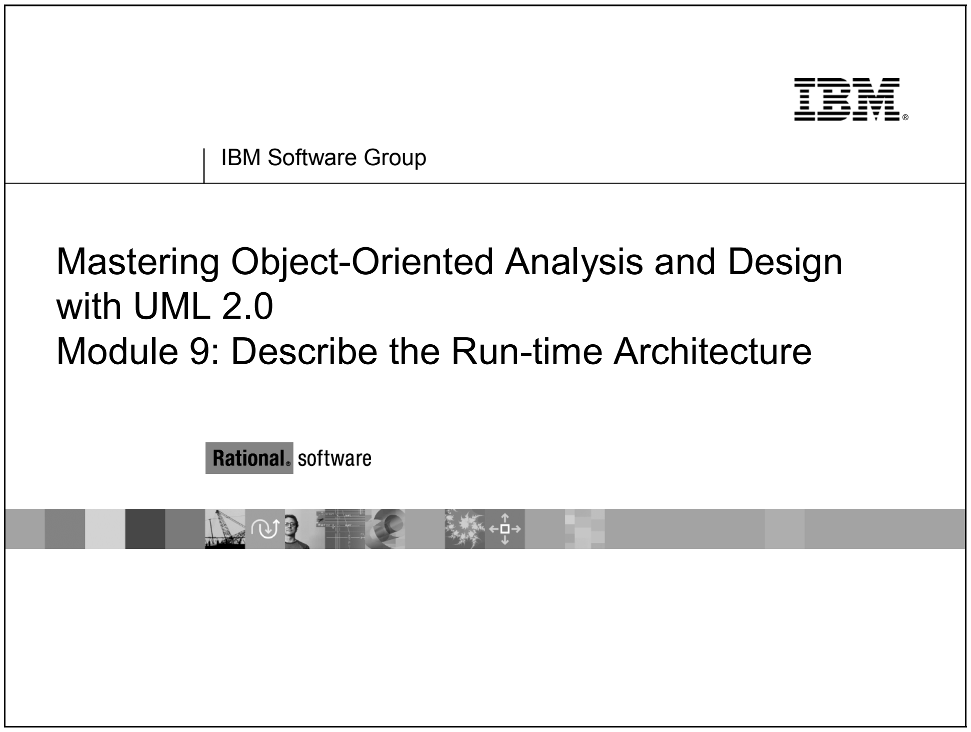
Module 9
Describe the Run-Time Architecture



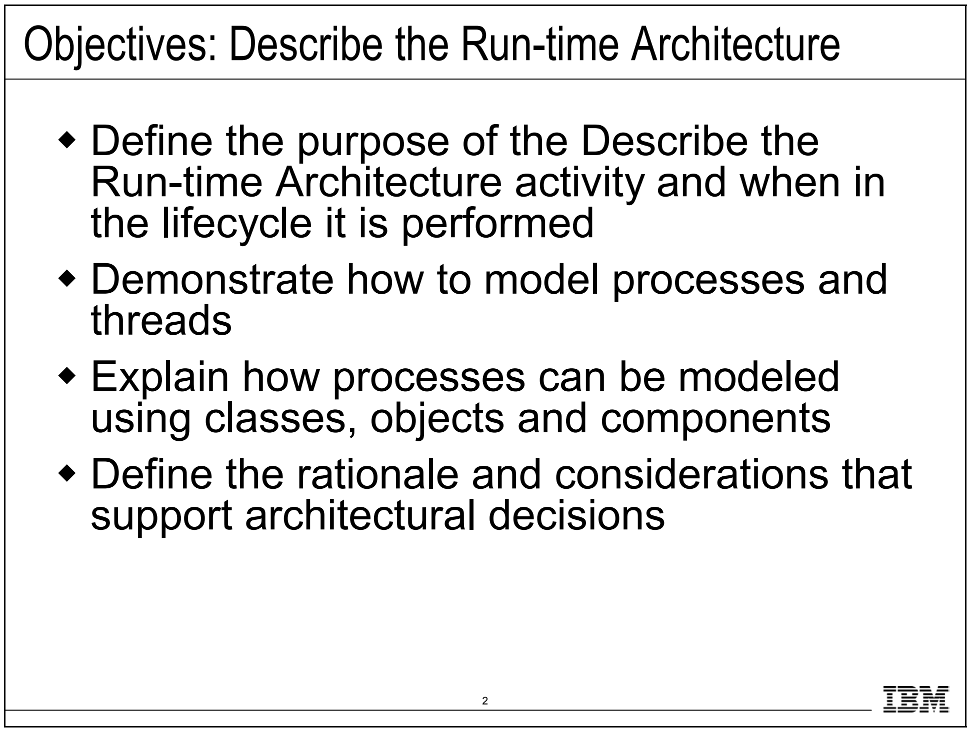
IBM Software Group

Mastering Object-Oriented Analysis and Design with UML 2.0
Module 9: Describe the Run-time Architecture

Topics

Describe the Run-time Architecture Overview 9-4
What Is Concurrency? 9-6
Identifying Processes and Threads 9-15
Creating and Destroying Processes and Threads 9-19
Mapping Processes onto the Implementation 9-22
Design Elements-to-Processes Strategies 9-26
Review 9-32

Objectives: Describe the Run-time Architecture



Objectives: Describe the Run-time Architecture

- Define the purpose of the Describe the Run-time Architecture activity and when in the lifecycle it is performed
- Demonstrate how to model processes and threads
- Explain how processes can be modeled using classes, objects and components
- Define the rationale and considerations that support architectural decisions

2

The **Describe the Run-time Architecture** activity focuses on the identification and modeling of the independent system of control of flows (for example, process and threads) and the ways in which they communicate.

The focus of the **Describe the Run-time Architecture** activity is on developing the Process View of the architecture.

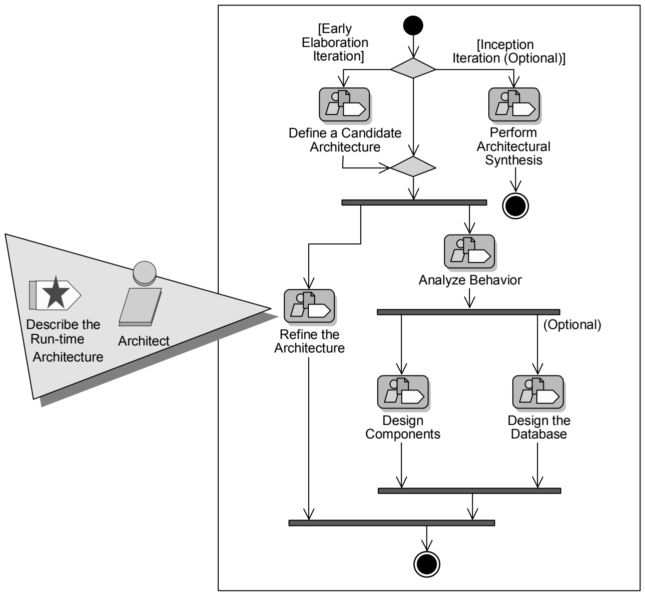
In this module, you will describe *what* is performed in **Describe the Run-time Architecture** but will not describe *how* to do it. Such a discussion is of interest in an architecture course, which this course is not.

The goal of this module is to give the student an understanding of how to model the Process View using the UML.

A comprehension of the rationale and considerations that support the architectural decisions is needed in order to understand the architecture, which is the framework in which designs must be developed.

Describe the Run-time Architecture in Context

Describe the Run-time Architecture in Context


[Early Elaboration Iteration]
[Inception Iteration (Optional)]
Define a Candidate Architecture
Perform Architectural Synthesis
Analyze Behavior
Describe the Run-time Architecture
Architect
Refine the Architecture
(Optional)
Design Components
Design the Database

3

As you may recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process. **Describe the Run-time Architecture** is an activity in the Refine the Architecture workflow detail.

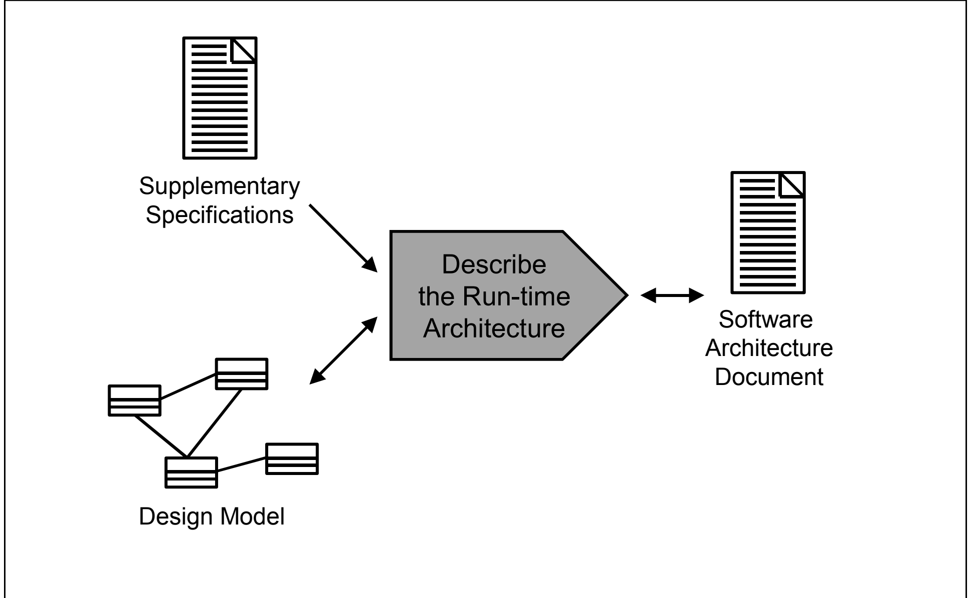
At this point, the subsystems, their interfaces, and their dependencies is normally defined. The initial design classes and the packages in which they belong have also been defined.

In **Describe the Run-time Architecture**, the independent threads of control are identified, and the design elements (subsystems and classes) are mapped to these threads of control. The focus is on the Process View of the architecture.

If the system under development needs only to run one process, then there is no need for a separate Process View. In such a case, **Describe the Run-time Architecture** can be skipped.

Describe the Run-time Architecture Overview

Describe the Run-time Architecture Overview


Supplementary Specifications
Describe the Run-time Architecture
Software Architecture Document
Design Model

4

The architect performs the **Describe the Run-time Architecture**, once per iteration.

Purpose

- To analyze concurrency requirements, to identify processes, identify inter-process communication mechanisms, allocate inter-process coordination resources, identify process lifecycles, and distribute model elements among processes.

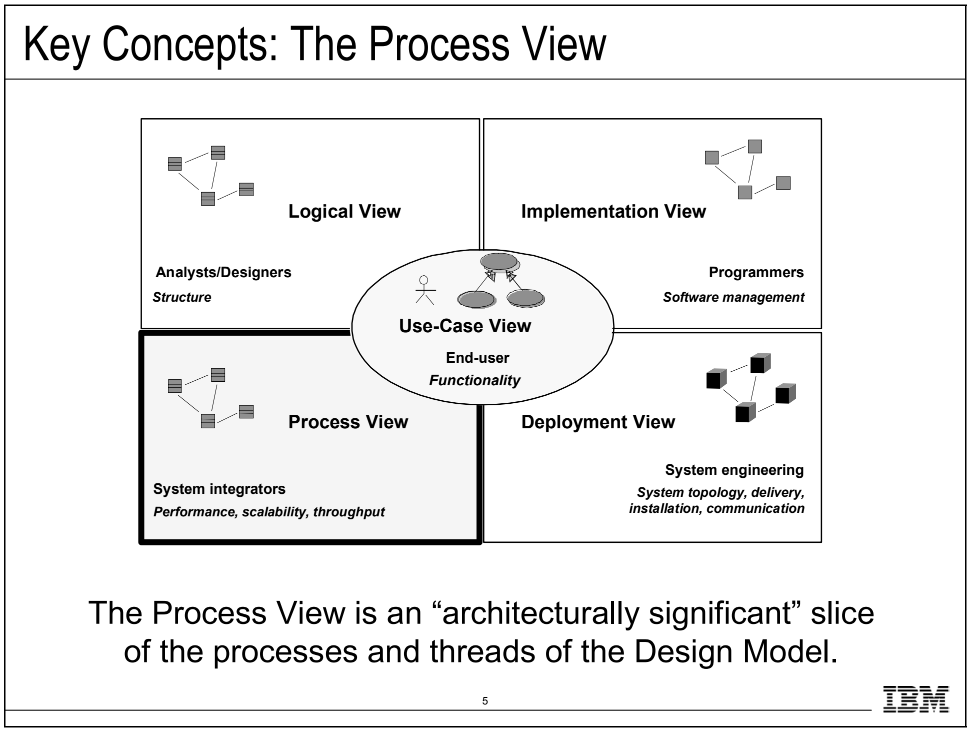
Input Artifacts

- Supplementary Specifications
- Design Model
- Software Architecture Document

Resulting Artifacts

- Software Architecture Document
- Design Model

Key Concepts: The Process View



Key Concepts: The Process View

The Process View is an “architecturally significant” slice of the processes and threads of the Design Model.

In **Describe the Run-time Architecture**, we will be concentrating on the Process View. Before we discuss the details of what occurs in **Describe the Run-time Architecture**, you need to review what the Process View is.

The above slide describes the model Rational uses to describe the software architecture. For each view, the stakeholder interested in it and the concern addressed in it are listed.

The Process View describes the planned process structure of the system. It is concerned with dynamic, run-time decomposition and takes into account some nonfunctional requirements, such as performance and availability. It also includes some derived requirements resulting from the need to spread the system onto several computers.

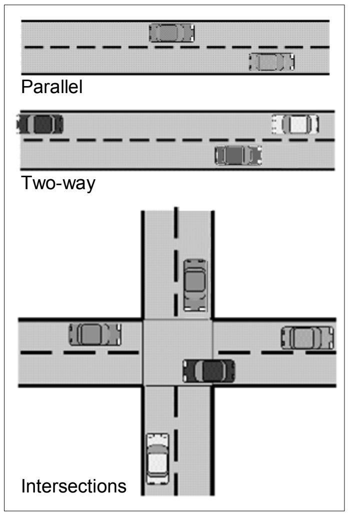
In the Process View, the system is decomposed into a set of independent tasks and threads, processes, and process groups.

The Process View describes process interaction, communication, and synchronization.

What Is Concurrency?

What Is Concurrency?

- Example of concurrency at work:
  - Parallel roads require little coordination
  - Two-way roads require some coordination for safe interaction
  - Intersections require careful coordination



6

Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon, of course. In the real world, at any given time many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency.

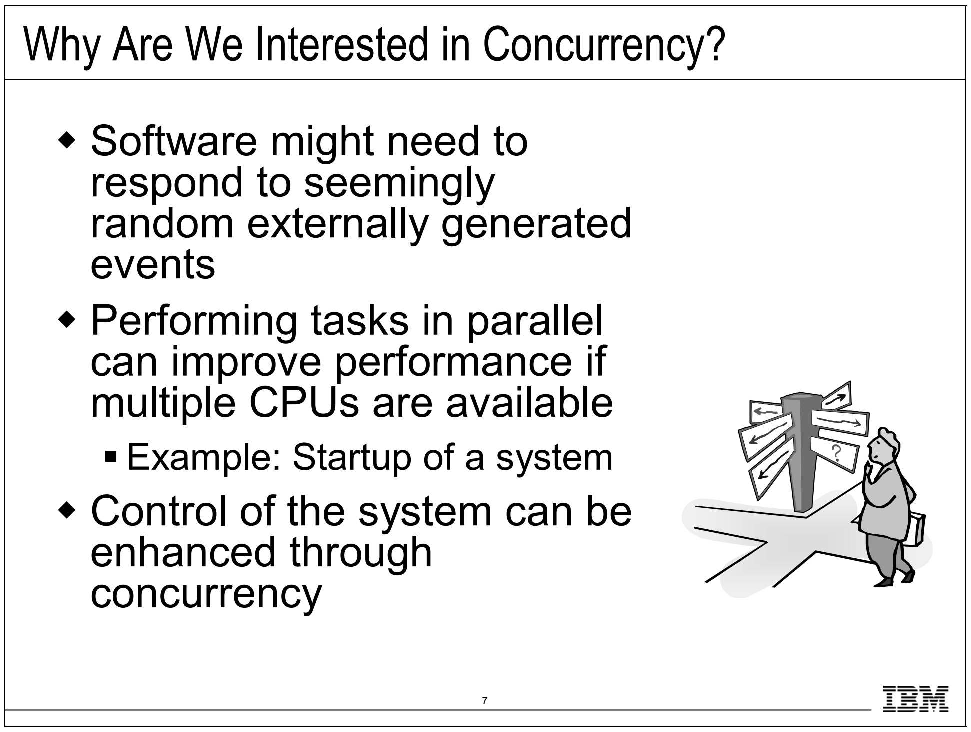
When dealing with concurrency issues in software systems, you must consider two important aspects:

- Being able to detect and respond to external events occurring in a random order.
- Ensuring that these events are responded to in some minimum required interval.

If each concurrent activity evolved independently, in a truly parallel fashion, managing them would be relatively simple: we could just create separate programs to deal with each activity. However, this is not the case. The challenges of designing concurrent systems arise mainly because of the interactions that happen between concurrent activities. When concurrent activities interact, some sort of coordination is required.

Vehicular traffic provides a useful analogy. Parallel traffic streams on different roadways having little interaction cause few problems. Parallel streams in adjacent lanes require some coordination for safe interaction, but a much more severe type of interaction occurs at an intersection, where careful coordination is required.

Why Are We Interested in Concurrency?



Why Are We Interested in Concurrency?

- Software might need to respond to seemingly random externally generated events
- Performing tasks in parallel can improve performance if multiple CPUs are available
  - Example: Startup of a system
- Control of the system can be enhanced through concurrency

7

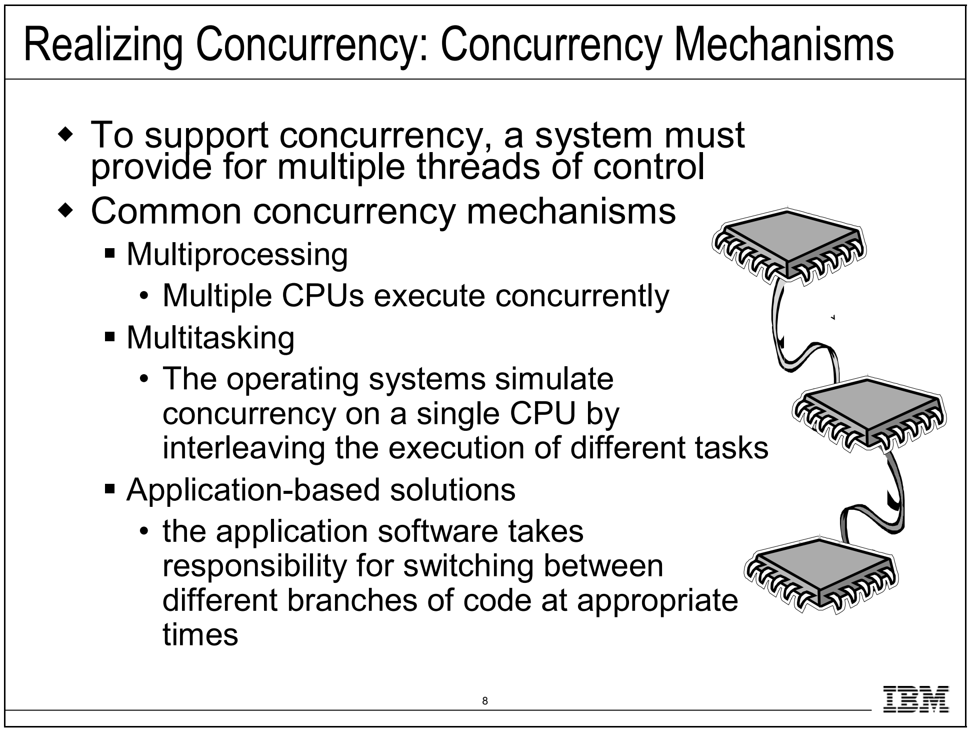
Some of the driving forces behind finding ways to manage concurrency are external. That is, they are imposed by the demands of the environment. In real-world systems, many things are happening simultaneously and must be addressed “in real-time” by software. To do so, many real time software systems must be “reactive.” They must respond to externally generated events that might occur at somewhat random times, in somewhat random order, or both.

Designing a conventional procedural program to deal with these situations is extremely complex. It can be much simpler to partition the system into concurrent software elements to deal with each of these events. The key phrase here is “can be,” since complexity is also affected by the degree of interaction between the events.

There also can be internally inspired reasons for concurrency. For example, performing tasks in parallel can substantially speed up the computational work of a system if multiple CPUs are available. Even within a single processor, multitasking can dramatically speed things up by preventing one activity from blocking another while waiting for I/O. A common situation in which this occurs is during the startup of a system. There are often many components, each of which requires time to be made ready for operation. Performing these operations sequentially can be painfully slow.

Controllability of the system can also be enhanced by concurrency. For example, one function can be started, stopped, or otherwise influenced in midstream by other concurrent functions — something extremely difficult to accomplish without concurrent components.

Realizing Concurrency: Concurrency Mechanisms



Realizing Concurrency: Concurrency Mechanisms

- To support concurrency, a system must provide for multiple threads of control
- Common concurrency mechanisms
  - Multiprocessing
    - Multiple CPUs execute concurrently
  - Multitasking
    - The operating systems simulate concurrency on a single CPU by interleaving the execution of different tasks
  - Application-based solutions
    - the application software takes responsibility for switching between different branches of code at appropriate times

8

When the operating system provides multitasking, a common unit of concurrency is the process. A process is an entity provided, supported, and managed by the operating system whose sole purpose is to provide an environment in which to execute a program. The process provides a memory space for the exclusive use of its application program, a thread of execution for executing it, and perhaps some means for sending messages to and receiving them from other processes. In effect, the process is a virtual CPU for executing a concurrent piece of an application.

Many operating systems, particularly those used for real-time applications, offer a "lighter weight" alternative to processes, called "threads" or "lightweight threads."

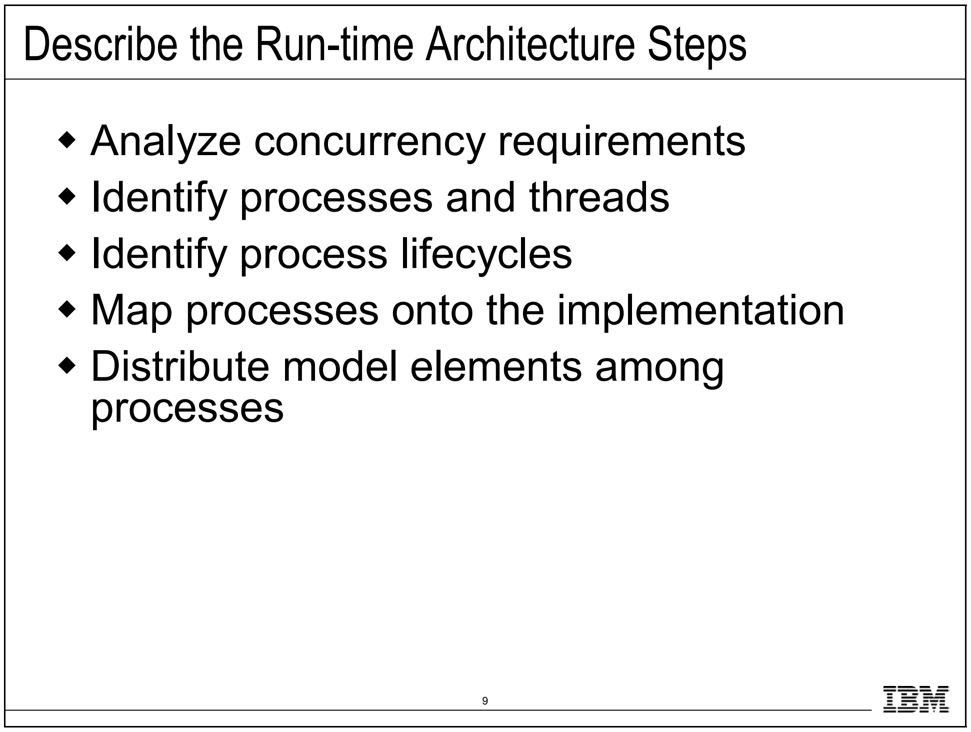
Threads are a way of achieving a slightly finer granularity of concurrency within a process. Each thread belongs to a single process, and all the threads in a process share the single memory space and other resources controlled by that process.

Usually each thread is assigned a procedure to execute.

Of course, multiple processors offer the opportunity for truly concurrent execution. Most commonly, each task is permanently assigned to a process in a particular processor, but under some circumstances tasks can be dynamically assigned to the next available processor. Perhaps the most accessible way of doing this is by using a "symmetric multiprocessor." In such a hardware configuration, multiple CPUs can access memory through a common bus.

Operating systems that support symmetric multiprocessors can dynamically assign threads to any available CPU. Examples of operating systems that support symmetric multiprocessors are SUN's Solaris and Microsoft's Windows NT.

Describe the Run-time Architecture Steps



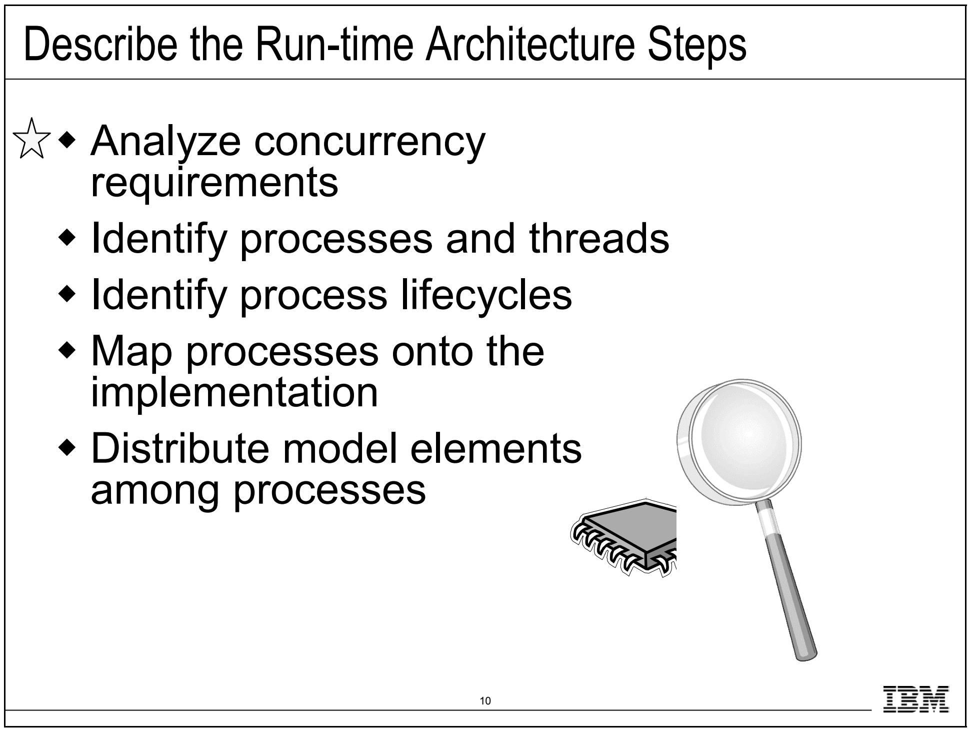
Describe the Run-time Architecture Steps

- Analyze concurrency requirements
- Identify processes and threads
- Identify process lifecycles
- Map processes onto the implementation
- Distribute model elements among processes

9

You will discuss the above topics in the **Describe the Run-time Architecture** module. Unlike the designer activity modules, you will not discuss each step of the activity, since the objective of this module is to understand the important concurrency concepts, not to learn *how* to design the concurrency aspects of the architecture.

Describe the Run-time Architecture Steps

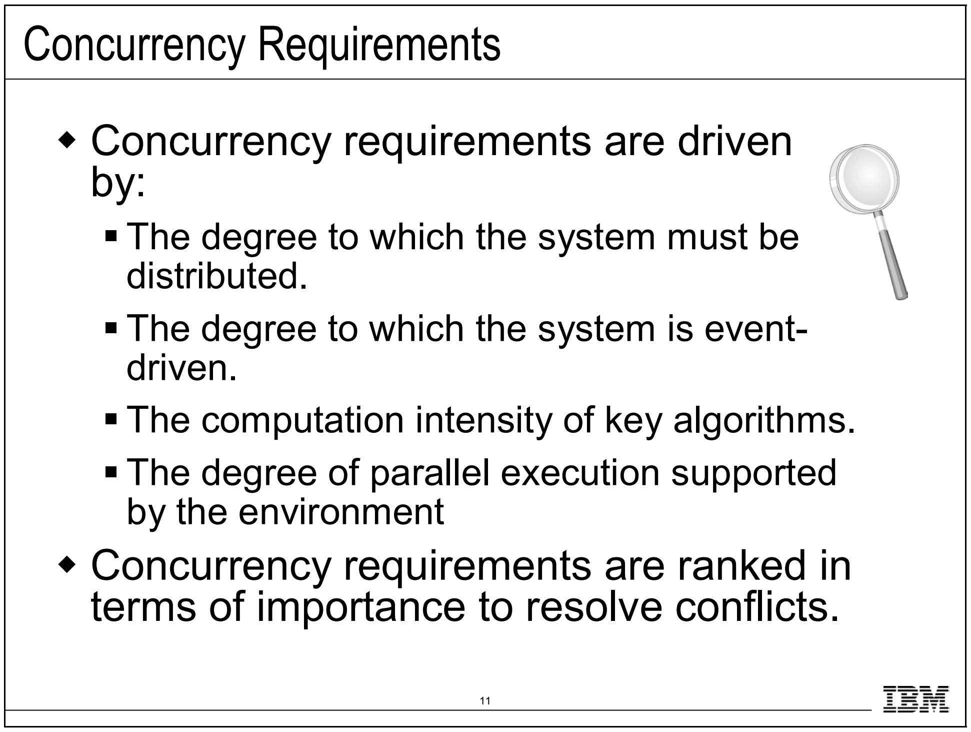


Describe the Run-time Architecture Steps

- ☆ Analyze concurrency requirements
- Identify processes and threads
- Identify process lifecycles
- Map processes onto the implementation
- Distribute model elements among processes

10

Concurrency Requirements



Concurrency Requirements

- Concurrency requirements are driven by:
  - The degree to which the system must be distributed.
  - The degree to which the system is event-driven.
  - The computation intensity of key algorithms.
  - The degree of parallel execution supported by the environment
- Concurrency requirements are ranked in terms of importance to resolve conflicts.

11

Concurrency requirements define the extent to which parallel execution of tasks is required for the system. These requirements help shape the architecture.

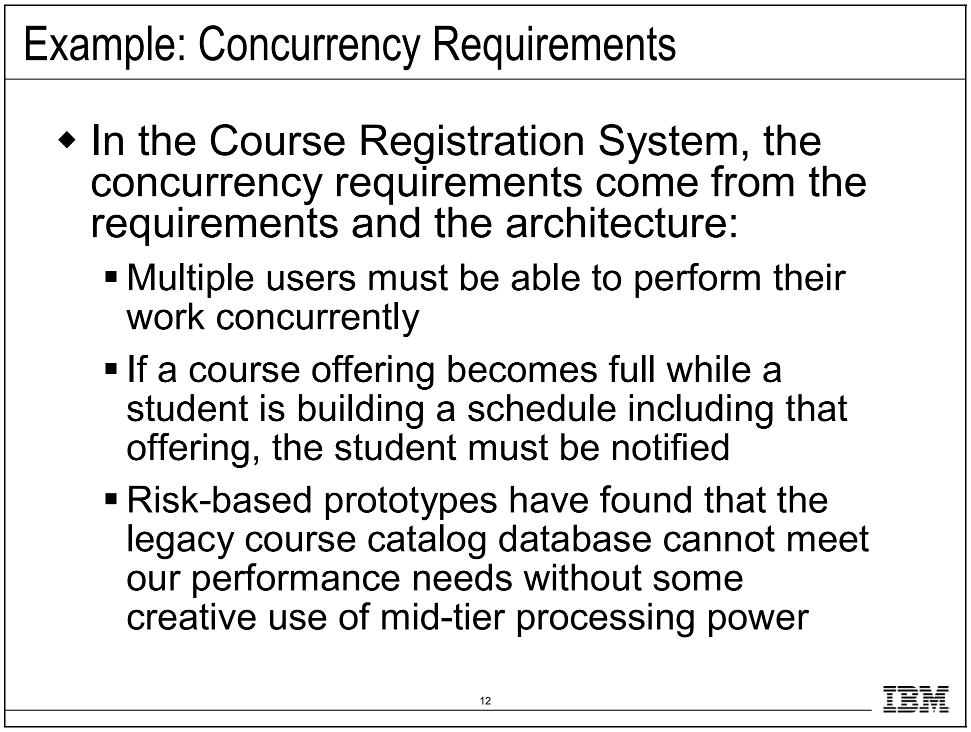
A system whose behavior must be distributed across processors or nodes virtually requires a multi-process architecture. A system that uses some sort of Database Management System or Transaction Manager also must consider the processes that those major subsystems introduce.

If dedicated processors are available to handle events, a multi-process architecture is probably best. On the other hand, to ensure that events are handled, a uni-process architecture may be needed to circumvent the “fairness” resource-sharing algorithm of the operating system: It may be necessary for the application to monopolize resources by creating a single large process, using threads to control execution within that process.

In order to provide good response times, it might be necessary to place computationally intensive activities in a process or thread of their own so that the system still is able to respond to user inputs while computation takes place, albeit with fewer resources. If the operating system or environment does not support threads (lightweight processes), there is little point in considering their impact on the system architecture.

The above requirements are mutually exclusive and might conflict with one another. Ranking requirements in terms of importance will help resolve the conflict.

Example: Concurrency Requirements



Example: Concurrency Requirements

- In the Course Registration System, the concurrency requirements come from the requirements and the architecture:
  - Multiple users must be able to perform their work concurrently
  - If a course offering becomes full while a student is building a schedule including that offering, the student must be notified
  - Risk-based prototypes have found that the legacy course catalog database cannot meet our performance needs without some creative use of mid-tier processing power

12

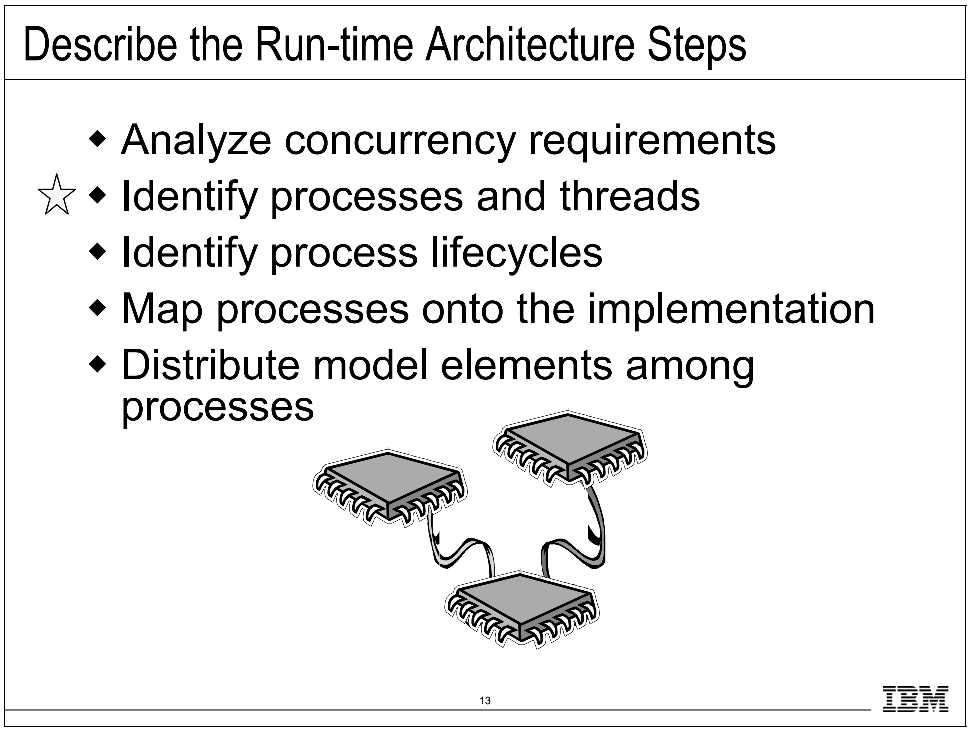
The above concurrency requirements were documented in the Course Registration System Supplemental Specification (see the Course Registration Requirements Document).

The first requirement is typical of any system, but the multi-tier aspects of our planned architecture will require some extra thought for this requirement.

The second requirement demonstrates the need for a shared, independent process that manages access to the course offerings.

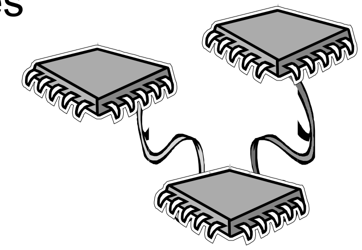
The third issue leads us to use some sort of mid-tier caching or preemptive retrieval strategy.

Describe the Run-time Architecture Steps



Describe the Run-time Architecture Steps

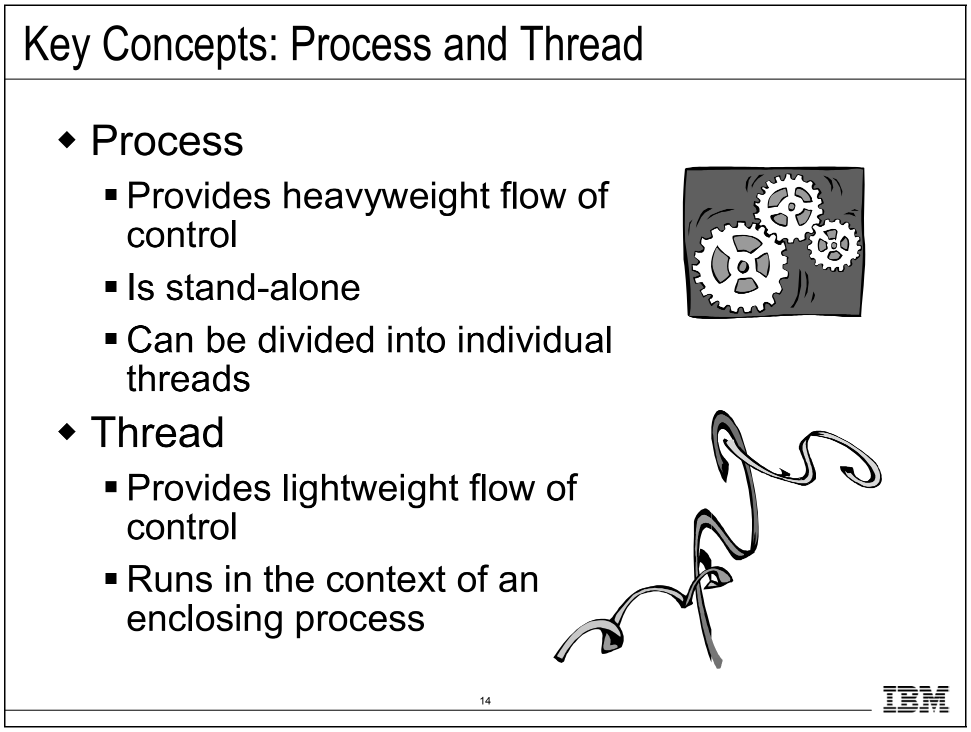
- Analyze concurrency requirements
- ☆ Identify processes and threads
- Identify process lifecycles
- Map processes onto the implementation
- Distribute model elements among processes



13

We have discussed what concurrency requirements are and how they drive the identification of the independent threads of control (for example, processes and threads) that will exist in the system. Now we will learn how to model those threads of control.

Key Concepts: Process and Thread



Key Concepts: Process and Thread

- Process
  - Provides heavyweight flow of control
  - Is stand-alone
  - Can be divided into individual threads
- Thread
  - Provides lightweight flow of control
  - Runs in the context of an enclosing process

14

Process: A unique address space and execution environment in which instances of classes and subsystems reside and run. The execution environment can be divided into one or more threads of control.

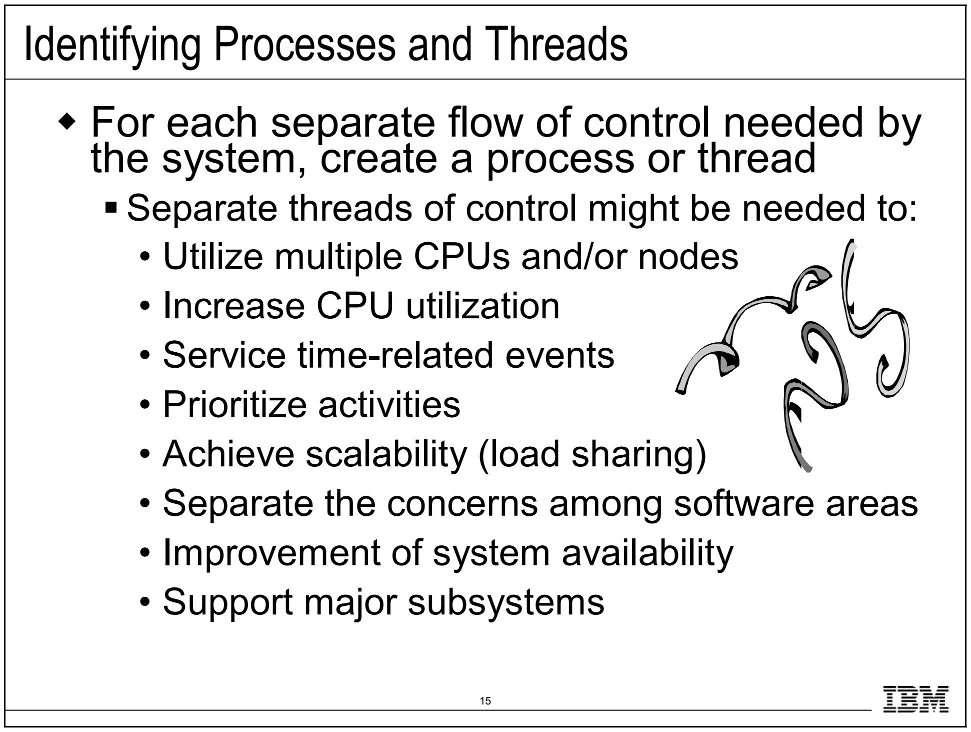
Thread: An independent computation executing within the execution environment and address space defined by an enclosing process.

From *UML Toolkit* by Hans-Erik Eriksson and Magnus Penker —

The difference between process and thread has to do with the memory space in which they execute:

- A process executes in its own memory space and encapsulates and protects its internal structure. A process can be viewed as being a system of its own. It is initiated by an executable program. A process can contain multiple threads (that is, a number of processes can execute within a single process, sharing the same memory space).
- A thread executes in a memory space that it can share with other threads.

Identifying Processes and Threads



Identifying Processes and Threads

- For each separate flow of control needed by the system, create a process or thread
  - Separate threads of control might be needed to:
    - Utilize multiple CPUs and/or nodes
    - Increase CPU utilization
    - Service time-related events
    - Prioritize activities
    - Achieve scalability (load sharing)
    - Separate the concerns among software areas
    - Improvement of system availability
    - Support major subsystems

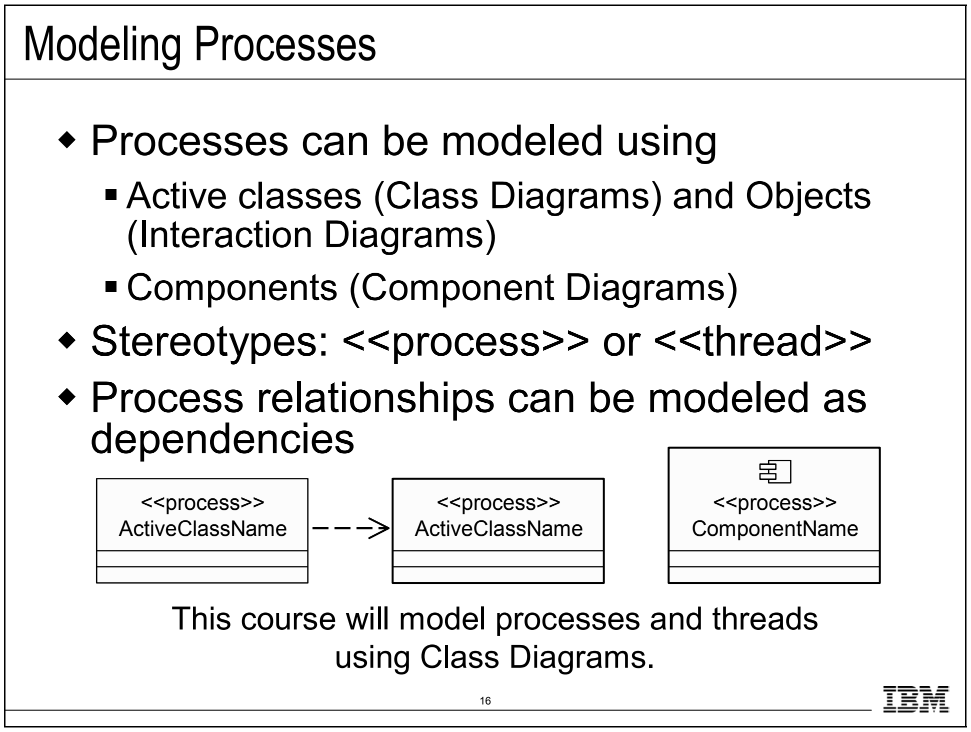
15

For each separate flow of control needed by the system, create a process or a thread (lightweight process). A thread should be used in cases where there is a need for nested flow of control. (Within a process, there is a need for independent flow of control at the subtask level.)

For example, we can say (not necessarily in order of importance) that separate threads of control might be needed to:

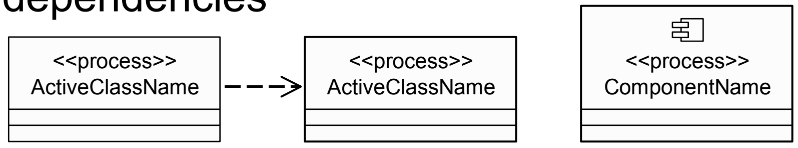
- **Utilize multiple CPUs:** There may be multiple CPUs in a node or multiple nodes in a distributed system.
- **Increase CPU utilization:** Processes can be used to increase CPU utilization by allocating cycles to other activities when a thread of control is suspended.
- **Service time-related events:** For example, timeouts, scheduled activities, periodic activities.
- **Prioritize activities:** Separate processes allow functionality in different processes to be prioritized individually.
- **Achieve scalability:** Load sharing across several processes and processors.
- **Separation of concerns:** Separating concerns between different areas of the software, such as safety.
- **Improve system availability:** Higher system availability from backup and redundant processes.
- **Support major subsystems:** Some major subsystems might require separate processes (for example, the DBMS, and Transaction Manager).

Modeling Processes



Modeling Processes

- Processes can be modeled using
  - Active classes (Class Diagrams) and Objects (Interaction Diagrams)
  - Components (Component Diagrams)
- Stereotypes: <<process>> or <<thread>>
- Process relationships can be modeled as dependencies



This course will model processes and threads using Class Diagrams.

16

You can use “active” classes to model processes and threads. An active class is a class that “owns” its own thread of execution and can initiate control activity, contrasted with passive classes that can only be acted upon. Active classes can execute in parallel (that is, concurrently) with other active classes.

The model elements can be stereotyped to indicate whether they are processes (<<process>> stereotype) or threads (<<thread>> stereotype).

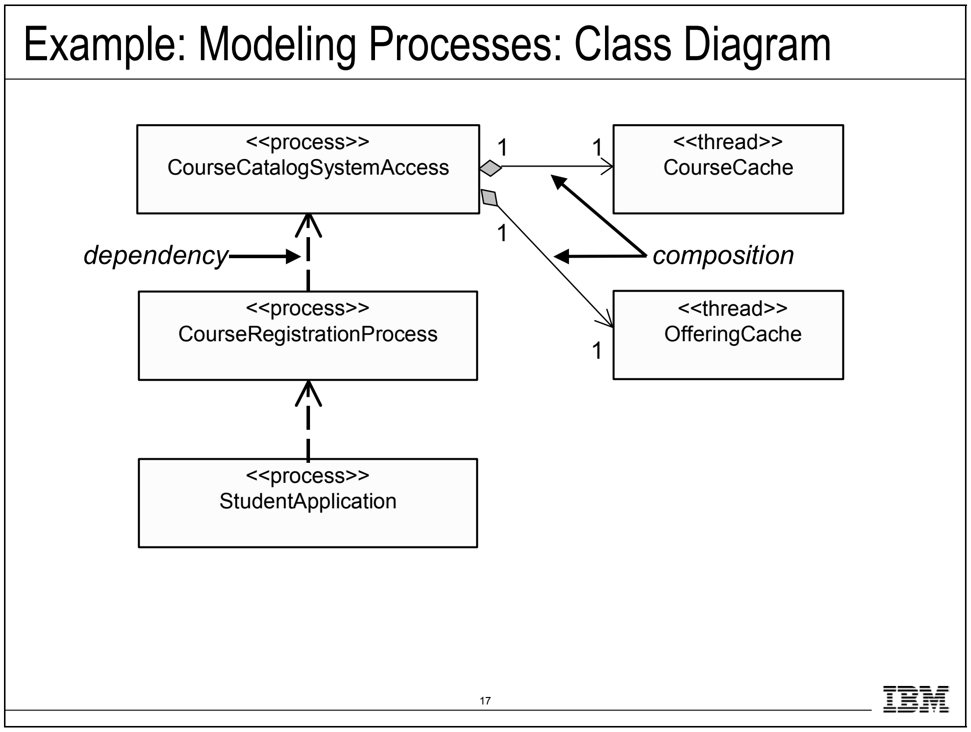
Note: Even though you use “active” classes to model processes and threads, they are classes only in the meta-modeling sense. They aren’t the same kind of model elements as classes. They are only meta-modeling elements used to provide an address space and a run-time environment in which other class instances execute, as well as to document the process structure. If you try to take them further than that, confusion may result.

Process communication is modeled using dependency relationship whether you use classes or components to represent your processes.

In cases where the application has only one process, the processes may never be explicitly modeled. As more processes or threads are added, modeling them becomes important.

The Class Diagram will be used to represent the Process View for the remainder of the course.

Example: Modeling Processes: Class Diagram



The above example demonstrates how processes and threads are modeled. Processes and threads are represented as stereotyped classes. Separate processes have dependencies among them. When there are threads within a process composition is used. The composition relationship indicates that the threads are contained within the process (that is, cannot exist outside of the process).

The StudentApplication process manages the student functionality, including user interface processing and coordination with the business processes. There is one instance of this process for each student who is currently registering for courses.

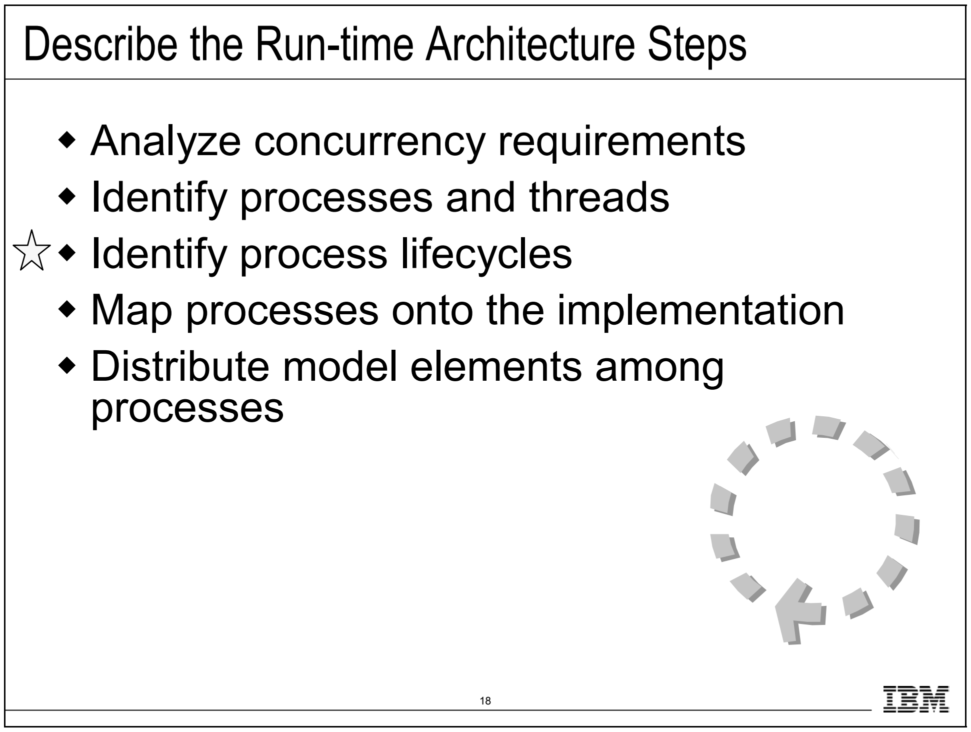
The CourseRegistrationProcess encapsulates the course registration processing. There is one instance of this process for each student who is currently registering for courses.

The CourseRegistrationProcess talks to the separate CourseCatalogSystemAccess process, which manages access to the legacy system. CourseCatalogSystemAccess is a separate process that can be shared by multiple users registering for courses. This allows for a cache of recently retrieved courses and offerings to improve performance.

The separate threads within the CourseCatalogSystemAccess process, CourseCache, and OfferingCache are used to asynchronously retrieve items from the legacy system. This improves response time.

The above example is a subset of the Process View of the Course Registration system.

Describe the Run-time Architecture Steps



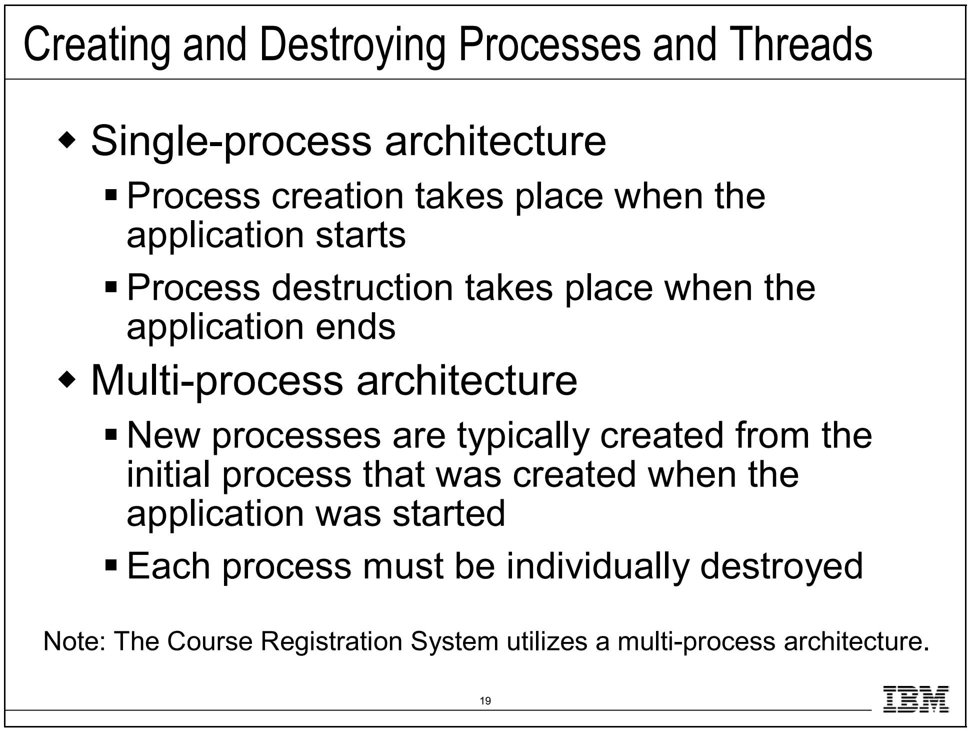
Describe the Run-time Architecture Steps

- Analyze concurrency requirements
- Identify processes and threads
- ☆ Identify process lifecycles
- Map processes onto the implementation
- Distribute model elements among processes

18

Now that we have identified processes and threads, we must determine when those processes and threads are created and destroyed.

Creating and Destroying Processes and Threads



Creating and Destroying Processes and Threads

- Single-process architecture
  - Process creation takes place when the application starts
  - Process destruction takes place when the application ends
- Multi-process architecture
  - New processes are typically created from the initial process that was created when the application was started
  - Each process must be individually destroyed

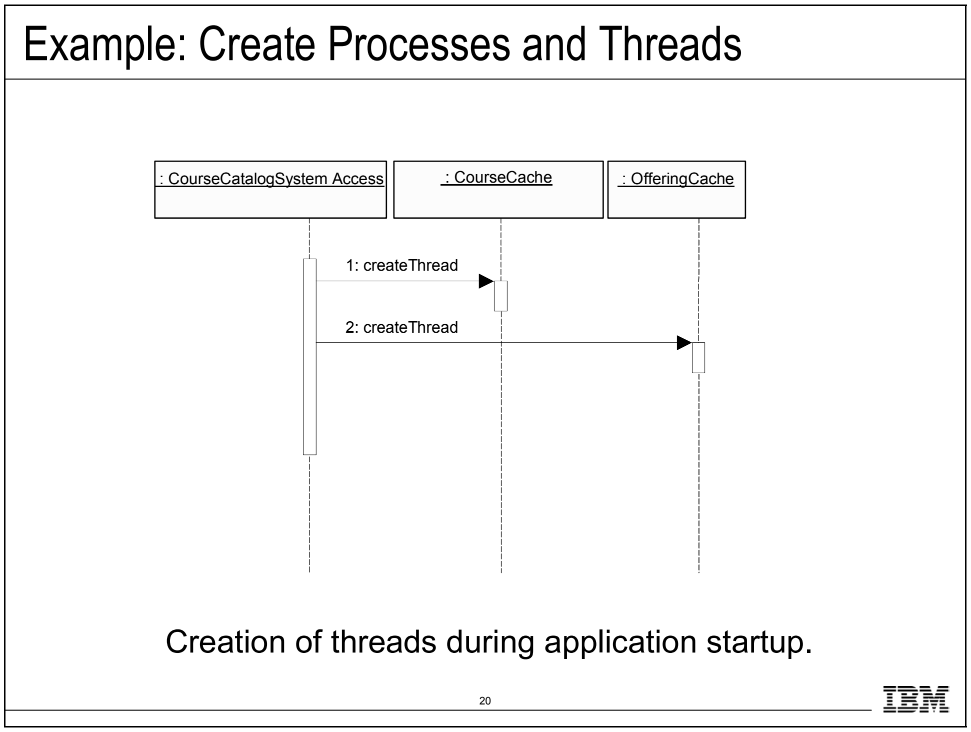
Note: The Course Registration System utilizes a multi-process architecture.

19

Each process or thread of control must be created and destroyed. In a single-process architecture, process creation occurs when the application is started, and process destruction occurs when the application ends. In multi-process architectures, new processes (or threads) are typically spawned or forked from the initial process created by the operating system when the application is started. These processes must be explicitly destroyed as well.

The sequence of events leading up to process creation and destruction must be determined and documented, as well as the mechanism for creation and deletion.

Example: Create Processes and Threads



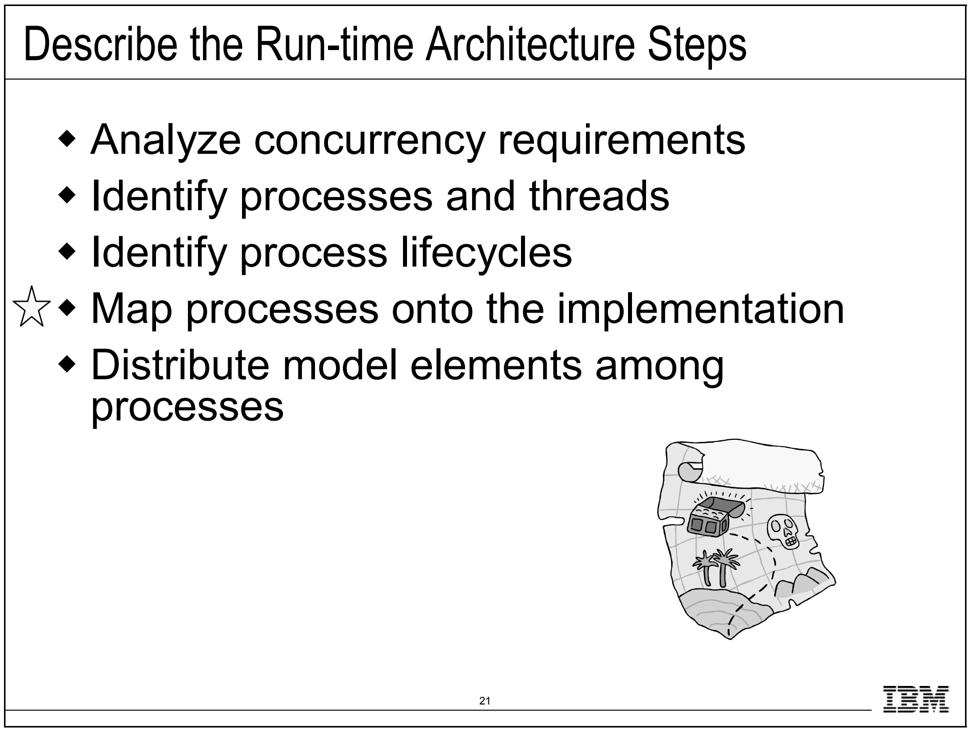
Example: Create Processes and Threads

Creation of threads during application startup.

20

In the Course Registration System, a main process is started which is responsible for coordinating the behavior of the entire system. It in turn spawns a number of subordinate threads of control to monitor various parts of the system — the devices in the system and events emanating from the Course Catalog System. The creation of these processes and threads can be shown with **classes** in UML, and the creation of instances of these active classes can be shown in a sequence diagram, as shown above.

Describe the Run-time Architecture Steps



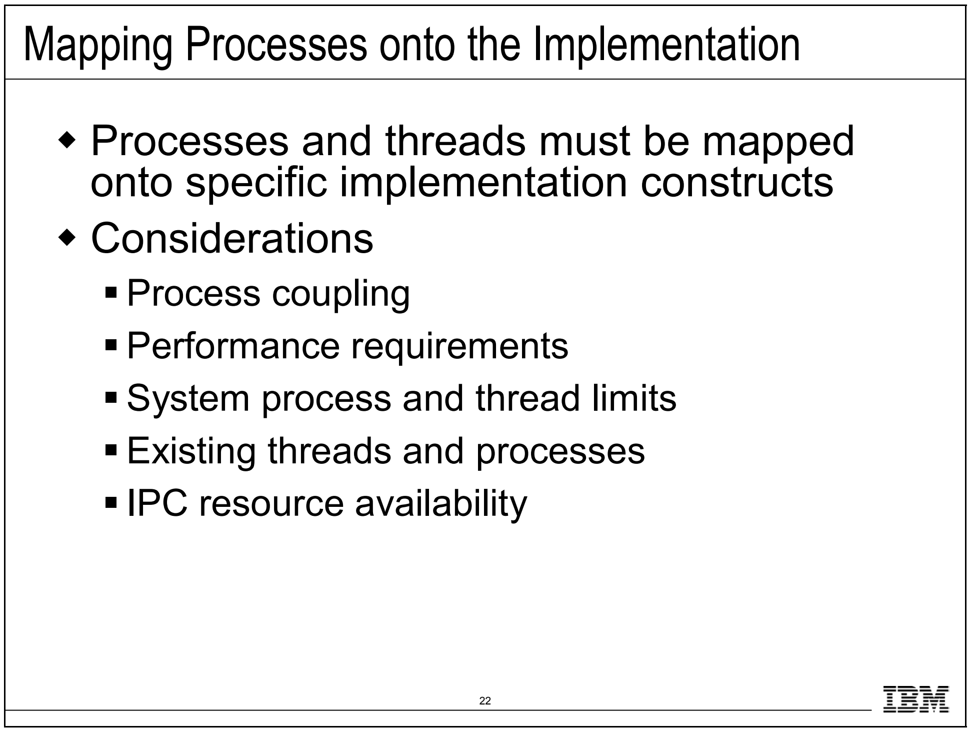
Describe the Run-time Architecture Steps

- Analyze concurrency requirements
- Identify processes and threads
- Identify process lifecycles
- ☆ Map processes onto the implementation
- Distribute model elements among processes

21

At this point, we have defined the flows of control (for example, processes and threads). Now we need to map these "flows of control" onto the concepts supported by the implementation environment.

Mapping Processes onto the Implementation



Mapping Processes onto the Implementation

- Processes and threads must be mapped onto specific implementation constructs
- Considerations
  - Process coupling
  - Performance requirements
  - System process and thread limits
  - Existing threads and processes
  - IPC resource availability

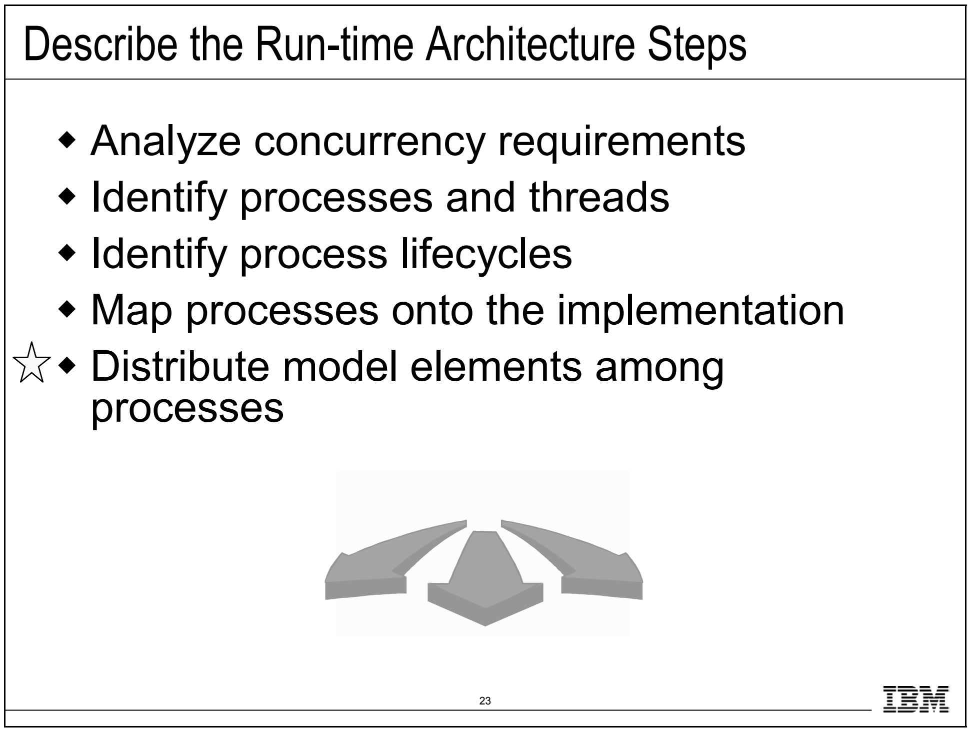
22

Conceptual processes must be mapped onto specific constructs in the operating environment. In many environments, there are choices of types of processes, at the very least process and threads. The choices are based on the degree of coupling (processes are stand-alone, whereas threads run in the context of an enclosing process) and the performance requirements of the system (intra-process communication between threads is generally faster and more efficient than that among processes).

In many systems, there may be a maximum number of threads per process or processes per node. These limits might not be absolute, but might be practical limits imposed by the availability of scarce resources.

The threads and processes already running on a target node need to be considered, along with the threads and processes proposed in the process architecture.

Describe the Run-time Architecture Steps



Describe the Run-time Architecture Steps

- Analyze concurrency requirements
- Identify processes and threads
- Identify process lifecycles
- Map processes onto the implementation
- ☆ Distribute model elements among processes

23

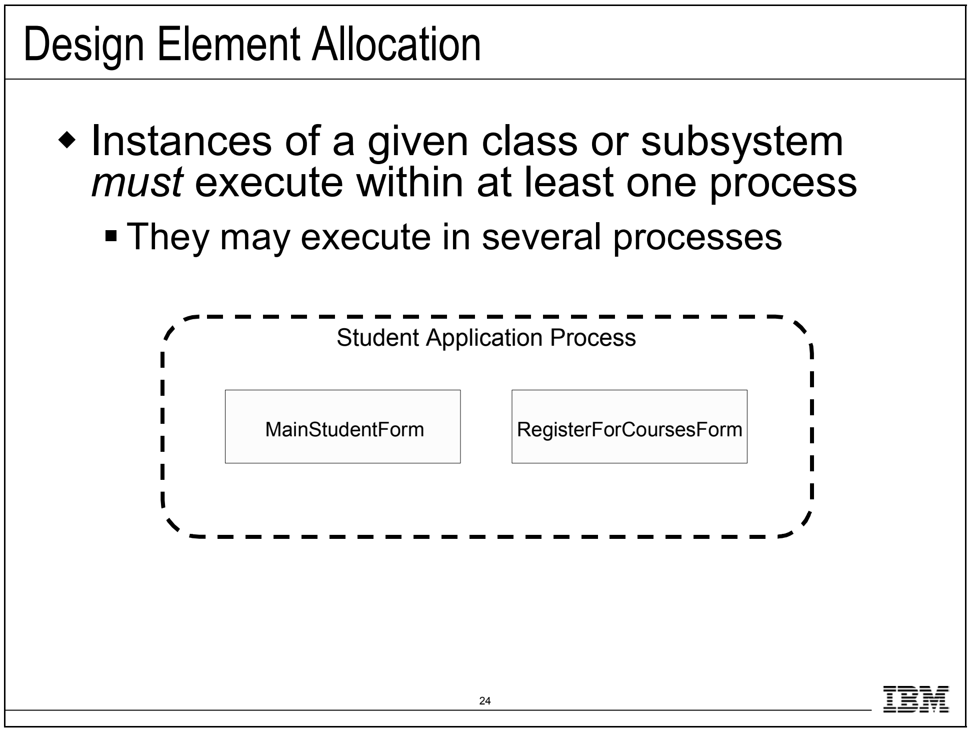
IBM

At this point, the processes that are to run in the implementation environment have been defined. Now it is time to determine the processes within which the identified classes and subsystems should execute.

Instances of a given design element (class or subsystem) must execute within at least one process; they might in fact execute in several different processes. The process provides an "execution environment" for the design element.

This step is where the “Design Model meet the Process View,” and where consistency is established and maintained between these two very important aspects of the system.

Design Element Allocation



Design Element Allocation

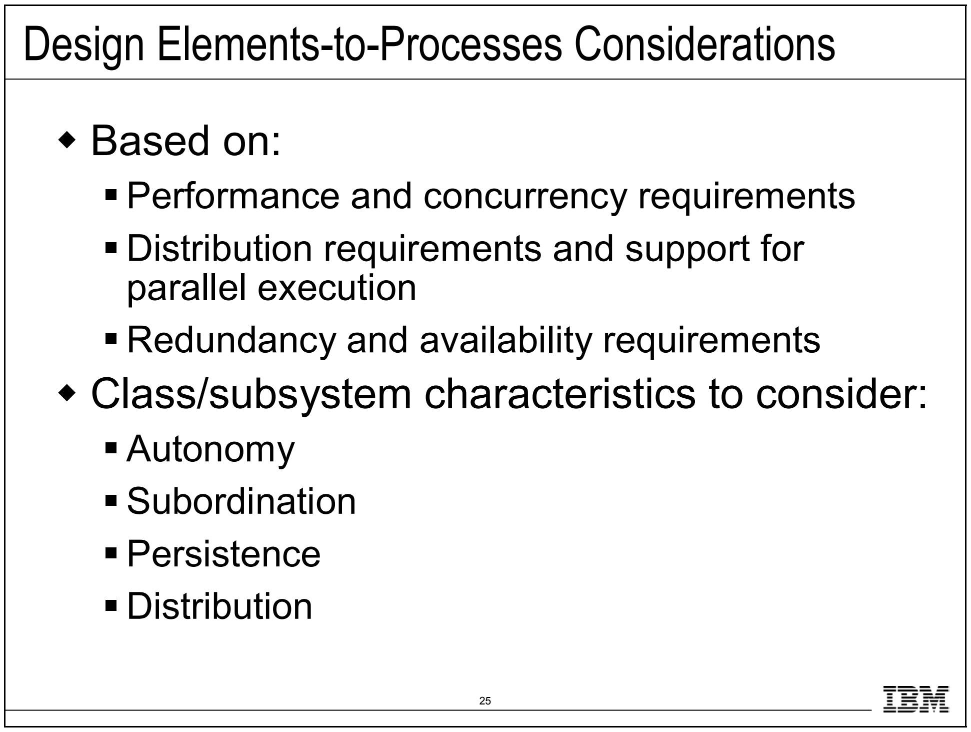
- Instances of a given class or subsystem *must* execute within at least one process
  - They may execute in several processes

Student Application Process
MainStudentForm
RegisterForCoursesForm
24

The purpose of this step is to determine the processes within which classes and subsystems should execute. Instances of a given class or subsystem must execute within at least *one* process; it might in fact execute in several different processes. The process provides an "execution environment" for the class or subsystem.

On this slide, the box represents the execution environment for the instances of the MainStudentForm and RegisterForCoursesForm.

Design Elements-to-Processes Considerations



Design Elements-to-Processes Considerations

- Based on:
  - Performance and concurrency requirements
  - Distribution requirements and support for parallel execution
  - Redundancy and availability requirements
- Class/subsystem characteristics to consider:
  - Autonomy
  - Subordination
  - Persistence
  - Distribution

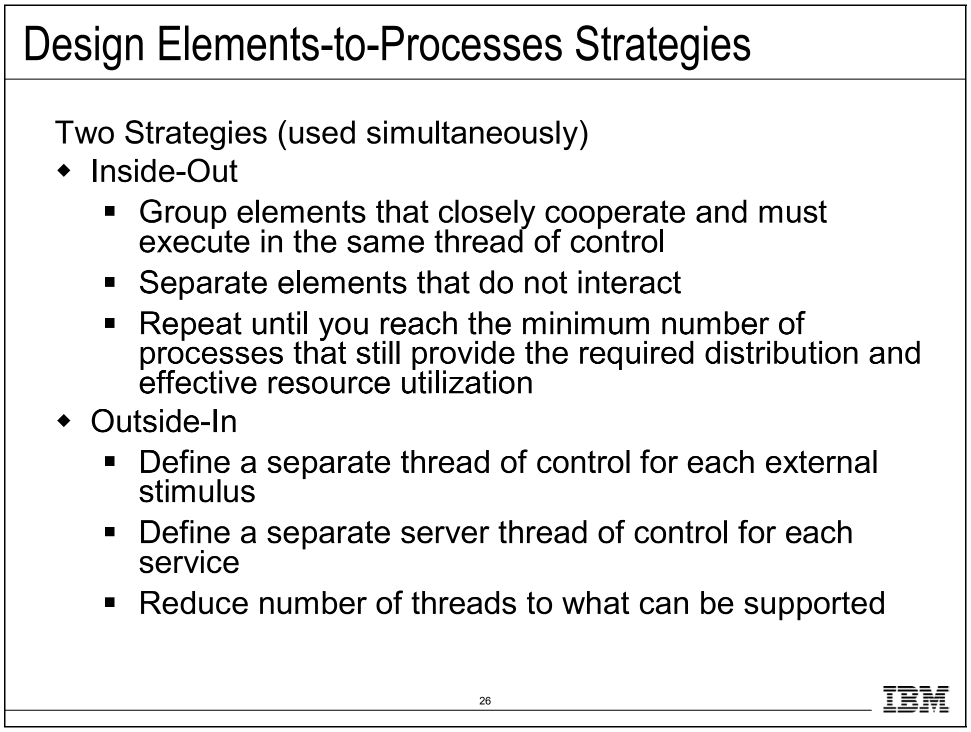
25

When deciding which classes and subsystems to map to which processes, there are some important characteristics to consider:

- Whether or not the class or subsystem is active, passive, or protected.
- What is its lifetime? Is it contained within, or does it contain, other classes/subsystems.
- Is it persistent?
- Should its state, operations, or both be distributed?

Each of these considerations affects how classes and subsystems are allocated to processes.

Design Elements-to-Processes Strategies



Design Elements-to-Processes Strategies

Two Strategies (used simultaneously)

- Inside-Out
  - Group elements that closely cooperate and must execute in the same thread of control
  - Separate elements that do not interact
  - Repeat until you reach the minimum number of processes that still provide the required distribution and effective resource utilization
- Outside-In
  - Define a separate thread of control for each external stimulus
  - Define a separate server thread of control for each service
  - Reduce number of threads to what can be supported

26

IBM

Classes and subsystems can be allocated to one or more processes and threads.

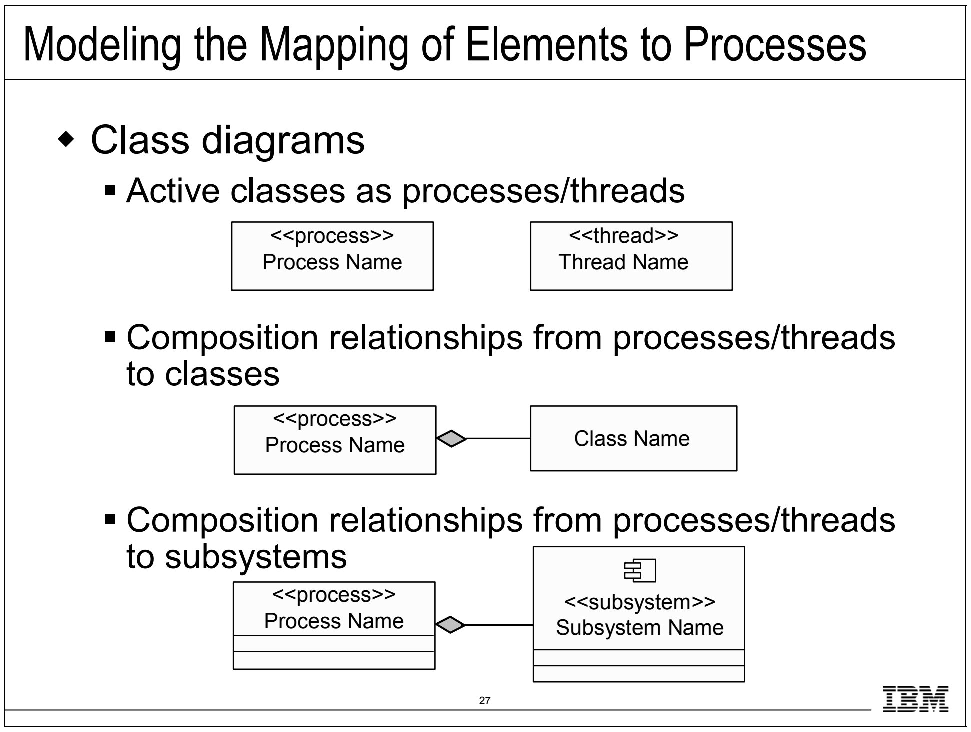
Inside-Out

- Group classes and subsystems together in sets of cooperating elements that (a) closely cooperate with one another and (b) need to execute in the same thread of control. Consider the impact of introducing inter-process communication into the middle of a message sequence before separating elements into separate threads of control.
- Conversely, separate classes and subsystems that do not interact at all, placing them in separate threads of control.
- This clustering proceeds until the number of processes has been reduced to the smallest number that still allows distribution and use of the physical resources.

Outside-In

- Identify external stimuli to which the system must respond. Define a separate thread of control to handle each stimuli and a separate server thread of control to provide each service.
- Consider the data integrity and serialization constraints and then reduce this initial set of threads of control to the number that can be supported by the execution environment.

Modeling the Mapping of Elements to Processes



Modeling the Mapping of Elements to Processes

- Class diagrams
  - Active classes as processes/threads
  - Composition relationships from processes/threads to classes
  - Composition relationships from processes/threads to subsystems

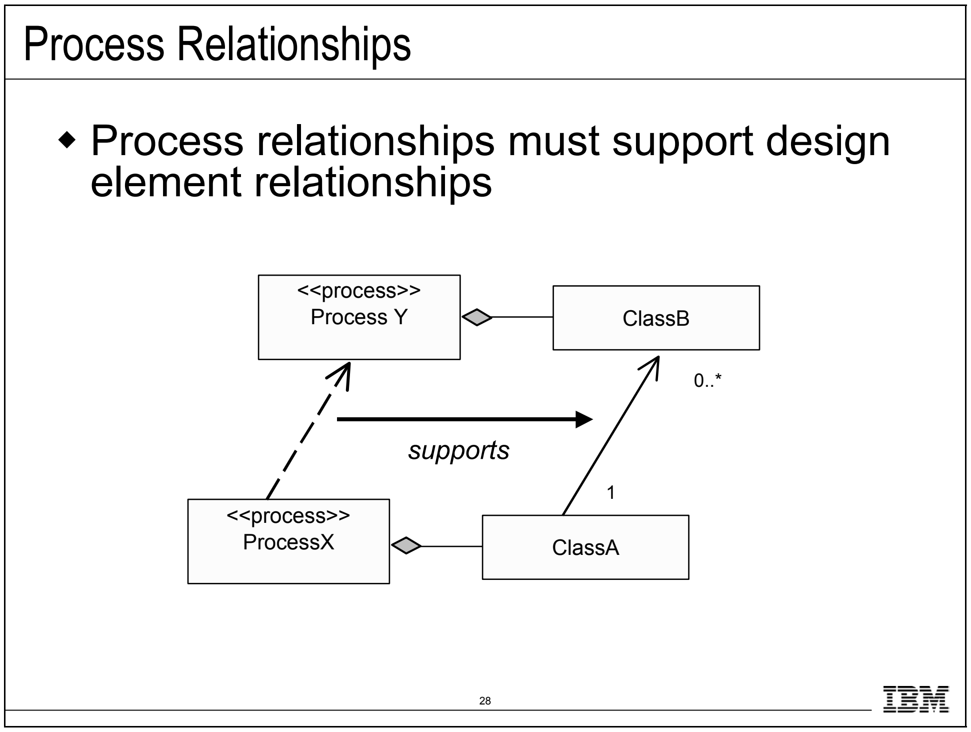
<<process>>
Process Name
<<thread>>
Thread Name
<<process>>
Process Name
Class Name
<<process>>
Process Name
<<subsystem>>
Subsystem Name
27
IBM

Processes and threads are composed of instances of design elements (that is, classes and subsystems). To show this allocation, class diagrams are drawn that model the processes and threads as active classes, and show the composition of the active classes (for example, composition relationships drawn from the process elements to the design elements that execute within it).

The relationship between a process (active class) and the elements it contains is *always* a composition (that is, aggregation-by-value) relationship, since processes contain instances of classes and subsystems.

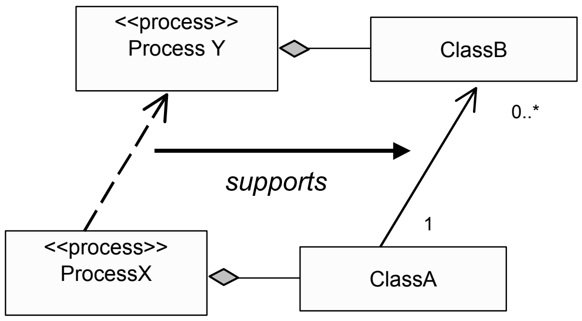
Note: It is only necessary to model the top-most design element that is mapped to the process or thread. You do not need to model all the design elements that the top-most element has relationships with unless they are in different threads.

Process Relationships



Process Relationships

- Process relationships must support design element relationships

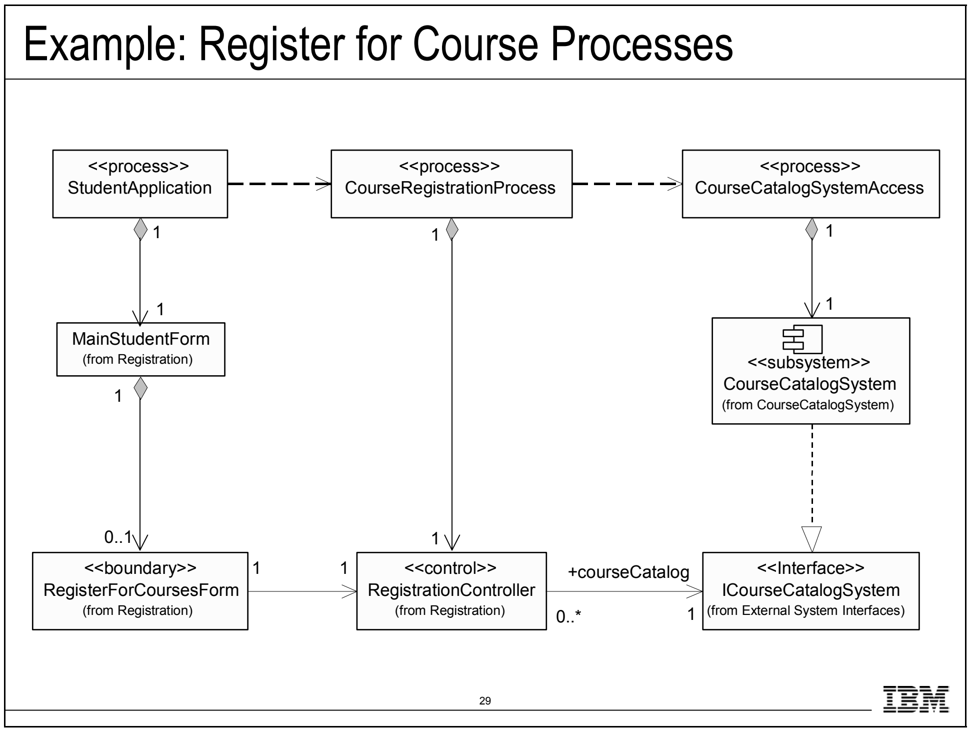


28

The process relationships can be derived from the class relationships. If two classes must communicate and they have been mapped to different processes, then there must be a relationship between the two processes.

Thus, the process relationships should be justified by the associated design element relationships.

Example: Register for Course Processes



Example: Register for Course Processes

<<process>> StudentApplication
<<process>> CourseRegistrationProcess
<<process>> CourseCatalogSystemAccess
1
1
1
1
MainStudentForm (from Registration)
1
<<subsystem>> CourseCatalogSystem (from CourseCatalogSystem)
0..1
1
<<boundary>> RegisterForCoursesForm (from Registration)
1
1
<<control>> RegistrationController (from Registration)
+courseCatalog
0..*
1
<<Interface>> ICourseCatalogSystem (from External System Interfaces)
29

The above slide demonstrates how to use composition to model the mapping of design elements to the processes on which they run.

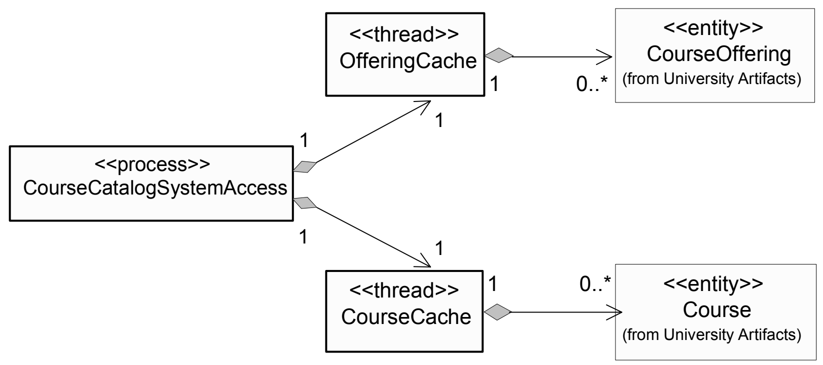
- The classes associated with the individual user interfaces have been mapped to the application process.
- The classes associated with the individual business services have been mapped to the controller process.
- The class associated with access to the legacy course catalog system has been mapped to the access process.

Notice how the process relationships (modeled using dependency relationships) support the relationships between the design elements mapped to the processes.

The above example is a subset of the Process View for the Course Registration System, specifically the processes that support the Register for Courses use-case realization.

Example: Register for Course Processes (continued)

Example: Register for Course Processes (continued)

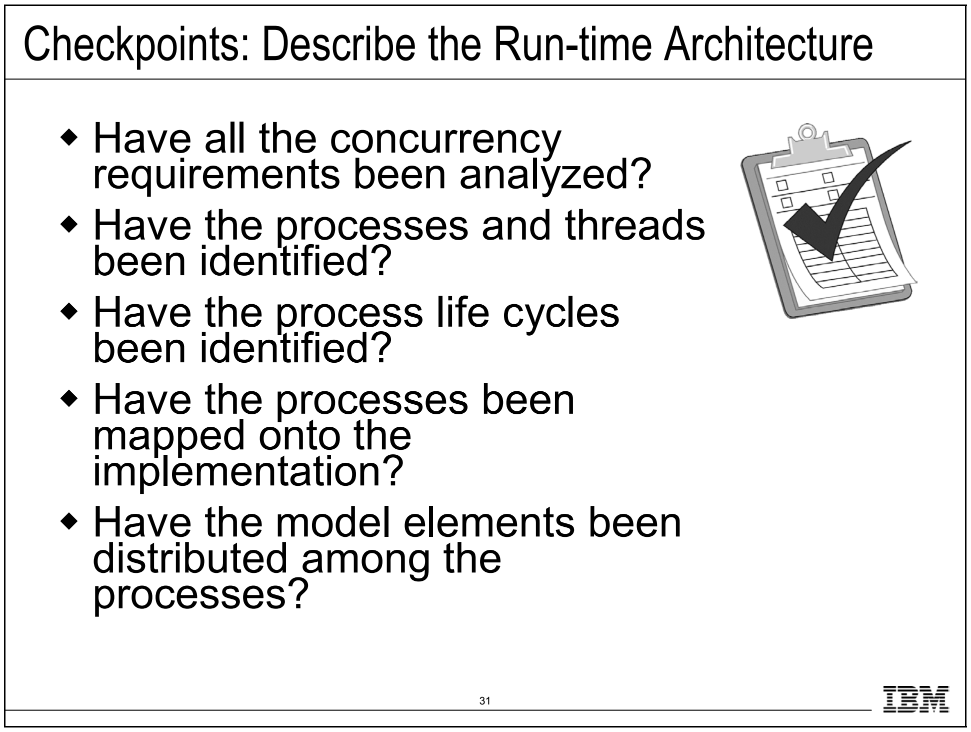


30

The above example demonstrates how to use composition to model the mapping of design elements to the threads on which they run.

Note the additional composition relationship from the process to the threads it contains.

Checkpoints: Describe the Run-time Architecture

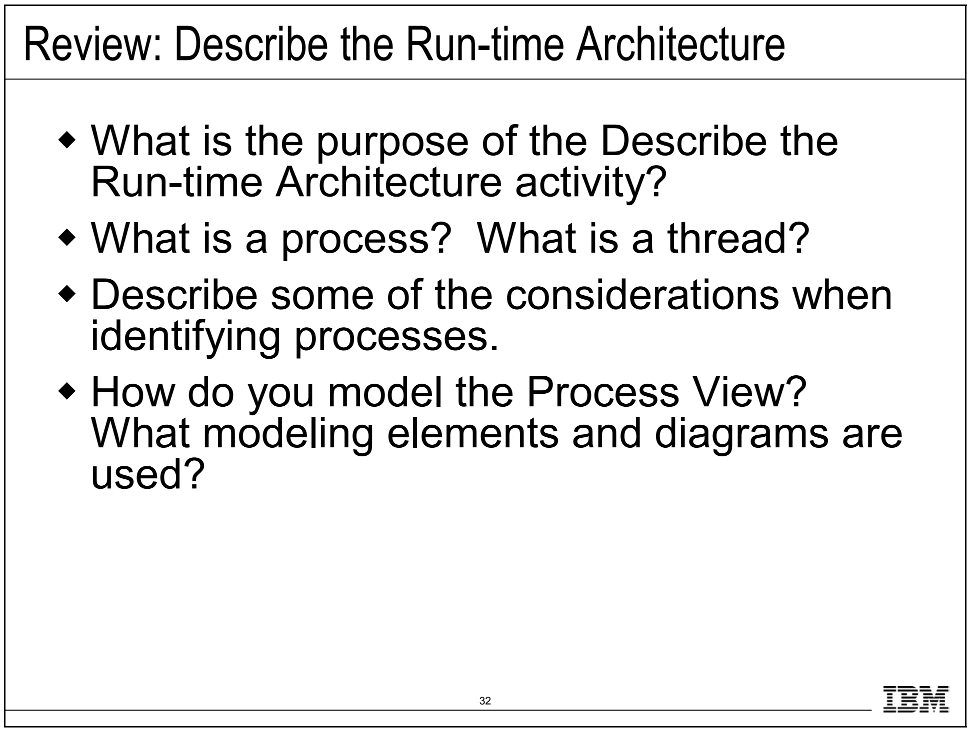


Checkpoints: Describe the Run-time Architecture

- Have all the concurrency requirements been analyzed?
- Have the processes and threads been identified?
- Have the process life cycles been identified?
- Have the processes been mapped onto the implementation?
- Have the model elements been distributed among the processes?

31

Review

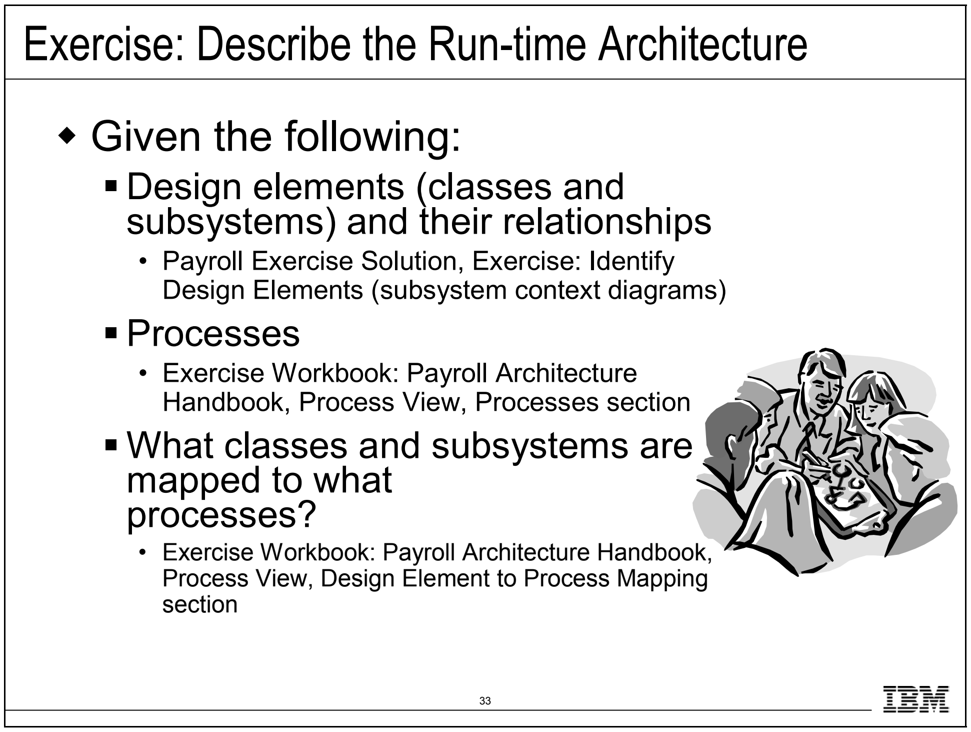


Review: Describe the Run-time Architecture

- What is the purpose of the Describe the Run-time Architecture activity?
- What is a process? What is a thread?
- Describe some of the considerations when identifying processes.
- How do you model the Process View? What modeling elements and diagrams are used?

32

Exercise: Describe the Run-time Architecture



Exercise: Describe the Run-time Architecture

- Given the following:
  - Design elements (classes and subsystems) and their relationships
    - Payroll Exercise Solution, Exercise: Identify Design Elements (subsystem context diagrams)
  - Processes
    - Exercise Workbook: Payroll Architecture Handbook, Process View, Processes section
  - What classes and subsystems are mapped to what processes?
    - Exercise Workbook: Payroll Architecture Handbook, Process View, Design Element to Process Mapping section

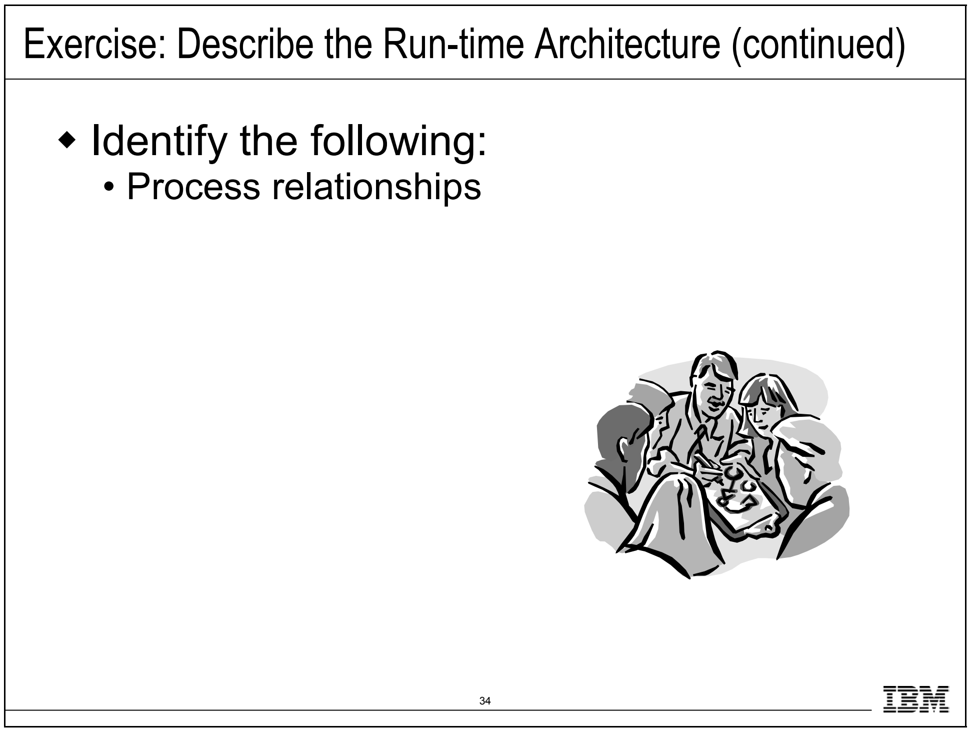
33 IBM

In this exercise, a part of the process architecture will be given textually. The entire process architecture will not be derived, since such a derivation is out of the scope of this course. The exercise allows you to identify the necessary process relationships and produce a visual model of the process architecture.

References to the givens:

- Design elements and their relationships:
  For subsystems and interfaces: Payroll Exercise Solution, Exercise: Identify Design Elements (subsystem context diagrams).
  For other design elements, see their associated Analysis element relationships: Payroll Exercise Solution, Exercise: Use-Case Analysis, Part 2 (VOPC diagrams).
- Processes: Exercise Workbook: Payroll Architecture Handbook, Process View, Processes section.
- Classes and subsystems: Exercise Workbook: Payroll Architecture Handbook, Process View, Design Element to Process Mapping section.

Exercise: Describe the Run-time Architecture (continued)



Exercise: Describe the Run-time Architecture (continued)

- Identify the following:
  - Process relationships

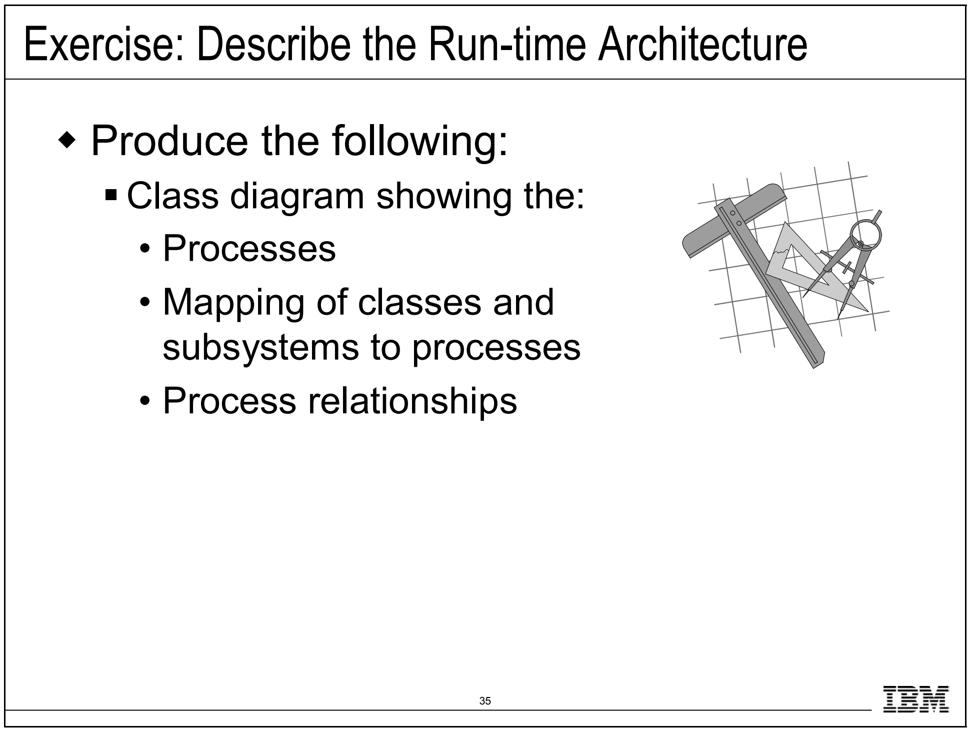
34

The process relationships can be derived from the class relationships. If two classes must communicate and they have been mapped to different processes, then there must be a relationship between the two processes.

Refer to the following slides if needed:

- Key Concept: Processes and Threads – 9-14
- Modeling Processes – 9-16
- Example: Modeling Processes: Class Diagram – 9-17

Exercise: Describe the Run-time Architecture



Exercise: Describe the Run-time Architecture

- Produce the following:
  - Class diagram showing the:
    - Processes
    - Mapping of classes and subsystems to processes
    - Process relationships

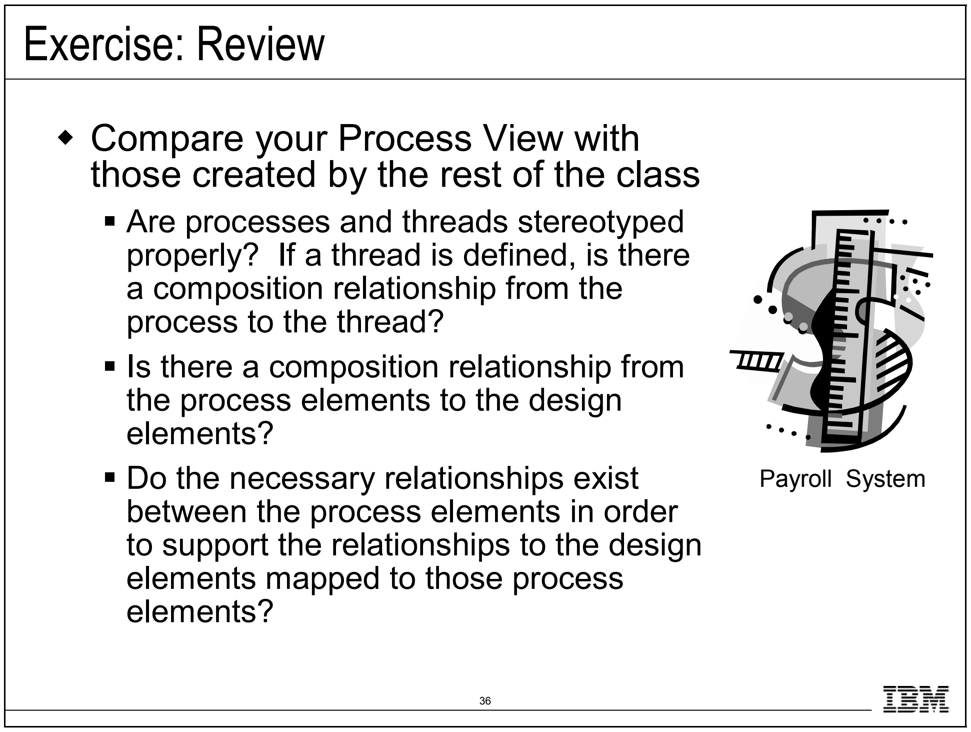
35

The Payroll process class diagram that is produced should include the design element relationships that justify the associated process element relationships. All of the design element relationships do not have to be shown. Only the design element relationships that justify the associated process element relationships must be included.

Refer to the following slides if needed:

- Modeling the Mapping of Elements to Processes – 9-27
- Process Relationships – 9-28
- Example: Register for Courses Processes – 9-29 and 9-30

Exercise: Review



Exercise: Review

- Compare your Process View with those created by the rest of the class
  - Are processes and threads stereotyped properly? If a thread is defined, is there a composition relationship from the process to the thread?
  - Is there a composition relationship from the process elements to the design elements?
  - Do the necessary relationships exist between the process elements in order to support the relationships to the design elements mapped to those process elements?

Payroll System

36

After completing a model, it is important to step back and review your work. Some helpful questions are the following:

- Are processes and threads stereotyped properly? If a thread is defined, is there a composition relationship from the process to the thread? Processes and threads should be modeled as <<process>> and <<thread>> stereotyped classes, respectively. If a <<thread>> is defined, there must be a composition relationship from some <<process>> to the << thread>>, since a <<thread>> *never* exists outside the context of a <<process>>.
- Is there a composition relationship from the process elements to the design elements? There should be composition relationships modeled *from* the process elements to the design elements mapped to those process elements.
- Do the necessary relationships exist between the process elements in order to support the relationships to the design elements mapped to those process elements?