About the Speaker

Desmond D’Souza is founder of Kinetium. He is co-author and developer of the CATALYSIS method (Addison Wesley 1998), and is a respected authority and speaker at companies and conferences internationally. He was previously senior vice president of component-based development at Platinum Technology and at Computer Associates, working on methods, tools, and architectures for component-based development. He founded ICON Computing, an object and component technology methods and services company that was acquired by Platinum in 1998. Mr. D’Souza has worked with object and component technology since 1985.

Kinetium provides solutions for component-based development, modeling, and architecture. To learn more about the strategies, methods, modeling, architecture, and technology of component-based development and e-Business, you can contact Desmond at dsouzad@acm.org
Outline

Introduction
What problem are we setting out to address?

Components
What they are, how they interact, how to describe them

Architecture
What it is, why it is essential, how to describe it

Frameworks
The basic idea

Reuse
What it is (and is not), reuse at all levels

Systematic Reuse with Frameworks
Making models, designs, code reusable

Summary
Catalysis in Perspective

Talk Preview: Success With Components

1. Business Driven
understand business first relate disparate views

2. Traceability
bridge Business and IT precise shared vocabulary critical business questions early refinement across levels

3. Pluggable Components
precise interface-centric design traceable business components

4. Architecture
infrastructure interfaces, connectors plug standards interoperability

5. Reuse Process
develop for reuse develop with reuse first reuse problem model then architectures then interface code

= Integrated Business and Systems Engineering

Catalysis Component-Based Development for E-Business
What is a Component?

- A package of software that can be independently replaced. It both provides and requires services based on specified interfaces. It conforms to architectural standards to interoperate with others.

- Totally separate interface from implementation
- Component package can include installable, interface, specs, models, tests, docs, …
- Granularity from 1-class JavaBean to multi-tiered business component with UI, DB
How do Components Interact?

Components interact via clearly specified interfaces

Focus on behavior; stored data, implemented procedures, other interfaces remain completely hidden from clients

Specifying a Component for its Client(s)

**Services Provided**: Interface Spec
- e.g. pickup(goods)
- ? inventory decreased by quantity of goods

**Services Required**: Interface Spec
- e.g. inventory below minimum
- ? restock event raised

**Events Raised**: Notifications of state changes
- e.g. inventory below minimum
- ? restock event raised

**Logical model of component state**: state attributes for each interface
- The interface operations and events are specified based on this
  - e.g. inventory and minimum, attributes used to specify restock event

**Result**: Precise model of information exchanged, assumptions, guarantees
**Client 1: Interface Client**

What does the implementor of this component need to know ...

… about this component?

A client only knows about the relevant **interface** specification
- operations, events, logical state through that interface
- each interface has its own ops, events, logical model of state
- warehouse inventory, staffing, storage maintenance: different views

**Client 2: Component Assembler**

What does the creator of this component assembly need to know ...

… about this component?

Assembler needs to know full **component specifications**
- all interfaces provided and required
- how the logical state models (and events, operations) are related
e.g. storage out for maintenance? less space available for delivery
**Client 3: Component Implementor**

What does the implementor of this component need to know?

- implements all provided interfaces, assuming all required interfaces
- chooses physical state representation (or recursive assembly)
- has additional implementation component dependencies (controversial)
- exhibits specified behavior and logical state view for each interface
- conforms to specified architecture common for these components

---

**“Plug” together - Symmetry and Caution**

- “Plugging-in” parts will only work if the two ends are compatible
- Client must specify required behavior
- Implementor must specify provided behavior
- Needs a symmetrical, precise, black-box view of every component
- We want to “plug” together even dynamically, in cyberspace!
- Need some shared standards for connecting plugs to sockets
Type = Precision in Interface Specs

Note: Behavior Specs can be made precise using UML/Object Constraint Language (OCL)

Software Interface as Contract

~~~
**Contract Name**
The Development Agreement between Joe Inc and Fred Ltd.

**Terms and Definitions**
 tested = ...
correct = ...
product = ...
extension = ...
horrible thing = ...

**Body of Contract**
...Joe Inc shall deliver a tested and correctly functioning product to Fred Ltd by the delivery date, subject to extensions. If he should fail to do so, then Fred Ltd can do many and various horrible things to Joe...
~~~

Type specification = Contract
Type name = Contract name
Type interface = Contract body
Type model = Terms and Defs

This is no coincidence!
Different Aspects of a Component - I

From Catalysis-based component standard with Microsoft / MDC
www.mdcinfo.com

Different Aspects of a Component - II
**Different Aspects of a Component - III**

- **ToolSpecification**: Component specified in terms of its interface and interfaces of components it needs. Developed packages binary into an installable.
- **ToolAssembly**: How components are assembled in an executing application.
- **ToolSource**: Produced by compiler and developers.
- **ToolBinary**: After user installs a module.
- **ToolModule**: Component Architecture - rules and constructs applicable at each of these levels.

**Different Aspects of a Component - IV**

- **Component**: Full-lifecycle includes black-box spec, assembly of sub-components, source, binary, installable module, deployment.
  - Component Specification - a component specified as a collection of ports. This style of specification is suitable for assembling the component with other components to produce a larger component.
  - Component Assembly - a static configuration of components, whose ports are wired together with connectors.
  - Component Source - defines the lowest level manually created “source” code for a component that will be related to its compiled form.
  - Component Binary - the installable, executable binary for a component (e.g., class file bytecodes for a JavaBeans component). Binaries.
  - Component Module - packaged installable collection of binaries and other needed parts.
  - Component Deployment - deployed, registered, and ready for discover and instantiation.
  - Component Architecture - rules and constructs applicable at each of these levels.
Summary - Components

- Interface centric, collaboration patterns
- Symmetrical, precise, black box views
- Refinement - separate interface from implementation
- Full-lifecycle component model - specification, design/assembly, module, deployment

Outline

Introduction
What problem are we setting out to address?
Components
What they are, how they interact, how to describe them

Architecture
What it is, why it is essential, how to describe it

Frameworks
The basic idea

Reuse
What it is (and is not), reuse at all levels

Systematic Reuse with Frameworks
Making models, designs, code reusable

Summary
Catalysis in Perspective
Modeling Business Components

1. Higher-level parts... abstract the objects

2. Higher-level late-bound connectors and properties
   … abstract the protocol, defer binding

3. Separate views of customer for flexibility (can ship to any receiver)
   - must integrate in conceptual models
   - must integrate data in implementation
   - shared objects with multiple interfaces
   - federated data + cross-component links

4. Precise interfaces for 3rd party assembly

---

Business Components - Tiered Architecture

- interface to compose UI parts
- "local" connector
- "distributed" connector
- "distributed" connector

Can package application components as:
- vertical slice
- interfaces to connect Business Logic

Component packaging + reuse at all levels
- interface to compose UI parts
- "local" connector
- "distributed" connector
- "distributed" connector
**Component “Kits”**

- Components are never stand-alone
  - Only meaningful in collections that work well together
  - A component “Kit”

- But the parts must work together in many assemblies
  - Can only happen if they *interoperate* at the appropriate levels

- And each part must be itself flexibly and adaptable
  - … often by configuring its smaller-grained “components”

- So, a component kit is a (potentially open-ended) set of parts built on a coherent architecture

---

**Components without Architecture = Failure!**

- For separately built components to work together they **must** share…

- Standard “horizontal” infrastructure services and interfaces
  - transactions, security, directory, request broker, interface repository...
  - OMG, Microsoft rapidly defining many global “standards”

- Standard “vertical” models of domain concepts
  - What is a “Customer”, “Phone Call”, “Order”, etc.
  - components must use same “domain language” at interfaces
  - OMG defines “Vertical” architectures standards as well

- Standard “connector” mechanisms between components
  - Synchronous / asynchronous message, event, workflow, mobile code
  - Location transparency: CORBA, DCOM

- Other architecture standards
  - Architectural tiers, implicit context passing, lock and connection release...
What is Architecture?

The set of principles and decisions, rules, or patterns about any system that keep its designers from exercising needless creativity
- Desmond D’Souza

- It is not about any specific size, scale, domain, or infrastructure
- Can range from “3-tier C/S” to “use Corba OTS” to “get/set method name rule”
- Includes business architectures: “all operations support are geographically centralized” or “record client company information at first client inquiry”
- Based upon Frameworks

Is This an “Architecture”?

This is an abstract view of the implementation
- It uses the language of properties, events, methods
- ... and of connectors between these “connection points”
- It has a mapping to Java code patterns i.e. a refinement
- This design is an instance of the Java Beans style: design + code
**Architecture as View based on Style**

Event / Property / Method View

ORM Style  
Concurrency Style  
Func-Req Style  

Architecture Style = Language + Rules (for some viewpoint)

Functional View

Object / Relational View

Concurrency View

Architecture = an Abstraction or View of the implementation

**Varying Degree of Generative Style**

- Architectural styles to keep 2 attributes in sync

  - Style 0: “The Cowboy” - do it any way you want
  - Style 1: “2 copies + update protocol” construct defined, use at will
  - Style 2: “1 copy in shared memory” construct defined, use at will
  - Style 3: both Style 1 and Style 2 available, choose at will
  - Style 4: **Whenever** you have a requirement to keep 2 attributes in sync with each other across a distribution boundary with infrequent updates, **use** the “2 copies + update protocol” design
Catalysis - Architecture Style in UML

- Architecture style defined in separate package
- In general, realization refines specification in a way that conforms to the architecture style
- Style constrains realization and/or refinement

Summary - Architecture

- ... it limits needless creativity

- Wide range:
  0% (cowboy) ................. 100% (compiler)

- Architecture defines / uses specific constructs, language, patterns, rules

- Architecture definition sharable across projects
Summary - Component Architecture

- Connectors couple Ports (connection points) of Components
  - Connector abstracts interaction protocol and intermediaries
  - Port abstracts internal structure as connection point
  - Architecture style defines set of port / connector types
- Ports and connectors provide a thinking / design-time tool
  - Implementation is considerably more complex
- Dynamic run-time assembly requires objectified port / connector
  - Alternately, some form of reflective access to components
- Frameworks provide succinct application of all the above

Outline

Introduction
  What problem are we setting out to address?
Components
  What they are, how they interact, how to describe them
Architecture
  What it is, why it is essential, how to describe it
Frameworks
  The basic idea
Reuse
  What it is (and is not), reuse at all levels
Systematic Reuse with Frameworks
  Making models, designs, code reusable
Summary
  Catalysis in Perspective
A Framework is a Skeletal Solution

- Framework defines overall structure of parts and relationships
  - trusses, beams, floor, how they fit together, rules
  - but some specifics are deferred
  - number of floors, layout, wall placement, windows, doors
- You “plug-in” the specifics when “instantiating” the framework
- subject to constraints the framework imposes on the bits you plug in
- A framework helps define and enforce some aspect of architecture

Many Variants by Framework “Plug-In”s
Framework Concept at All Levels

Business Framework
- Reservations
  - resource
  - owner
- room
- hotel

UI Framework
- Master-Detail
  - title
  - abstract
- room
- reservations

Technical Framework
- Event Broadcast
  - event
  - method
- returned
- notify member
- checked out
- clean room

Multiple frameworks used in any app
- Library System
- Hotel System

Outline

Introduction
  - What problem are we setting out to address?
Components
  - What they are, how they interact, how to describe them
Architecture
  - What it is, why it is essential, how to describe it
Frameworks
  - The basic idea
Reuse
  - What it is (and is not), reuse at all levels
Systematic Reuse with Frameworks
  - Making models, designs, code reusable
Summary
  - Catalysis in Perspective
What is Reuse?

What to reuse:
- Code
- Interfaces
- Designs
- Problem Domain Models

How to reuse:
- Don’t reuse code without spec
- Payback includes specs, architecture, … code

What can a Reusable Component include?

- Name + description
- Interface + Specification
- Tests, documentation, example uses...
- Architectural assumptions
- Implementation (executable, design, source, template…)

Model-driven approach keeps these largely technology independent

Good architecture keeps this infrastructure independent

Traceable linkage to...

(Refinement)
Outline

Introduction
What problem are we setting out to address?
Components
What they are, how they interact, how to describe them
Architecture
What it is, why it is essential, how to describe it
Frameworks
The basic idea
Reuse
What it is (and is not), reuse at all levels
Systematic Reuse with Frameworks
Making models, designs, code reusable
Summary
Catalysis in Perspective

Model Frameworks - Generic Models

allocate resources to jobs if resource capability meets job req ...

generalize

allocate room to seminar session if room facility meets session needs...

allocate machine time to batch lot if machine capability meets lot processing ...

Generic model, design, or code

room session time lot

allocate resources to jobs if resource capability meets job req ...

generalize

allocate room to seminar session if room facility meets session needs...

allocate machine time to batch lot if machine capability meets lot processing ...

- A generic model / design / implementation component whose
  - Defines the broad generic structure and behavior
  - Provides plug-points for adaptation
  - Reuse starts with commonality in problems themselves!
Resource Allocation Framework

ResourceAllocation <<framework>>

```
<Requirement> * meets * <Capability>

* requirement capability *

<Job> * schedule 0.1 allocated <Resource>

when: DateRange *
```

invariants

**Job**: // only allocate resource whose capability matches requirements

allocated <> nil implies allocated.capability.meets ->includes (self.requirement)

**Resource**: // resource not double-booked: its jobs dates do not overlap

schedule->forall (j,k | j <> k implies not j.when.overlaps(k.when))

"Applying" frameworks to build a Model

Built by plugging into pre-existing framework (twice)

Seminar Scheduling

```
RoomFacility capability resource Room

Topic requirement ResourceAllocation job

SeminarSession when: DateRange

InstructorSkill * skills Certification * certs Instructor

inv capability == certs.skills
```

### Some Business Model Frameworks

<table>
<thead>
<tr>
<th>Framework</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource Allocation</strong></td>
<td>Assign a resource to a job if the resource capability meets the job requirement, watching for overbooking</td>
</tr>
<tr>
<td><strong>Customer Trends</strong></td>
<td>Track a customer’s preferences for different products by monitoring how frequently he/she has indicated an interest in that product (e.g., by purchasing, calling, requesting samples, …)</td>
</tr>
<tr>
<td><strong>Production and Inventory</strong></td>
<td>Manage just-in-time inventory of some products by tracking the number of items of that product in inventory, and placing an order for the production facility when inventory drops below some threshold</td>
</tr>
</tbody>
</table>

Note: these could be used in very different combinations
A Complete Seminar Business Model

- Built by specializing three different pre-existing model frameworks

Range of Frameworks

- Systematic Reuse with Frameworks
  - Domain Models
  - Design Patterns
  - Abstract and Concrete frameworks via Refinement
  - Architectural Connectors
  - JavaBeans Frameworks
  - Layered Frameworks - Fundamentals to Domains
### Design Patterns as Frameworks

#### Subject-Observer

<table>
<thead>
<tr>
<th>&lt;Subject&gt;</th>
<th>obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>s: State</td>
<td>sub = &lt;Observer&gt;</td>
</tr>
<tr>
<td>inv: changed(s)=&gt;</td>
<td>isProjection(Subject): boolean</td>
</tr>
<tr>
<td>obs-&gt;forall (o</td>
<td>update()</td>
</tr>
<tr>
<td>obs.update(self))</td>
<td>post: isProjection(sub)</td>
</tr>
</tbody>
</table>

**externally**: subject and observers always appear to be in sync

**inv**: Subject->forall (s | s.obs->forall (isProjection(s))

### Applying Design Patterns

- **Power-Switch-Display**
  - **PowerSwitch**
    - isOn: boolean
    - turnOn
    - turnOff
  - **SwitchDisplay**
    - isRed: boolean
    - update

- **SubjectObserver**
  - subject
    - [s = isOn]
  - observer
    - [isProjection(s) = s.isOn <=> isRed]

- Application defines mappings of types, attributes, actions
Factory Pattern: Generative Aspects

Factory

make<Abstract>(): <Abstract>
post
<Abstract>.new = Set { result }

<Concrete>Factory

make<Abstract>(): <Concrete>

Abstract

Concrete

generated factory class, make methods

placeholder

Applying Factory Framework

Thread Management

NT Thread

Abstract

Concrete

Factory

Import with substitution

Solaris Thread

Concrete

Abstract

Concrete

Factory

Factory
The Model is Automatically Generated

Frameworks: Two More Dimensions

- Frameworks can be described at different levels of refinement
- Frameworks themselves are composed of smaller frameworks
Framework for Architectural Connector

Here is what I mean by "Workflow" in any domain...

A connector abstracts some interaction protocol
Connector is used by “plugging” into that framework
Different “connector” frameworks: workflow, events, properties

Summary - Generative Architecture

Architectural style defines language and rules for valid realizations of some specification
Style = Set of <spec, realization, refinement>

Style either defined as constraint or “generative”

Generative style = construct + its realization pattern

Frameworks capture any model pattern
Framework is a package
Pattern application is import + substitute
**Framework for JavaBean <<property>>**

- <<property>> stereotype means a read-write accessor
- Stereotype implies import with substitution
  - import property [ X\EditorCore, property\selection, T\Element ]

**Implementation Frameworks**

- Frameworks can include both models and implementations
- An implementation framework configures a particular set of code components to realize a particular model framework
- Like any framework, it leaves some code “plug-points” for customization - via delegation, sub-classing, code-generation...
**Component Framework: Seminar System**

- Partial implementation with specs of the missing pieces

**Framework for Architectures - All Levels**

- **Business Models**
  - Barter
  - Trader
  - Authorizer

- **Domain Models**
  - Resource Allocation
  - Account Settlement
  - User Interface Patterns

- **Design Patterns**
  - Corba-CGI Gateway
  - Data Marshalling
  - Thread Pooling
  - Subject-Observer
  - 2-Way Link
  - Moving Window

- **Fundamentals**
  - Total Ordering
  - Groups
  - Range
  - Descriptors

- Constructive approach to modeling and design with full traceability
- Libraries and commerce of frameworks of models, designs, and code
**The Vision of Layered Frameworks**

Layered frameworks - fundamentals to domain-specific

Example of Catalysis frameworks in business

- CBOP (Consortium of Business Object Promotion), Japan
- Business Domains: Wholesale sales, Financial Accounting

### Problem Domain to Business Solution

- Problem Domain Models
- Product / tool models based on domain models
- Business Model
- SW Distribution Tool
- Service Request Tool
- SW Distribution Tool
- Service Request Tool
- Business Process Model
- Customer Service
- Deployment

**Problem domain models**

+ process models
+ product models

= Business Solution Model
**Enterprise Models - Package Partitions**

- **Performance Monitoring - Essential**
  - Equipment
  - Statistics
  - Plans
- **Network Expansion - Essential**
  - Equipment
  - Statistics
  - Plans
- **PM Tool Requirements Spec**
  - Spec of a PM tool
  - PM Tool
  - Equipment
  - Statistics
- **Customer Billing**
  - Equipment
  - Customer
  - Bill

**Net Expansion Biz Process - As-Is**

**Net Expansion Biz Process - To-Be**

**Domain, Process, and Machine**

- **Domain Concepts**
  - Essential Model
    - Independent of variable human/machine process
- **Machine Specification**
  - Includes minimal model of Environment
- **Machine Design**
  - Using specs of sub-components C1, C2
- **Process Model**
  - Detailed (uses Machine)
    - Specific detailed process including Machine

- **Horizontal partition**
- **Vertical partition**

- **Equipment**
- **Statistics**
- **Plans**
- **Alarm**
- **Simulator**
- **PM Tool**
- **Design of a PM tool**

- **Import and “say more” about Equipment**

- **Rules constraining design**

- **Uniform structure of specification, refinement, architecture**

- **Clear separation of specification, implementation, usage**
## Component and Enterprise Similarities

<table>
<thead>
<tr>
<th>Components</th>
<th>Enterprise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels of abstraction</td>
<td>Problem Domain vs. Business Process vs. Application Spec vs. Application Impl</td>
</tr>
<tr>
<td>Multiple Views</td>
<td>Security, Transactions, naming</td>
</tr>
<tr>
<td>Architecture standards</td>
<td>Approvals levels, escalation and notification, centralized support operations,…</td>
</tr>
</tbody>
</table>

**Interface vs. Implementation**
- *Shipper* has a different view of *Customer* than *OrderTaker*.

**Problem Domain vs. Business Process vs. Application Spec vs. Application Impl**
- Customer Care department vs. Network Expansion department.

**Customer Care department vs. Network Expansion department**
- Approval levels, escalation and notification, centralized support operations,…

### System Integration - Structure and Route

1. **Business model, approximate spec**
2. **Reverse Eng. Component specs**
3. **Design the Integration Based on Architecture Specs**

<table>
<thead>
<tr>
<th>Spec (A)</th>
<th>Spec (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation A</td>
<td>Implementation B</td>
</tr>
</tbody>
</table>

**Fixed underlying structure, different route and techniques**
Reuse - Two Distinct Processes

Reuse - Investment in Building Assets
Component-Based Development

Component-based Development: a development approach in which

… all artifacts — from executable code to interface specifications, architectures, and business models …

… scaling from complete applications and systems down to individual components …

… can be built by assembling, adapting, and “wiring” together existing components into a variety of different configurations.
Outline

Introduction
What problem are we setting out to address?

Components
What they are, how they interact, how to describe them

Architecture
What it is, why it is essential, how to describe it

Frameworks
The basic idea

Reuse
What it is (and is not), reuse at all levels

Systematic Reuse with Frameworks
Making models, designs, code reusable

Summary
Catalysis in Perspective

What is Catalysis™?

A next-generation standards-aligned method
For open distributed component systems
from components and frameworks
that reflect and support an adaptive enterprise

Precise models and systematic process
UML partner, OMG standards, TI/MS standards
Dynamic non-“stovepipe” systems

Compose pre-built interfaces,
models, specs, implementations...
...all built for extensibility

From business to code

More info at www.catalysis.org and www.platinum.com
Catalysis has been in development and use since 1992
Supports components, OO, legacy, heterogeneous systems
**Catalysis: Beyond UML**

- **UML +** simple consistent approach, process, techniques
  - **Traceability** from business models to code
  - Business-driven, improved change management, quality assurance
  - **Precision**, with clear unambiguous models and documents
  - Uncover issues early, explicit shared vocabulary and understanding
  - **Component** Based Development
  - Interface-centric flexible assembly from parts based on common architecture
  - **Reuse** of designs, specs, problem domain models, architectures, ...
  - Consistent and rapid architecture via patterns and frameworks
  - **Scalability** from small to large teams and projects
  - Consistency, completeness, adoption spectrum, incremental development
  - **Process** that is flexible yet repeatable, with multiple “routes”
    - In terms of flexible process patterns with full process implementation

---

**Traceability from Business to Code**

Zoom in/out of use-case (user task)
(abstract action or detailed dialog)

Zoom in/out of objects
(external or internal view, including software)
Precise Traceability via Refinement

Post: Funds transferred, know-how gained, instructor used

Instructor usage = ...instructor, qual, session...
Client know-how = ...students, skills
$ = ...session, cost, ...
buy course = <schedule + deliver + pay>

Sharp abstraction without zooming into details
Trace mapping defined in "refinement model"
Better tests, inspection, traceability, design reviews

How do Components Interact?

Components interact via clearly specified interfaces

Use higher-level parts and connectors
Focus on individual interfaces
- precise external behavior (provided + required)
- all implementation aspects completely hidden
Systematic Framework-based Reuse

Business Framework
- Reservations
  - resource
  - owner
  - book
  - library
  - room
  - hotel
- UI Framework
  - Master-Detail
    - master
    - detail
  - title
  - abstract
  - room
  - reservations
- Technical Framework
  - Event Broadcast
    - event
    - method
    - returned
    - notify member
    - checked out
    - clean room

Multiple frameworks used in any app

Scalable to Enterprise Modeling

- Separation - separate views with dependencies
- Integration - horizontal & vertical
- Sharing - frameworks for shared models, repeating patterns
- Object-Relational map
- Service-Level Agreement
- Seamless - from biz to code
- Synchronization - between all models and “biz world”
Typical Business System with Catalysis

**Requirements**
Understand problem, system context, arch and non-functional requirements.

**System Specification**
Describe external behavior of target system using problem domain model.

**Architectural Design**
Partition technical and application architecture components and their connectors to meet design goals.

**Component Internal Design**
Design interfaces and classes for each component: build and test.

Outside (+ project constraints)
boundary
inside

**UI Design**
dialog flow, prototype, usability

**DB Design**
class mapping, transactions, etc.

Catalysis Component-Based Development for E-Business www.kinetium.com www.catalysis.org

CBD/Catalysis Process - Web Site version

**Business context, problem definition, solution constraints**

**PROJECT INITIATION**

**PROTOTYPING ITERATIONS**

**SOLUTION DEFINITION**

**TIMEBOXES**

**EVOLUTIONARY DELIVERY**

**INCREMENTS**

**ITERATIVE DEPLOYMENT**

**ROLLOUT ITERATIONS**

Analyze, design, build, test cycles

Deliver solutions
**Process - High Level Package Structure**

Catalysis Component-Based Development for E-Business www.kinetium.com www.catalysis.org

---

**Three Modeling Scopes or Levels**

<table>
<thead>
<tr>
<th>Level/Scope</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain/Business</strong></td>
<td>“Outside”: Identify Problem, Solution establish problem domain terminology understand business process, roles, collaborations build <em>as-is</em> and <em>to-be</em> models</td>
</tr>
<tr>
<td><strong>Component Spec</strong></td>
<td>“Boundary”: Specify Component scope and define component responsibilities define component/system <em>interface</em> specify desired component operations</td>
</tr>
<tr>
<td><strong>Internal Design</strong></td>
<td>“Inside”: Implement the Spec define <em>internal architecture</em> define internal components and collaborations recursively design insides of each component</td>
</tr>
</tbody>
</table>
Three Modeling Constructs

- Model external behavior of an object
- Model interactions of a group of objects
- Map concrete to abstract model

Recurring patterns of collaborations, types, designs, etc. define generically, “plug-in” to specialize

Three Principles

- Abstraction
  - clear views
  - technology insulation
- Precision
  - expose gaps early
  - accurate shared understanding
- Pluggable Parts
  - no duplicate work
  - consistency by reuse
  - quicker development

Abstract descriptions (requirements, architecture, specifications) become robust, reliable, traceability

Reuse of parts includes implementation, interfaces, specifications, requirements architecture, patterns, ...

Reusable parts can be composed reliably and predictably
Adoption Spectrum - Think Big, Start Small

The Key to Catalysis

Minimize the Magic

- Minimize the “magic” that happens in a development process
  - Gaps between business process, software solutions, technical infrastructure
  - Capture known designs, techniques, processes, architectures, …
  - Common vocabulary across business, analyst, architect, programmer
  - Common core techniques for requirements, non-functionals, design, specs...

- Full lifecycle coverage
  - Business problem driven with traceability from requirements to code
  - Rapid application development with reuse of all levels
  - Combine IT Engineering and Business Engineering into one whole
**Experiences with Catalysis**

- **EDS**: Internet Multimedia division
  - Adopted as a required part of development standards 1998
- **Lockheed Martin**: Defense projects
  - Adopted as integral part of standard since 1997
- **USAA**: Insurance
  - Successful application on risk-profiling project 1998
- **Yellow Services**: Travel and Transportation
  - In current use in Enterprise Systems Management domain modeling and CBD
- **Credit Suisse**: Asset Management
  - Adopted as integral part of standard 1998
- **Texas Instruments WORKS**: Factory Automation 1997-1998
  - Successful and “deep” use on capacity planning and scheduling
  - Successful “lite” use on overall project
- **Daimler Benz**:
  - Used since 1998 with good results
  - “easy to understand core, very consistent and complete overall method”
- **Visa / Chicago**, **BMW**, **Nortel**, **Olivetti**, **Siemens**, **Dutch Ministry of Taxes**, **KPMG/Germany**, **LCM/Italy**, and more …

---

**Success With Components**

1. **Business Driven**
   - Understand business first
   - Relate disparate views

2. **Traceability**
   - Bridge Business and IT
   - Precise shared vocabulary
   - Critical business questions early refinement across levels

3. **Pluggable Components**
   - Precise interface-centric design
   - Traceable business components

4. **Architecture**
   - Infrastructure
   - Interfaces, connectors
   - Plug standards
   - Interoperability

5. **Reuse Process**
   - Develop for reuse
   - Develop with reuse
   - First reuse problem model
   - … then architectures
   - … then interface, code
References


UML 1.3 Specification: uml.shl.com

UML 2.0 Working Group documents: uml.shl.com


Catalysis overviews and discussions: www.catalysis.org