Modeling Legacy Architecture with UML

Jeroen van Tyn

Armstrong Consulting, Inc.
511 Second Street, Suite 100
Hudson, WI 54016
(800) 575-4160
www.armstrongconsulting.com

Goal of Today's Presentation

To present and discuss the process of modeling non-OO software architecture using UML and Rose

- What our mission was
- What the presenting situation was
- What we tried
- What we settled on

Agenda

- Overview of legacy system
- Situation,management charter, stakeholder requests
- Challenges
- Existing source constructs
- Various modeling approaches
- Modified 4+1 View of Architecture
- Summary and questions

Context: Motorola’s OMC-R System

- Operations Management Center - Radio
- Manages and configures radio wave-based wireless communications networks
  - Cell sites, transmitters, etc. that carry radio transmissions to/from cell phones, 2-way pagers, wireless Internet devices
  - Largest customer is Nextel

- Major product releases twice per year
  - Fast pace of wireless technology development demands software support for new features
- 1 million+ lines of procedural (non-OO) code
  - C, stored procedures

Situation

- No big picture of the product's software architecture
  - Existing documentation uneven and very low-level
- Lack of reliable brain-based system knowledge
  - Staff migration, staff turnover over software lifespan (1993 to the present)
- Distributed development team
  - Chicago, Texas, India
- No effective repository of project artifacts
  - Largely textual (Framemaker): very little visual modeling
  - No modeling tools other than sequence diagram drawing tool
  - Document management (CCM) ineffective

Management Charter

- #1: Reduce defects in software development
  - Many problems caught very late in the release development cycle

- #2: Facilitate ability to enhance the product
  - Cumbersome impact analysis for proposed features
- #3: Enable componentization
  - Network element configuration = most attractive feature set for componentization, incorporation into Motorola’s

Enable the above by modeling the software architecture using UML

Stakeholder Requests

- Management
  - Previous slide
  - Provide basis for establishing formal architectural analysis into the software development workflow
- Architects
  - Big picture of the system: major components & structures, relationships between them
  - Support impact analysis for adding new features
  - Context for documenting, developing and educating others about the system
  - Single-source repository for all architectural and design artifacts

Stakeholder Requests (cont'd)

- Subsystem Leads
  - Show how subsystems interact on a high level
  - Provide clear definition of subsystem boundaries (interfaces provided, interfaces required)
  - Support impact analysis for designing and implementing new features
  - Context for educating new developers about the system
- Developers
  - Roadmap for learning the design of the system

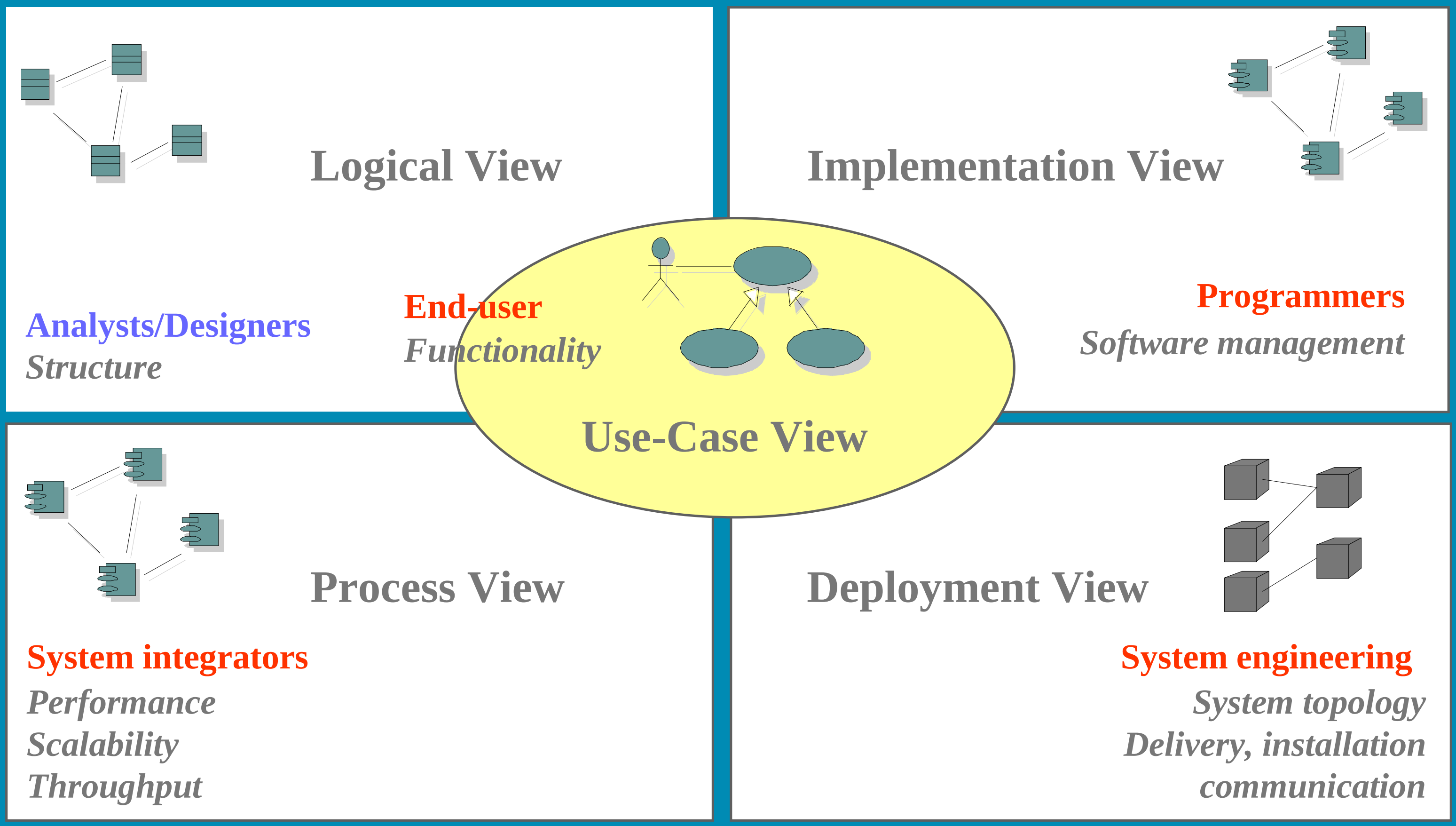
Challenges

- Non-OO source constructs
  - Implementation of functional decomposition (!)
  - Sparsely documented C programs, stored procedures
- Design artifacts largely out of date
  - Wide variety of detail, format, completeness
- No analysis artifacts
- Heterogeneous requirements artifacts
  - Combine high-level requirements with low-level design
- Many disparities and contradictions across artifacts
- Architects new to the system
  - No “gray-hairs” to rely on: everyone learning

Existing Source Constructs

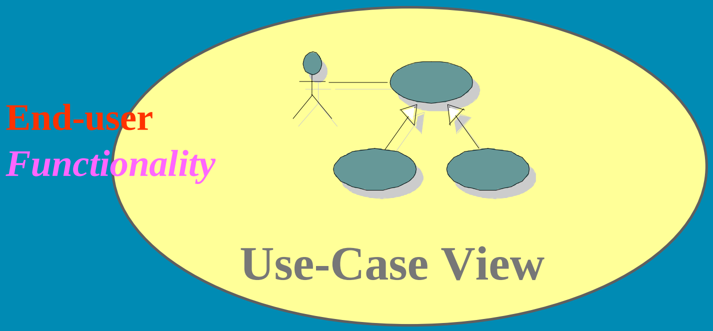
- C programs
- Processes (EXE’s) communicating via modified IPC
  - Asynchronous messaging simulating synchronous behavior via coded “wait” loops
- Large numbers of complex stored procedures
- Several hundred business rules
  - No traceability to design, implementation: all done by hand analysis

Status: 4+1 View of Architecture

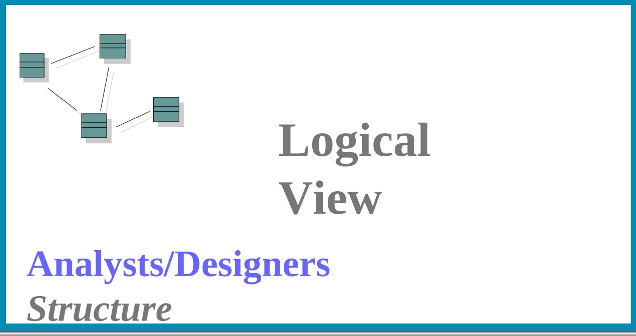


Status: 4+1 View of Architecture (cont'd)

- Use Case View
  - System functionality poorly documented from an outside perspective
  - No visual modeling

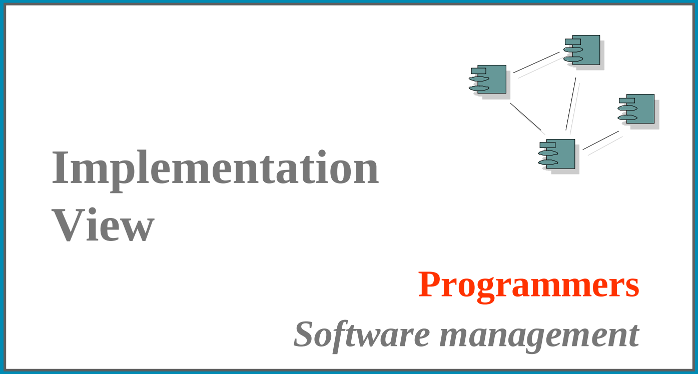


- Logical View
  - No analysis model: everyone has their own concept of key abstractions
  - Design model spotty, mostly textual and very detailed

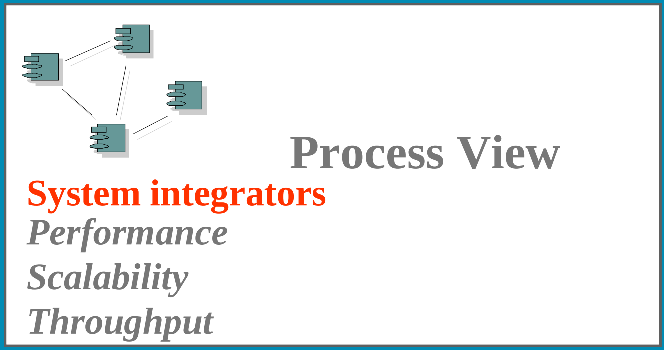


Status: 4+1 View of Architecture (cont'd)

- Implementation View
  - Software constructs already in place, so not a priority

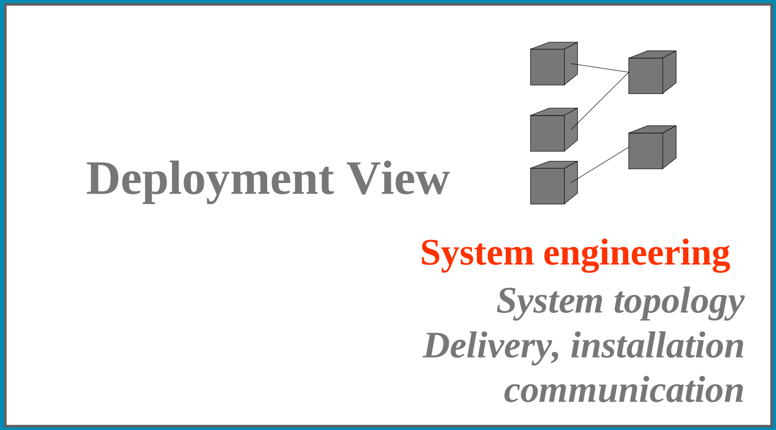


- Process View
  - Processes in place
  - Communication between them documented, but not up to date



Status: 4+1 View of Architecture (cont'd)

- Deployment View
  - Very well understood
  - Fairly well documented



Approach

- Raise the level of abstraction
  - Focus on Use Case View and Logical View
- Gain agreement on basic functionality, key concepts
- Define modeling levels
  - What will be in the Logical View?
- Adapt and extend UML as needed
- Work top-down and bottom-up simultaneously
  - Top-down: create requirements artifacts, analyze from there
  - Bottom-up: start with existing code, abstract upward

Top-down Effort

- Existing system = implementation of functional decomposition (!)
- Use Case modeling
  - Looked to user interface for Use Cases
  - Actors fairly readily identified
  - Challenge: keep Use Cases at a high enough level, yet with enough detail to decide which are architecturally significant
- Analysis Modeling/Use Case Realizations
  - Big challenge: no agreement on key abstractions (big surprise to team members)
  - Very difficult to separate conceptual entities from implementation

Bottom-up Effort

- Goal: meet the top-down model somewhere in the middle
- Where to start? What equals a UML “splat” (classifier)?

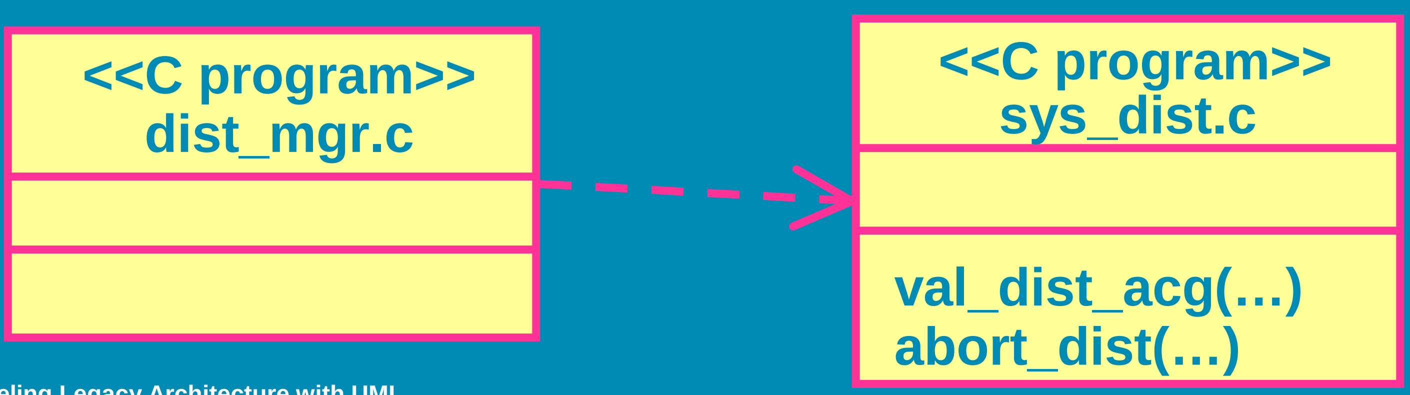
Three major tries at this...

Try #1: C Program $\cong$ UML Class

- C program is a collection of (more or less) related functions
  - Corollary to class with operations
- State is an issue
  - State captured in files, tables, other data structures: requires further modeling conventions

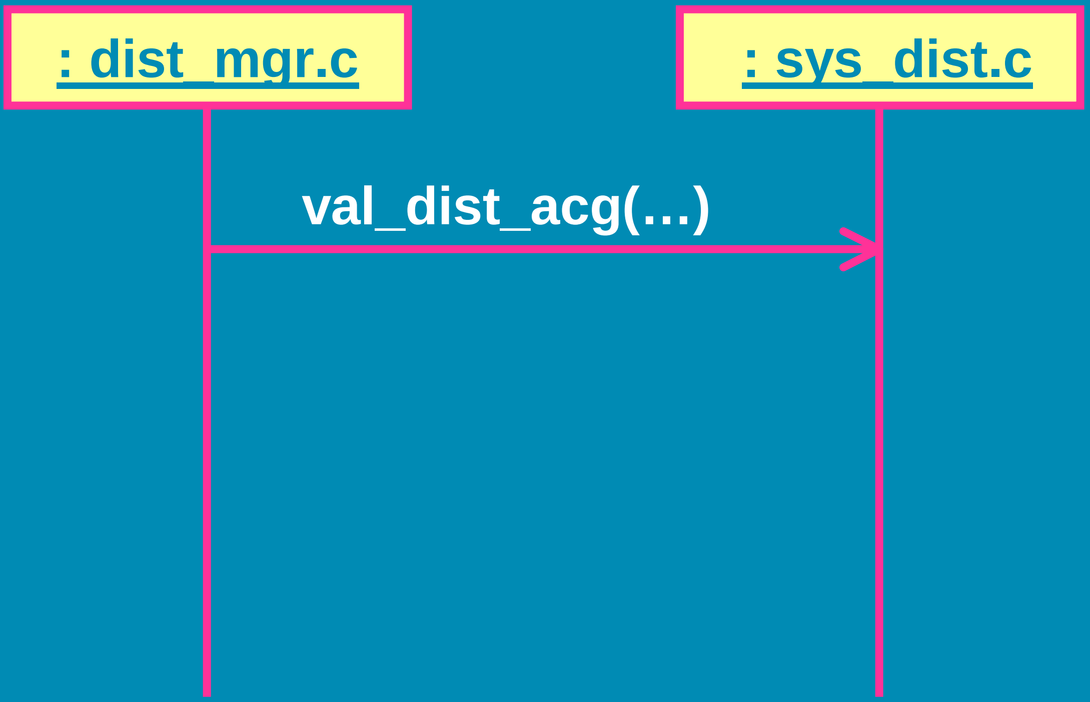
Class Diagrams

- C function = operation on class
  - Parameters = function parameters
- No attributes: programs themselves are stateless
- Structural relationships
  - No instances, so association, aggregation, composition are irrelevant
  - Dependencies between caller & “callee” (next slides…)



Interaction Diagrams

- Interaction diagrams
  - Message = invocation of function in another C program

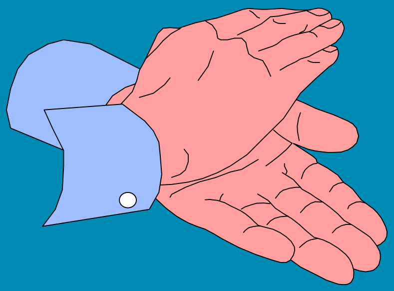


Packages

- Packages
  - Package structure parallels UNIX directory structure
  - Package dependencies required significant hand analysis of code

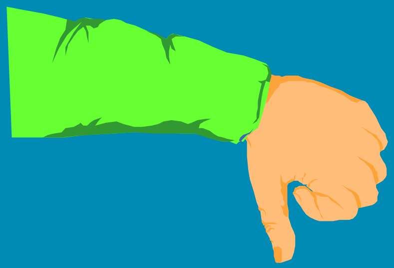
Rose Demo: C Programs as UML Classes

Evaluation of Try #1



- Pro:
  - "Classes" easy to identify
  - Automatic reverse engineering
    - Custom program converts C headers to C++ format
    - Rose C++ reverse engineering capability
    - Interaction diagramming easily done in Rose

Evaluation of Try #1 (cont'd)



- Con:
  - Too many classes: not an effective big picture
  - Not architectural in nature: too constrained by implementation
  - Doesn't effectively show communication between larger constructs
  - Doesn't yield new information
    - Code profiling tools, UNIX commands, MAKE files just as effective

Try #2: OMC-R Process ≅ UML Subsystem

- OMC-R process = EXE “instance”
  - Process “contains” C programs
- Processes communicate via IPC
  - Motorola-extended version of standard IPC
  - Implemented as C functions making calls to common IPC library: parameters include receiving process ID, message ID and message parameters

Sender code fragment (inside LMUI process):

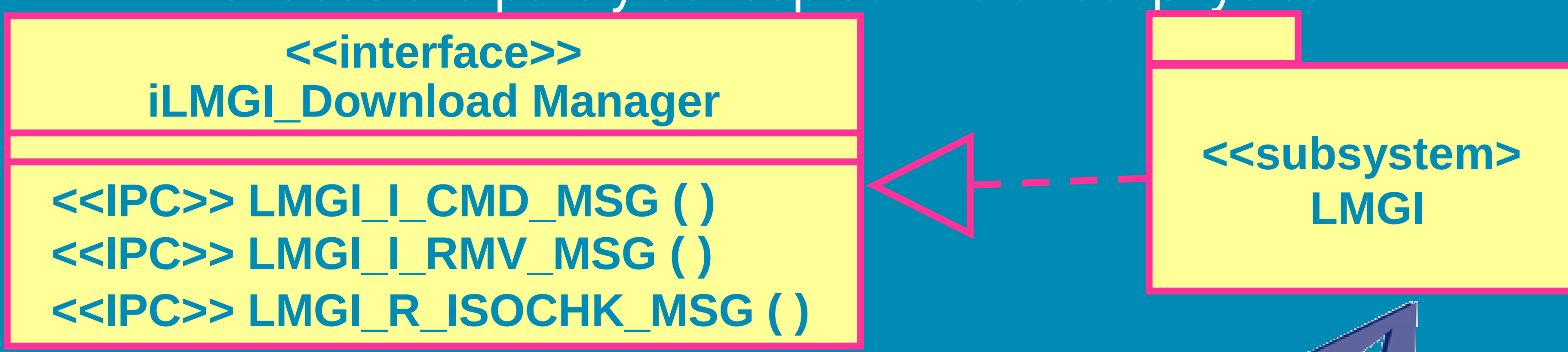
ipc_mq_send_msg (<receiver ID>, LMGI_I_CMD_MSG, …)

Receiver code fragment (inside LMGI process):

ipc_mq_receive_msg (<my process ID>, LMGI_I_CMD_MSG, …)

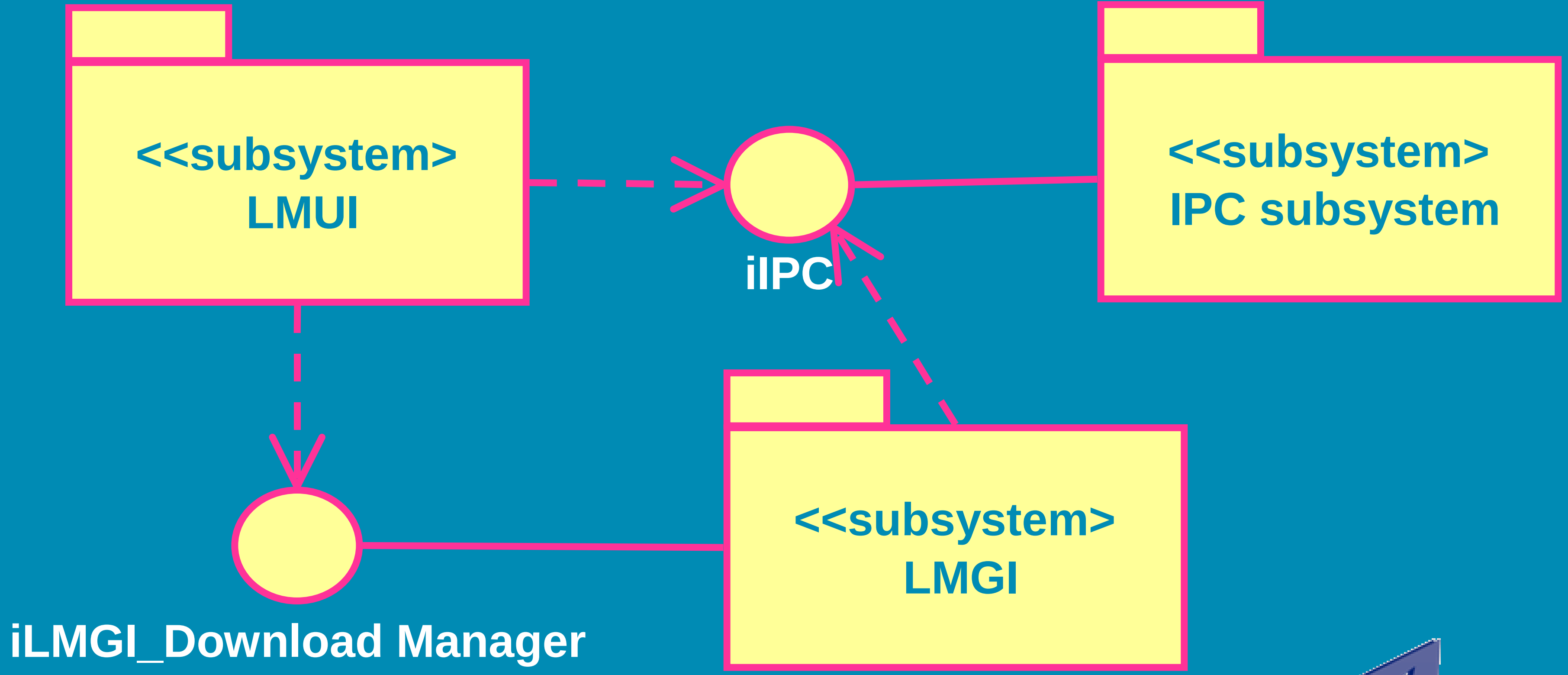
Class Diagrams

- Subsystem encapsulation parallels implementation
  - C programs make direct function calls only to other C programs within the same containing process
  - Calls to other processes requires IPC call
- Interface = functionally coupled set of IPC messages received by contained C programs
  - Interfaces are purely conceptual: no direct physical



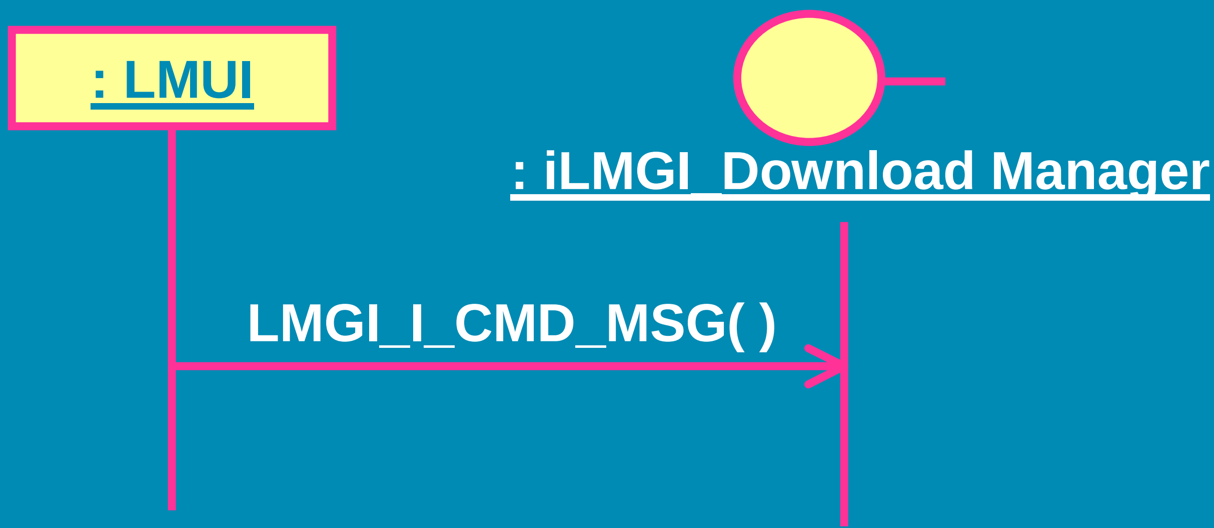
Class Diagrams (cont'd)

- Subsystem dependency: C program inside a sub-system sends IPC message to some other process
- Sender and receiver both dependent on IPC subsystem



Interaction Diagrams

- Message = name of incoming IPC message
  - Remember: IPC messages are on interfaces **(!)**

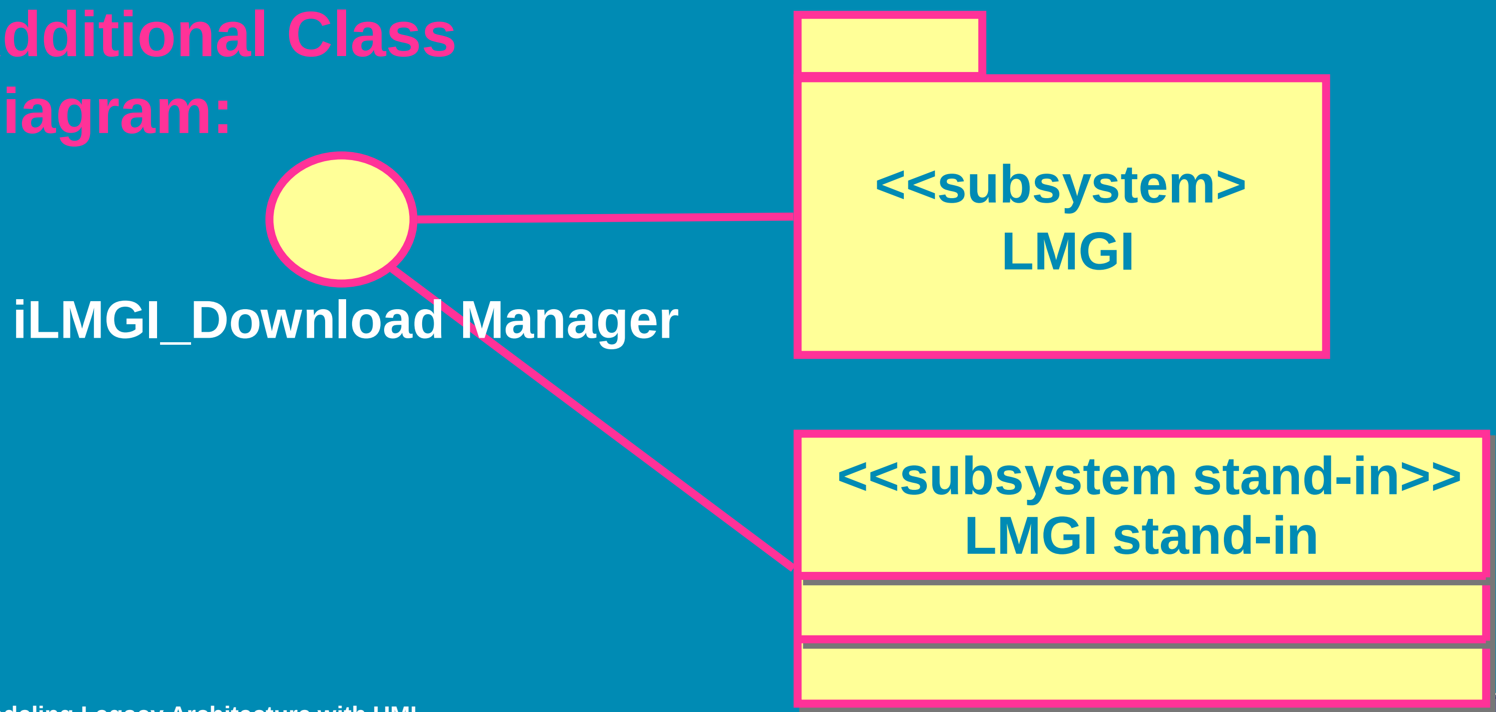


- But how useful is this? …all interaction diagrams would show just one “step”
  - We’re interested in showing a conversation between a set of subsystems...

Subsystem Interaction Diagrams

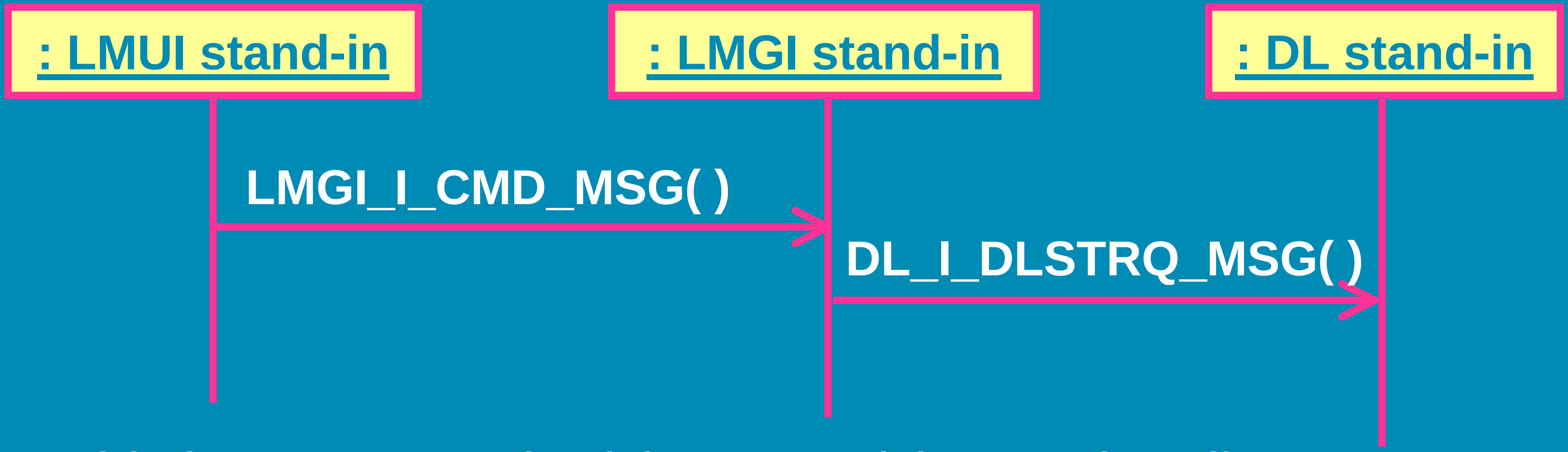
- Rose can’t show subsystem directly on an interaction diagram
- “Subsystem stand-ins” solve the problem
  - Subsystem stand-in = class the realizes same set of interfaces that the subsystem realizes

Additional Class Diagram:



Subsystem Interaction Diagrams

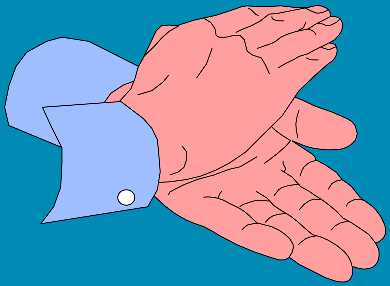
- Interaction diagram using subsystem stand-ins:



- This is a non-standard (non-RUP) interaction diagram
  - Combines external view of subsystems receiving messages with internal view of interface message realizations ! !
  - ***But it models what we wanted to show ! !***

Rose Demo: Processes as UML Subsystems

Evaluation of Try #2



- Pro:
  - Larger-grained picture = helpful
  - Grouping IPC messages into interfaces = helpful
  - Can flesh out subsystem realizations as needed for understanding
  - Interaction diagrams (using subsystem stand-ins) = useful

- Con:
  - Processes still too fine-grained and implementation-constrained
  - Stakeholders want info on incoming AND outgoing messages
  - Architecture should show “externally visible properties”
    - Received messages are only part of the story

Try #3: “Subsystem” ≅ Conceptual Component

- OMC-R “subsystems”: larger conceptual chunks
  - “Subsystem” (“TNM functional area”) = a collection of C programs
  - Run in (perhaps) several processes
  - Are functionally distinct
  - Conceptually layered
    - Application (CM, FM, PM)
    - Infrastructure (IPC, SNMP, event handling) = collections of implemented architectural mechanisms
  - Communicate strictly via IPC messages
  - Arguably “own” important data structures/stores (state)

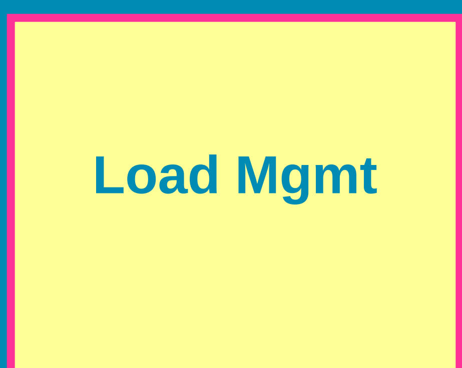
Notation

- Blend of two sources:
  - *Applied Software Architecture* by Hofmeister, Nord & Soni
  - Rational's UML Profile for Real-time Modeling
- Concepts:
  - component
  - port
  - connector
  - protocol
  - binding

Conceptual Component

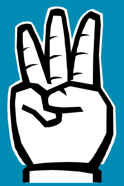
- UML definition: “a physical, replaceable part of a system that packages implementation and conforms to and provides the realizations of a set of interfaces”
- Conceptual component:
  - is conceptual, not (necessarily) physical
  - decomposable into other components

Elided notation



Canonic notation

Port

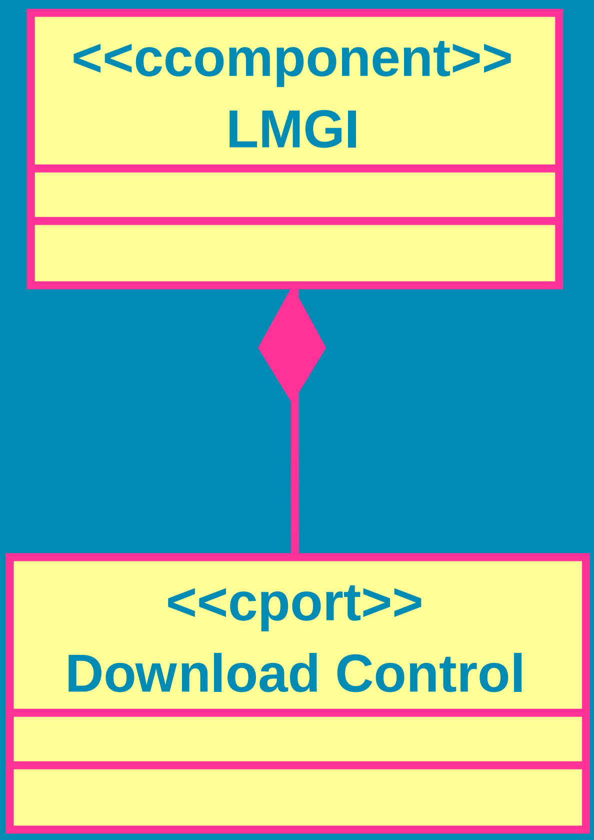


- An interaction point for a conceptual component
  - In context of a particular collaboration with another component
- Is different from a UML interface in that it:
  - defines both incoming and outgoing messages
  - may have its own implementation
  - is associated with a ***protocol*** that mandates how the incoming and outgoing messages can be ordered

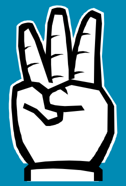
Port (cont'd)

Elided notation

Canonic notation



Connector

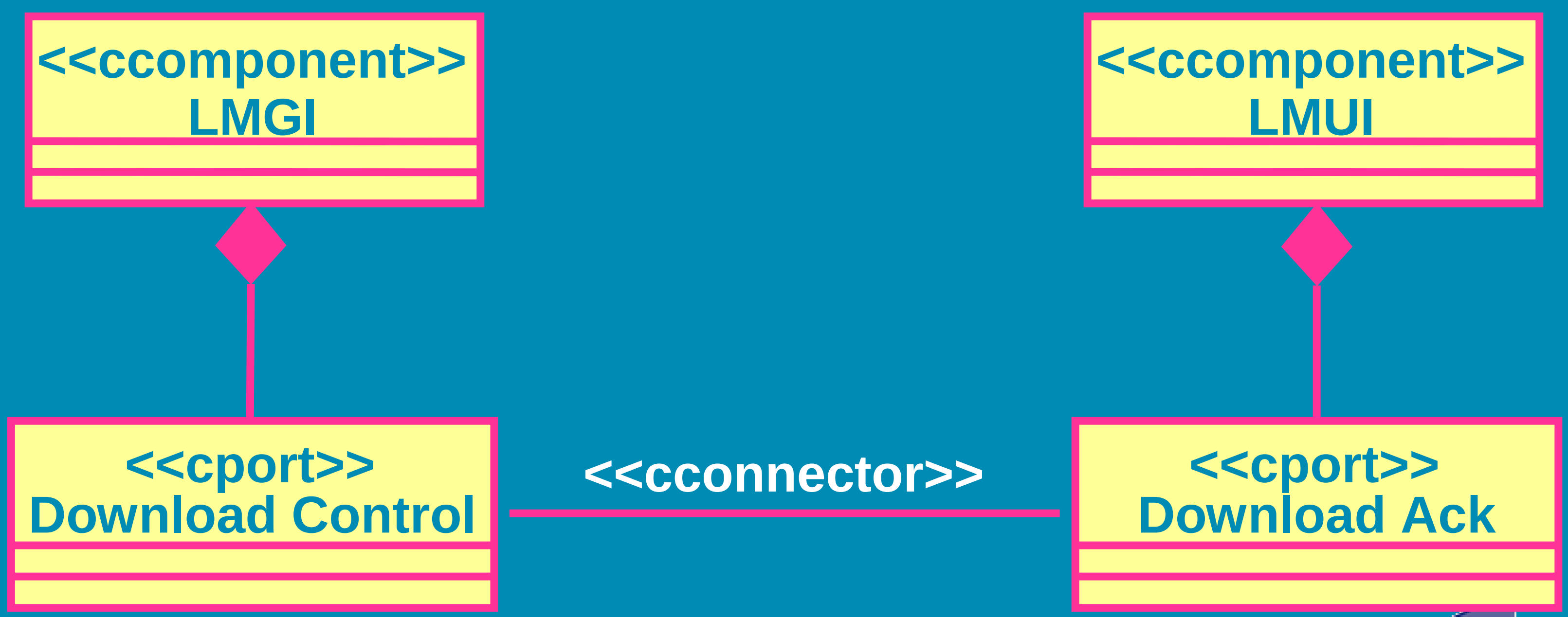


- Represents a communication channel between two ports that play complementary roles in a protocol
- Is different from a UML association in that it
  - connects ports instead of classes
  - places the protocol restriction on the ports that it connects

Class Diagram: Components, Ports, Connectors

Elided notation:

Canonic notation:



Protocol

- A specification of desired behavior that can take place over a connector
  - Explicit specification of contractual agreement between participants in the protocol
  - Each participant plays a specific protocol role
  - May specify valid communication sequences, documented by state machines and/or sequence diagrams
- Binary protocol
  - Involves just two participants: ***base*** and ***conjugate*** roles
  - Conjugate role transposes incoming and outgoing message sets of the base role

Protocol (cont'd)

Extended UML:

<<binary protocol>>
Download Commands

incoming
LMGI_I_CMD_MSG ( )

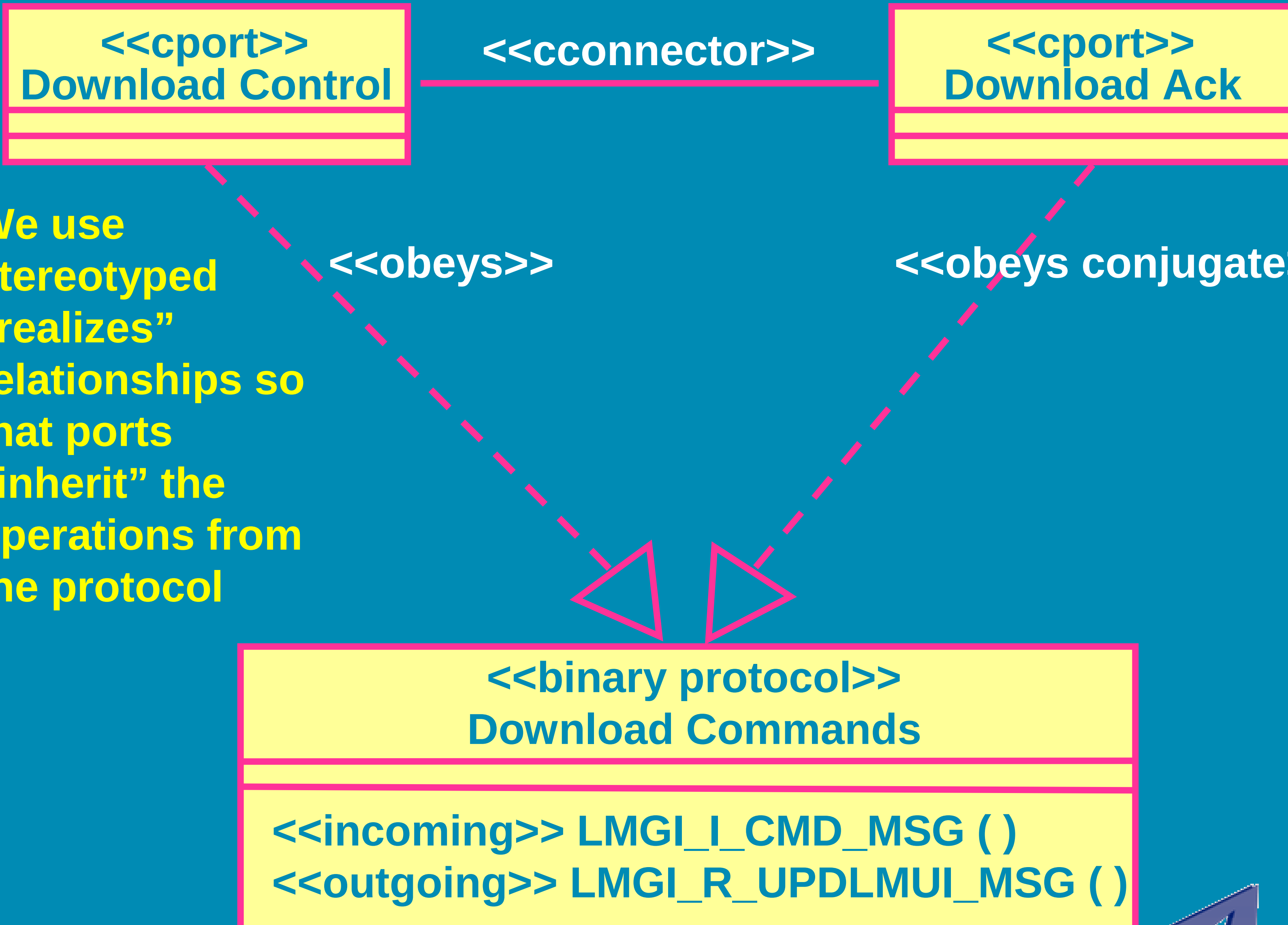
outgoing
LMGI_R_UPDLMUI_MSG ( )

UML with stereotypes:

<<binary protocol>>
Download Commands

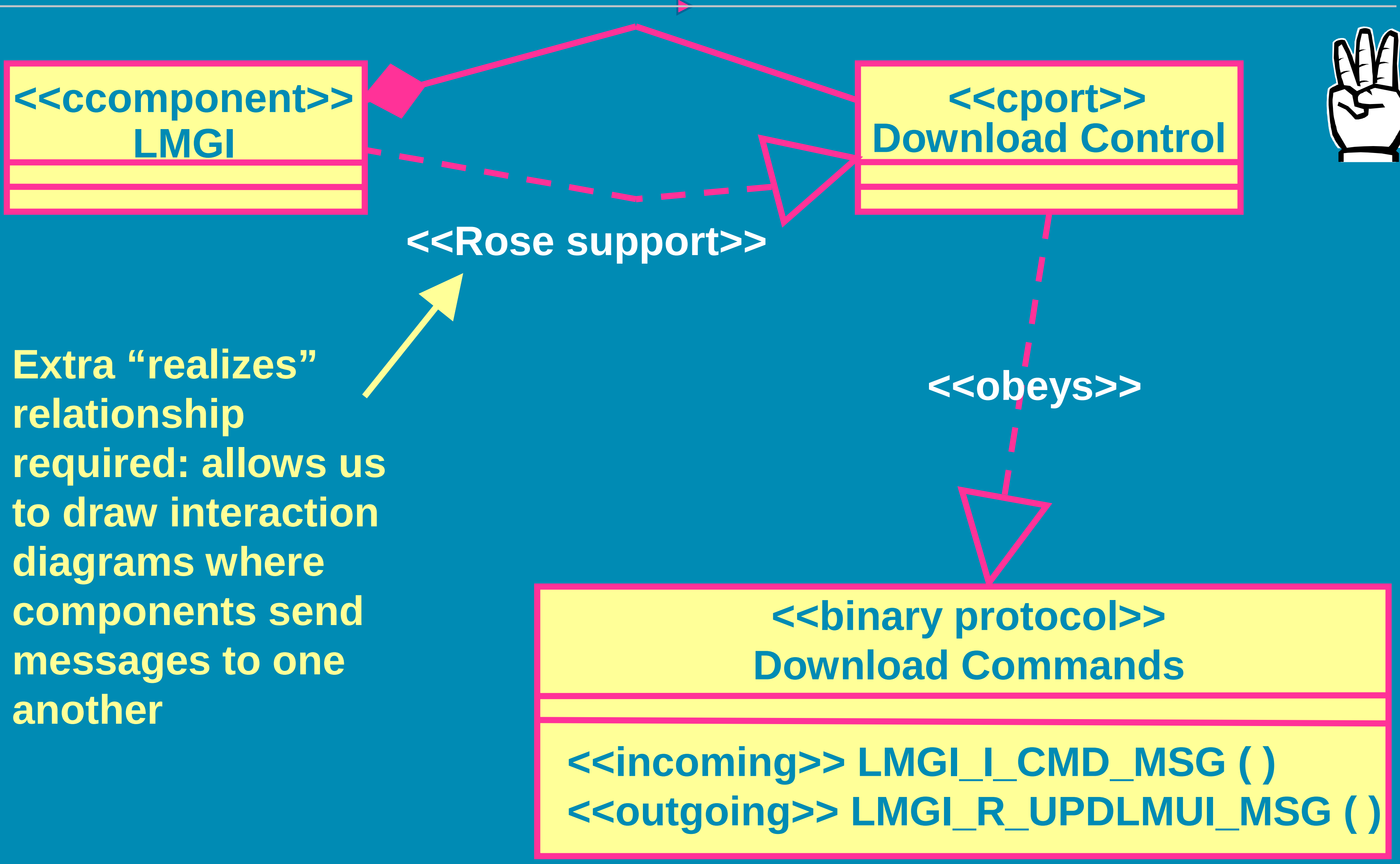
<<incoming>> LMGI_I_CMD_MSG ( )
<<outgoing>> LMGI_R_UPDLMUI_MSG ( )

Class Diagram: Ports and Protocols



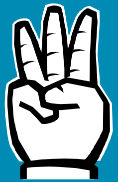
We use stereotyped “realizes” relationships so that ports “inherit” the operations from the protocol

Class Diagram: Rose Support



Extra “realizes” relationship required: allows us to draw interaction diagrams where components send messages to one another

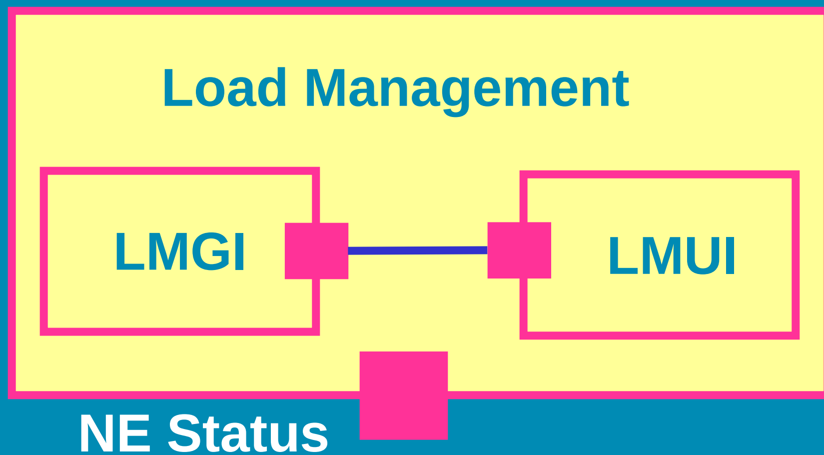
Interaction Diagrams (new slide!!)



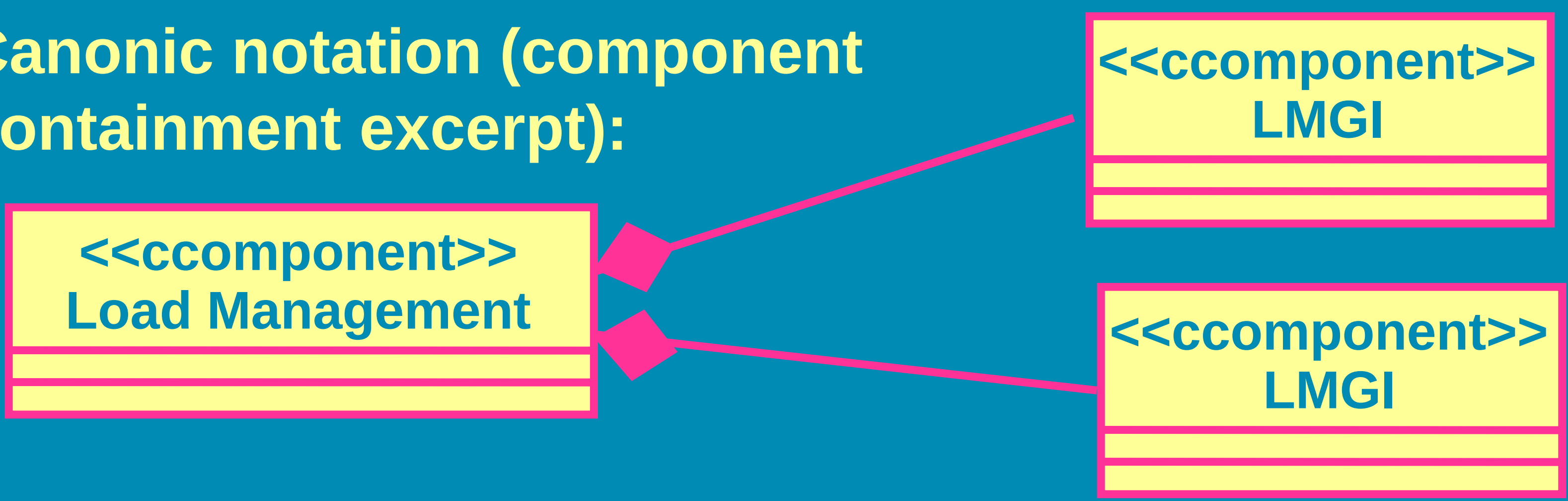
- I'll show an example in Rose here…

Notation: Scalability

Elided notation:



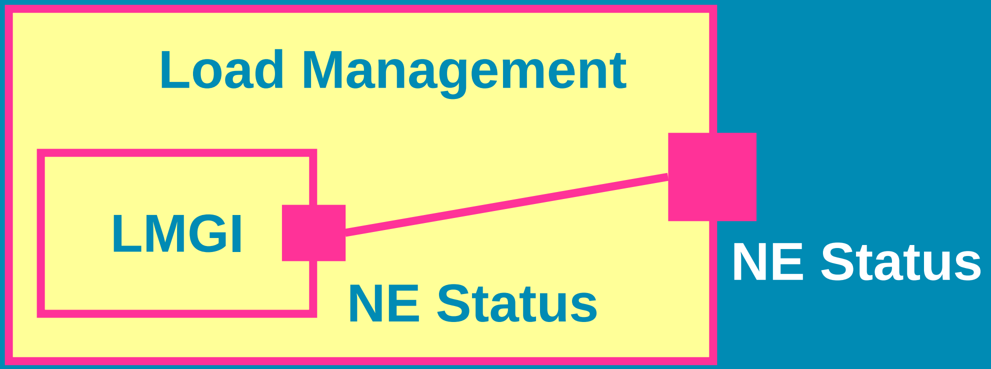
Canonic notation (component containment excerpt):



Notation: Port Binding

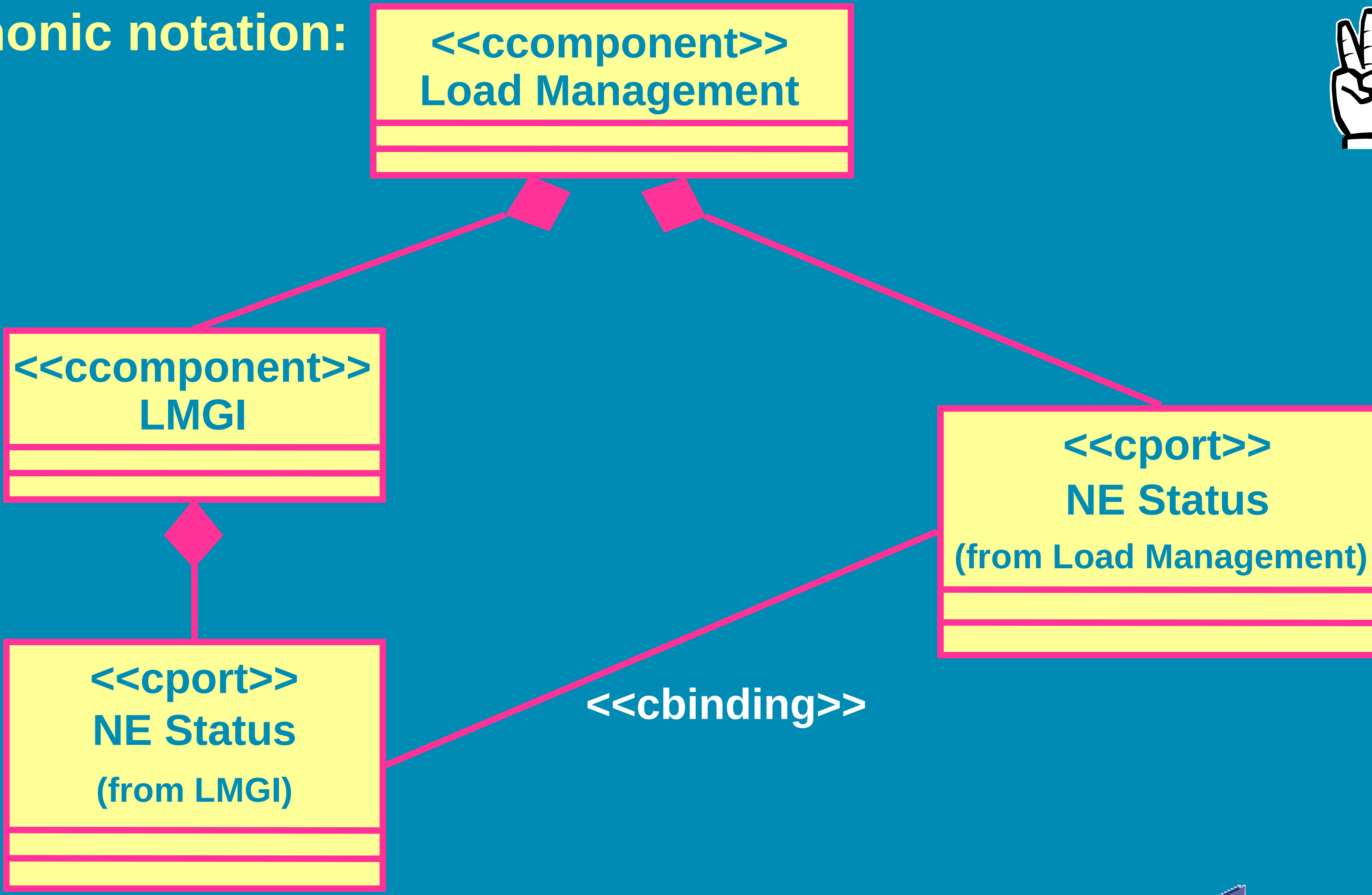
- Port of enclosed component may be bound to port of enclosing component
  - Enclosing component's port may or may not obey the exact same protocol as enclosed component's port obeys
  - Multiple ports from enclosed components may bind to a single port of an enclosing component

Elided notation:



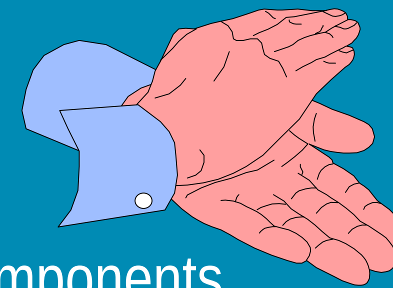
Notation: Port Binding (cont'd)

Canonic notation:

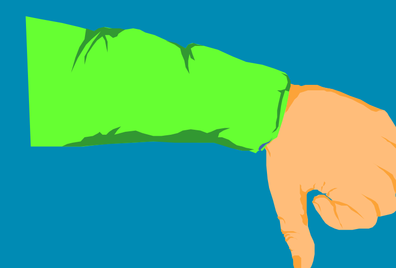


Rose Demo: Conceptual Components

Evaluation of Try #3



- Pro:
  - Notation is conceptually scaleable
  - Captures constraints on interactions between components
  - Fits with the real-time, asynchronous nature of OMC-R system
  - Provides big-picture view of system
- Con:
  - Difficult to model in Rose: Rose Realtime is better, but UML stereotypes mean somewhat different things
  - Interaction diagrams difficult, error-prone
  - Modeling state is still a big issue



Modified 4+1 View of Architecture

- Use Case View
  - Captured functional requirements separate from solutions
- Logical View
  - Analysis model
    - Crucial element for gaining understanding, agreement on the fundamental system concepts
  - Architectural model
    - High-level functional abstractions in the solution space
  - Design model
    - Combination of abstract elements and concrete elements directly traceable to code
- Process, Implementation, Deployment Views

Summary Reflections

- Architectural modeling is an immature art
  - Many different approaches, all have strengths and weaknesses
- Modeling non-OO constructs with UML presents special challenges
  - We couldn’t find anyone else who was doing this…
  - How to capture state?
- Visual modeling with Rose
  - Complicated diagrams
  - Model organization is crucial (deep package structure)

Summary Reflections (cont'd)

- Extended UML notation for architectural modeling
  - Conceptual modeling was very useful, scaleable
  - Lower-level modeling (Hofmeister/Nord/Soni's Module and Execution views) was confusing, so we omitted it
  - The notational conventions we ultimately adopted served main purpose of showing big picture of legacy architecture
  - Ongoing refinement of notational conventions
- In short: if it promotes clarity, use it; else leave it out

Questions

Thank you for your attention and participation!