# Raising the Level of Development: Models, Architectures, Programs

Dr. James Rumbaugh
IBM Distinguished Engineer

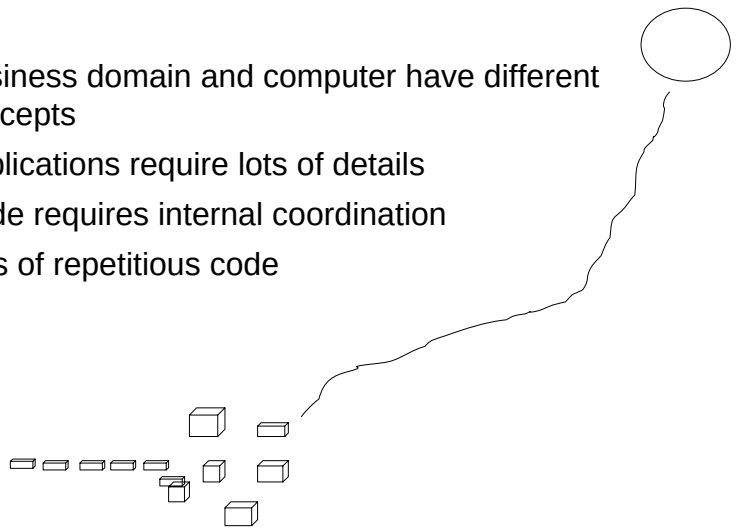
Rational. software





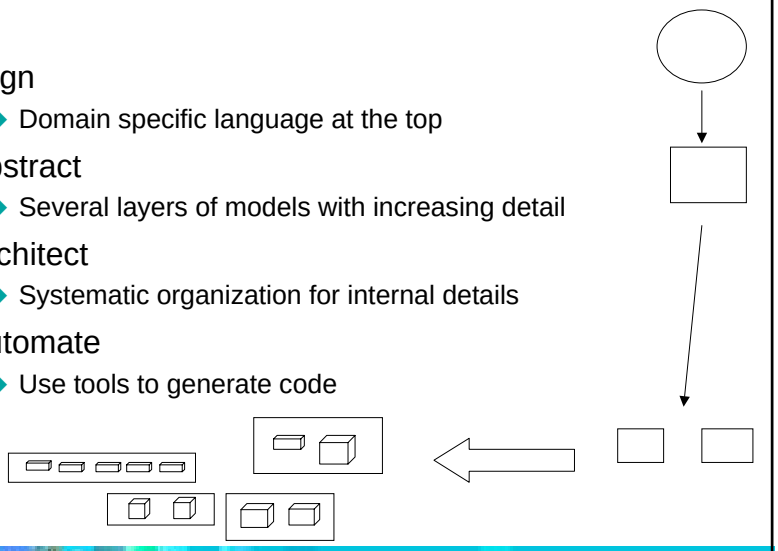
## Why Is Software Difficult?

- Business domain and computer have different concepts
- Applications require lots of details
- Code requires internal coordination
- Lots of repetitious code



# Bridging the Gap

- Align
  - Domain specific language at the top
- Abstract
  - Several layers of models with increasing detail
- Architect
  - Systematic organization for internal details
- Automate
  - Use tools to generate code





# Code-Only Development

- Deceptively simple
- Lack of planning leads to trouble
  - Weak architecture
  - Bad decomposition
  - Inconsistent formats and interactions
  - Bad for teamwork
  - Lots of detailed work
  - Brittle for changes

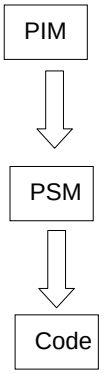
# Model Driven Architecture (MDA)

- Build high-level models
  - Expressed in domain concepts
- Generate platform-specific code
  - Into specified architectures
- Using automated tools
  - Based on standards



# MDA Approach

- Object Management Group (OMG) initiative
- Platform Independent Model (PIM)
  - Captures logic of the application
- Implementation profile
  - Platform
  - Technology decisions
- Platform Specific Models (PSM)
  - Target multiple platforms
  - Translator incorporates platform knowledge



# Needed for MDA

- Problem domain models
  - UML or domain-specific language (DSL)
- Architecture frameworks
  - Make assumptions about infrastructure
  - Use them to remove details from models
- Automated tools
  - Modeling tools
  - Repositories
  - Translators

IBM Software Group | Rational software

# Needed for MDA

- Problem domain models
- Architecture frameworks
- Automated tools

# UML

- Build models in UML (Unified Modeling Language)
- Higher level than programming languages
- Adapt UML to application domains with profiles
  - Special domain concepts
  - Constraints on the use of general UML constructs
- UML is still a general-purpose language
  - May be a lot of repetitious details



# Domain Specific Language (DSL)

- Syntax and semantics for a particular purpose
- Reduce conceptual gap
  - Capture business-level content
  - More intuitive syntax
- Reduce mindless repetition
  - Predefined control and data patterns
  - Eliminate repetitious control specification
  - Generate low-level details

# DSL Examples

- yacc
  - Unix compiler-compiler
- GUI Builder
  - Visually construct user interface
- Desktop publishing editor (FrameMaker, Word)
  - See the final output while editing
- MIDI
  - Music specification language
- Mathematica
  - Mathematics notation



# SDL

- Specification and Description Language (SDL)
- International Telecommunications Union (ITU) standard
- Special syntax and logic for telephone systems
- SDL concepts added to UML 2.0
- SDL can be modeled as a UML profile

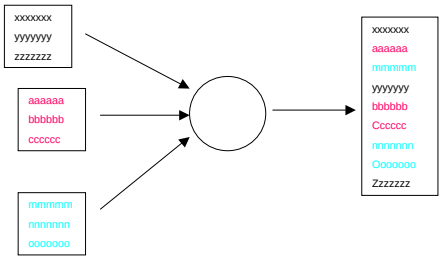
# DSL Styles

- Declarative
  - Say what to do
  - Avoid control logic
- Imperative
  - Say how to do it
  - Explicit control logic
- Hybrid
  - High-level declarative
  - Low-level imperative



# Aspect Oriented Programming (AOP)

- Separate different functional aspects
  - Distinct syntax for each aspect
  - Aspects cut across implementation code
  - Examples: Security, persistence, concurrency, logging
- Translator weaves aspects into code



# AOP is a Kind of DSL

- DSL and AOP depend on good translators
- Current AOP languages are awkward
  - Too much explicit coordination
- Expect AOP features from DSL effort



# Needed for MDA

- Problem domain models
- Architecture frameworks
- Automated tools

# Architecture

- Decomposition into subsystems
- Topology
- Interaction rules
- Data formats
- Resource management
- Hooks for future extensions
- Scaffolding for testing



# Architecture Framework

- Framework = reusable architecture pattern
- Includes:
  - Skeleton execution environment
  - Predefined subsystems and topology
  - Interaction and data rules, formats, interfaces
  - Attachment points for plug-ins
  - Component libraries of useful functionality
  - Sample applications
- Implementation development environments (IDE)
  - Eclipse, .NET, J2EE

# Framework Advantages

- Enable DSL to eliminate repeated patterns
- Ensure consistency
- Excellent for incremental design
- Enforce design trade-offs



# Specifying Frameworks

- UML models
  - Structure models
  - Interaction models
  - Interface models
  - Patterns
- DSL syntax
- Code
- Text descriptions

# Other Experience-Based Content

- Patterns
  - Solution to a common problem
  - Parameters
  - Design, modeling, architecture level
  - Has structure, interaction rules, trade-offs, guidelines
- Components
  - Module ready to plug in
  - Has UML models, code, text descriptions, keywords



# Needed for MDA

- Problem domain models
- Architecture frameworks
- Automated tools

# Modeling Tools

- Build models
  - Edit models
  - Apply patterns
  - Organize large models
  - Coordinate among large teams
  - Check for errors and bottlenecks
- Generate code
  - Load technology and architecture assumptions
  - Trace generated code to model



# Repositories

- Standard format
- Support multiple tools and languages
- Access and update models
- MOF (Meta-Object Protocol)
- XML (Extended Markup Language)

# Translation

- Map from DSL and UML to code
- Use platform knowledge
- Optimize code
- Reformat to match target language
- Query-View-Transformation (QVT) project to define standard format



# Translator Requirements

- Understand source and target syntax
- Generated code must be efficient enough
  - Plenty of experience building good compilers
- Support programming language plug-ins
  - Lots of existing code
  - Handle unexpected situations
- Incremental
  - Maintain traceability
  - Small change to source → small change to target

# Transformation Styles

- Partial
- Full
- Embedded code in model
- Built-up transformation



# Partial Transformation

- Transform source to code
  - Implementation details are missing
  - Usually only data structures and code outlines
- Then edit the code manually
  - Gets code stared quickly
  - Finish the job by programming
- Round-trip transformation possible
  - Maintain traceability links
  - Reverse transformation often ambiguous
  - Difficult to maintain synchronization
- Quick way to start using models

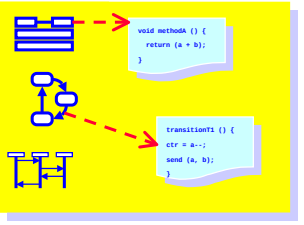
# Full Generation

- One-way transformation from source to code
  - Generate code with all details
  - Don't touch the generated code
  - Have to anticipate all details in the DSL
- Compiler
- Format conversion
  - UML-to-XML
- Good in tightly defined situations



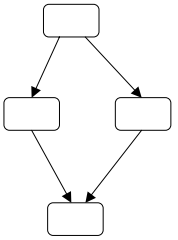
# Embedded Code in Model

- Embed code fragments in the model
  - When DSL is inadequate
- Generate code from model
  - Insert embedded code into generated code
- Edit the source model to make changes
  - Regenerate code when needed
  - Keeps the entire specification together
- Flexible but mixes levels



# Built-up Transformation

- Explicitly model the transformation itself
  - Assemble out of smaller pieces
- Adjust the transformation to change the result
  - Design = building a transformation
- Source model doesn't change
  - Reuse on many source models
- Can have multiple transformations
- Powerful but tricky to build





# One-way and Two-way Transformation

- One-way—generate source to target
  - Can't modify target code
  - Can insert plug-ins to code
  - No synchronization problems
  - Can add and modify nested code without regeneration
- Two-way—round trip engineering
  - Maintain traceability links
  - Modify source or target
  - Propagate changes to other model
  - Much more difficult to implement

# Executable Models

- Allow execution with partial specification
  - Dummy functions
  - Missing data
  - Unknown decisions
- Incomplete models execute with less detail
  - May require user guidance
- Test architecture early
- Add detail later



# Executable UML

- Plain UML is not executable
  - Pick action language (built-in actions too primitive)
  - Select semantic variation points
  - Supply missing execution semantics
    - Tasking, external formats
- It is still a general-purpose language
  - DSLs may be more expressive for most purposes
- It may be a useful intermediate format
  - Output of DSL mappings
  - Input to code/platform mappings

# Standards

- Models
- Architecture frameworks
- Code plug-ins
- Encourage an ecology of vendors
  - Big vendors: Large generic tools
  - Small vendors: Niche-specific plug-ins
  - Use plug-ins with generic tools
  - Standards guarantee access to entire market
  - Example: Photoshop filters



# Status

- MDA is still early
- DSLs are still rare but coming
- Generic frameworks exist
  - Domain frameworks will be developed
- Transformation technology under development
  - QVT standard
  - Much existing compiler experience
- MDA will greatly improve software development


Thank
You