
- Use-Case Model
- User Experience Model
- Analysis Model
- Design Model
- Implementation Model
- Deployment Model

Requirements Set → Analysis Set → Realization Set
Modeling “Conventions”

- Agreements on how to represent specific system aspects in the system models
  - Proposed way of describing (problem) domain concepts in models
- Captured in different forms
  - Informal guidelines
  - Structured guidelines
  - Formal languages and UML Profiles in particular
- May span multiple models
  - Capture mappings between concepts in different domains
Profiles Tighten UML Semantics

A profile contains model elements which have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged definitions and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

— UML 1.4 Reference Manual
A domain concept is a component managed by an EJB application container.

JSR-26 profile stereotyped the Subsystem to represent elements of Enterprise Java Beans.

UML contains Subsystem modeling element that is used to represent collective behavior of enclosed elements.
Example “Conventions”

RUP Guidelines
Example “Conventions”

- Business Model
- Use-Case Model
- Analysis Model
- User Experience Model
- Design Model
- Implementation Model
- Deployment Model
- EDOC Profile
- WAE Profile
- EAI Profile
- DC Profile
- JR-26

Technology Profiles
Examples Of Traceability (and Mappings)

- Design use cases trace to user experience use cases
- Design elements trace to user experience screens and forms
- Design constraints trace to user experience constraints
- Design use case realizations trace to system use cases
- Design constraints trace to use case constraints
- Design use case realizations trace to analysis use cases
- Design elements trace to analysis elements
- Design element interactions trace to analysis element interactions
- Implementation elements trace to design element
- Implementation element packaging traces to design elements distribution and concurrency
RTE Of Executable Artifacts

- Business Model
- Use-Case Model
- Analysis Model
- User Experience Model
- Design Model
- Implementation Model
- Deployment Model
- RTE Engine
- Transformation Rules
- EAI Profile
- DC Profile
- JR-26
- Executable Artifacts
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
What Is A Pattern?

- Expert solution to recurring problem in specific context
  - Codifies best practices identified by experience
  - Can be used to recognize, analyze and refactor bad designs
- Constrains structure and behavior of participating elements
- Solution description not solution instance
  - Applied to create specific solution instances
- Can be combined to create solution frameworks
- Variability points permit adaptation and customization
  - Class substitution, method override, parameterization
Types of Patterns

- Can be differentiated by various factors
  - Levels of abstraction
  - Software development life cycle phases
  - Application domains
  - Technology domains

- Examples
  - Language idioms
  - Design patterns
  - Analysis patterns,
  - Architectural patterns
Defining Patterns

- Pattern template describes pattern information structure
  - Combination of formal and informal information
- Formal information usually captured via UML models
  - Class diagram shows static dependencies between elements (participants) of solution and implementation strategy
  - Interaction (Sequence or Collaboration) diagram shows interactions between participants in solution or implementation strategy
**Architectural Pattern**

- A “template” for a commonly used solution to a high-level design problem
  - Handling HTML requests
  - Security
  - Locating application server component/service
  - Passing data collections by value, etc.

- Can be articulated at different levels of detail
  - “Pattern level”
  - Implementation strategy level

Note: Examples taken from the Sun J2EE Architecture Patterns
The Front Controller Pattern
Front Controller Interactions

1. Client → FrontController: Send Request
2. FrontController → Dispatcher: Delegate Request
3. Dispatcher → View: Forward Request
4. View → Helper: Send Request
5. Helper → Client: Forward Request
6. FrontController → View: Process Request
Dispatches may include multiple JSPs to produce different parts of the same page.
The Business Delegate Pattern

The Business Delegate returns a service or a service factory to the Business Delegate.
Systematic application of Front Controller and Business Delegate leads to a very J2EE-characteristic architecture for handling user interactions

- All requests handled by the same design element
- Web server container handles concurrency and scalability
- Front controller responsible for authentication and authorization
- Dispatchers responsible for all interaction business logic
- Views (JSP) only responsible for assembling web page fragments
Authentication and Authorization

This sequence of:
- pre-processing
- view invocation, and
- post-processing

can be repeated multiple times
Characteristic Structure: Use-case-specific Dispatchers

- FrontController
- SignInDispatcher (from Presentation)
- CreateAccountDispatcher (from Presentation)
- ManageAccountDispatcher (from Presentation)
- CreateAuctionDispatcher (from Presentation)
- SignInDispatcher (from Presentation)
- BidDispatcher (from Presentation)
- BrowseDispatcher (from Presentation)
Characteristic Structure: Use-case-specific Dispatchers

- FrontController
- SignInDispatcher (from Presentation)
- ManageAccountDispatcher (from Presentation)
- CreateAuctionDispatcher (from Presentation)
- SignOutDispatcher (from Presentation)
- BidDispatcher (from Presentation)
- BrowseDispatcher (from Presentation)
- CreateAccountDispatcher (from Presentation)
- CreditBureauDelegate
- CreateAccountDispatcher()
doGet()
doPost()
CreditValues
Jump : String
address : String
city : String
state : String
zip : String
account_number : String
expiration : String
CreditValues()
AccountManagerDelegate
AccountValues
username : String
password : String
email : String
unique_id : String
AccountValues()
createaccount_email_view
createaccount_view
createaccount_results_view
createaccount_start_view
createaccount_verify_view
createaccount_entry_view
creditcheck_view
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
Reusable Assets

- Architecture Frameworks
- Patterns
- Components
- Models
  - Business Model, Use-Case Model, etc.

.... are all examples of reusable assets
In general terms: a *reusable software asset* is a software artifact or a set of related artifacts that have been created or harvested with an explicit purpose of applying it repeatedly in subsequent development efforts. An asset has a description of the way(s) in which it should be used and applied.

*Reusable Asset Specification v 1.0.4*
Asset-Based Development

- Development based on economically-significant reuse of software assets
- Reusable Asset Specification (RAS) defines standard way of packaging assets
Variability points can be context-independent or relevant only to a specific context. A Context can impact each artifact in a different way. Artifact Context represents "interpretation of a Context" for a specific Artifact. For each Artifact there is at least one Context that the Artifact is in. => There is at least one context in an Asset (a "root context").
For each variability point there must exist at least one activity that describes how to bind that variability point (what to do with it).

1. There may be a set of context-independent Activities that "operate" on a group of Artifacts.

2. For each Context there may be a set of activities that apply only to that context. Each Activity in the group may "touch" multiple Artifacts.

3. For each Artifact there may be a set of Activities that describe how to apply the Artifact. Some of these Activities are context-dependent.
Asset Management & Consumption

Producer
  - Harvester
  - Packager

Broker
  - Analyst
  - Architect
  - Designer
  - Developer
  - Assembler
  - Tester

Consumer
  - Search
  - Analyze
  - Purchase
  - Deliver
  - Host
  - Maintain
  - Catalog
  - Apply
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
Automation Of System Development

- Automation ➔ development activities supported or preformed by the tools
  - Support of model development and transformations
  - Support of model-to-code transformations (aka RTE)
  - Support of asset development, analysis and management
Automation Of Model Development & Transformation

- Application of patterns
- Integration (assembly) of components
- Model transformations
- Model integrity validation
Code generation
- Driven by general model-to-code transformation rules
- Method-level details controlled by context-specific template

Code reverse-engineering

Model and code synchronization
Support for asset packaging
Management of asset repositories
Tools for asset search and identification
Agenda

- Enterprise Applications
- Industry in Transition
- Software Fabrication
- Component-based development
- Model-driven development
- Pattern-oriented development
- Asset-based development
- Automation
- Standardizing the process
S/W manufacturing is about repeatability
- Common deployment infrastructure
- Common development infrastructure
- Common application domain
- Common solution elements

... product-line development principles are an example of software manufacturing within a single organization and a closely relates family of systems
Three Supporting Processes

- **Platform Engineering**
  - Deployment Platform Engineering
    - Component platforms
  - Development Platform Engineering
    - Tolling and automation

- **Domain Engineering**
  - Domain modeling, scoping, development of architecture frameworks

- **Component Engineering**
  - Development of domain-and-platform-specific reusable patterns and components
Emphasis Of Solution Manufacturing Process

- Selection and refinement of application framework(s)
- Functional requirements modeling
- User experience modeling
- Design of use-case realizations
  - Reuse common patterns and components
- Solution component manufacturing
- Solution assembly
  - Code generation, code modifications, deployment unit packaging
- Solution testing
Software Fabrication Is Not One Thing

- Models
- Patterns
- Automation
- Standard Process
- Asset-Based Development
- Components

Software Fabrication
Rational Areas of Emphasis

- Model-driven development, of course
- Automation
  - Model-to-model transformations, in particular using patterns
  - RTE
- Asset-based development
  - Standardization of software assets
  - Support for asset market via RDN
- Process, of course
- Component-platform-specific tool support
questions?
Thank You

This presentation will be posted in the Partner Resource Library shortly after this conference via:
http://www.rational.com/partners_only/index.jsp
Have your user id and password handy.