Managed Architecture of Existing Software as a Practical Transition to MDA

Nikolai Mansurov
Chief Scientist & Architect
Overview

- Introduction: The Complete Life-Cycle of software production
- The challenges of the complete-life cycle
  - New Development
  - Evolution
    - Program Comprehension
    - Architecture Erosion
    - The increasing cost of production
  - Modernization
- Model-Driven architecture (MDA)
- Managed Architecture
  - Architecture Excavation
  - Architecture Management
  - Architecture Enforcement
  - Refactoring
- Managed Architecture and MDA
- Remaining challenges
- Conclusions/Key points
Production of software is evolutionary

- Production of software involves multiple releases
  - ...despite the fact that many textbooks emphasize only the initial release, when the system is built
  - Software production after the initial release has an additional constraint of dealing with existing code

- Changes are made to software after the initial release in order to:
  - Develop new functionality
  - Fix bugs
  - Adapt to new operating environments
  - Improve quality

From Ian Sommerville, Software Engineering, 2000
Traditional definition of maintenance

- Software maintenance has been defined within the ANSI standard [ANSI83] as

"the modification of software products after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment."
Software production with existing code

- Changes to existing code may have different *magnitude*:
  - Traditional maintenance (repair)
    - Perfective maintenance
    - Adaptive maintenance
    - Corrective maintenance
    - Preventive maintenance
  - Major new features to existing software
    - Major modifications
    - Scaling
  - Modernization (beyond maintenance)
    - Porting to a new platform
    - Migration to a new technology
    - Migration to COTS components
    - Modularization and refactoring
  - Redevelopment

From Ian Sommerville, *Software Engineering*, 2000
Changes of *increasing* magnitude are required to address changing business needs.

From Seacord, Plakosh, Lewis, SEI, 2003
Evolution of existing code has significant economic impact

- More than one half of all programmers are already working with existing code (repair + enhancement)
- The number of programmers working with existing code grows faster than the number of programmers developing new software
  - Larger amounts of software already developed
  - Large business value accumulates in existing code over time
  - Increasing cost of new development
  - The useful lifespan is increasing
  - Bigger churn of platforms and technologies
  - More defects

From Deursen, Klint, Verhoef, 1999

© 2004, Klocwork Inc.
Cost of evolution overshadows the cost of the initial release

- The cost of post-initial evolution has grown from 40% to 90% of the total production cost
- The cost of the initial release is only 10% of the total production cost
- Software is getting larger, more feature-rich
- Overall cost of development is increasing
- The useful lifespan is increasing
- Production cost is increasing over time

From Lehman, 1999
Complete life-cycle: initial release and post-build evolution

new development challenges

value

new development

Release 1 Release 2 Release 3

evolution

evolution challenges

© 2004, Klocwork Inc.
Where do we go from here?

- OMG’s Model-Driven Architecture (MDA) is a new development approach focused at complete life-cycle of software production. It emphasizes
  - Modeling using UML, MOF, etc,
  - separation of Platform-independent and Platform-dependent concerns,
  - transformation of models (eventually – into the code)
  - shifting away from maintenance of the code to maintenance of the model and the transformation specifications
- MDA addresses the program comprehension challenge: once started with an upfront model the knowledge is never lost
- MDA addresses modernization challenges

- Managed Architecture is an new development approach focused at evolution of existing software assets. It emphasizes
  - excavation of an Architecture Model,
  - proactive enforcement of architecture integrity
- Addresses program comprehension challenge: re-capture preserve and accumulate the knowledge of the software,
- Eliminates or slow down architecture erosion by visualization and impact analysis
- Refactoring of the Architecture Model drives modernizations

- Managed Architecture can also kick-start transition into MDA when the upfront model is not available
Challenges of software production: new development

- Developing new software is challenging:
  - need to understand requirements
  - make decisions

![Diagram showing the process of software development and client needs vs. implemented features.](image)

- How the client has explained his need
- What was implemented
- What was in the requirements specification
- What was in the design specification
- What the client needed

Adapted from www.oper.ru
Challenges of software production: evolution

- Software production after the initial release has an additional *constraint* of dealing with *existing code*
- Evolution of existing software *adds* new challenges:
  - Make new decisions to change existing code and not to break existing features
  - Understanding existing decisions
  - The hierarchy of decisions requirements-architecture-design-code often not available; most information needs to be recovered from code
  - High volume and complexity of information
  - Evolution pace of documentation and code are different– there can be significant lags
  - Maintenance often assigned to junior staff
  - Loss of knowledge
  - Knowledge “walks away”
Challenges of multi-release production

From Schach, Tomer, 2001
Program comprehension challenge

- The costs of performing program comprehension have been widely cited as being between 50 and 90 percent of the overall cost of performing maintenance [Standish84]
- Program comprehension during software maintenance involves the acquisition of knowledge about programs, as well as accompanying documentation and operating procedures

From Boldyreff, 2002
High volume and complexity of information

- Assuming 4,000,000 lines of software printed on listing paper covers 1 mile
  - average data processing program is 0.06 mile
  - complete works of Shakespeare is 0.3 miles
  - BT customer support software 10 miles
  - Royal Bank Toronto 205 miles
  - total COBOL and C 500,000,000 miles

- 1/10 mile is baseline for person to understand
- Growth rate per year 2 Billion lines
- Estimations indicate Windows XP contains upwards of 50 million lines of code. The source code for NT doubles every 800 days.

From Boldyreff, 2002
The Cost of Program Comprehension

- Program comprehension consumes a significant proportion of effort and resources.
- HP estimated that reading code costs $200 million a year.
- Maintenance expenditure increases as the age of the software increases since documentation becomes out of data or if it is carried out by maintainers other than the original developers.
- Changes are necessary so these problems must be overcome.

From Boldyreff, 2002
Program comprehension and production cost

new development
Architecture Erosion

- **Architecture Erosion**: Positive feedback between
  - the loss of software architecture coherence
  - the loss of the software knowledge
    - less coherent architecture requires more extensive knowledge
    - if the knowledge is lost, the changes will lead to a faster deterioration
- **loss of key personnel = loss of knowledge**
## Lehman’s laws on program change dynamics

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuing change</td>
<td>A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.</td>
</tr>
<tr>
<td>Increasing complexity</td>
<td>As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.</td>
</tr>
<tr>
<td>Large program evolution</td>
<td>Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release.</td>
</tr>
<tr>
<td>Organisational stability</td>
<td>Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.</td>
</tr>
<tr>
<td>Conservation of familiarity of</td>
<td>Over the lifetime of a system, the incremental change in each release is approximately constant.</td>
</tr>
</tbody>
</table>
Program change

Prior to making a change it is essential to understand the software product as a whole. During maintenance this involves

- having a general knowledge of what the software does and how it relates to its environment
- identifying where in the system the changes are to be effected
- having an in-depth knowledge of how the parts to be corrected or modified work
The Impact of software change

- Productivity decreases for maintenance staff
- Cost increase for maintaining software
- Code is harder to understand
- Documentation become out of date
- The number of errors reported increases
- The original design becomes corrupted
Strategies for comprehension

- Types of strategies applied are
  - **systematic strategy**: using this strategy, the entire program is examined to establish the interactions between components. Maintenance begins only once the understanding process is complete i.e. when the maintainer considers the mental model to be accurate.
  - **as-needed strategy**: using this strategy the maintainer only attempts to gain an understanding of part of the program. In this case, only a partial model of the software is devised before the modifications process begins.
Plans

- Much discussion by researchers in the field of program comprehension has concentrated on program plans.

- Two levels of program plans are discussed:
  - At the lowest level we have what are termed 'rules of programming discourse'. These rules are said to be composed of the conventions of programming such as the use of meaningful variable names.
  - The other types of plans are 'program plans'. These are said to be stereotypic action sequences such as a high level functional description of a loop. Programs are said to consist of a number of plans.

- If plans are well implemented, they can provide the maintainer with a high level of understanding of the program under maintenance.
Plans become de-localized over time

- Over time plans become de-localised, interleaved, merged or nested with other plans and thus their benefits for maintenance can be reduced.
- The problem with the information plans provide is that the use of a as-needed strategy with de-localised plans can lead to incorrect assumptions being made about the way a program functions.
- For this reason, while the as-needed strategy is a more realistic approach to tackling maintenance, it is also a risky process. Both systematic and as-needed strategies need to be applied but at different levels of detail.
Results of analyzing files

- Increase in size of most applications
- Increase in the number of files
- Some files more highly maintained, others hardly ever maintained
- Plans become de-localised
Results of analyzing functions

- New functions are added to code
- Objects are created via a splitting process
- Call depth increases
- Over time the number of functions increases
- Call graphs have specific traits than can be used to spot legacy tendencies
Results of analyzing data

- Localized data items become de-localized
- Interfaces between modules become more complex
Implications for comprehension

- De-localised plans - modules of functionality are split
- More to comprehend as code becomes larger and more complex
- Larger impacts of change due to ‘globalization’ of data
- Documentation not kept up to date with changes
The cost of production increases over time

- The cost of changing software increases over time, and is not linear, but exponential
  - Brooks attributes the exponential rise in costs to the cost of communication
  - Maintenance corrupts the software structure so makes further maintenance more difficult
  - The number of error reports increases
  - Productivity of the maintenance staff decreases
  - Code becomes harder to understand

- **Architecture Erosion: Positive feedback between**
  - the loss of software architecture coherence
  - the loss of the software knowledge
    - less coherent architecture requires more extensive knowledge
    - if the knowledge is lost, the changes will lead to a faster deterioration

- loss of key personnel = loss of knowledge
- Exponential cost of production may lead to *project failures* (Negative ROI)

From C.Mangione, CIO Update, 2003
Need to manage the cost of production

- Managing features vs “software infrastructure” to avoid negative ROI
  - Manage new functionality and features
  - Improve “software infrastructure”:
    - Eliminate or slow down architecture erosion
    - Upgrade the robustness of the software infrastructure to balance the addition of the new features
- Measures to eliminate or slow down architecture erosion:
  - Preserve and accumulate knowledge
  - Restore architecture coherence
  - Proactively slow down erosion by visualization of the architecture and impact analysis
  - Refactor software to introduce better firewalls (localize design plans, scope the need for understanding and change; scope the impact of changes)
Challenges of the post-build production

What features were working in the first release

What was in the maintenance documentation

What was understood by reading the code

What was in the second release

What is the production cost

Adapted from www.oper.ru
Modernization

- Modernization (beyond maintenance)
  - Porting to a new platform
  - Migration to a new technology
  - Migration to COTS components
  - Modularization and refactoring
When do you need to modernize?

Modernization can be caused by several factors:

- Major New functionality/Changes in functionality
- Modularization to improve reuse of the core software assets, esp. in an SPL
- Migration to standard COTS components
- Platform change or upgrade
- Modernization of software development technology
- Decision to improve robustness of software
- Outsourcing, reorganization of the company
Aspects of modernization

Modernization may involve several different aspects:

- Source code (programming language constructs)
- Design (algorithms, design patterns)
- Architecture (packages, components and their dependencies)
- Business rules (business processes, scenarios, requirements)
Modernization and production cost

new development  maintenance
Model Driven Architecture

From R. Soley, OMG, 2003

© 2004, Klocwork Inc.

Automated Solutions for Understanding and Perfecting Software
What is Model Driven Architecture?

- **A New Way to Specify and Build Systems**
  - *Based on modeling with UML*
  - Supports full lifecycle: analysis, design, implementation, deployment, maintenance, evolution & integration with later systems
  - Builds in Interoperability and Portability
  - Lowers initial cost and maximizes ROI

- Applies directly to the mix you face:
  - Programming language
  - Operating system
  - Network
  - Middleware

From R. Soley, OMG, 2003
Building an MDA Application

Start with a Platform-Independent Model (PIM) representing business functionality and behavior, undistorted by technology details.

From R.Soley, OMG, 2003

MODEL DRIVEN ARCHITECTURE®

Automated Solutions for Understanding and Perfecting Software

© 2004, Klocwork Inc.
Generating Platform-Specific Model

MDA tool applies a standard mapping to generate *Platform-Specific Model* (PSM) from the PIM. Code is partially automatic, partially hand-written.

From R.Soley, OMG, 2003
MDA tool applies an standard mapping to generate *Platform-Specific Model* (PSM) from the PIM. Code is partially automatic, partially hand-written.
Generating Implementations

MDA Tool generates all or most of the implementation code for deployment technology selected by the developer.

From R. Soley, OMG, 2003
Integrating Legacy & COTS

MDA Tools for reverse engineering automate discovery of models for re-integration on new platforms.

From R. Soley, OMG, 2003
MDA helps manage production cost

...by addressing program comprehension challenge...

...and modernization challenges...

new development

maintenance=continued development
Existing code becomes the barrier for adopting MDA

new development maintenance

MDA ???
Managed Architecture

- Managed Architecture is an new development approach focused at evolution of existing software assets. It emphasizes
  - excavation of an Architecture Model,
  - proactive enforcement of architecture integrity
- Addresses program comprehension challenge: re-capture preserve and accumulate the knowledge of the software,
- Eliminates or slow down architecture erosion by visualization and impact analysis
- Refactoring of the Architecture Model drives modernizations
Architecture Capability Maturity

- Single aspect of SEI-CMM – refers to the architecture of the existing software
  - **Level 1:** Initial architecture
  - **Level 2:** Repeatable architecture
    - Some packaging rules, some use of libraries, some code reuse
  - **Level 3:** Defined architecture
    - Components and their interfaces are formally defined and maintained together with the rest of the code,
    - modelling tools like Rational Rose are used
    - middleware or component environment is used,
  - **Level 4:** Managed architecture
    - Visualization of existing software is available, feedback between the “as designed” architecture and the “as built” architecture is available, metrics of existing architecture are used, architecture integrity is enforced
  - **Level 5:** Optimizing architecture
    - On-going architecture improvement is part of the overall development process
Managed Architecture: overview

- **Visualize Architecture** of the “as built” software as a *model*
  - “Excavation”, because the model is likely to be created manually by abstracting from existing code
  - Abstraction level is *sufficient*, when the architect can
    - Explain the model to the team
    - Use the architecture model to detect anomalies
    - Plan refactorings through the model
    - Enforce integrity of the model

- **Align the “as built” model with the “as designed” description**, if available

- **Use the architecture model** to detect anomalies and effects of architecture erosion

- **Use the architecture model** for impact analysis

- **Use the architecture model** for efficient program comprehension

- **Change the architecture model** to plan and implement architecture improvement
Architecture Excavation summary

- Build Architecture Model of existing code
  - Scalable
  - precise
  - Can be automatically updated as changes are made to the code
  - Can be refactored
- Model visualizes the effects of architecture erosion
- Model allows understanding code
- Eliminates or slows down erosion:
  - Restore architecture cohesion, therefore slowing down architecture erosion
  - Proactively slow down erosion by impact analysis
  - Plan modernizations to improve the robustness of the software infrastructure as the new features are being added
- Model preserves and accumulates the architectural knowledge about the software
- Architecture model is shared among the whole development team
- Model is changed to drive new development
Excavated high-level architecture

before

after
Why SW Architecture is so hard?

- Because we think code is the most important thing our development teams produce
  - Human beings will optimize any single indicator of success
- Because architecture is not visible or tangible in any way
  - You can’t manage what you can’t measure
- Because all our processes and tools are focused on the “programming in the small” problems (i.e. code)
  - code inspections processes
  - debuggers
  - xref
  - tools to measure code coverage under test
  - tools to detect memory leaks
  - run time profilers
  - IDE’s, Languages, SCM systems
OK – How do we manage it?
inForce System
SWOT Source Code Analysis

- Comprehensive
- System level understanding
- Every build

- Architecture and design issues
- Code defects and metrics

- Coding and design rule violations
  - Encapsulate build rules
  - Set baseline architecture
  - Define and enforce APIs
inSight Architect
Design Analysis

- System Design
  - Understand
  - Visualize
  - Customize
- On-the-fly changes
- Every build
  - “Analyze what you build”: all compile flags, directives, macros expanded prior to analysis.

- External environment
- Design anomalies

© 2004, Klocwork Inc.
inSpect System
Source Code Analysis

- Expansive hyperlinked report
- System-scoped recommendations
- Detailed dependency analysis
Managed Architecture: Enforcing Integrity

- Architecture rules
- Metrics
- Strategies of enforcement
  - Desktop
  - Clearcase
  - Post-build
…and how to enforce architecture?
inForce Developer
Productivity and Quality Control

inForce Developer report in gvim editor

1- Pick a problem here
Coding problems
Security vulnerabilities
Design infractions
Metric violations
Include anomalies

2- Jump to the line to analyze and fix it here

View and edit source code

© 2004, Klocwork Inc.
Managed Architecture and Production cost

excavation

new development maintenance management

© 2004, Klocwork Inc.

Automated Solutions for Understanding and Perfecting Software
Managed architecture: Refactoring

- Use the architecture model to proactively refactor software
  - Initial refactorings to remove physical irregularities
  - Minor refactorings to restore architecture cohesion
  - Larger refactorings to improve architecture robustness
  - Major refactorings (modernizations, reuse, redevelopment)
Refactorings, supporting harvesting for reusable components

- Locate functionality to be reused, encapsulate in a new virtual component
- Identify additional dependencies of the new component
- Follow dependencies, analyze and add to the new component
- Refactor remaining dependencies (usually, inversion, or adaptor layer)
- Confirm results
Refactorings, supporting adoption of standard COTS components

- Locate functionality to be retired, encapsulate in a new virtual component
- Split new component into core and adaptor
- Identify architecture mismatches with the new COTS component
- Refactor to match
- Introduce placeholders for new components
- Identify new adaptor layer to the COTS component
- Implement refactoring
- Add missing functionality to placeholders
- Create stub for the new COTS component
- Confirm results
Refactorings, supporting platform changes and upgrades

Platform – all system software, including operating system, standard run-time libraries (e.g. graphical libraries), middleware, any other external 3rd-party run-time components (e.g. Web-browser, Web-server, Application Server, etc.)

- Locate old platform usage, encapsulate in a new virtual component
- Identify adaptor layer
- Refactor adaptor
- Identify inverted dependencies
- Refactor inverted dependencies
- Introduce placeholders for new functionality
Refactorings, supporting modernization of software development technology

Software development technology – all modeling languages, programming languages, etc., and the corresponding tools (such as compilers, macroprocessors, build automation systems, code generators, simulators, etc.)

- Migration, related to the changes in the development environment
- Porting into another programming language
- Migration into MDA
Production cost: Big-Bang Transformation and Refactoring

- Big-Bang Transformation
- Refactoring
- Release 1
- Release 2
- Maintenance
- New Development
- Refactoring

© 2004, Klocwork Inc.
Software evolution with Managed Architecture

Analysis phase
- Launch
- Inventory Assessment
- Architecture Excavation
- Conceptual Analysis
- Robustness Analysis

Implementation phase
- Review
- Cleanup
- Integration

Step applied to each component (in parallel)
Step applied to the whole system

© 2004, Klocwork Inc.

Automated Solutions for Understanding and Perfecting Software
Transition to MDA

- Architecture Models for Managed Architecture are not UML, because of scalability, the need to evolve with the code, and specific “existing code” understanding concerns, like links to the code, navigation, etc.
- Transition to UML is straightforward
Generating UML component view
Interface of the selected component

<table>
<thead>
<tr>
<th>Identifier</th>
<th>in File</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAXParseException</td>
<td>sax.cpp</td>
<td>3</td>
</tr>
<tr>
<td>EntityResolver</td>
<td>EntityResolver.cpp</td>
<td>7</td>
</tr>
<tr>
<td>ErrorHandler</td>
<td>ErrorHandler.cpp</td>
<td>6</td>
</tr>
<tr>
<td>Locator</td>
<td>Locator.cpp</td>
<td>4</td>
</tr>
<tr>
<td>Parser</td>
<td>Parser.cpp</td>
<td>3</td>
</tr>
<tr>
<td>DocumentHandler</td>
<td>DocumentHandler.cpp</td>
<td>2</td>
</tr>
<tr>
<td>InputSource</td>
<td>InputSource.cpp</td>
<td>13</td>
</tr>
<tr>
<td>HandlerBase</td>
<td>HandlerBase.cpp</td>
<td>1</td>
</tr>
<tr>
<td>AttributeList</td>
<td>AttributeList.cpp</td>
<td>2</td>
</tr>
<tr>
<td>SAXException</td>
<td>SAXException.cpp</td>
<td>3</td>
</tr>
<tr>
<td>DTDHandler</td>
<td>DTDHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>AttributeList</td>
<td>AttributeList.cpp</td>
<td>1</td>
</tr>
<tr>
<td>DocumentHandler</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>characters</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>endDocument</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>endElement</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>ignorableWhitespace</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>processingInstruction</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>resetDocument</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>setDocumentLocator</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>startDocument</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>startElement</td>
<td>DocumentHandler.cpp</td>
<td>1</td>
</tr>
<tr>
<td>DTDHandler</td>
<td>DTDHandler.cpp</td>
<td>6</td>
</tr>
<tr>
<td>notationDecl</td>
<td>DTDHandler.cpp</td>
<td>2</td>
</tr>
<tr>
<td>unparsedEntityDecl</td>
<td>DTDHandler.cpp</td>
<td>2</td>
</tr>
<tr>
<td>resetDeclType</td>
<td>DTDHandler.cpp</td>
<td>2</td>
</tr>
<tr>
<td>InputSource</td>
<td>EntityResolver.cpp</td>
<td>10</td>
</tr>
<tr>
<td>EntityResolver</td>
<td>EntityResolver.cpp</td>
<td>10</td>
</tr>
<tr>
<td>resolveEntity</td>
<td>EntityResolver.cpp</td>
<td>4</td>
</tr>
<tr>
<td>ErrorResolver</td>
<td>ErrorResolver.cpp</td>
<td>20</td>
</tr>
<tr>
<td>warning</td>
<td>ErrorHandler.cpp</td>
<td>4</td>
</tr>
</tbody>
</table>

© 2004, Klocwork Inc.
Architecture-driven migration to MDA

PIM

PSM

PIC

PSC

CM

existing code

code

rules

enforce

refactor

generate

integrate

integrate

generate

excavate

integrate

generate

Architectural-driven migration to MDA
Architecture-driven migration to MDA

- First step is to understand existing software assets
- Then identify and isolate platform-specific assets
  - Refactor existing assets into platform-independent layer and platform-specific layer
- Perform Architecture Excavation (recovery and refactoring)
  - Top-level model
  - As many levels as necessary, as little as possible
  - Physical irregularities are removed
- Decoupling platform-specific from platform-independent layers
- Clean-up to improve design and architecture (optional)
Generate UML model(s) for platform-independent components, import into UML tool
  • Contains components and interfaces
  • Does not model behavior
  • Is precise with respect to the existing (re-factored) code

Use generated models to generate platform-specific skeletons (e.g. to migrate to new platform)

Use generated model to interface newly developed components to it

Generate Adaptors to integrate automatically generated code with legacy components (optional)

Generate rules to enforce excavated design
Managed Architecture can kick-start MDA

excavation

new development

maintenance

© 2004, Klocwork Inc.
Remaining challenges

- Maturation of MDA tools
- Addressing behavior modeling, not just components and their interfaces
For vendors of software-intensive products the evolution of existing code base is a vital part of day-to-day software development and maintenance. Modernization is used when the regular maintenance can no longer address the changes.

Managed Architecture is a new development approach focused on evolution of existing software assets. It emphasizes:

- excavation of an Architecture Model,
- proactive enforcement of architecture integrity

- Addresses program comprehension challenge: re-capture, preserve and accumulate the knowledge of the software,
- Eliminates or slow down architecture erosion by visualization and impact analysis,
- Refactoring of the Architecture Model drives modernizations

Conclusions/Key points
Conclusions/Key points

- Transition to MDA has strategic, long-term benefits for organizations. Modernization in the context of MDA creates the model of the code for the purpose of importing it into MDA-enabled development environment.
- Managed Architecture Approach can be used as a migration strategy into MDA for the projects where the upfront model is not available.
Conclusions/Key points

- Architecture-driven migration to MDA:
  - the main priority is to gain intellectual control over the architecture of existing software.
  - Short-term benefits from gradually improving architecture and maintenance robustness of existing software
  - Use Managed Architecture approach to get
    - Big picture of the system: major components and structures, relations between them
    - Common context for documenting, developing and educating others about the system as it evolves, especially the legacy components
    - Common repository for all architectural and design artifacts related to
  - Use MDA tools to do
    - platform-independent modeling and validation
    - platform-specific modeling
    - Automatic code generation
    - Common context for documenting, developing and educating others about the system as it evolves
  - Use Managed Architecture techniques to enforce integrity of achieved architecture improvements