NDIA Systems Engineering Conference 2011

Technical Basis of Model Driven Engineering

©2011 Google - Imagery ©2011 Google - Terms of Use - Edit in Google Map Maker

Dr. William Bail

The MITRE Corporation
NAVSEA PEO IWS
The University of Maryland

24 Oct 2011

Agenda

- Overview of tutorial
- Models
- Model Driven Engineering (MDE)
- Overview of software development activities
- Software development artifacts
- Model Driven Architecture® (MDA®)
- Assessment
- Recommendations

Overview of tutorial

- Intend to provide a perspective on the underlying nature of model driven engineering
  - What does it really do for us?
- Cover
  - nature of models
  - the concepts of model driven engineering and where they fit into traditional software development life cycle
  - advantages and limitations of MDE
  - considerations in applying MDE
  - basic economics of MDE
- Will not cover
  - Specific tools and approaches, “how-to”, what to buy
- Focus is on underlying principles

Agenda

- Overview of tutorial
- ➡ Models
- Model Driven Engineering (MDE)
- Overview of software development activities
- Software development artifacts
- Model Driven Architecture® (MDA®)
- Assessment
- Recommendations

Terminology – models

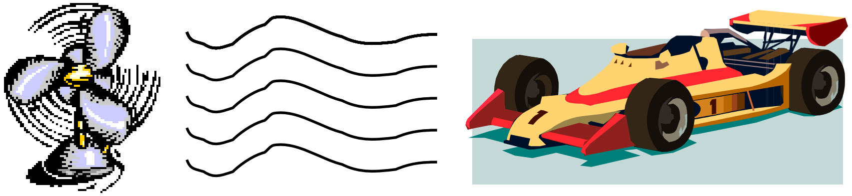
- ☐ *Model*
    - A representation of an object that captures some (but not all) of the object's attributes
- ☐ Models are not complete (by design and intent)
    - They do not capture ALL attributes of the object
    - Just the ones that are "useful" or "of interest"
    - If a model captured all attributes, would be the same as replicating the object – defeats purpose
- ☐ Models are used to reason about an object (*e.g.*, a system)
    - to infer its characteristics
    - *e.g.*, use a model to infer how hard it will be to build the system
    - *e.g.*, use a model to infer how well the system will perform

Two categories of models

- ❑ *Analytic* models
  - – used to infer features of the software system based on structure or algorithms
    - ➢ e.g., block diagram of a system, Unified Modeling Language ® UML class diagram, …
- ❑ *Executable* (or *behavioral*) models
  - – used to infer characteristics of the system's behavior based on simulation/execution of parts of the system
- ❑ Some models can serve both roles
- ❑ Others are limited to a single role unless they are enhanced

Sample models *(page 1 of 5)*

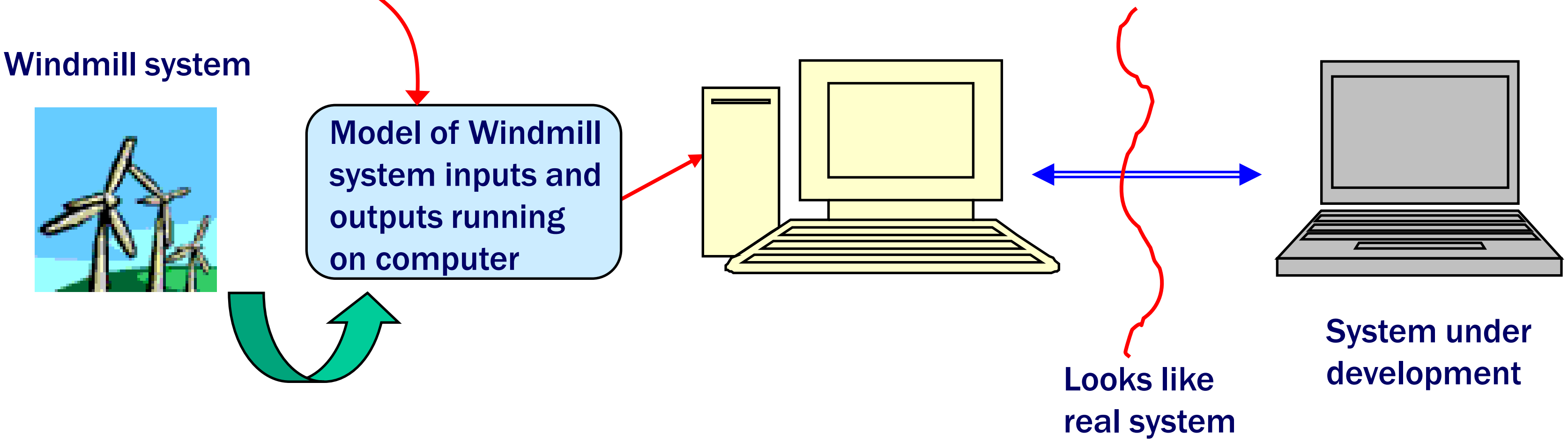
- Create a scale model of a race car to measure its behavior in a wind tunnel



- This is an executable model
  - We are exercising an aspect of the system
- If we wanted an analytic model, we would
  - Express the shape of the race car mathematically
  - Apply known aerodynamics effects to the shape
  - Infer car's behavior based on the analysis

Sample models *(page 2 of 5)*

- Create a model of one system to assist in developing and testing another system with which it will interact
- This is an <u>executable</u> model
  - We are actively *simulating* the environment in which a system will operate, using a model of the environment
  - We are executing the system (or a model of the system)


Windmill system

Model of Windmill system inputs and outputs running on computer

Looks like real system

System under development

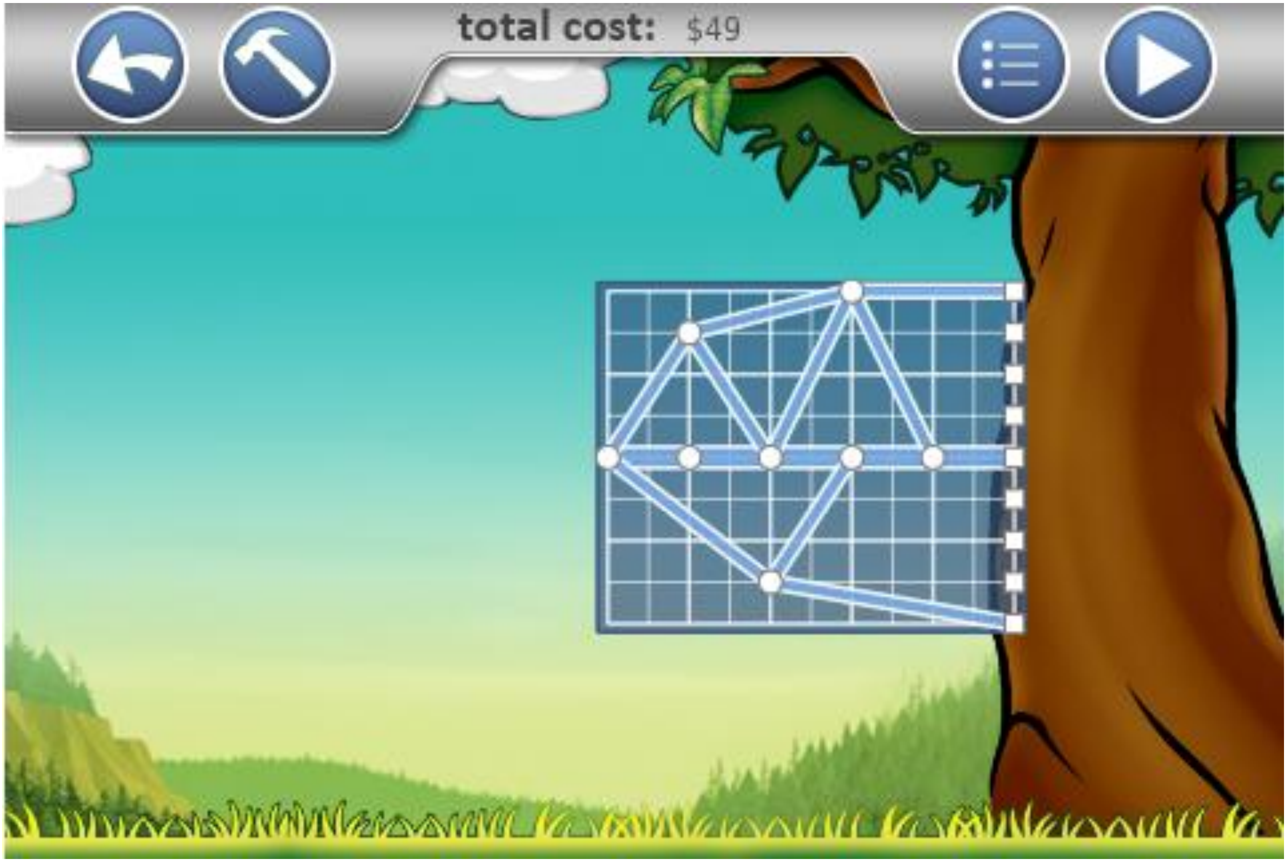
Sample models *(page 3 of 5)*

- ❑ Build a scale model to assess the design of a building before it is built

- ❑ This is an <u>analytic</u> model
  - – We are assessing the building's characteristics based on its structure, to analyze
    - ➢ How hard it might be to build
    - ➢ How convenient it might be for people to move about within the building (*e.g.*, fire escapes)

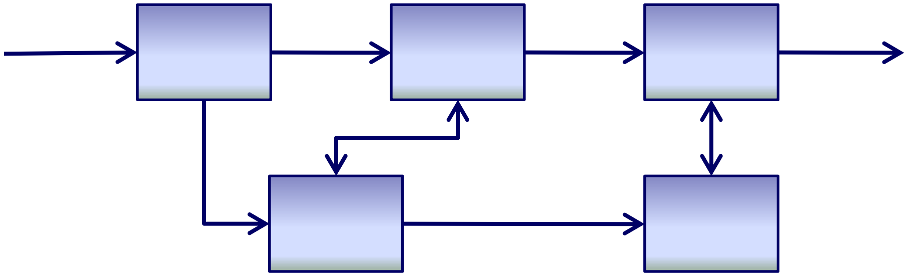
Sample models *(page 4 of 5)*

- Simple Physics – game for the iPhone



Sample models *(page 5 of 5)*

- ☐ Create architectural design in AADL (Architecture Analysis and Design Language) of a system, to include components and their interconnections
- ☐ Based on projected I/O rates and interconnection capacities, project expected performance of system



- ☐ This is an <u>analytic</u> model
  - – We are assessing the design's potential data throughput and latency based on its structure

More sample models

- Build a model of a hardware VLSI chip in VHDL to assess whether the design will be effective
- Build a brassboard of a single board computer to assess its characteristics
- Build software prototypes to
  - Validate operational concepts, assess requirements and user interfaces
  - Explore alternative design approaches
  - Refine test approaches
- Write software source code that computes position from GPS satellites
  - The code is a model of the actual product

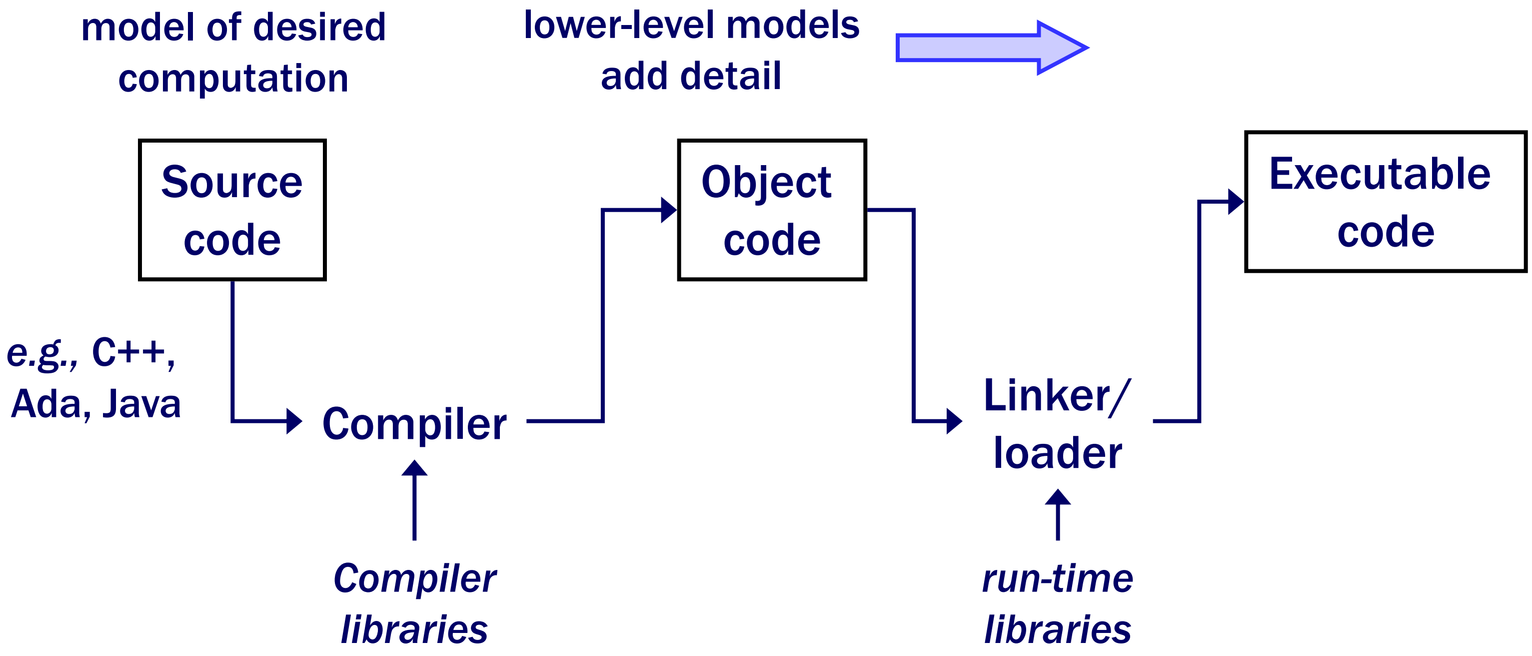
This last sample is key to how we view MDE and software development

Software programs as models

- ❑ A piece of software is a model
  - – Sometimes even an executable model
- ❑ Written in a programming language with precise semantics (so that the computer knows what to do)
- ❑ The program models the execution object (the algorithm)
- ❑ *Consider* – code that contains all procedure interfaces but no code bodies
  - – Models the program structure (architecture)
- ❑ *Consider* – code that contains procedure bodies that contain non-optimized code for selected algorithms
  - – Models the procedure's functional behavior but not its performance

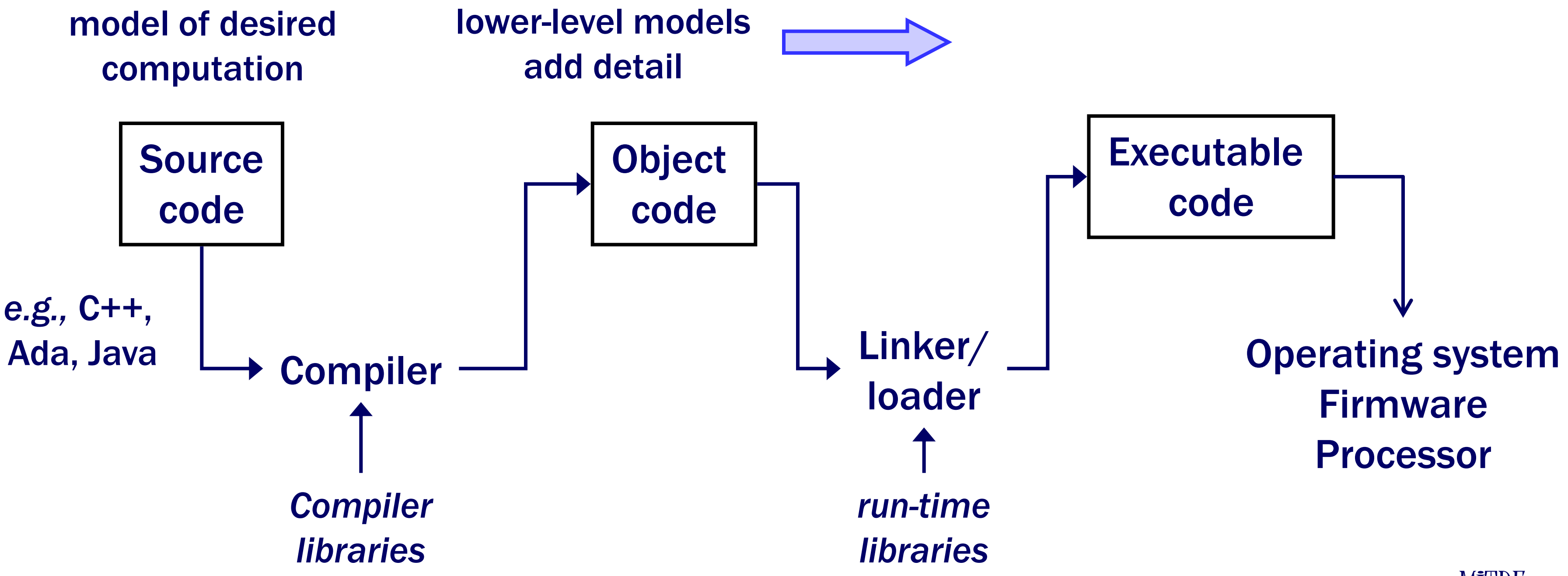
Code to executables

- Finished source code is still a model
  - Defines all semantics at some level but still requires additional detail (*e.g.*, run-time libraries)
  - Source code is not complete



But wait – there's more…

- The final product is the system
  - Fully integrated with all of its necessary parts to make it complete



model of desired computation

lower-level models add detail

Source code

e.g., C++, Ada, Java

Compiler

Compiler libraries

Object code

Linker/ loader

run-time libraries

Executable code

Operating system
Firmware
Processor

Prototypes

- A software prototype is a form of model
- Created to investigate a specific aspect of the system-under-development
  - and then thrown away *(or should be)*
  - Generally not designed to be production-quality or complete
  - *e.g.*, usually omits error handling and IA considerations, coding standards, documentation, …
  - Adding production-quality features after the fact is usually not cost-effective
- Sample uses
  - Evaluate user interfaces, perform usability studies
  - Investigate alternative design structures
  - Measure alternative code patterns
  - Evaluate algorithms

Useful items that can be modeled

- ❑ *System environment* – capture aspects of the system's operational environment to aid in requirements definition and testing
- ❑ *System design* – describe the software design to reason about how the software will behave (*structural modeling*)
- ❑ *System behavior* – simulate the system's behavior for analysis and refinement (*behavioral modeling*)
- ❑ *User interfaces* – capture aspects of the user's interface and present to the users to determine suitability
  - Can be static images or dynamic simulations
- ❑ Many others….

MITRE

When and where models can be used

- ☐ At all levels of development
  - Software
  - Systems
  - Systems of systems (SoS)
- ☐ During all activities performed as a part of software development (ref: IEEE/EIA 12207.0)
  - Software requirements analysis
  - Software architectural design
  - Software detailed design
  - Software coding and testing
  - Software integration
  - Software qualification testing

Where else….

- ☐ As well as all other processes and activities defined by IEEE/EIA 12207.0

Process	Activities
Acquisition	Initiation Request-for-proposal [-tender] preparation Contract preparation and update Supplier monitoring Acceptance and completion
Supply	Initiation Preparation of response Contract Planning Execution and control Review and evaluation Delivery and completion
Development	Process implementation System requirements analysis System architectural design Software requirements analysis Software architectural design Software detailed design Software coding and testing Software integration Software qualification testing System integration System qualification testing Software installation Software acceptance support
Operation	Process implementation Operational testing System operation User support
Maintenance	Process implementation Problem and modification analysis Modification implementation Maintenance review/acceptance Migration Software retirement

Agenda

- ☐ Overview of tutorial
- ☐ Models
- ➡ ☐ Model Driven Engineering (MDE)
- ☐ Overview of software development activities
- ☐ Software development artifacts
- ☐ Model Driven Architecture® (MDA®)
- ☐ Assessment
- ☐ Recommendations

Model Driven Engineering

- A development strategy that uses models as stepping stones to creating the final product
- Models are iteratively refined and matured, with more detail added, until the model becomes the product
  - Rather than throwing the model away after each iteration
- Requires associated tools to transform between stages
- Advantages
  - Reduces defect content, since humans separated from the transformation process (humans make mistakes which result in latent defects)
  - Reduces rework, thereby reducing cost and time-to-deployment
  - Increases predictability, by relying on tools rather than humans

Tools cannot add information –
they can only transform

Sanity-check

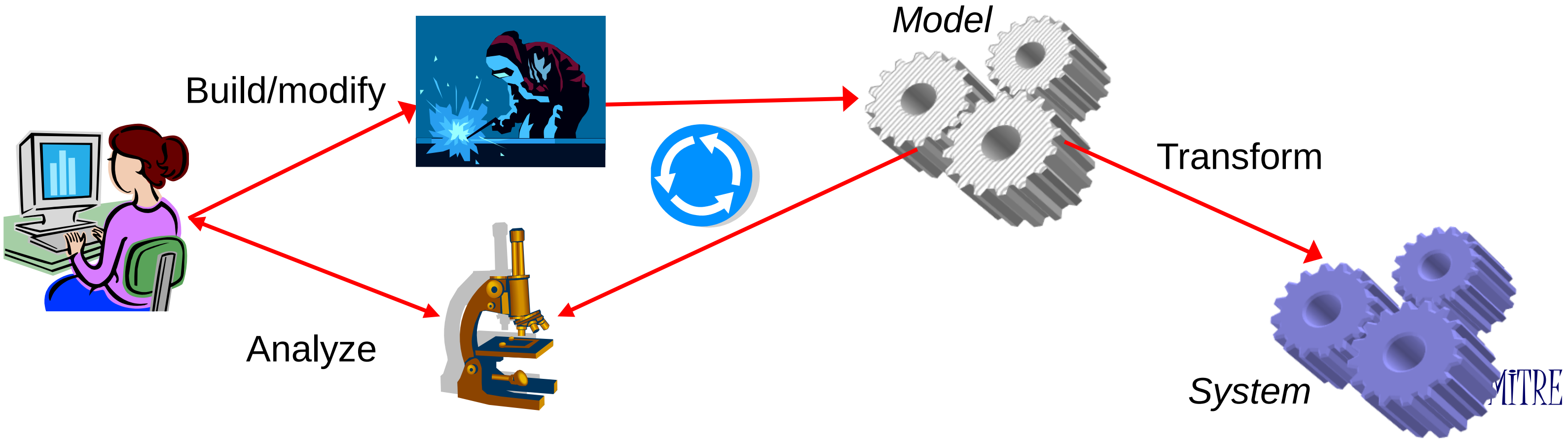
- Note that this includes what we do today
  - Particularly with incremental and iterative development
  - Each iteration product goes through increasing detail (design, write code, verify)$^{+n}$
- Each iteration produces a model of the final system
  - Not the complete product
  - but maturing towards that product
- Our notion of a model has evolved
  - and continues to evolve
- MDE aims at exploiting the power of modeling and pushing practice to the “left”

Model Driven Engineering (MDE)

- ☐ Term coined by Prof. Doug Schmidt/Vanderbilt University
  - Ref: Schmidt, Douglas C. "Model-Driven Engineering". *IEEE Computer*, 39(2), February 2006.
- ☐ Covers many specific types of techniques and tools
  - *e.g.*, Model Driven Architecture®, Model Driven Development™, Model Integrated Engineering, Model Based Programming™…
  - *e.g.*, OMG Model Driven Architecture® MDA®, Microsoft® Domain Specific Language (DSL) Tools (Visual Studio 2005), Lockheed-Martin's Model Centric Software Development (MCSD), others…

Core elements of MDE

- ❑ ***Modeling languages***
  - Used to define the models, *i.e.*, the structures and algorithms of the system being modeled/developed
- ❑ ***Transformation engine and generator tools*** **to assist in**
  - Building the models
  - Maturing the models as more information is embedded
  - Constructing the system by transforming the models to source code/executable form



MITRE

Modeling languages

- Sometimes called “notations”
- Used to express the attributes of the entity being modeled
- Two types
  - *Descriptive* (used to create analytic models)
    - Express system attributes but requires analysis to make inferences about system behavior and attributes
    - Often do not provide complete semantics. *e.g.*, core UML
  - *Semantic* (used to create executable models)
    - Express system attributes including both structure and behavior
    - If semantics are sufficiently rich and precise, semantic models can be executable
    - *e.g.*, UML Action Semantics, Ada, Java, C++, …

Language forms

- Language pedigree
  - Most common but less precise are natural languages
  - Most useful are artificial languages
- *Textual*
  - Expressed in text (*e.g.*, Ada, C++, Java, AADL, …)
  - Generally defined by context-free grammars
    - Constrained by description to be context sensitive
  - Potential straightforward mapping from text to semantics
  - Can be directly processed by tools
- *Graphical*
  - Expressed as diagrams and images (e.g., UML, Petri Nets, …)
  - Form and semantics defined by exemplars
  - Must be mapped to text to be processed
    - Some have both forms (*e.g.*, AADL, EXPRESS – a data modeling language)

Impact of form

- ❑ Textual languages
  - Often harder to read and understand
  - Tend to be more precise semantically
  - Can be directly processed
- ❑ Graphical languages
  - Tend to be easier to understand (but not always)
  - In general, less precise semantically due to potential variations in presentation
    - ➢ Exceptions exist – State Diagrams, Petri Nets
  - Cannot be directly processed – need to be converted into a text form
- ❑ Languages need both forms to be fully expressive
  - And have consistent precise semantics in both forms

Important language attributes

- *Level of standardization* – if language is not standard, risk of getting locked into a specific vendor and model
- *Precision of definition* – ambiguous or imprecise language definitions can lead to ineffective models and non-standard implementations
- *Popularity* – using rarely-used languages, even if adhering to standards, runs the risk of being locked-in
- *Completeness* – if a language is incomplete, risk of not being able to express important system attributes
- *Suitability* – language should be appropriate for the application domain being modeled
- *Level* – if language is too low-level (such as C/C++), model tightly linked with implementation

Sample modeling languages

- SysML – a domain-specific visual modeling language for systems engineering applications
- UML, UML2 – Unified Modeling Language, recently updated with new standard, visual modeling language primarily for object-oriented software development
- AADL – Architecture Analysis and Design Language
- MATLAB – tool-based language primarily for mathematical modeling
- Petri Nets – used to model discrete distributed systems
- IDEF – including IDEF0 for functional modeling, and IDEF1 for information modeling
- StateCharts – based on state machines, extended by Harel, supported by a toolset
- CSP (Communicating Sequential Processes) – used to model synchronization among concurrent processes
- pUML – Precise UML – extension of UML to correct its imprecision and ambiguity
- MAML – Multi-Agent Modeling Language – derivative of Swarm, extension of Objective-C

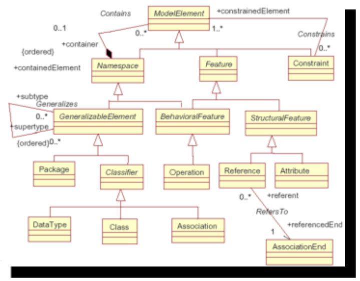
Role of "meta-languages"

- Also called "metamodels"
- You may see this term regarding the development of models
  - Fancy term for a common concept
- Denotes notations used to define other languages
- *Examples*
  - EBNF (Extended Backus-Naur Form) – standardized by ISO/IEC Standard 14977:1966
    - Used to define language syntax
  - OMG MOF (Meta Object Facility) – used to define UML
- Multiple levels exist
  - Metamodels, metametamodels, metametametametamodels, … m^{20}models…

Meta-model layers

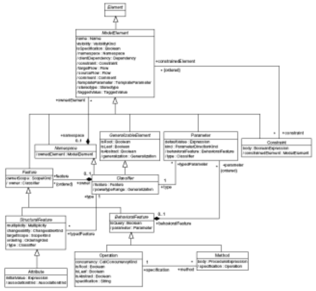
Meta Meta-Model Layer
M3

A definition of the structure and semantics of the notation used to define the modeling languages. e.g. the notation used in M2



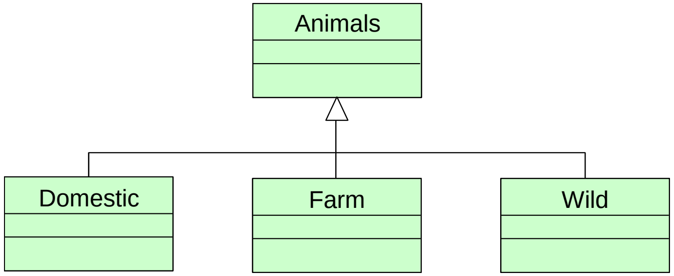
Meta- Model Layer
M2

A definition of the structure and semantics of the modeling language, *e.g.* definition of UML.



Model Layer
M1

A description of the objects. This layer defines a model of the system and uses the modeling language.

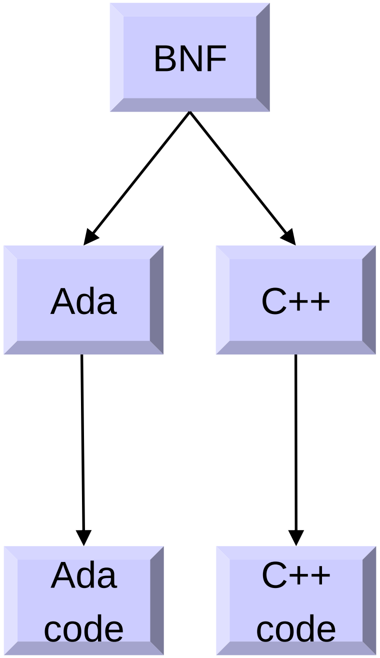


Information Object Layer
M0

The objects that we wish to describe.

Example of metamodel levels

Meta Meta Model	Definition of BNF (ISO/IEC 14977)
Meta Model	Definition of Ada using BNF (MIL STD 1815A)
Model	Ada or C++ code used to define data structures for personnel records
Information Object	Personnel records in a database system populated with data



e.g.,

```
class Employee
{
        string Name;
        unsigned EmpNum;
        …
};
```

Name	Emp-Num	Position	Date-of-hire	Department
Fred	36	Mgr	3/1/2003	Sales
Helen	1004	CEO	5/2/2003	Corp
Cynthia	899	CIO	6/1/2003	Manuf
John	722	Mkt	4/1/2003	Sales
Ralph	234	Drone	12/1/2003	Manuf

Modeling tools

- Used to develop, analyze, and refine model products
- Three types
  - *Manipulation* tools – which help in defining, enhancing, and refining the model as it evolves (*e.g.*, Rational Rose RT)
  - *Analytic* tools – which aid in analyzing the model to determine properties (*e.g.*, MATLAB)
    - May involve simple analysis of structure or full simulation of behavior
  - *Transformation* tools - which convert/ translate/ transform the model into something that can be executed (*e.g.*, Kennedy-Carter iUML), Rose RT, Rhapsody, PathMate)
- Some tools combine these features

Important tool attributes

- *Level of standardization* – tool must work with standard formats to avoid risk of getting locked into specific vendors
- *Completeness* – tool must be able to exploit full capabilities of modeling language
- *Suitability* – tool should be appropriate for the application domain being modeled
- *Confidence* – tools must be certified to be correct so that trust can be placed on its operation
- *Ease of use* – tools operation must be straightforward and require minimal human intervention
- *Stability* – tool must be mature and stable to avoid constant updates
- *Predictability* – tool must incorporate minimal tailorability of core elements, to ensure that process is repeatable

Trade-offs with tools and languages

- ☐ Projects need to balance language and tool attributes
  - Not all potential system features are likely to be attained in any one tool/language
  - Need to perform trade-offs to weigh risk against benefits
- ☐ State-of-the-practice is continuously maturing
  - So balance of attributes shifting from day-to-day

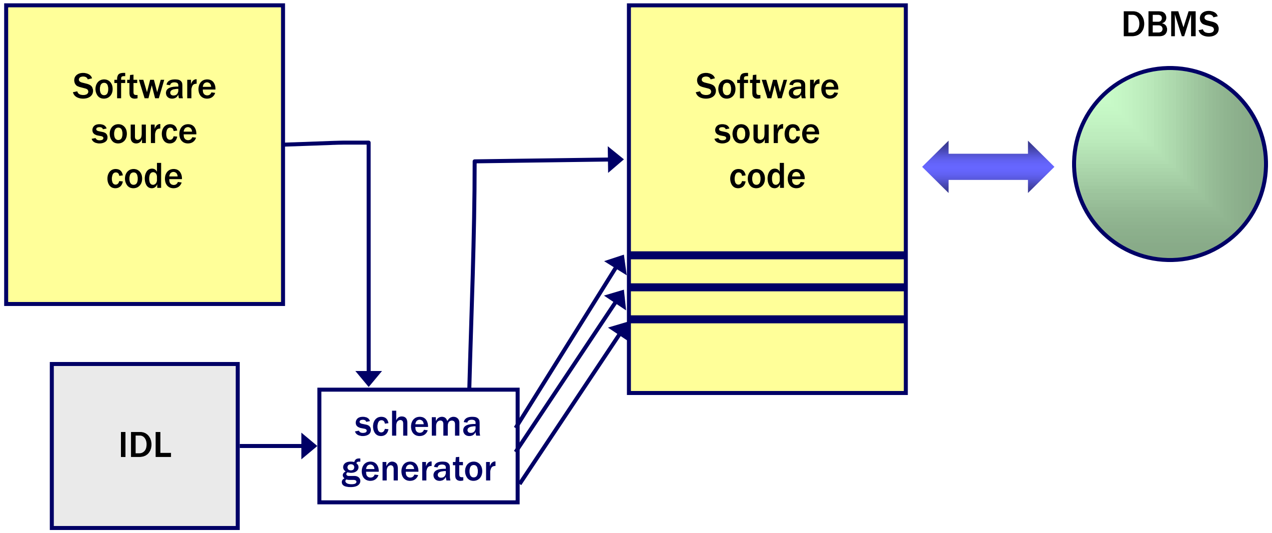
- ☐ Standardization potentially a big issue
  - Standards must be precise and clearly defined
  - and undergo an open community review process (*e.g.*, ISO, IEEE, SAE, …)
  - Some level of agreement needs to be achieved – not always easy!
  - Particularly important with MDE “action languages”

MDE legacy (part 1 of 3)

- ❑ MDE is not new
  - The concepts and the tools have been around for decades
  - We've done this before
- ❑ *e.g.*, Autocoding Toolkit® - originated in mid-1980s
  - Developed by MCCI (a company in Arlington, Virginia, USA)
  - Software modeled using data flow diagrams with the Processing Graph Method
    - ➢ Using SPGN – Signal Processing Graph Notation
  - Models were executable to verify correctness
  - "Automatically" translated into higher-level language for execution on target computers
  - Produces C or Ada source code
  - Info at http://www.mdatechnology.net/update.aspx?id=a5126

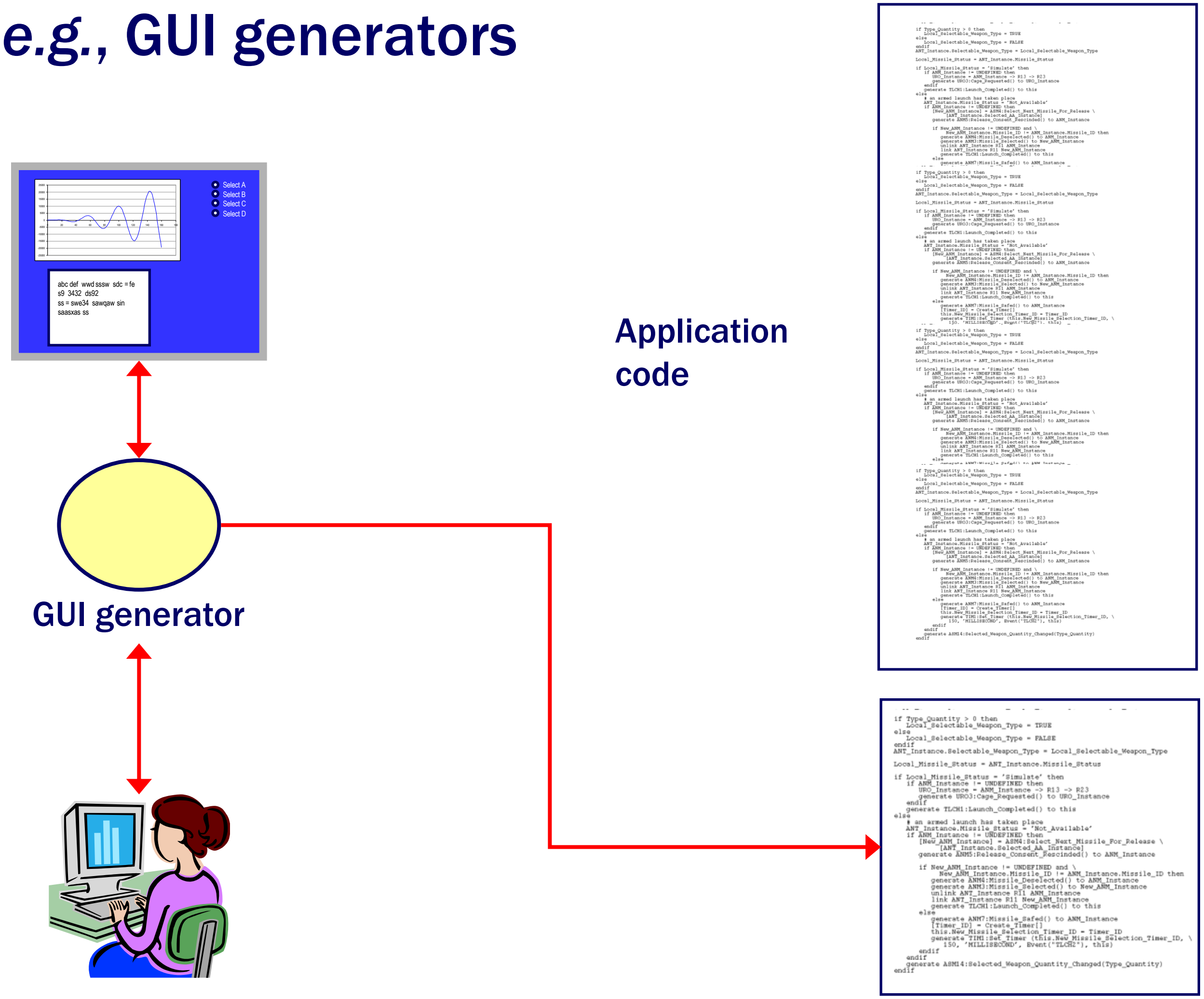
MDE legacy *(part 2 of 3)*

- e.g., database schema generators
- Uses Interface Definition Languages (IDL) to automatically generate code for database access



MDE legacy (part 3 of 3)

- e.g., GUI generators



```
if Type_Quantity > 0 then
   Local_Selectable_Weapon_Type = TRUE
else
   Local_Selectable_Weapon_Type = FALSE
endif
ANT_Instance.Selectable_Weapon_Type = Local_Selectable_Weapon_Type

Local_Missile_Status = ANT_Instance.Missile_Status

if Local_Missile_Status = 'Simulate' then
   if ANM_Instance != UNDEFINED then
      URO_Instance = ANM_Instance -> R13 -> R23
      generate URO3:Cage_Requested() to URO_Instance
   endif
   generate TLCH1:Launch_Completed() to this
else
   # an armed launch has taken place
   ANT_Instance.Missile_Status = 'Not_Available'
   if ANM_Instance != UNDEFINED then
      [New_ANM_Instance] = ASM4:Select_Next_Missile_For_Release \
           [ANT_Instance.Selected_AA_Instance]
      generate ANM5:Release_Consent_Rescinded() to ANM_Instance

      if New_ANM_Instance != UNDEFINED and \
           New_ANM_Instance.Missile_ID != ANM_Instance.Missile_ID then
         generate ANM4:Missile_Deselected() to ANM_Instance
         generate ANM3:Missile_Selected() to New_ANM_Instance
         unlink ANT_Instance R11 ANM_Instance
         link ANT_Instance R11 New_ANM_Instance
         generate TLCH1:Launch_Completed() to this
      else
         generate ANM7:Missile_Safed() to ANM_Instance
         [Timer_ID] = Create_Timer[]
         this.New_Missile_Selection_Timer_ID = Timer_ID
         generate TIM1:Set_Timer (this.New_Missile_Selection_Timer_ID, \
            150, 'MILLISECOND', Event("TLCH2"), this)
      endif
   endif
   generate ASM14:Selected_Weapon_Quantity_Changed(Type_Quantity)
endif
```

MITRE

Agenda

- ☐ Overview of tutorial
- ☐ Models
- ☐ Model Driven Engineering (MDE)
- ➡ ☐ Overview of software development activities
- ☐ Software development artifacts
- ☐ Model Driven Architecture® (MDA®)
- ☐ Assessment
- ☐ Recommendations

Intro to this section

- ❑ Content of this section may seem repetitive and obvious
  - "Software Engineering 101"
- ❑ But need to emphasize certain aspects of development
  - to show how and where they are / can be affected by following a model-driven approach
- ❑ Establishes a context for the MDE discussion
- ❑ Helps to highlight where modeling techniques can provide benefits
  - Let's identify where they can help
- ❑ Impact of MDE is subtle yet significant
- ❑ Applying MDE will result in changes to how we perform development

Software development

- Development of software generally
  - starts with an allocation of requirements to the software
  - ends with the completion of the product that implements those requirements
- The *software development process* involves six activities as defined by IEEE/EIA Std. 12207
- Each activity involves different sets of issues and concepts, and can be clearly differentiated (but are tightly interdependent)
  - *e.g.*, requirements analysis ≠ detailed design
- These activities used to be called "phases" (*e.g.*, the "Waterfall" approach)
  - but are now generally applied incrementally and iteratively rather than sequentially

Incremental vs iterative development

- ❑ ***Incremental*** – a scheduling strategy in which various parts of the system are developed at different times and integrated as they are completed
  - Each part developed once
- ❑ ***Iterative*** – a rework scheduling strategy in which time is set aside to revise and improve parts of the system
  - Each part developed once and iteratively revised and improved over time
- ❑ Development plans can be one, the other, or both
- ❑ Modeling is a key tool to support incremental and iterative development efforts
  - Particularly when models are iteratively refined over development process

* Cockburn, Alistair. "Using Both Incremental and Iterative Development". *Crosstalk*, May 2008. http://www.stsc.hill.af.mil/crosstalk/2008/05/0805Cockburn.html

Software maintenance

- Maintenance is also an important activity
  - Begins after development is complete
- Historically consumes 50-80% of total life cycle cost
- Within IEEE/EIA 12207, the maintenance process is separate from the development process
- Focuses on
  - Defect detection based on anomaly reports
  - Defect correction based on modification analysis
  - Adaptation to changes in environment
  - Enhancement/improvement of capabilities

Core software development activities

- *System architectural design* – identify the system components (software and hardware) that will form the system

Core software activities

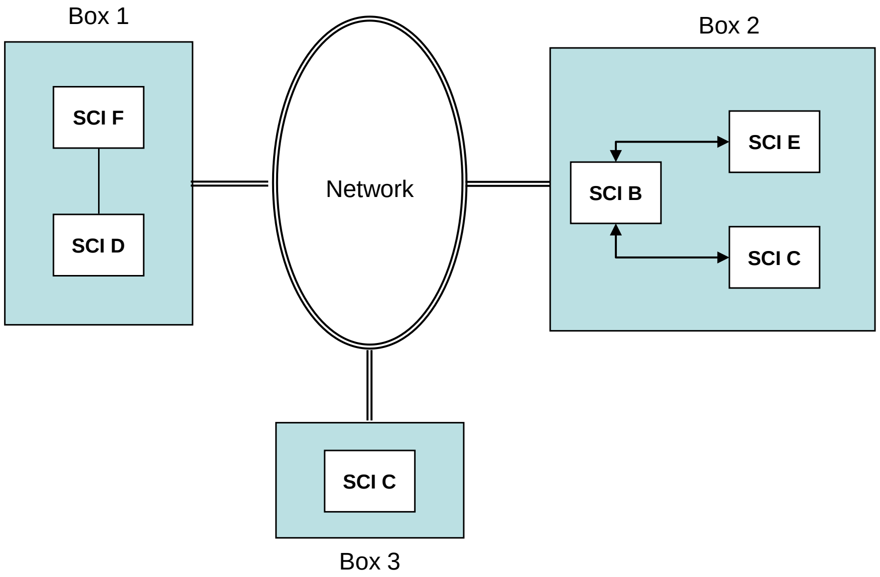
- *Requirements analysis* – determine the externally-visible behavior of the software elements (CSCIs)
- *Architectural design* – identify the software components that will form the CSCIs, and the interrelationships between these components
- *Detailed design* – determine the software modules and units that constitute the software components
- *Coding and testing* – create the modules and units in code and verify that the units are correctly written
- *Integration* – build the components and CSCIs from the units, and verify that the integrated pieces are correct
- *Qualification testing* – verify that the CSCIs are correctly built and implement the assigned requirements

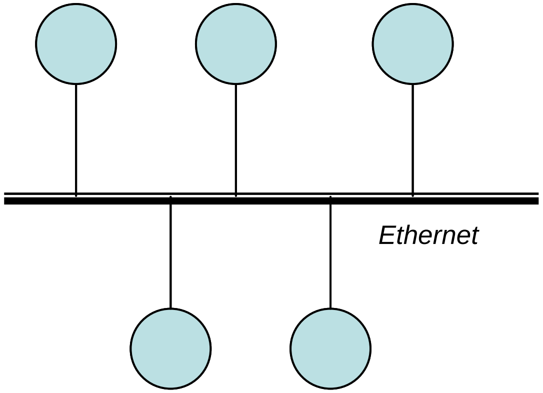
- *System integration* – support integration of the system CSCIs and HWCIs, correcting SW defects when discovered

Based on IEEE/EIA Std 12207

System architectural design

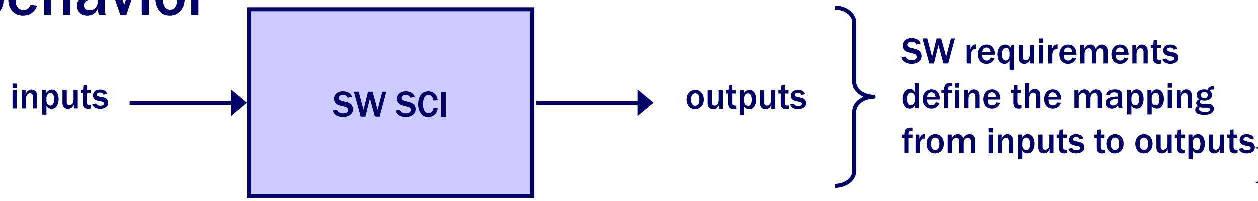
- Historically precedes software development
- Defines overall architecture of system, including software CIs, their location in the system, their means of communication, their required behavior





Software requirements analysis

- Input – “required capabilities” / allocated requirements
- Actions
  - Define typical scenarios of use (text, Use cases)
  - Define initial set of requirements (text, state machines,…)
  - Develop prototypes (pictures, GUI code,…)
  - Interact with users, analyze, and refine
- Output – “requirements”
  - Examples of forms used include text, use cases, scenarios, state models, sequence diagrams, GUI prototypes, usage models
- Note – requirements are descriptions of externally-visible behavior



inputs

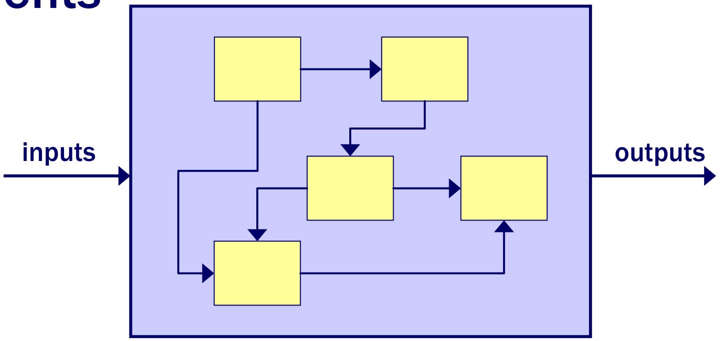
SW SCI

outputs

SW requirements
define the mapping
from inputs to outputs

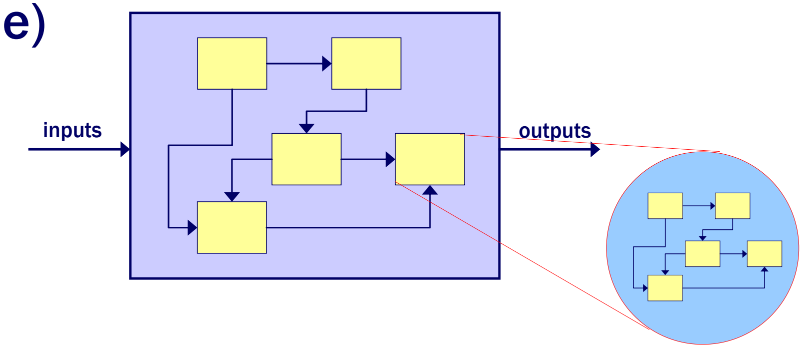
Software architectural design

- Input – “requirements”
- Actions
  - Identify major components that form the top-level structure
  - Define the functional behavior of each component (*responsibilities*)
  - Define the interfaces for the components
  - Develop prototypes
  - Analyze and refine
- Output – “architectural design”
  - List of components, their interfaces, and their interactions
  - Text, class diagrams, sequence diagrams, data flow diagrams, “4+1 views”, …



Software detailed design

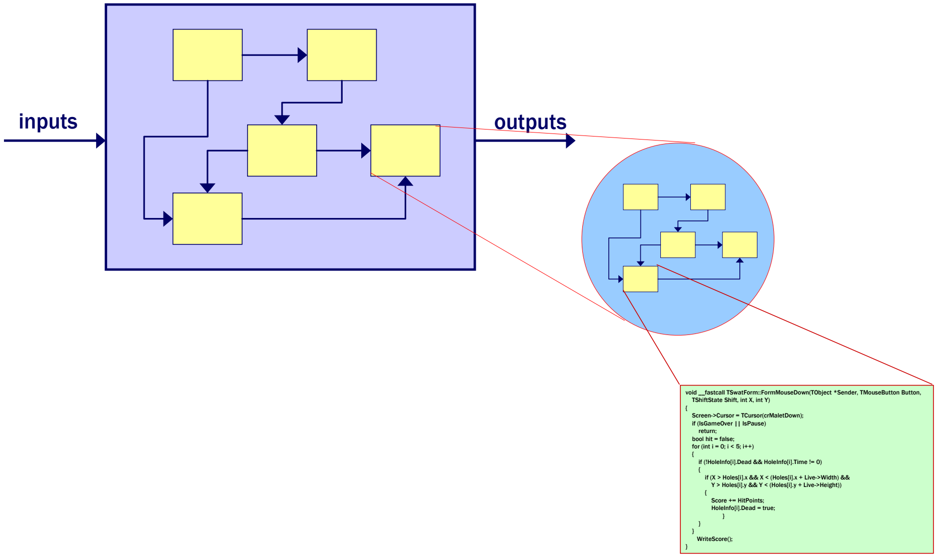
- Input – “architectural design”
- Actions
  - Identify internal structure of the major components
    - *i.e.*, down to smallest module level (units)
  - Define the behavior of each subcomponent and unit (using program design languages (PDLs), state machines, …)
  - Define the interfaces for the subcomponents and units (text)
  - Develop prototypes (figures / code)
  - Analyze and refine
- Output – “detailed design”
  - List of all code units and their interfaces and interactions
  - Text, class diagrams, sequence diagrams, data flow diagrams, activity diagrams, …



MITRE

Code and unit test

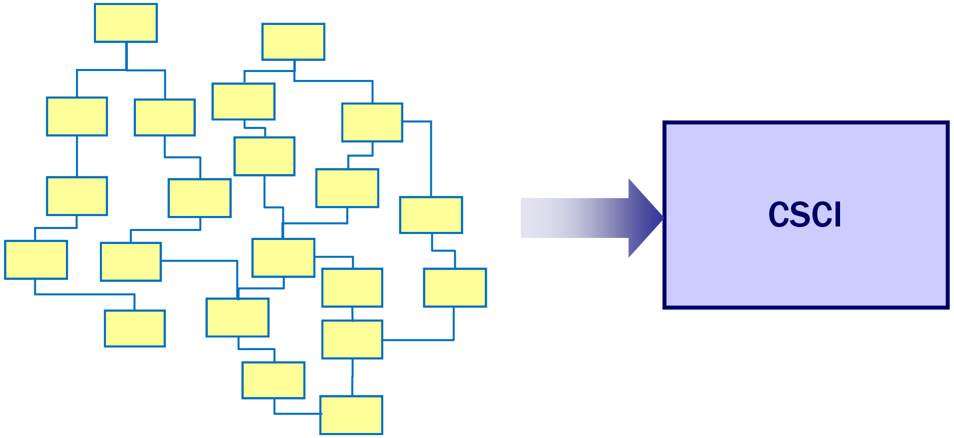
- ❑ Input – “detailed design”
- ❑ Actions
  - – Write code to implement the assigned behavior and interfaces for each unit
  - – Verify that the code is correct (unit test)
  - – Analyze and refine via prototypes and models
- ❑ Output – “fully coded units”
  - – C++, Ada2005, Java, Kennedy-Carter ASL…



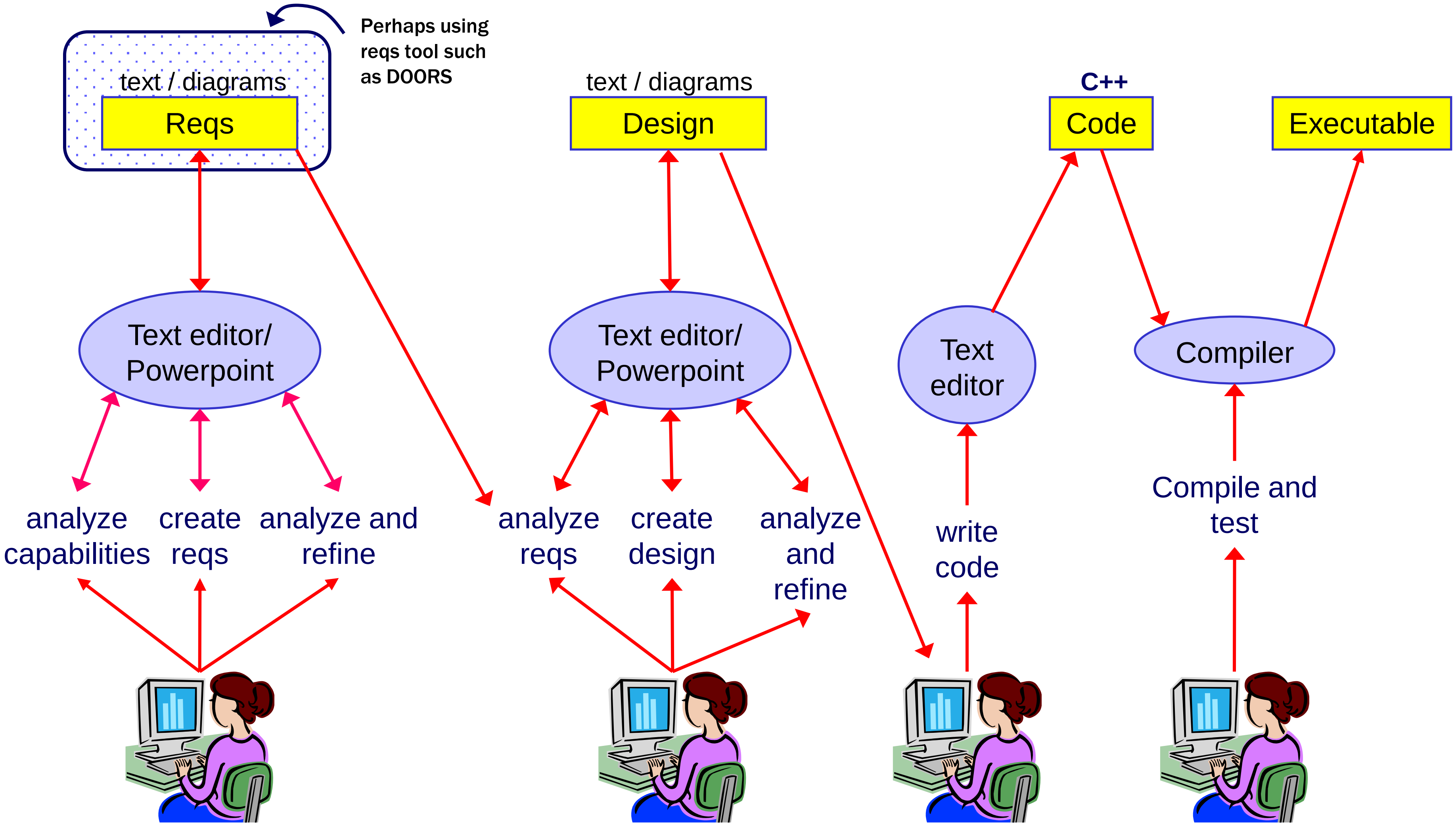
```
void __fastcall TSwatForm::FormMouseDown(TObject *Sender, TMouseButton Button,
   TShiftState Shift, int X, int Y)
{
   Screen->Cursor = TCursor(crMaletDown);
   if (IsGameOver || IsPause)
      return;
   bool hit = false;
   for (int i = 0; i < 5; i++)
   {
      if (!HoleInfo[i].Dead && HoleInfo[i].Time != 0)
      {
         if (X > Holes[i].x && X < (Holes[i].x + Live->Width) &&
            Y > Holes[i].y && Y < (Holes[i].y + Live->Height))
         {
            Score += HitPoints;
            HoleInfo[i].Dead = true;
                 }
      }
   }
     WriteScore();
}
```

Integration and test

- Input – “fully-coded units”
- Actions
  - Successively integrated units to form components and eventually the entire CSCI
  - Verify that the integrated code is correct at each stage
  - Analyze and refine
- Output – “complete integrated CSCI” that correctly implements the assigned requirements



Example 1 – minimal use of tools



Perhaps using reqs tool such as DOORS

text / diagrams

Reqs

Text editor/ Powerpoint

analyze capabilities

create reqs

analyze and refine

text / diagrams

Design

Text editor/ Powerpoint

analyze reqs

create design

analyze and refine

C++

Code

Text editor

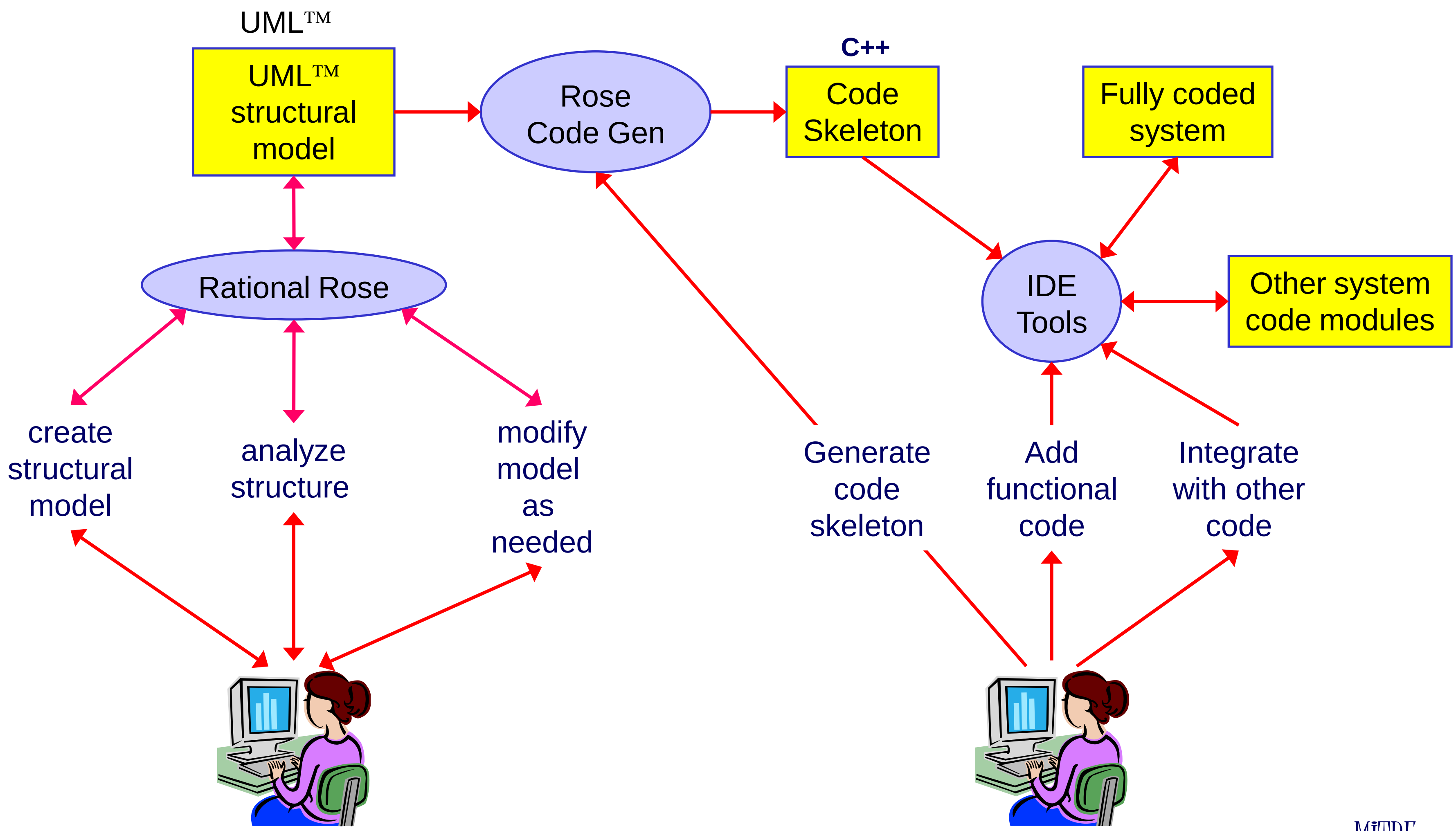
write code

Executable

Compiler

Compile and test

Example 2 – use of UML



UML™

UML™ structural model

Rose Code Gen

C++

Code Skeleton

Fully coded system

Rational Rose

IDE Tools

Other system code modules

create structural model

analyze structure

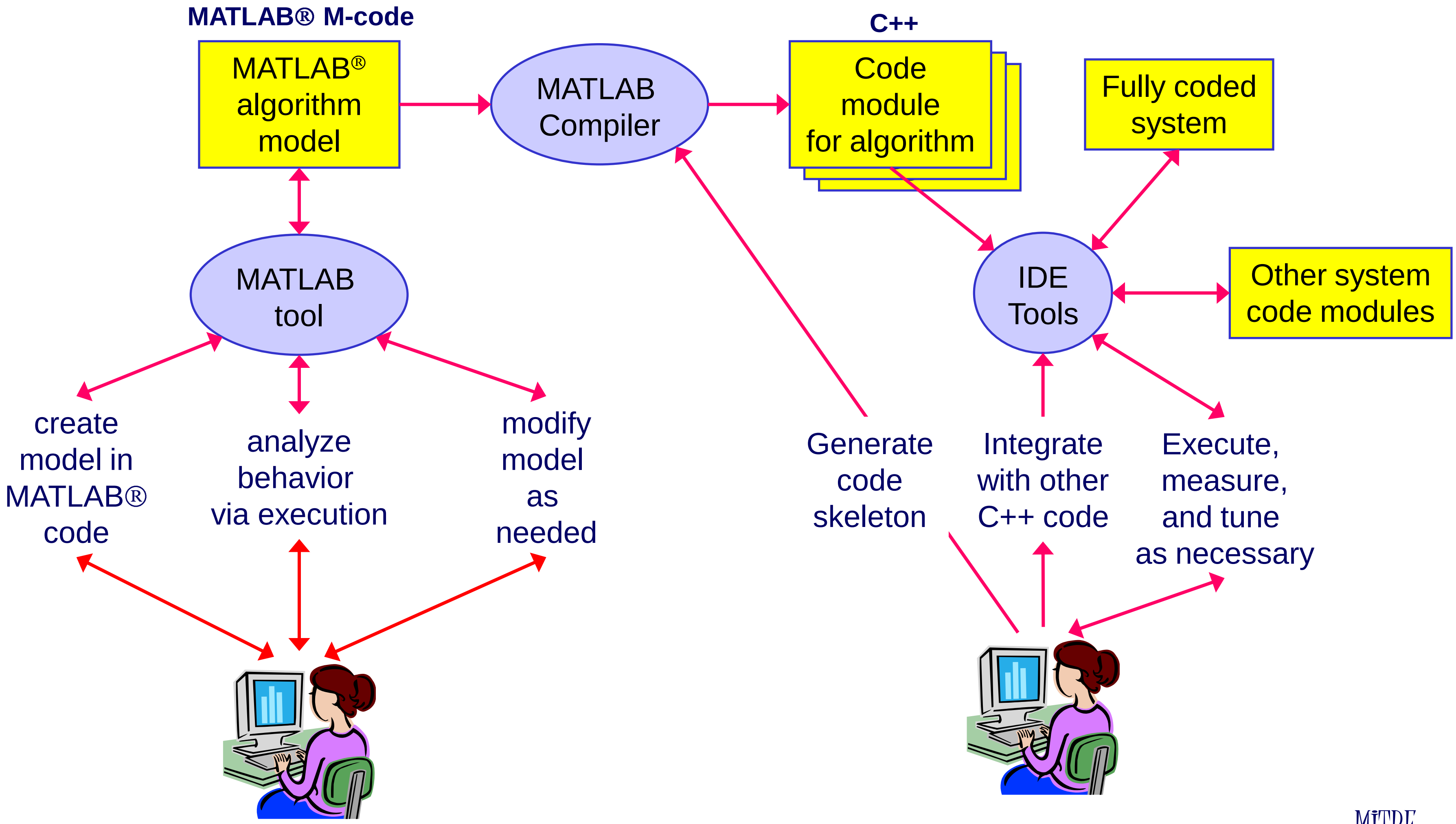
modify model as needed

Generate code skeleton

Add functional code

Integrate with other code

Example 3 – use of MATLAB



MATLAB® M-code

MATLAB® algorithm model

MATLAB Compiler

C++

Code module for algorithm

Fully coded system

MATLAB tool

IDE Tools

Other system code modules

create model in MATLAB® code

analyze behavior via execution

modify model as needed

Generate code skeleton

Integrate with other C++ code

Execute, measure, and tune as necessary

Agenda

- Overview of tutorial
- Models
- Model Driven Engineering (MDE)
- Overview of software development activities
- Software development artifacts
- Model Driven Architecture® (MDA®)
- Assessment
- Recommendations

Typical development products

Desired capabilities

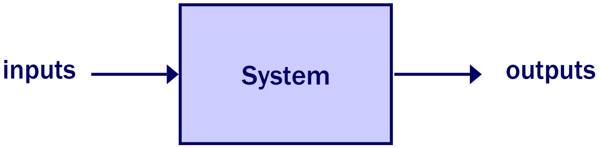
	Software Requirements Analysis	Software Architectural Design	Software Detailed Design	Software Coding and Testing	Software Integration
Purpose	Define software requirements at CSCI level – describe behavior, identify objects, classes, and external data (interfaces)	Define CSCI architecture - identify major classes, methods / functions, internal interfaces, and messages at component and module level	Define CSCI detailed design (unit level) – expand and refine classes/modules to include local data, the methods/functions and their interfaces	Write unit level code to implement class structure, methods, and functions Test unit code	Integrate units to form components, and verify. Integrate components to form CSCI and verify that CSCI meets its requirements.
Typical major products	Text description of reqs Use cases Scenarios ↓ State diagrams Context diagrams Class diagrams Sequence diagrams Models/prototypes	State diagrams Class diagrams Sequence diagrams Data flow diagrams Message definitions Models/prototypes	State diagrams Class diagrams Sequence diagrams Data flow diagrams Activity diagrams Algorithms Data definitions Pseudo-code	Code (text) Test cases Test results	Code (text) Test cases Test results

Different roles for code

- Defining architectural design
  - Specifications of components and their interrelationships
  - Can be written in parallel with architectural design
- Defining detailed design
  - Specifications of bottom-level components and their interrelationships
- Defining the semantics
  - Detailed semantics – procedures, operations, calculations, decisions, specific algorithms,…
- Supporting external interfaces and interactions
  - *e.g.*, user interfaces
  - Can be written in parallel with requirements analysis
- Interacting with host environment (resources)
  - OS calls
  - Middleware exploitation

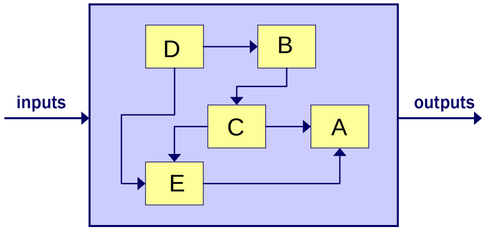
Relationship of activities and code

Requirements level



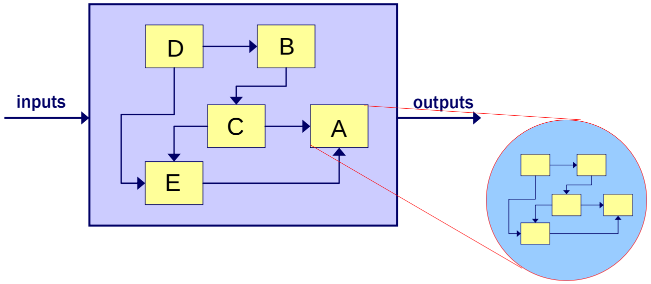
Procedure Main (inputs, outputs)

Architectural design level



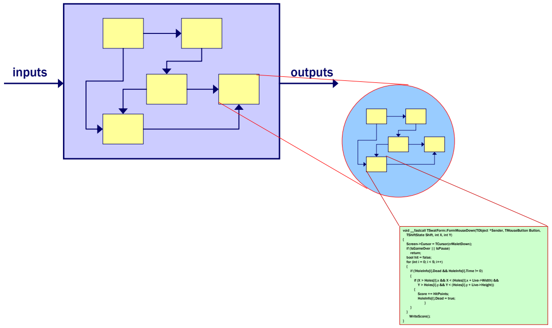
Procedure A (inputs, outputs)
Procedure B (inputs, outputs)
Procedure C (inputs, outputs)
Procedure D (inputs, outputs)
Procedure E (inputs, outputs)

Detailed design level



Procedure A (inputs, outputs)
  Procedure aa (inputs, outputs)
  Procedure ab (inputs, outputs)
  Procedure ac (inputs, outputs)
  Procedure ad (inputs, outputs)
  Procedure ae (inputs, outputs)

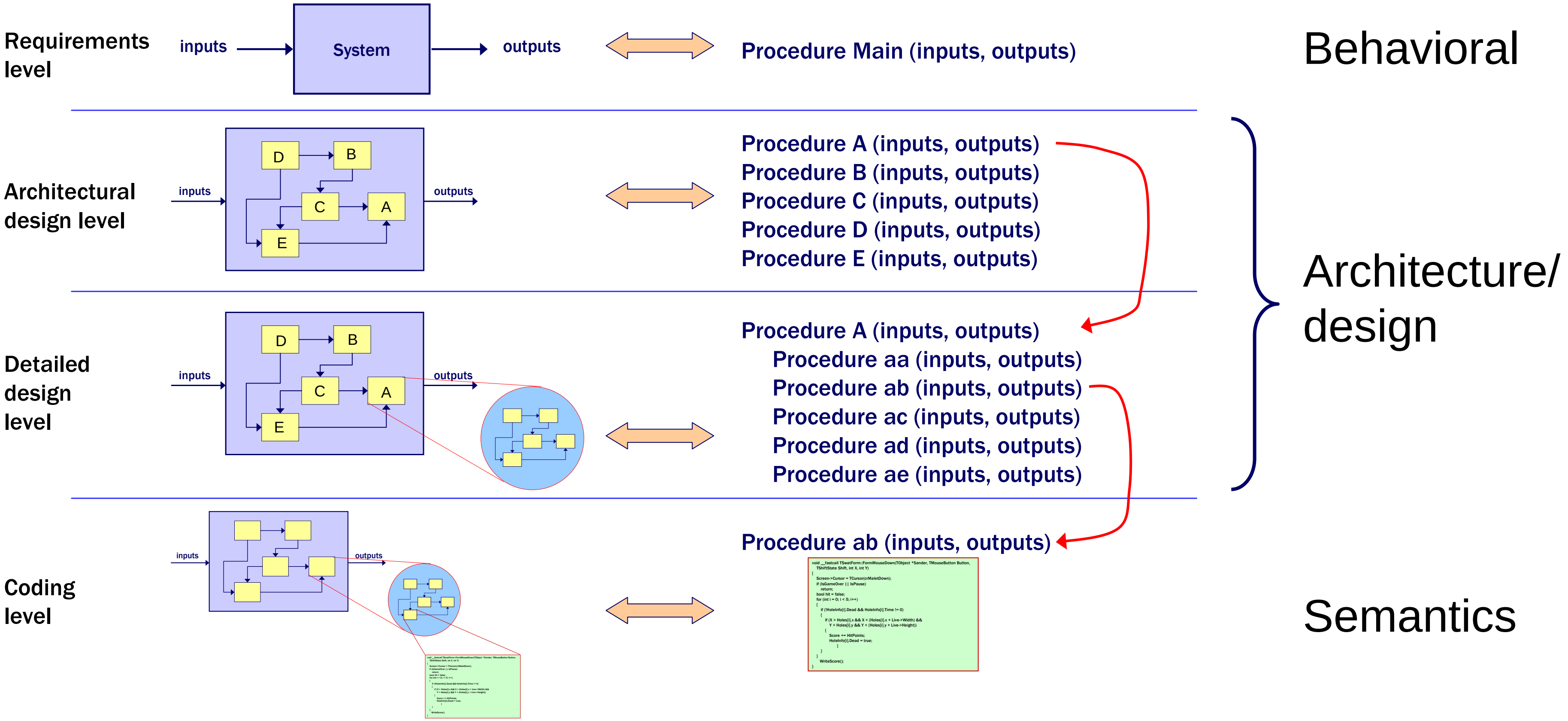
Coding level



Procedure ab (inputs, outputs)

```
void __fastcall TSwatForm::FormMouseDown(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
{
    Screen->Cursor = TCursor(crMaletDown);
    if (IsGameOver || IsPause)
        return;
    bool hit = false;
    for (int i = 0; i < 5; i++)
    {
        if (!HoleInfo[i].Dead && HoleInfo[i].Time != 0)
        {
            if (X > Holes[i].x && X < (Holes[i].x + Live->Width) &&
                Y > Holes[i].y && Y < (Holes[i].y + Live->Height))
            {
                Score += HitPoints;
                HoleInfo[i].Dead = true;
                    }
        }
    }
      WriteScore();
}
```

Code levels and roles



Requirements level

Procedure Main (inputs, outputs)

Behavioral

Architectural design level

Procedure A (inputs, outputs)
Procedure B (inputs, outputs)
Procedure C (inputs, outputs)
Procedure D (inputs, outputs)
Procedure E (inputs, outputs)

Detailed design level

Procedure A (inputs, outputs)
  Procedure aa (inputs, outputs)
  Procedure ab (inputs, outputs)
  Procedure ac (inputs, outputs)
  Procedure ad (inputs, outputs)
  Procedure ae (inputs, outputs)

Architecture/ design

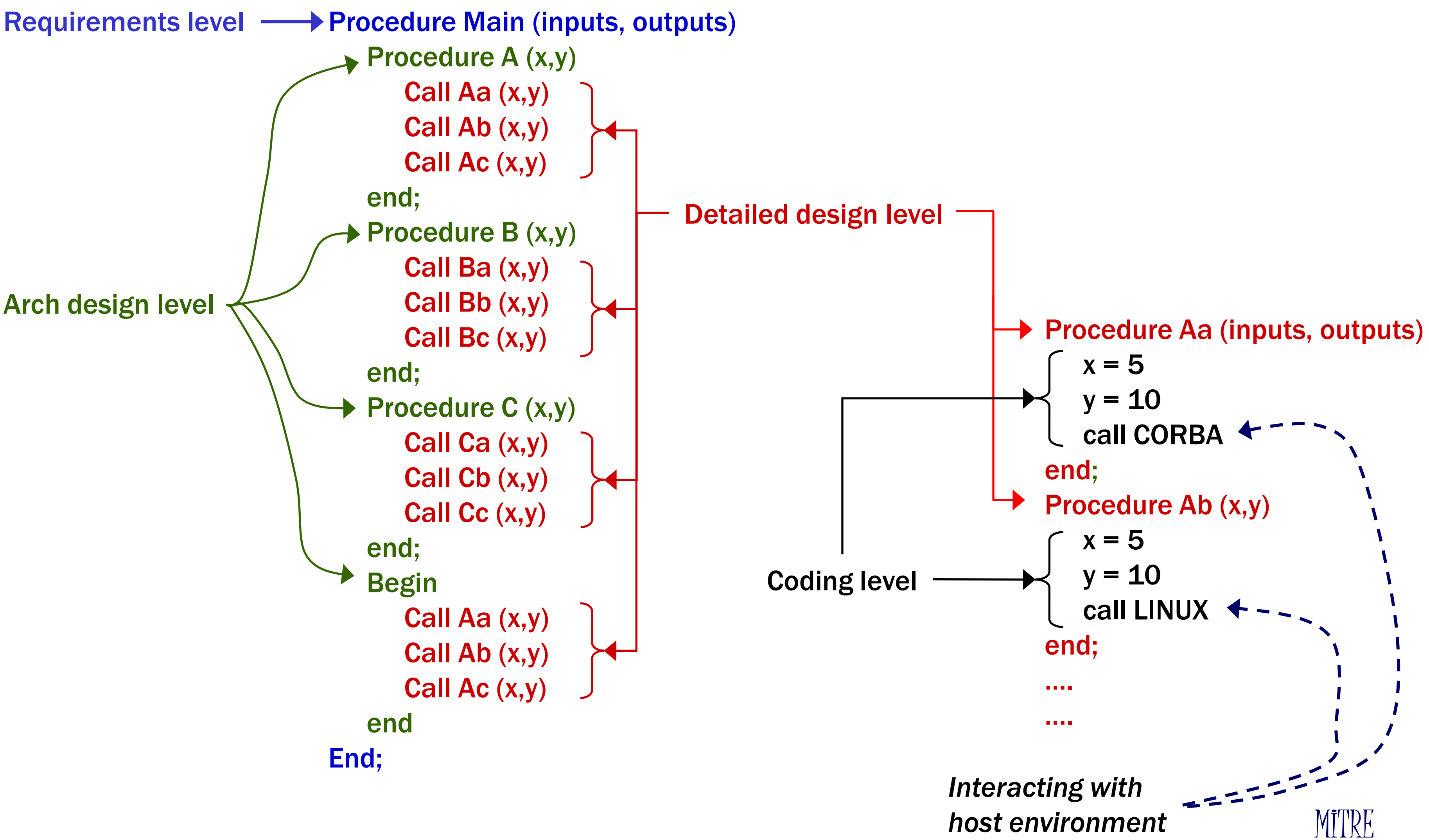
Coding level

Procedure ab (inputs, outputs)

```
void __fastcall TSwatForm::FormMouseDown(TObject *Sender, TMouseButton Button,
   TShiftState Shift, int X, int Y)
{
   Screen->Cursor = TCursor(crMaletDown);
   if (IsGameOver || IsPause)
      return;
   bool hit = false;
   for (int i = 0; i < 5; i++)
   {
      if (!HoleInfo[i].Dead && HoleInfo[i].Time != 0)
      {
         if (X > Holes[i].x && X < (Holes[i].x + Live->Width) &&
            Y > Holes[i].y && Y < (Holes[i].y + Live->Height))
         {
            Score += HitPoints;
            HoleInfo[i].Dead = true;
                 }
      }
   }
     WriteScore();
}
```

Semantics

Code structure



Requirements level

Arch design level

Detailed design level

Coding level

```
Procedure Main (inputs, outputs)
  Procedure A (x,y)
    Call Aa (x,y)
    Call Ab (x,y)
    Call Ac (x,y)
  end;
  Procedure B (x,y)
    Call Ba (x,y)
    Call Bb (x,y)
    Call Bc (x,y)
  end;
  Procedure C (x,y)
    Call Ca (x,y)
    Call Cb (x,y)
    Call Cc (x,y)
  end;
  Begin
    Call Aa (x,y)
    Call Ab (x,y)
    Call Ac (x,y)
  end
End;
```

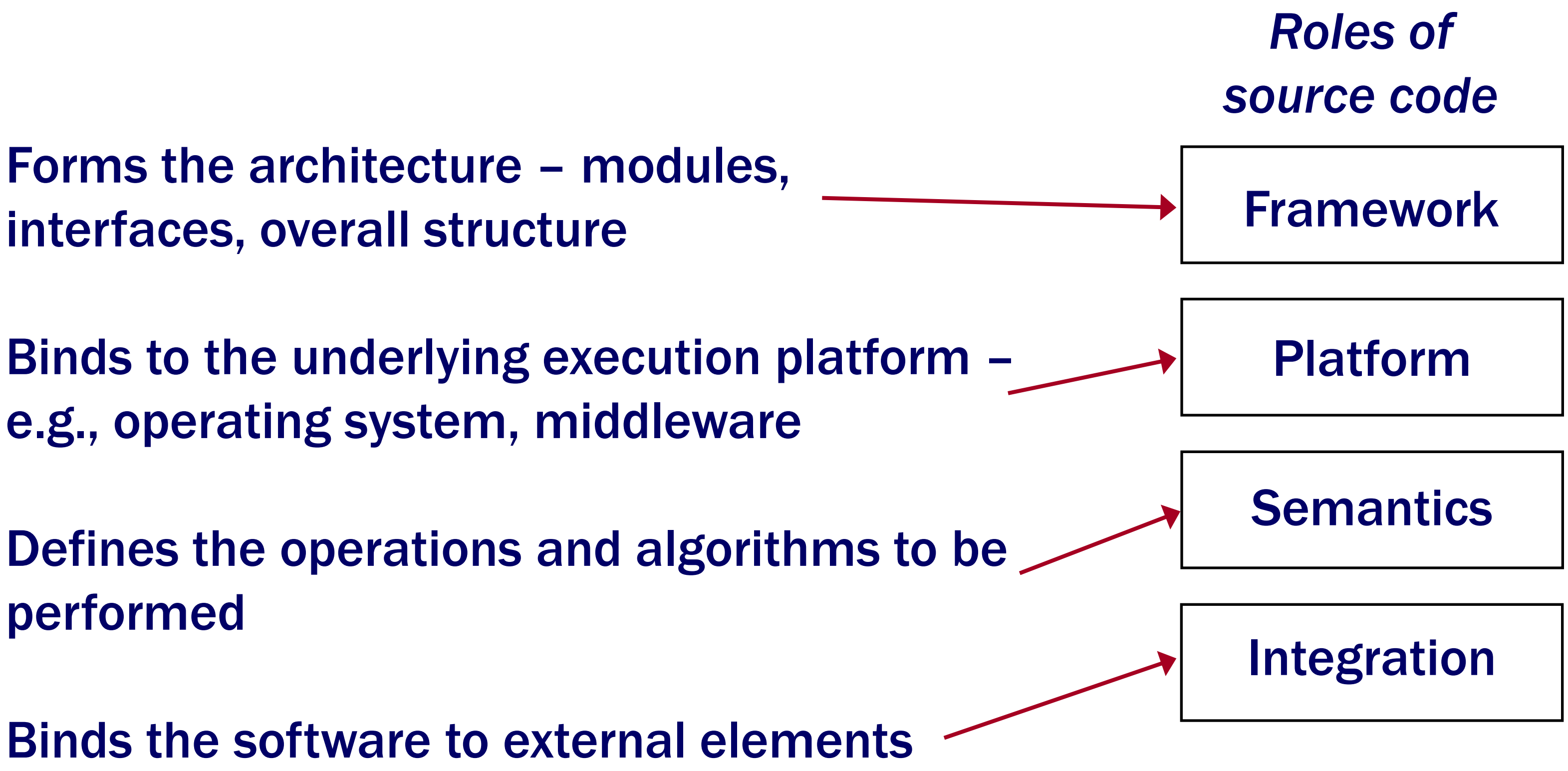
```
Procedure Aa (inputs, outputs)
  x = 5
  y = 10
  call CORBA
end;
Procedure Ab (x,y)
  x = 5
  y = 10
  call LINUX
end;
....
....
```

Interacting with host environment

Implications

- ❑ In any source code listing, possible to determine the levels (roles) that each SLOC supports (serves)
- ❑ Code can be written across entire development cycle
  - Reqs analysis → arch dsn → detailed dsn → code
- ❑ Separating code roles facilitates the use of tools
  - *e.g.*, specifying architecture using code allows use of tools to determine consistency of interfaces across system
- ❑ Dependencies to underlying computational platform can be isolated
  - Facilitating portability
- ❑ This observation is key to the use of models and other automatic techniques

Composition of source code



Roles of source code

Forms the architecture – modules, interfaces, overall structure → **Framework**

Binds to the underlying execution platform – e.g., operating system, middleware → **Platform**

Defines the operations and algorithms to be performed → **Semantics**

Binds the software to external elements → **Integration**

Agenda

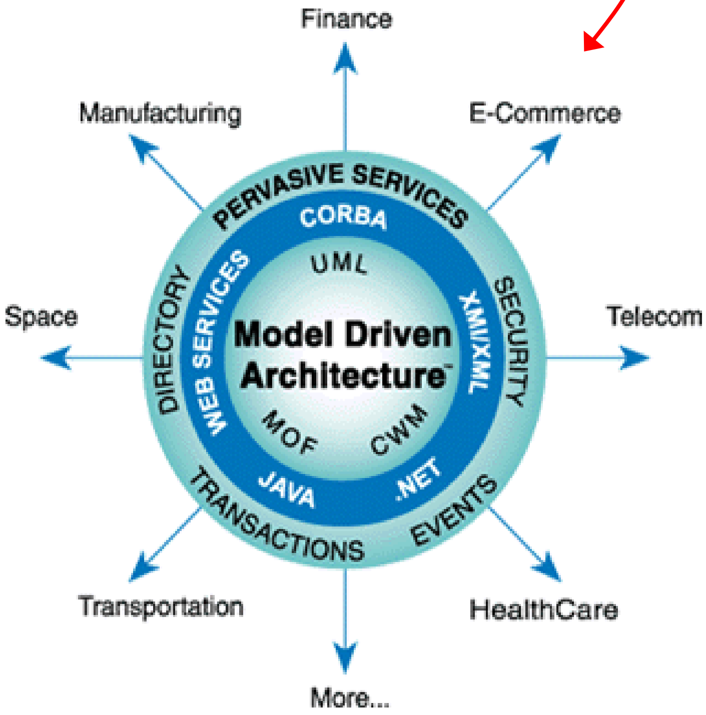
- ☐ Overview of tutorial
- ☐ Models
- ☐ Model Driven Engineering (MDE)
- ☐ Overview of software development activities
- ☐ Software development artifacts
- ➡ ☐ Model Driven Architecture® (MDA®)
- ☐ Assessment
- ☐ Recommendations

Model Driven Architecture®

- Trademarked term referring to the modeling approach defined by The Object Management Group® (OMG®)
  - A non-profit group that establishes standards by consensus
  - With significant participation by individual vendors
- Both the term and its abbreviation MDA® are registered trademarks of OMG®
  - As are Model Driven Development™, MDD™, Model Based Programming™, Model Based Development™, Model Driven Systems™, Model Based Management™, Model Driven Application Development™, + others…
- When these terms are used, they must refer to the technique defined by the OMG®
  - Cannot use MDA® to refer to another approach using different standards

The world of MDA®

- OMG® created a specific logo to refer to MDA®
- It includes the relevant standards that form the approach and the domains where it can be applied
- The "architecture" in the name refers to the architecture of tools that are defined by the OMG®
  - **Not** to the architecture of the system to be developed
- These tools include:
  - MOF™ – Meta-object Facility *
  - UML™ – Unified Modeling Language
  - XMI® – XML metadata exchange
  - CWM™ – Common Warehouse Metamodel



* Required for MDA modeling languages

MITRE

Core concept of MDA®

- According to OMG®

  "…the MDA® **separates business and application logic from underlying platform technology.** Platform-independent models of an application or integrated system's business functionality and behavior, built using UML and the other associated OMG modeling standards, can be realized through the MDA® on virtually any platform, open or proprietary, including Web Services, .NET, CORBA®, J2EE, and others. These platform-independent models document the business functionality and behavior of an application separate from the technology-specific code that implements it, insulating the core of the application from technology and its relentless churn cycle…."

- That is, MDA® supports portability

The MDA® approach

- Capture system requirements in a Computationally Independent Model (CIM)
- Model system design in a Platform Independent Model (PIM) using UML
  - Details about execution platform not included (*e.g.*, operating system, middleware,…)
- Augment PIM with system behavior details using Action Semantics
- Execute the system model to observe behavior
  - Refine model based on observations
- Transform model into Platform Specific Model
  - Details about execution platform added at this time
- Transform PSM into executable code (*e.g.*, C++)

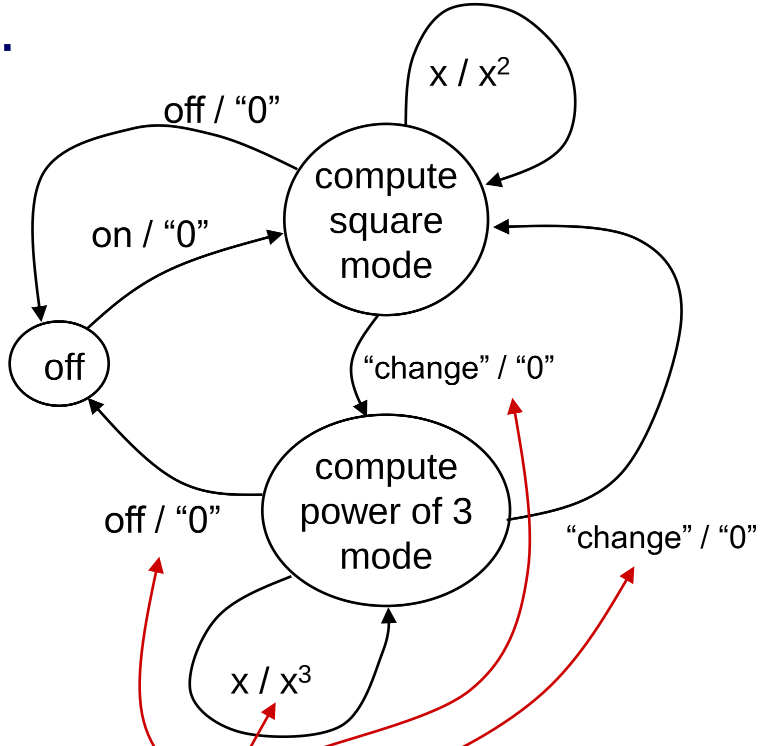
Details of defining semantics

- **Semantics typically defined in UML in two locations**
  - **Class methods**
  - **State entrance, exit, transition,...**

UML Class

Class Stack
Size: integer
Elements: Integer Array(1:Size)
Num_Entries: Integer
Push (In value)
Pop (Out value)
Get_Size (Out size)

Can supply action semantics code to perform these operations, thereby defining their semantics



Can supply action semantics code to compute these state transition actions thereby defining their semantics

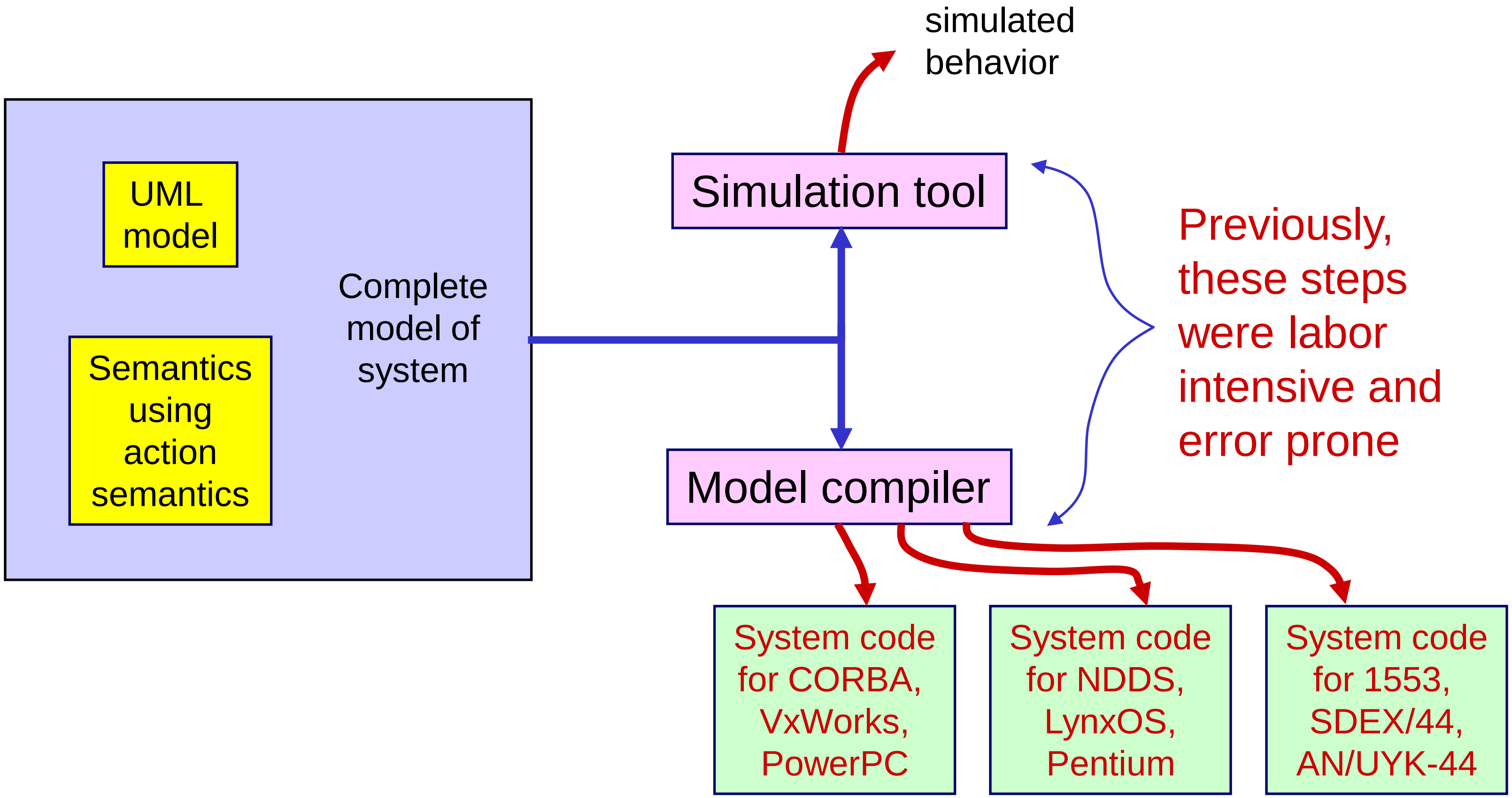
Sample action semantics

```
if Type_Quantity > 0 then
   Local_Selectable_Weapon_Type = TRUE
else
   Local_Selectable_Weapon_Type = FALSE
endif
ANT_Instance.Selectable_Weapon_Type = Local_Selectable_Weapon_Type

Local_Missile_Status = ANT_Instance.Missile_Status

if Local_Missile_Status = 'Simulate' then
   if ANM_Instance != UNDEFINED then
      URO_Instance = ANM_Instance -> R13 -> R23
      generate URO3:Cage_Requested() to URO_Instance
   endif
   generate TLCH1:Launch_Completed() to this
else
   # an armed launch has taken place
   ANT_Instance.Missile_Status = 'Not_Available'
   if ANM_Instance != UNDEFINED then
      [New_ANM_Instance] = ASM4:Select_Next_Missile_For_Release \
           [ANT_Instance.Selected_AA_Instance]
      generate ANM5:Release_Consent_Rescinded() to ANM_Instance

      if New_ANM_Instance != UNDEFINED and \
           New_ANM_Instance.Missile_ID != ANM_Instance.Missile_ID then
         generate ANM4:Missile_Deselected() to ANM_Instance
         generate ANM3:Missile_Selected() to New_ANM_Instance
         unlink ANT_Instance R11 ANM_Instance
         link ANT_Instance R11 New_ANM_Instance
         generate TLCH1:Launch_Completed() to this
      else
         generate ANM7:Missile_Safed() to ANM_Instance
         [Timer_ID] = Create_Timer[]
         this.New_Missile_Selection_Timer_ID = Timer_ID
         generate TIM1:Set_Timer (this.New_Missile_Selection_Timer_ID, \
            150, 'MILLISECOND', Event("TLCH2"), this)
      endif
   endif
   generate ASM14:Selected_Weapon_Quantity_Changed(Type_Quantity)
endif
```

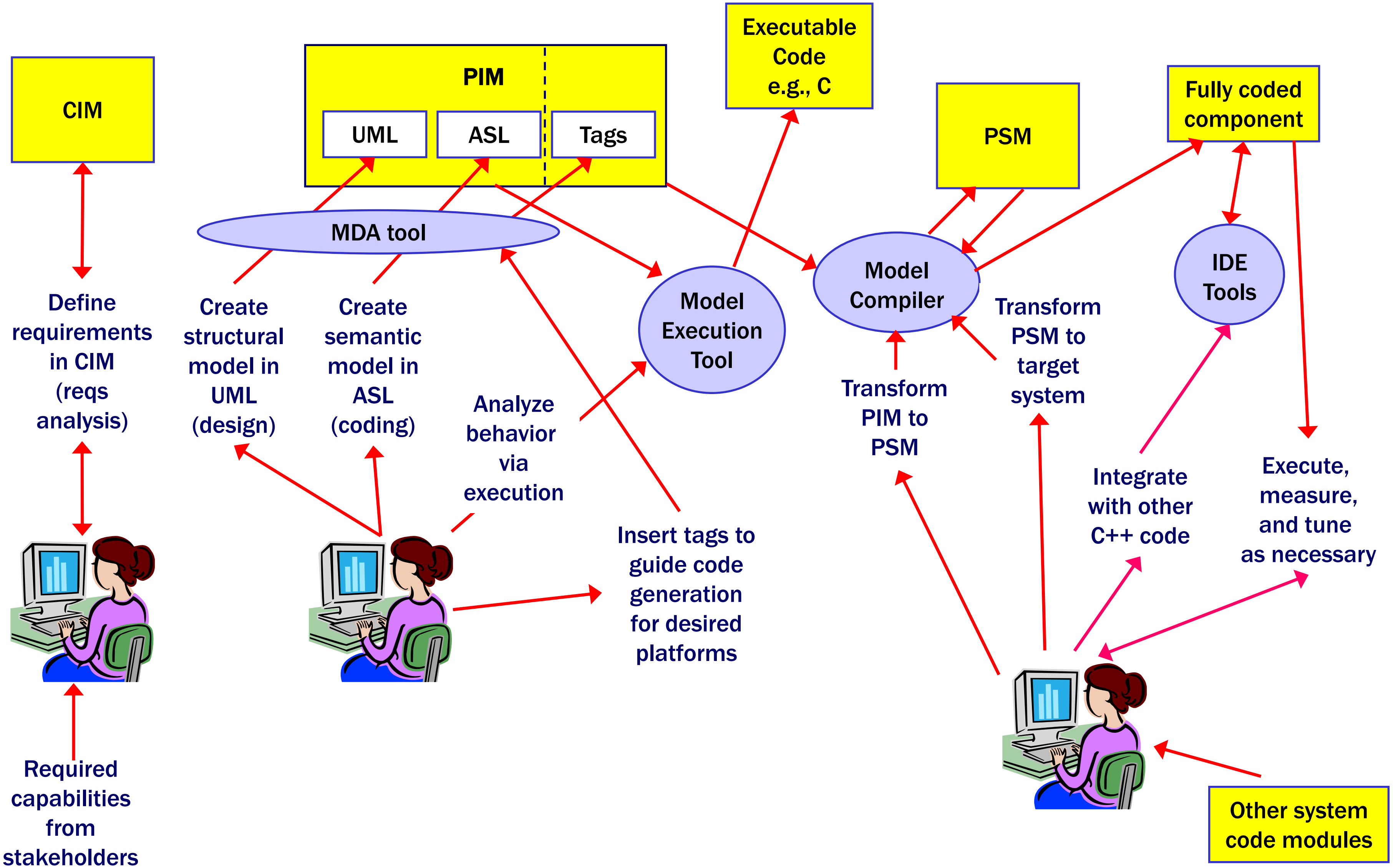
Kennedy-Carter. *Supporting Model Driven Architecture with eXecutable UML*

What it does for you

- **Allow observation of behavior directly from the "model"**
- **Support targeting the system to multiple run-time environments**



Example of MDA® process



CIM

PIM

UML

ASL

Tags

Executable Code e.g., C

PSM

Fully coded component

MDA tool

Model Execution Tool

Model Compiler

IDE Tools

Define requirements in CIM (reqs analysis)

Create structural model in UML (design)

Create semantic model in ASL (coding)

Analyze behavior via execution

Insert tags to guide code generation for desired platforms

Transform PIM to PSM

Transform PSM to target system

Integrate with other C++ code

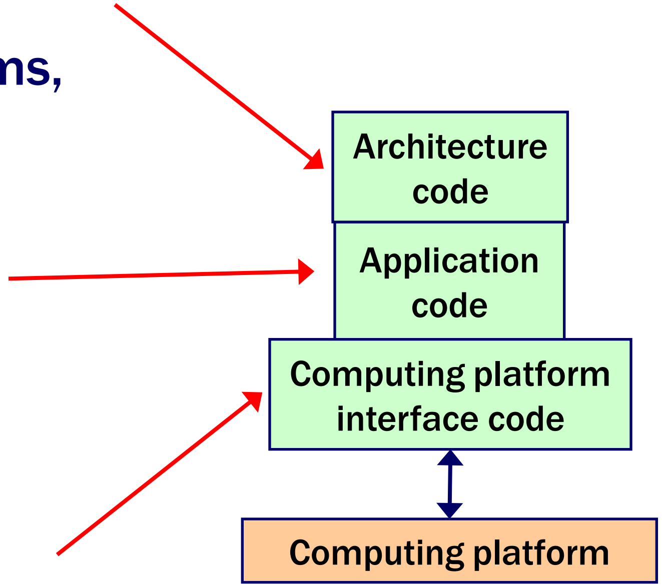
Execute, measure, and tune as necessary

Required capabilities from stakeholders

Other system code modules

What the pieces do

- UML allows the definition of the code that defines the architecture and detailed design of the system
  - Components and their interfaces
  - Using Class diagrams, state diagrams, sequence diagrams, …
- Action Semantics allows the definition of the semantics that defines the behavior
- The Action Semantics tags allow the definition of the alternate features of the various execution platforms that the component will operate on



Summary view

- ☐ MDA® relies on UML to define the overall structure of the system (classes, methods, components)
  - Uses UML to generate structural code
- ☐ MDA® relies on Action Semantics to define the semantics
  - Action Semantics = coding
  - The Action Semantics language could be C++, Kennedy-Carter's ASL, Pathfinder's PAL, or something else
  - It is inserted into the overall system structure appropriately
- ☐ MDA® relies on conditional compilation to achieve portability
  - Similar to a source code preprocessor

MDA® is a special case of MDE

Agenda

- ❑ Overview of tutorial
- ❑ Models
- ❑ Model Driven Engineering (MDE)
- ❑ Overview of software development activities
- ❑ Software development artifacts
- ❑ Model Driven Architecture® (MDA®)
- ➡ ❑ Assessment
- ❑ Recommendations

Overall assessment of MDA®/MDE

- ☐ Some users have experienced significant benefits from using MDA®
- ☐ MDA® facilitates early prototyping and evaluation by supporting executable model generation
- ☐ However, MDA® is not a silver bullet
  - It is not currently applicable for all situations
  - It does not replace modeling and simulation
- ☐ MDE, being the more general approach, is likely to be more responsive to emerging techniques and tools, *e.g.*,
  - Domain-specific languages
  - Powerful transformation engines
- ☐ MDE (including MDA®) is a rapidly emerging technology

Assessment – economics of MDE

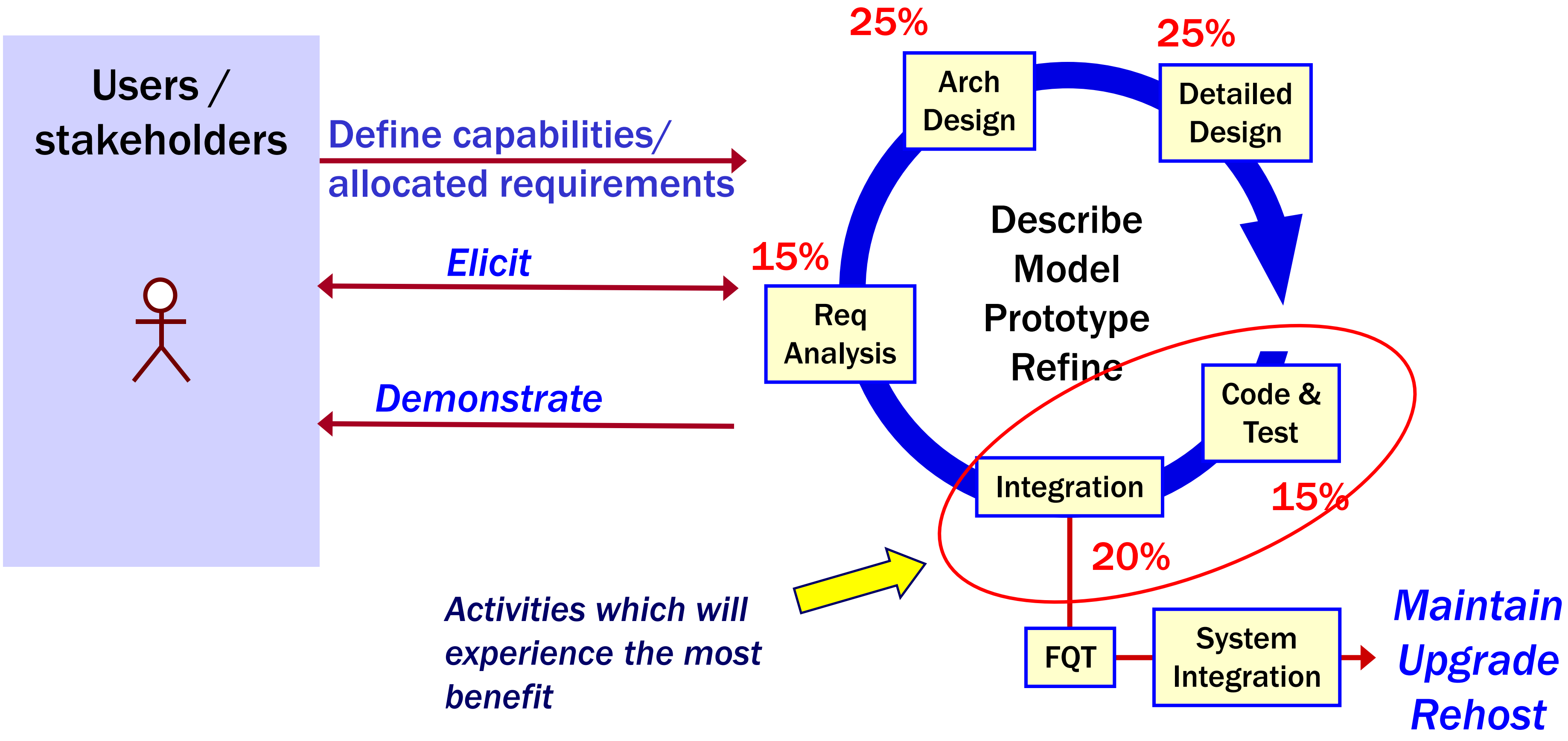
- Advantages of using MDE vary depending on
  - where applied in development life cycle
  - level of applicability to phase activities
- Advantages likely to increase as technologies mature
- Currently, benefit realized primarily in coding, integration, and maintenance
  - Tools automate some of detailed coding and integration work
  - Requirements analysis and design still required
- Users reporting benefits in reuse of models
  - Applied to similar systems

Effect on requirements

- Models generally do not address requirements, although they help to elicit requirements from users
  - This effect is realized primarily in reduction of requirements defects, hence rework late in the development cycle
  - Does not reduce requirements analysis effort
    - Still necessary to elicit, analyze, and refine requirements
  - Can help in capturing and documenting behaviors
    - Especially when using state diagrams
  - If executable, can use models to gain feedback from users if tools support such techniques
    - E.g., create sample dynamic user screens, allow users to interact, refine based on observations and experiences

Effort across development

Typical effort allocated to activities – actual experience varies by project



Users / stakeholders

Define capabilities/ allocated requirements

Elicit

Demonstrate

15% Req Analysis

25% Arch Design

25% Detailed Design

Code & Test 15%

Integration 20%

FQT

System Integration

Describe
Model
Prototype
Refine

Maintain
Upgrade
Rehost

Activities which will experience the most benefit

Source of effort allocations – Boehm, COCOMO, Reifer

Benefits of process discipline

- ☐ Applying techniques such as MDE forces a level discipline on developers
  - – In all activities from requirements analysis through coding
- ☐ As a result of this increased discipline
  - – Defect content is likely to drop
  - – Productivity is likely to increase
- ☐ MDE is not alone in realizing this benefit
- ☐ Other disciplined approaches have the same advantage
  - – as long as the discipline is applied

Productivity

- Large increases in productivity can be realized
- Key factors
  - Increased adherence to disciplined process (not unique to MDE)
  - Use of a higher-level Action Semantics Language
    - By raising level of code, more can be accomplished with fewer SLOC (more powerful expressiveness)
    - Experience shows that SLOC rate for writing ASL is about the same as for C++
      - But ASL → C++ expansion factor is important (can be as much as 3 to 1)
    - More efficient use of “mental energy”
  - Creation of reusable models
  - Creation of domain-specific model components
    - Similar to creating code libraries or class libraries

Assessment – design independence

- The models developed are not necessarily design independent
  - Unless they are at the requirements level
    - Possible with some tools/notations (e.g., state charts)
  - Otherwise they contain design and code
    - (ASL is after all a programming language)
  - Developers still need to create a design and write code
    - But some of it is handled by the default structure created by the tools
- Not necessarily a problem
- But, the models can be independent of the <u>computing infrastructure</u>
  - incl. middleware, target operating system, processor
  - Crucial to enhance portability

Assessment – maturity of tools

- ☐ Many available MDA®/MDE tools are becoming mature
  - Tools are maturing as more experience is gained
  - But not all – some tend to be immature
  - Relative to features, interoperability, robustness
- ☐ Limited interoperability across different vendor tools
  - Cannot easily move models, and so get locked into a vendor
  - True for UML-level models as well as Action Semantics code
- ☐ For safety-critical applications, essential to be able to verify correctness of the tools and model compilers
- ☐ Using model compiler is not a trivial task
  - Need to tailor and add tags to models – lots of details to worry about (e.g., multi-threading, priorities, …)
  - But such tailoring can be performed by a small percentage of project staff

Assessment – action semantics

- Action Semantics is a programming language
  - As implemented by some tool vendors, at a somewhat higher level of abstraction than C++
    - Tend to be proprietary to the vendor
  - Some tools use C++ directly
- Not implementation-independent
  - But can be host-platform independent
- To be used, need to be rich enough to cover crucial constructs needed by modern applications (e.g., exception handling)
  - Not the case for most such languages today
- Current UML 2.0 definitions too abstract
  - Other MDE tools depend on other notations with stronger semantics

Language considerations

- *Level of standardization* – if language is not standard, risk of getting locked into a specific vendor and model
  - *E.g.*, UML's Action Semantics defined at meta-language level, not language syntax level – leaves room for multiple interpretations
  - Implementations not standard
  - Will inhibit moving between tools and reuse
- *Precision of definition* – ambiguous or imprecise language definitions can lead to ineffective models and non-standard implementations
  - Implementations based on ambiguous guidance

Other assessment details

- MDA® does not remove the need to design
  - Developing a PIM involves designing and coding
  - Some non-MDA® techniques can avoid much design however
    - Possible to go from state charts or data flow diagrams directly to code
- MDA® will not support effort-free rehosting
  - Some effort required
    - to define the tags and
    - to fit the results to the operational environment
- MDA® does not currently provide any direct help for requirements analysis
  - Potentially the most critical activity in system development
  - But some MDE techniques can help in this area

Agenda

- Overview of tutorial
- Models
- Model Driven Engineering (MDE)
- Overview of software development activities
- Software development artifacts
- Model Driven Architecture® (MDA®)
- Assessment
- Recommendations

General strategy

- Weigh suitability of applying MDE approach to work to be performed
  - Requirements focus vs design/build
  - Degree of reuse / use of legacy content
  - Level of criticality of system
- Survey and assess tools and modeling languages to be applied
- Ensure adequate training for staff

Evaluating MDE approaches

Tool attributes	Tool score	Language attributes	Language score
Level of standardization		*Level of standardization*	
Completeness		*Precision of definition*	
Suitability		*Popularity*	
Confidence		*Completeness*	
Ease of use		*Suitability*	
Stability		*Level*	
Predictability		TOTAL	

The end

- Any questions?....

Contact information

Dr. William Bail

The MITRE Corporation

wbail@MITRE.org

571.205.7002