Object Oriented Analysis and Design Using UML

Course Code: CY450

Version 2.0

IBM®

Behavioral and Architectural Modeling

Unit 1

Basic Behavioral Modeling

Learning Objectives

- To understand the concept of dynamic modeling
- To learn about interactions, use cases and activity
- To describe interaction diagrams
- To gain knowledge about using case diagrams
- To comprehend the need for activity diagrams
- To learn about events and signals
- To understand state machines
- To study about time and space concept
- To describe statechart diagrams

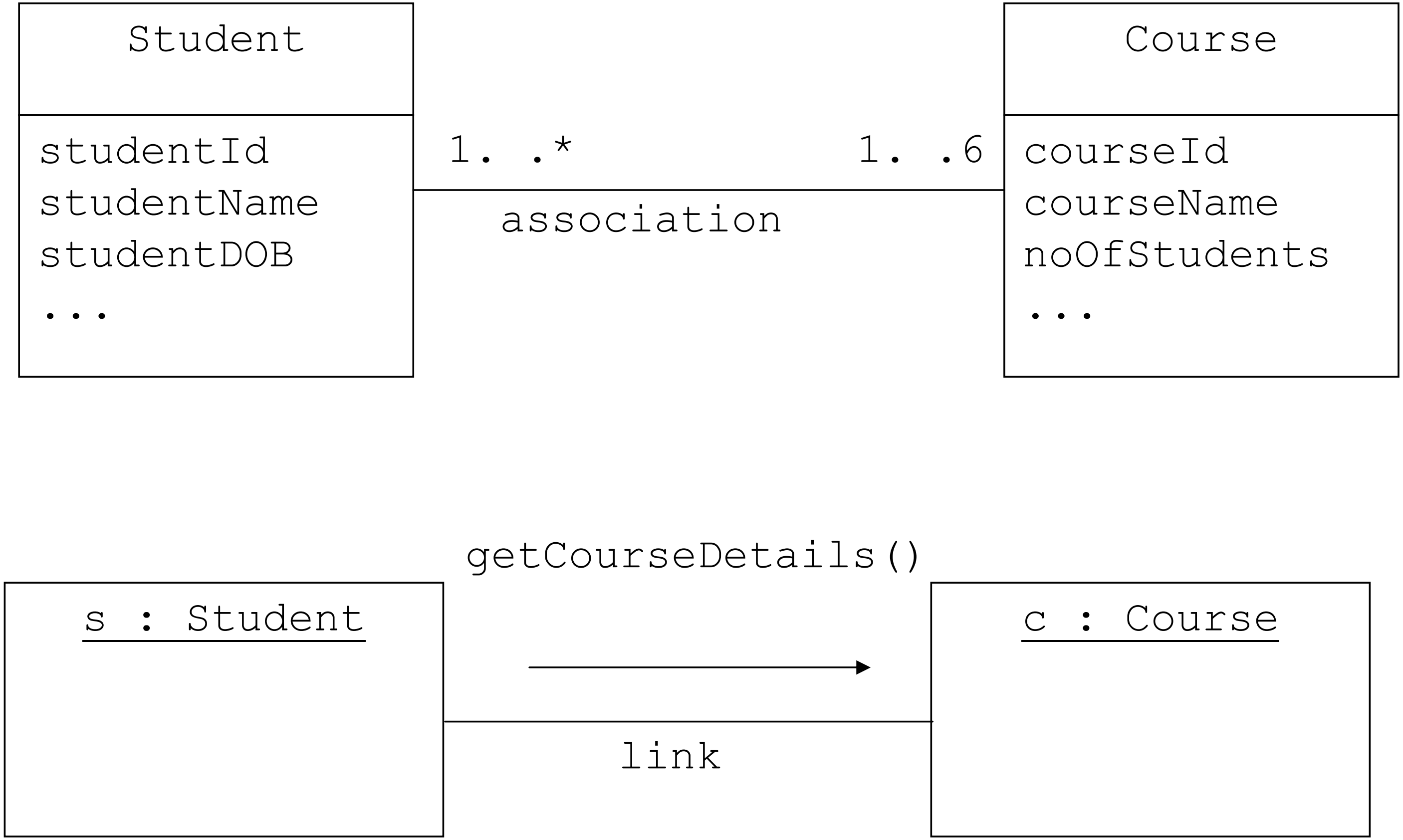
Modeling Dynamics

- Behavioral modeling enables us to capture the dynamic events that are part of a system
- Some of the dynamic events are listed below
  - Creation of objects
  - Destruction of objects
  - Sending of messages between objects (object interactions)
  - Capturing the state an object is currently in
- None of the above can be captured through a static model
- The dynamic semantics of a system can be depicted in the UML through the following diagrams
  - Interactions diagrams (sequence and collaboration diagrams)
  - Activity diagrams
  - Statechart diagrams

Interactions

- Objects work together to carry out the required behavior
- Objects interact with one another, as objects in the real world do
- A society of objects is represented as collaborations in the UML. Objects, when they interact with one another, also play a specific role
- Interactions are used to model the dynamic aspect of collaborations
- An interaction is a set of messages exchanged between objects in the system
- Interactions introduce a dynamic ordering of messages passed between two or more objects. Interactions can be modeled in two ways, viz. time ordering of messages or sequencing of messages

Links



Student

studentId
studentName
studentDOB
...

1. .*

association

1. .6

Course

courseId
courseName
noOfStudents
...

getCourseDetails()

s : Student

link

c : Course

Stereotypes Used With Links

- `association` – An association relationship between classes makes the object visible
- `self` – The object is visible as it is the dispatcher of the operation
- `global` – The object is available in the enclosing scope
- `local` – The object is in the local scope
- `parameter` – The object is a parameter

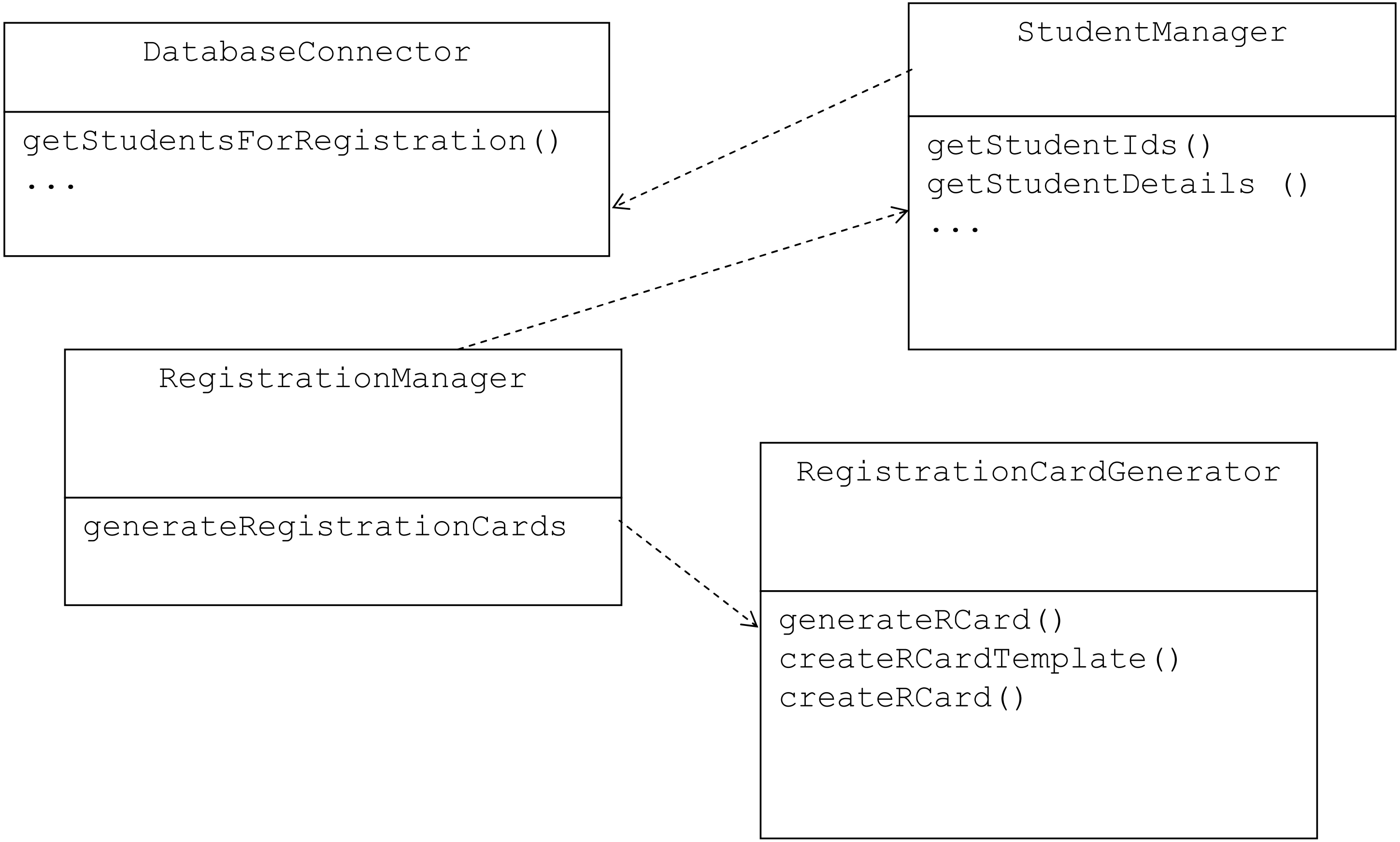
Messages

- Messages form an essential part of object interaction
- Having objects and links is not meaningful unless one object passes a message to another
- An action results when a message is passed between objects
- This is equivalent to submitting a withdrawal slip in a bank and the bank clerk handing over the cash, after performing a check on the account
- Submitting a withdrawal slip is the message and the handing over of the cash is the action
- Thus every message is followed by some action
- Action in object orientation is initiated with the passing of a message from one object to another
- The receipt of a message instance is typically considered as an instance of an event

Actions Modeled in UML

- `Call` – Invokes an operations on an object
- `Return` – Returns a value to the caller object
- `Send` – Sends a signal to an object
- `Create` – Creates an instance of a class
- `Destroy` – Destroys an already created instance

Abstractions and their Relationships



DatabaseConnector

getStudentsForRegistration()
...

StudentManager

getStudentIds()
getStudentDetails ()
...

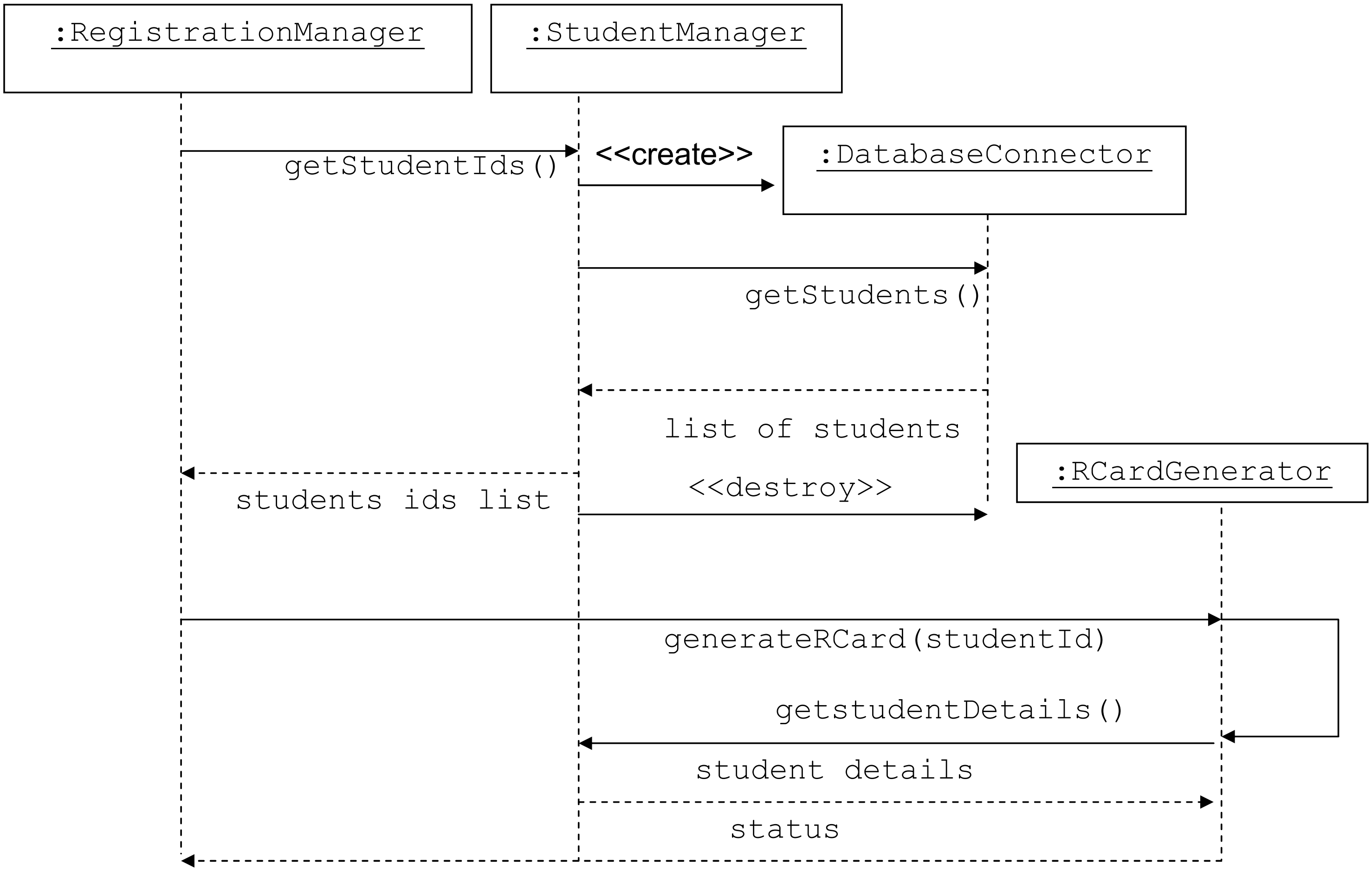
RegistrationManager

generateRegistrationCards

RegistrationCardGenerator

generateRCard()
createRCardTemplate()
createRCard()

The UML Way of Specifying Messages for Abstractions

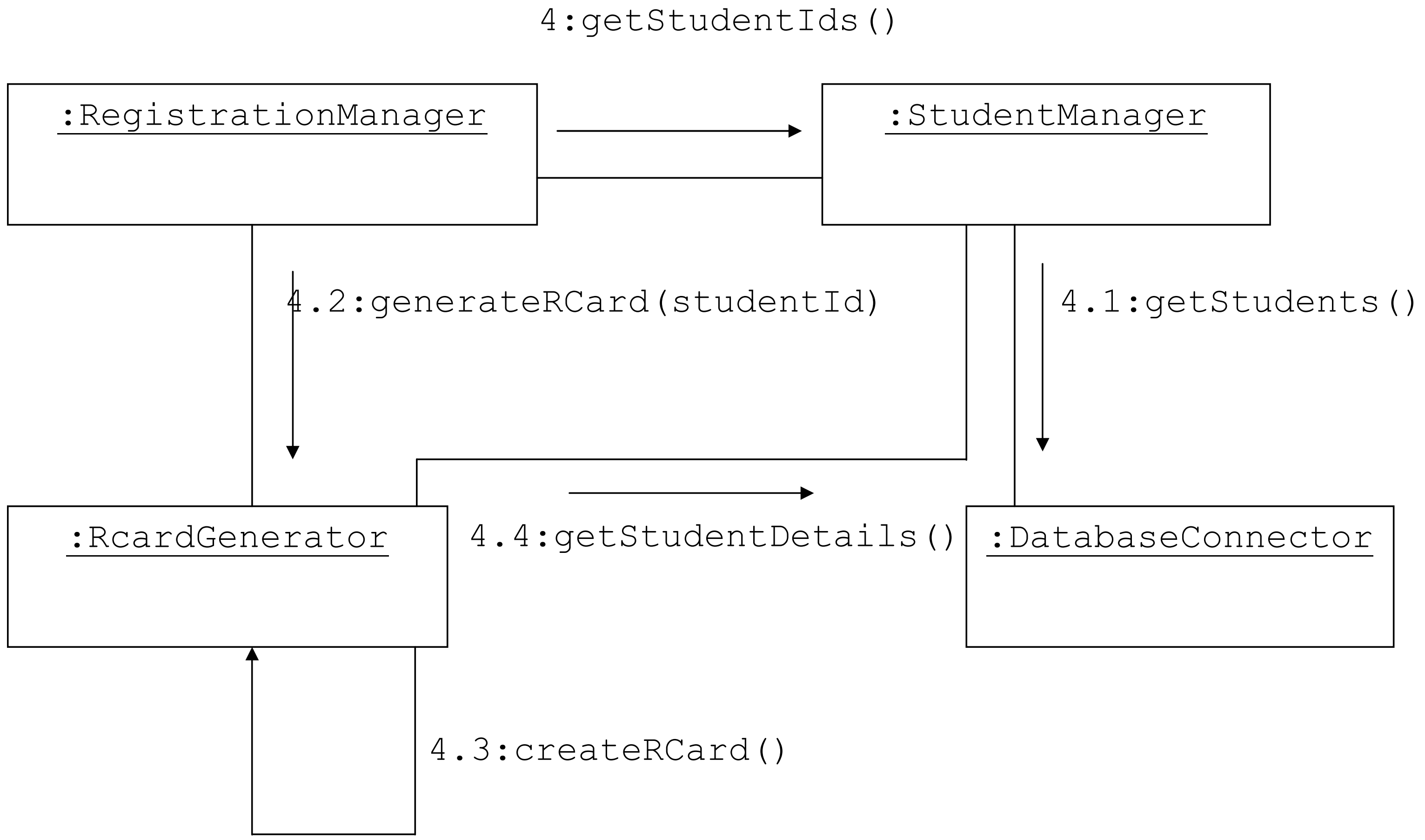


Procedural & Flat Sequencing

- Message passing can also be sequenced, by providing sequence numbers to each message
- This helps in understanding the flow of events
- The objects passing the messages are part of the sequence, shown in a typical object diagram style
- There are two kinds of sequencing, viz. *procedural* and *flat*
- Procedural sequencing is also called nested flow of control
    - This is the most common kind of sequencing used in most modeling efforts, which is typical of any operation
    - Operation `a` calls operation `b` and operation `b` in turn calls operation `c`
    - This is modeled using procedural sequence
- Flat sequencing is used to show the series of messages step by step
    - This is nonprocedural in nature.

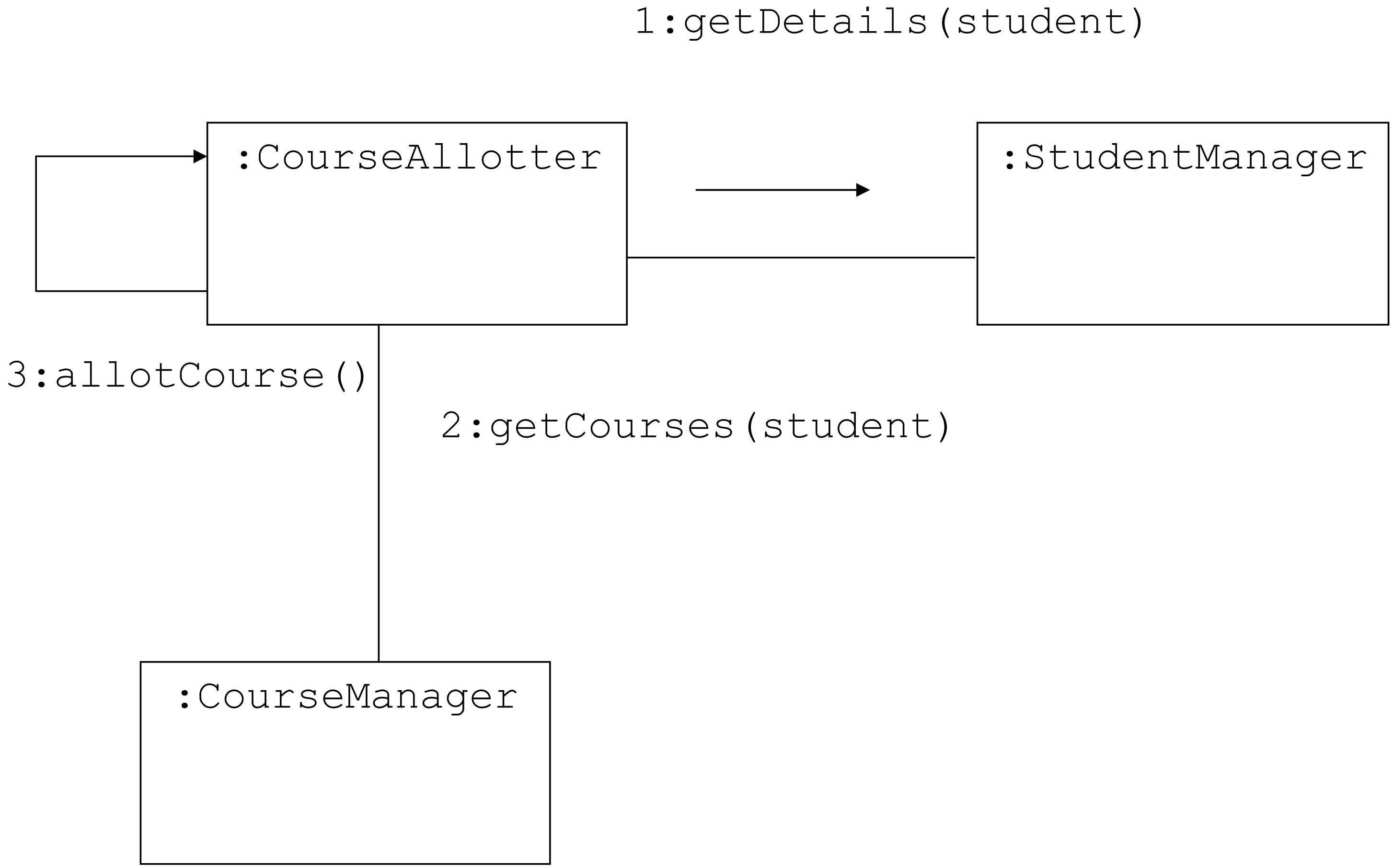
Procedural & Flat Sequencing …

continued



Procedural & Flat Sequencing …

continued



Constraints Associated With Links

- `new` – This is used to indicate that the link is created during the execution of the enclosing interaction
- `destroyed` – When the link is destroyed during the execution of the enclosing interaction
- `transient` – This is used to show that the link is created and destroyed during the execution of the enclosing interaction

Use Cases

- Formulated by Ivar Jacobson
- A use case is an interaction that a user or another system has with the system being designed
- The user or the system that has an interest in the system being designed is termed an *actor*
- An actor is an entity that interacts with the system
- A use case is a description of a set of actions that a system performs with respect to a particular actor interested in the system
- A use case involves interactions between actors and the system
- The actor itself could be another system
- The role of actors is important when using them with use cases
- The actor represents the role the users of the system play while interacting with the system

Usefulness of Use Cases

- in discovering requirements
- us capture the user's need by focusing on the task the user needs to perform with the system
- formulate system test plans
- in controlling iterative development

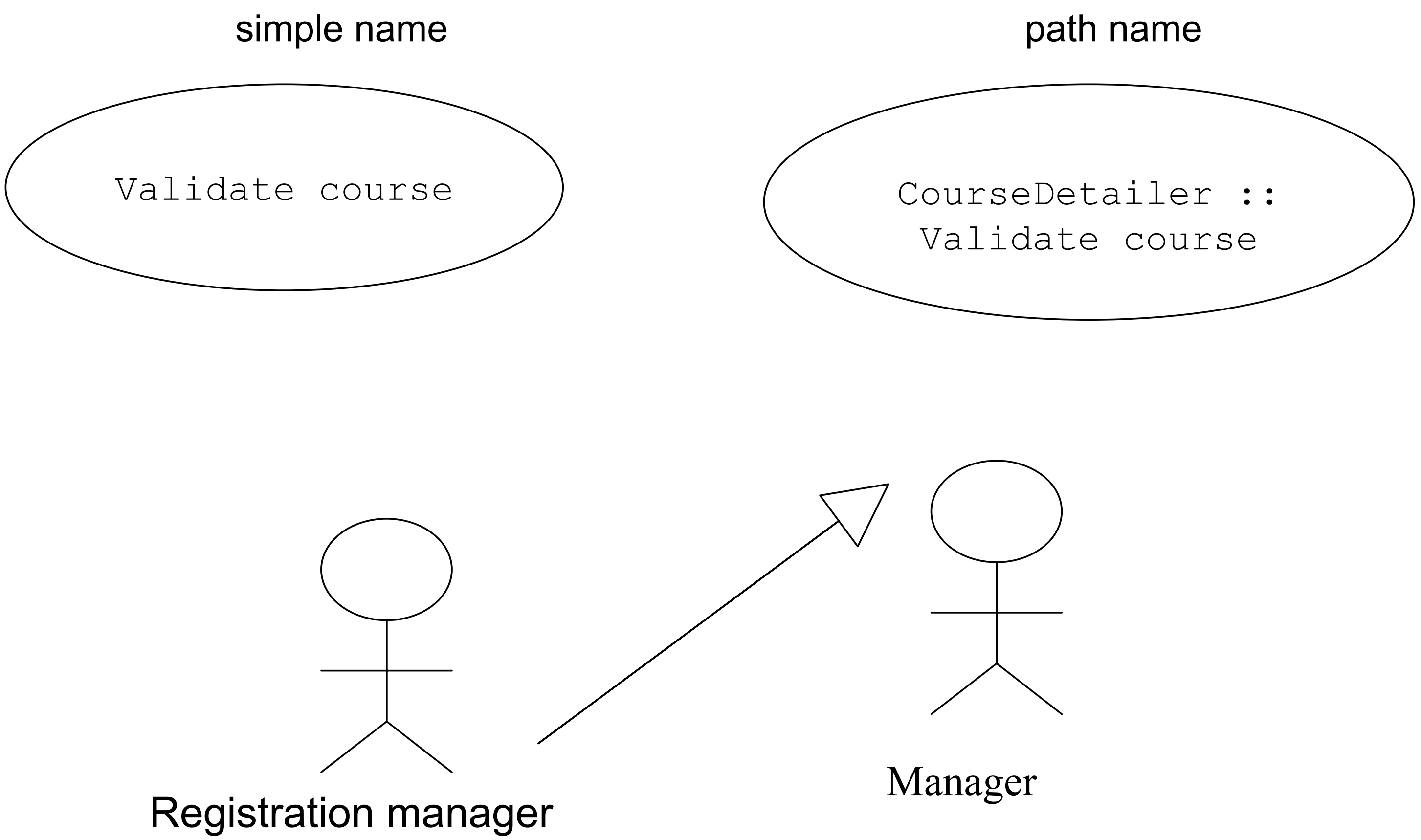
Typical Examples of Use Cases

- Generate idno
- Process loan
- Print report
- View statistics
- Allot course
- Create document
- Issue book

Actors

- For the above use cases mentioned in earlier slide, we may identify the following actors
  - Registration manager
  - Bank clerk
  - Office clerk
  - Stock exchange pundit
  - Course allotter
  - Author
  - Library clerk
- The actors are not the abstractions of the system
- They represent the external users who would use the system being designed

Use Case in UML



simple name

Validate course

path name

CourseDetailer ::
Validate course

Registration manager

Manager

Use case: Allot courses

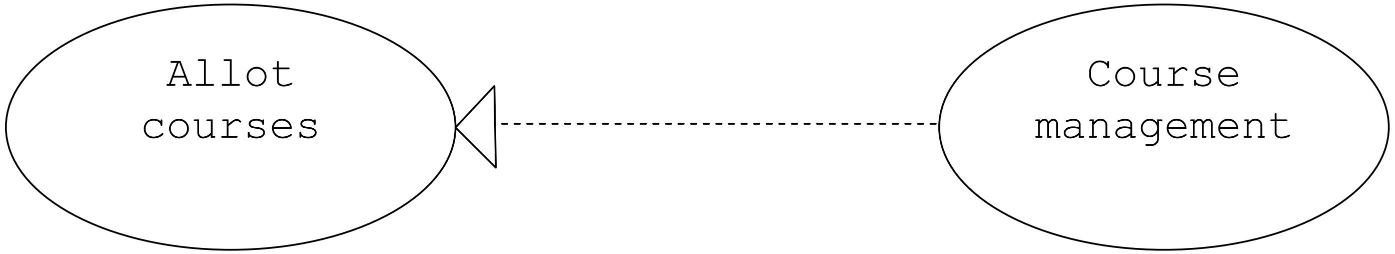
- The system prompts the RegistrationManager to enter the discipline code for which the course allotment needs to be done
- The RegistrationManager enters the discipline code through the keyboard
- The system checks for the validity of the discipline code
- The system then prompts the RegistrationManager to enter whether course is to be allotted to a single student, a range of students or all students
- The RegistrationManager chooses one of the three options
- The system displays the courses available for the current semester for allotment
- The RegistrationManager chooses the courses to be allotted
- The system displays the details to the RegistrationManager and waits for a confirmation
- The RegistrationManager confirms the allotment
- The system makes an entry into the student record for the current semester about the courses the student has been allotted
- The RegistrationManager exits the system

Scenarios

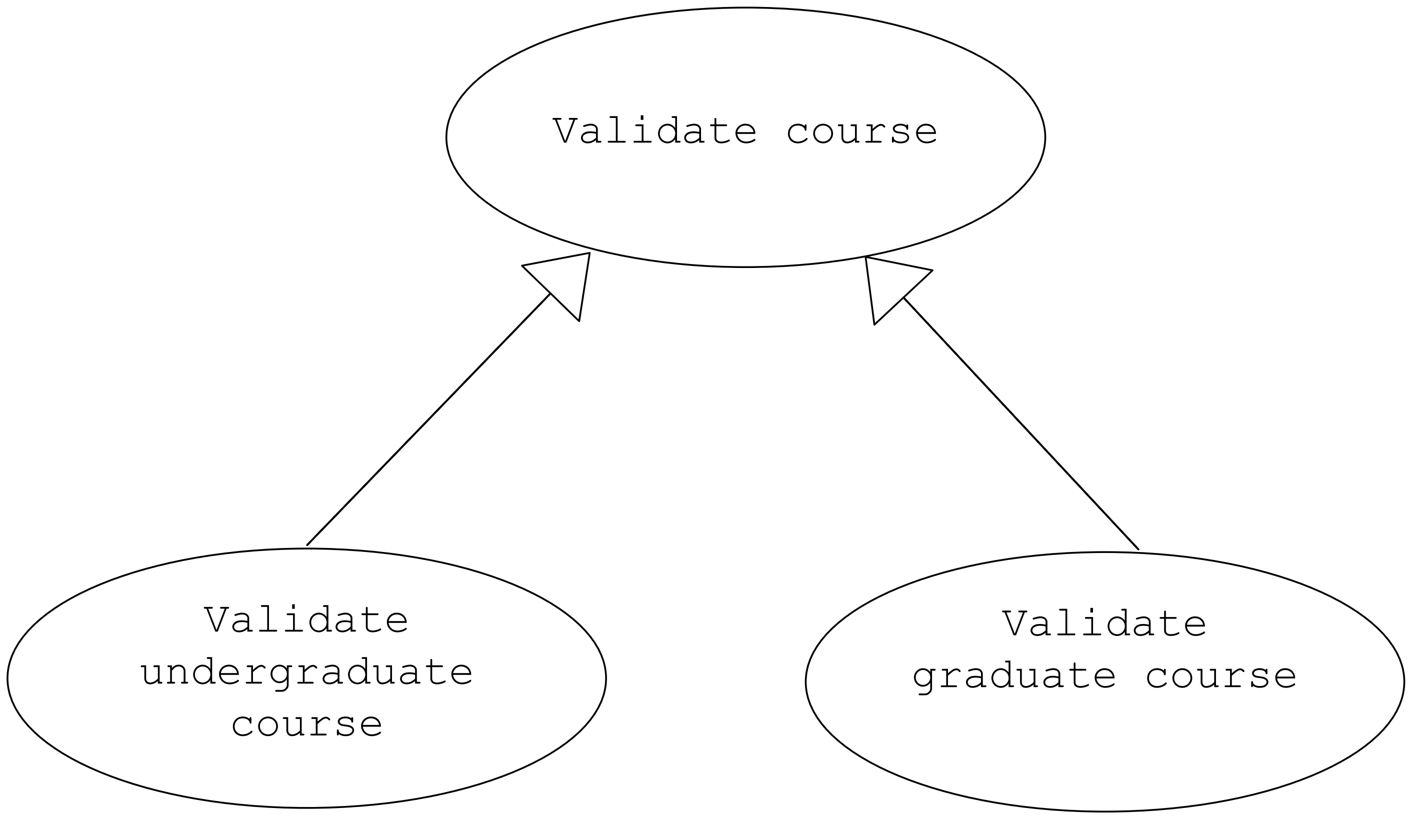
- A scenario is an instance of a use case
- There is normally an expansion from a use case to a scenario
- A single use case can result in a number of scenarios
- For most use cases, we may have primary and secondary scenarios
- Primary scenarios define the essential sequences and the secondary ones define the alternate sequences
- For instance, the use case `Generate graduate student list` can have variants such as generating lists based on certain inputs that are different for undergraduate, postgraduate and research students
- These variants can be modeled as alternate sequences

Collaborations

- A use case captures the intended behavior of the system
- This does not specify how the behavior will be carried out
- But eventually, every use case has to be implemented, creating a society of classes and other elements that help implement the behavior of the use case
- Collaboration is used in the UML to model the society of elements, both static and dynamic, that help implement the behavior of a use case
- A use case is realized by collaboration



Generalization

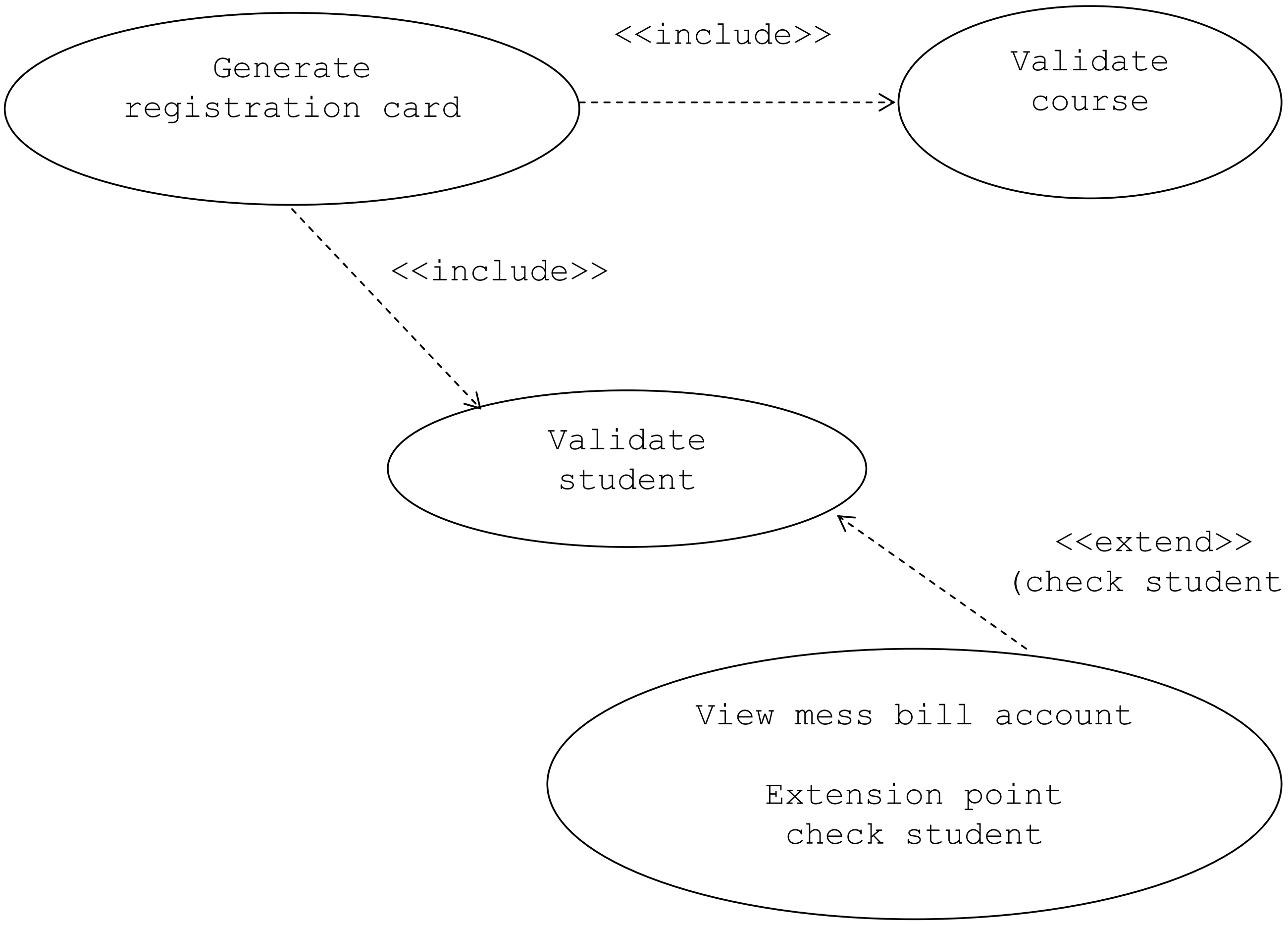


Validate course

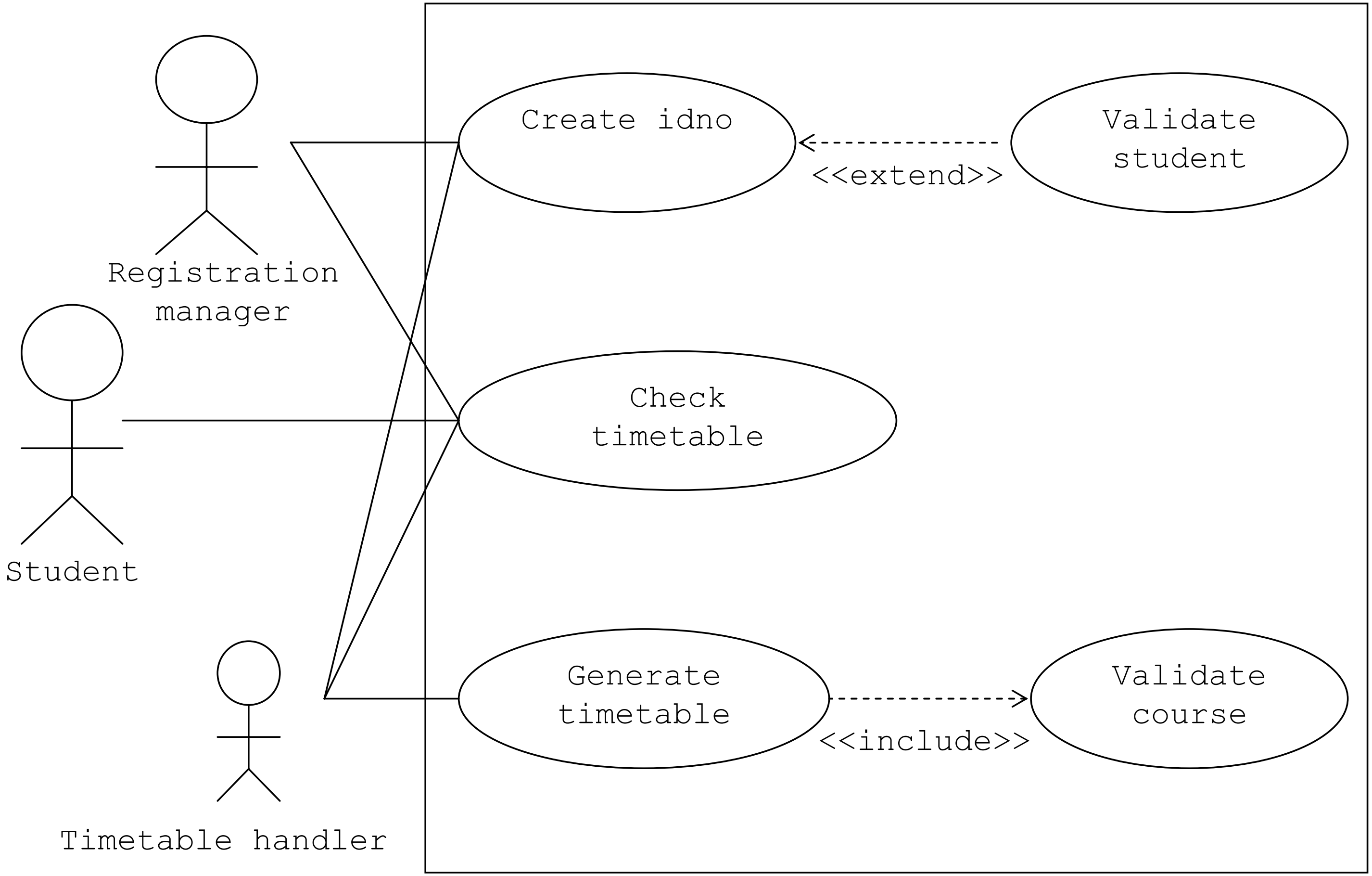
Validate
undergraduate
course

Validate
graduate course

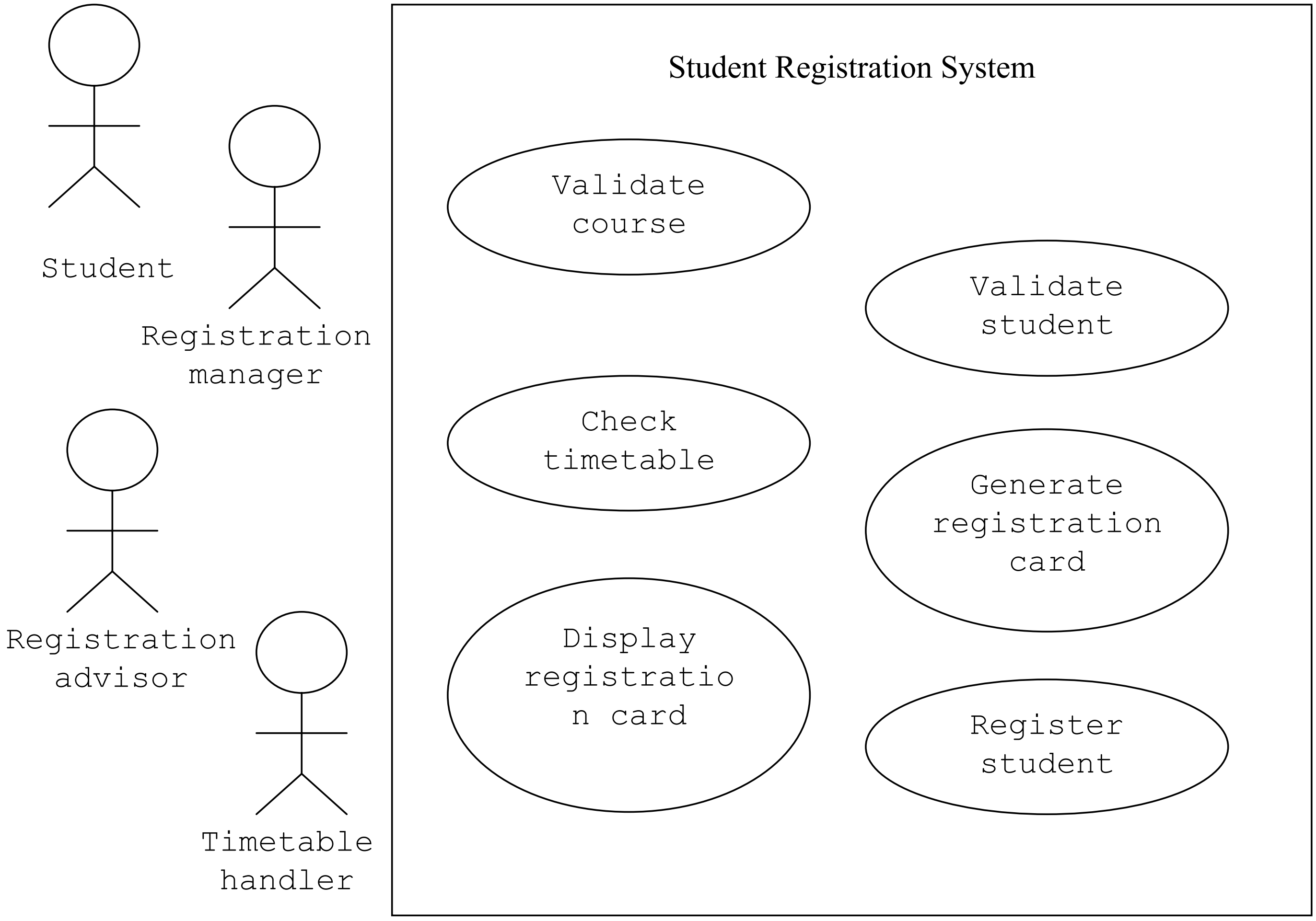
Extend and Include Relationships



Use Case Diagram



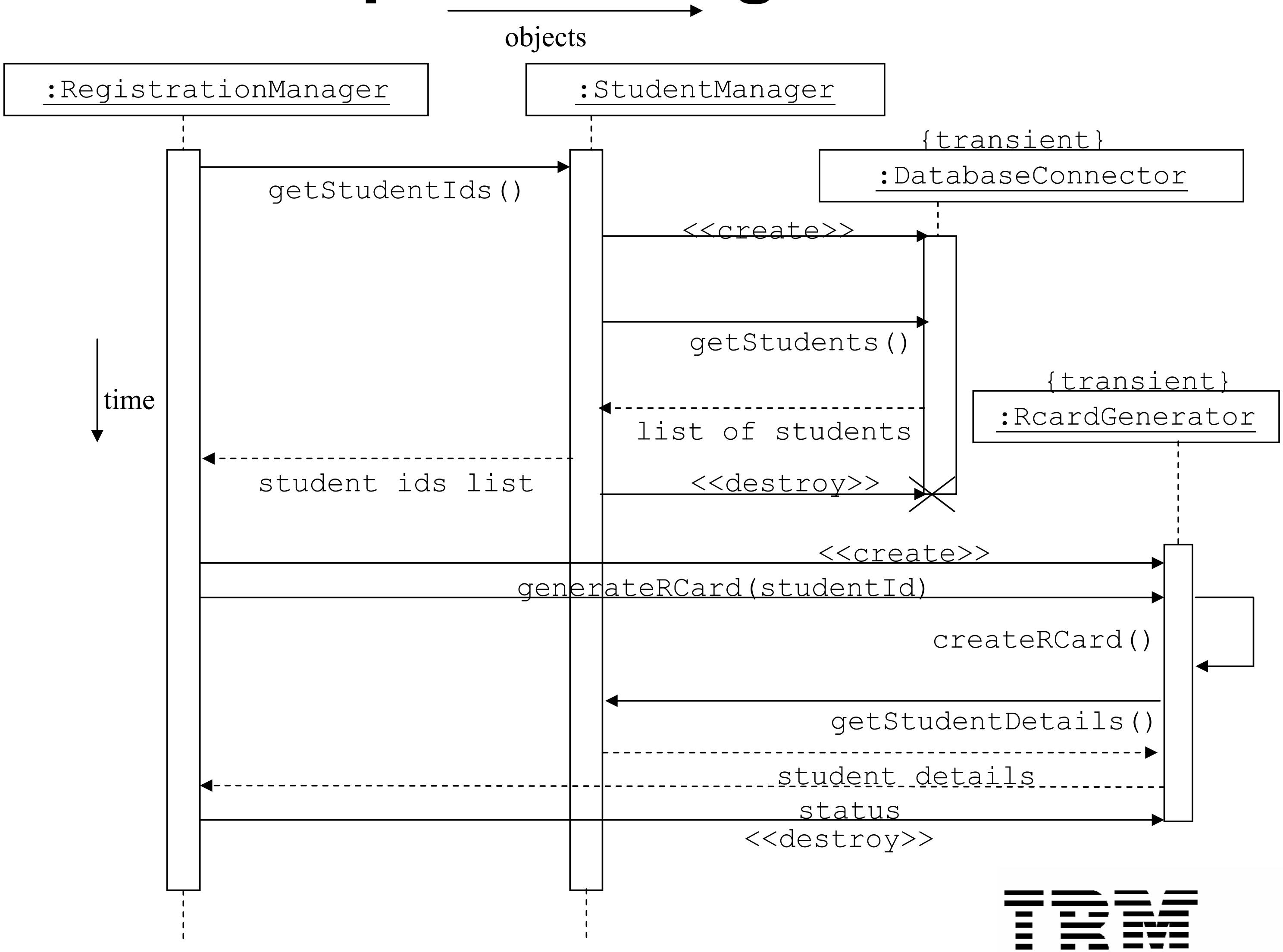
Modeling Requirements of System



Interaction Diagrams

- Interaction diagrams describe how groups of object collaborate to get a job done
- They capture the behavior of a single use case, showing the pattern of interaction among objects
- There are two kinds of interaction diagrams
- *Sequence diagrams*
  - Sequence diagrams describe the behavior of the system by emphasizing the time ordering of messages
- *Collaboration diagrams*
  - Collaboration diagrams represent collaborations emphasizing the structural organization of the objects that pass messages amongst themselves
  - The notation for a collaboration diagram is a set of arcs and vertices.

Sequence Diagrams



objects

:RegistrationManager

:StudentManager

{transient}

:DatabaseConnector

{transient}

:RcardGenerator

time

getStudentIds()

<<create>>

getStudents()

list of students

student ids list

<<destroy>>

<<create>>

generateRCard(studentId)

createRCard()

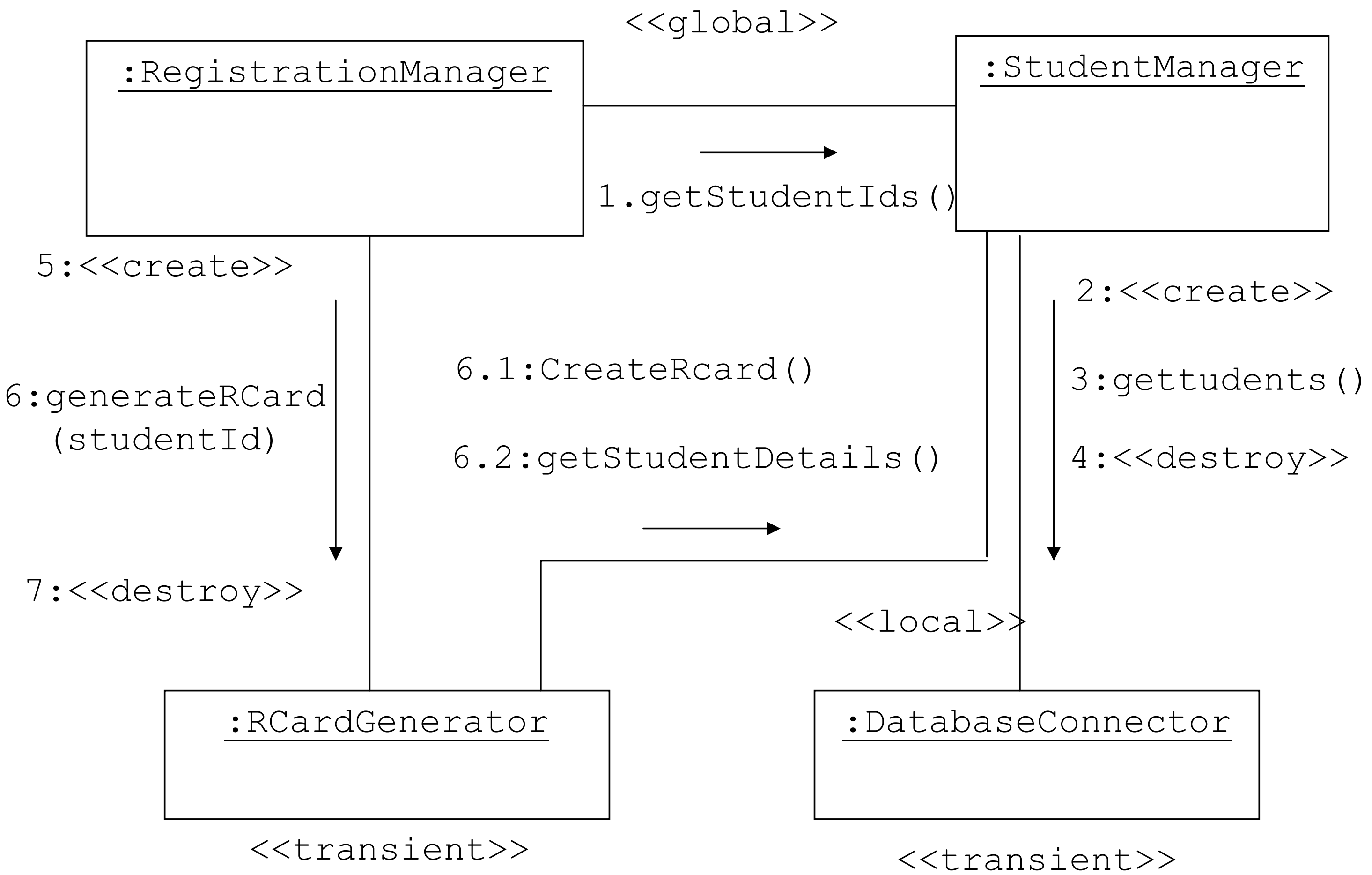
getStudentDetails()

student details

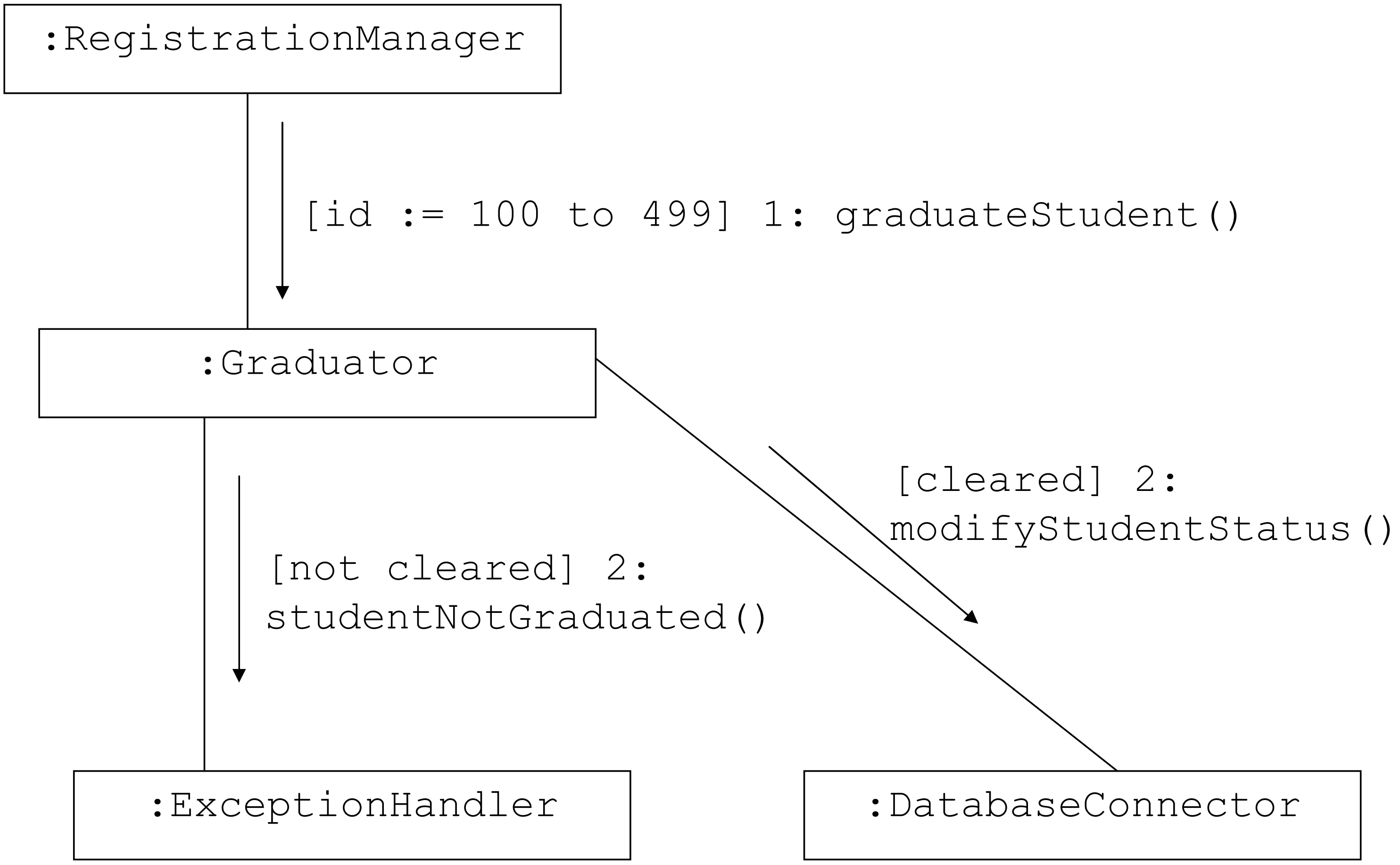
status

<<destroy>>

Collaboration Diagrams



Iteration & Branching in Collaboration Diagrams



:RegistrationManager

[id := 100 to 499] 1: graduateStudent()

:Graduator

[not cleared] 2: studentNotGraduated()

[cleared] 2: modifyStudentStatus()

:ExceptionHandler

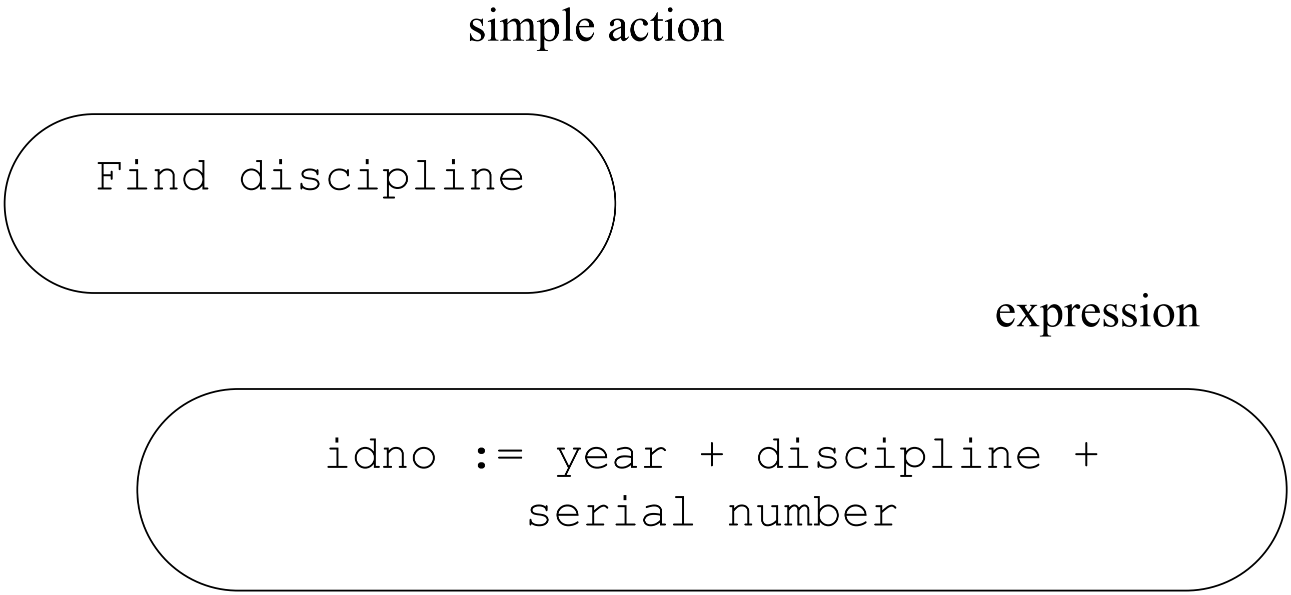
:DatabaseConnector

Activity Diagrams

- Activity diagrams are used to show the flow from one activity to another activity
- An activity is defined as an ongoing non-atomic execution within a state machine
- Each activity results in an *action*
- An action results in a state change of the object or in a value being returned to the caller
- There is a similarity between interaction and activity diagrams
- The difference lies in the fact that the interaction diagrams represent objects that pass messages between them, while the activity diagrams represent the operations that are passed between the objects
- The difference is subtle but presents different views to the users.
- An activity diagram consists of objects, action states, activity states and transitions

Action States

- Action states are executable atomic computations like calling an operation, sending a signal, creation of an object, destruction of an object or even a simple expression
- An action state can be a simple action or an expression. An action state cannot be further decomposed



simple action

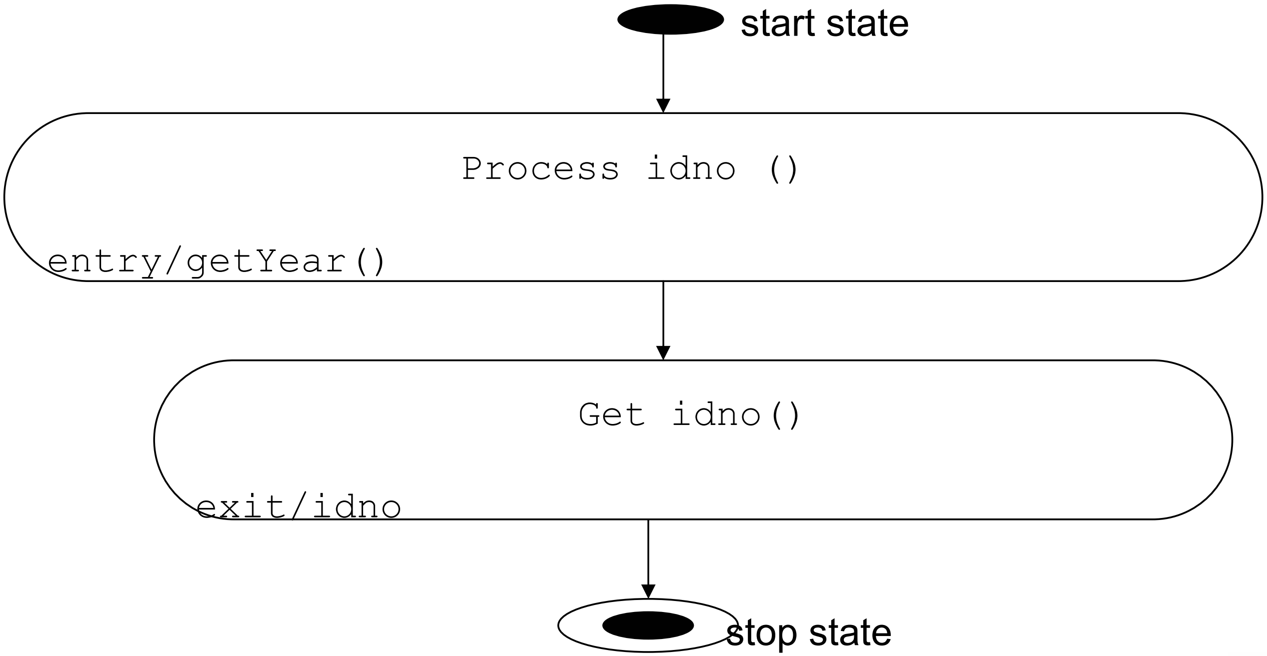
Find discipline

expression

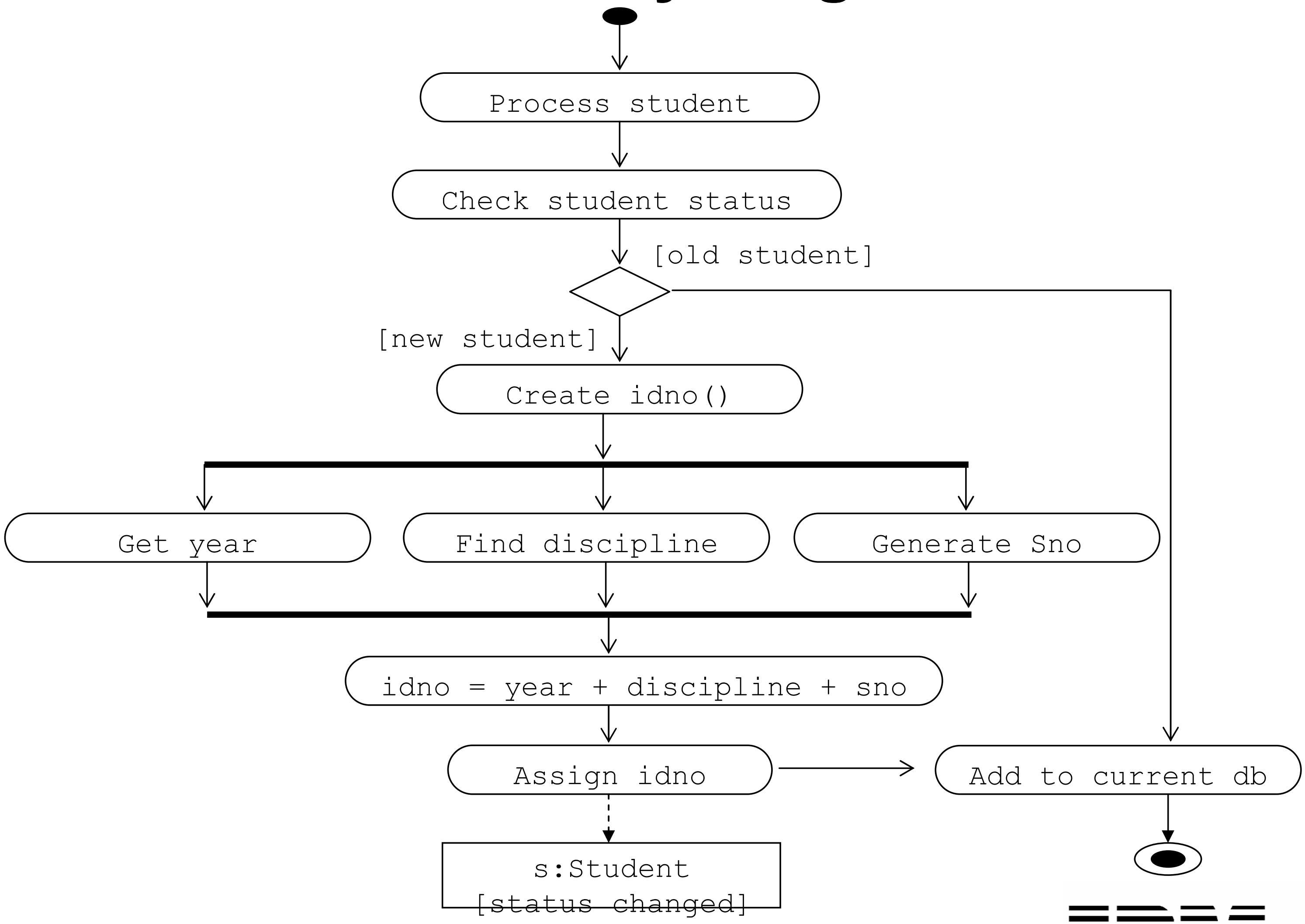
idno := year + discipline +
serial number

Transitions

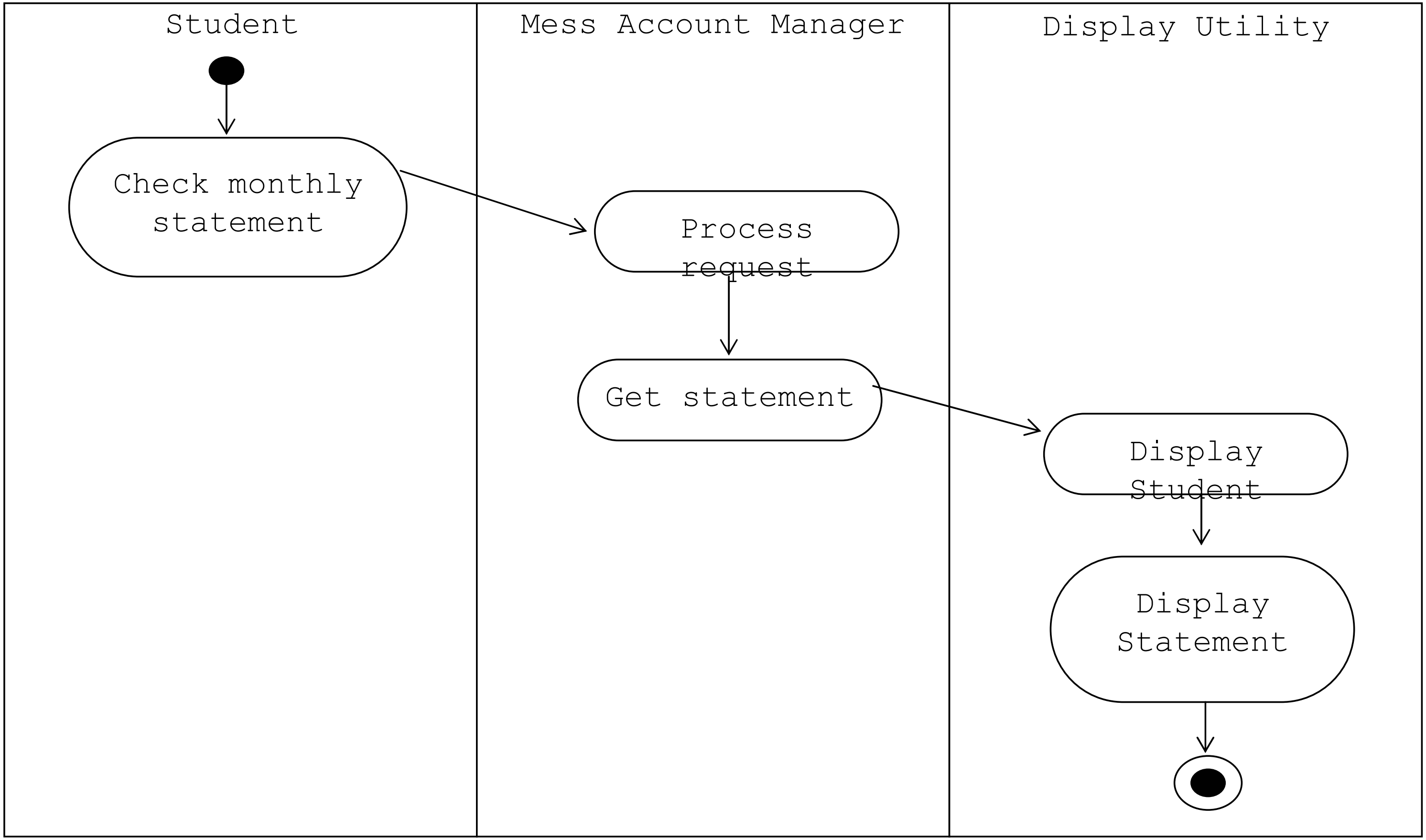
- When an action of a state is completed, the flow of control passes to the next action or activity state
- The flow from one state to another is shown through transitions, which is usually a directed line with the arrowhead pointing towards the next state



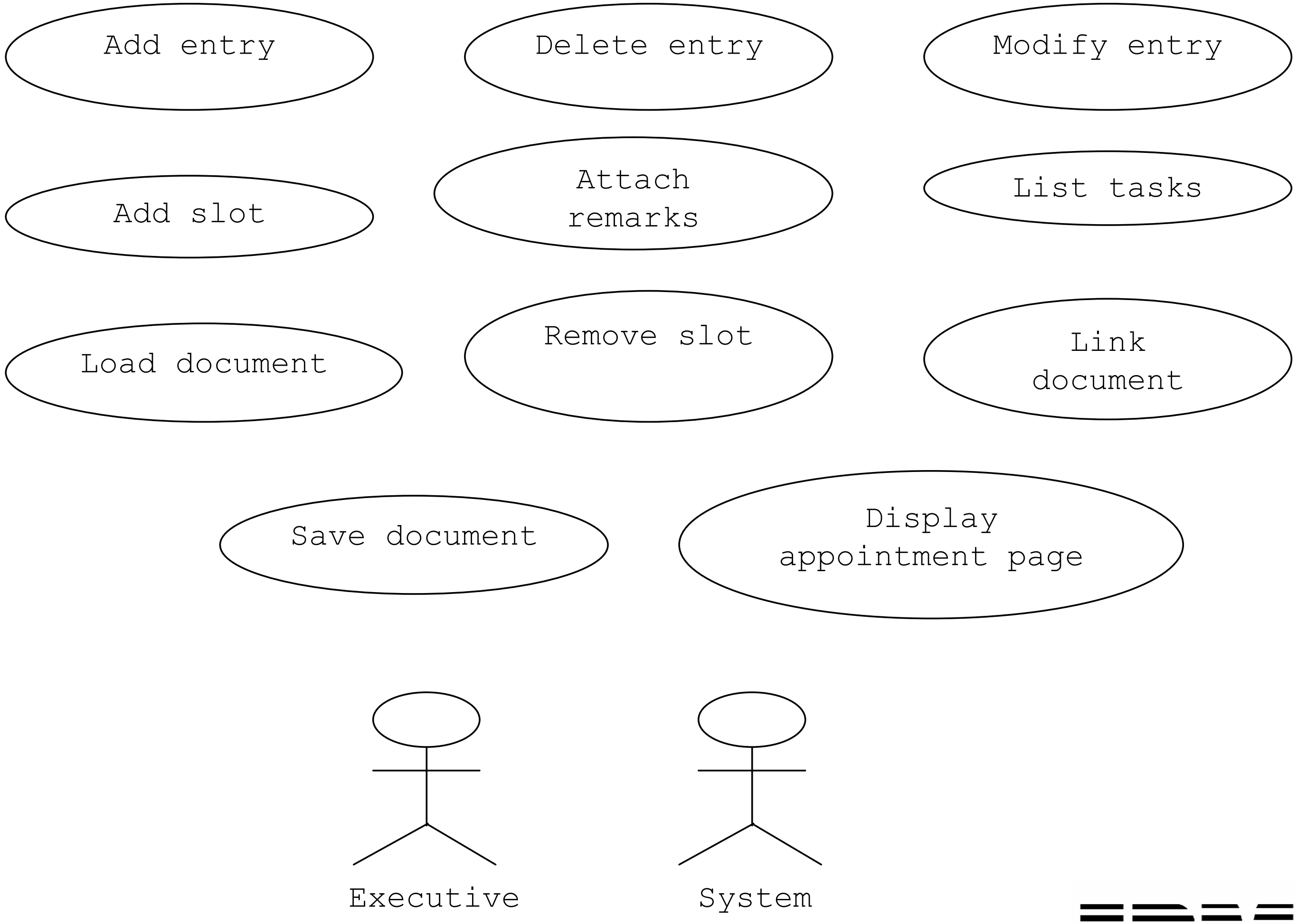
An Activity Diagram



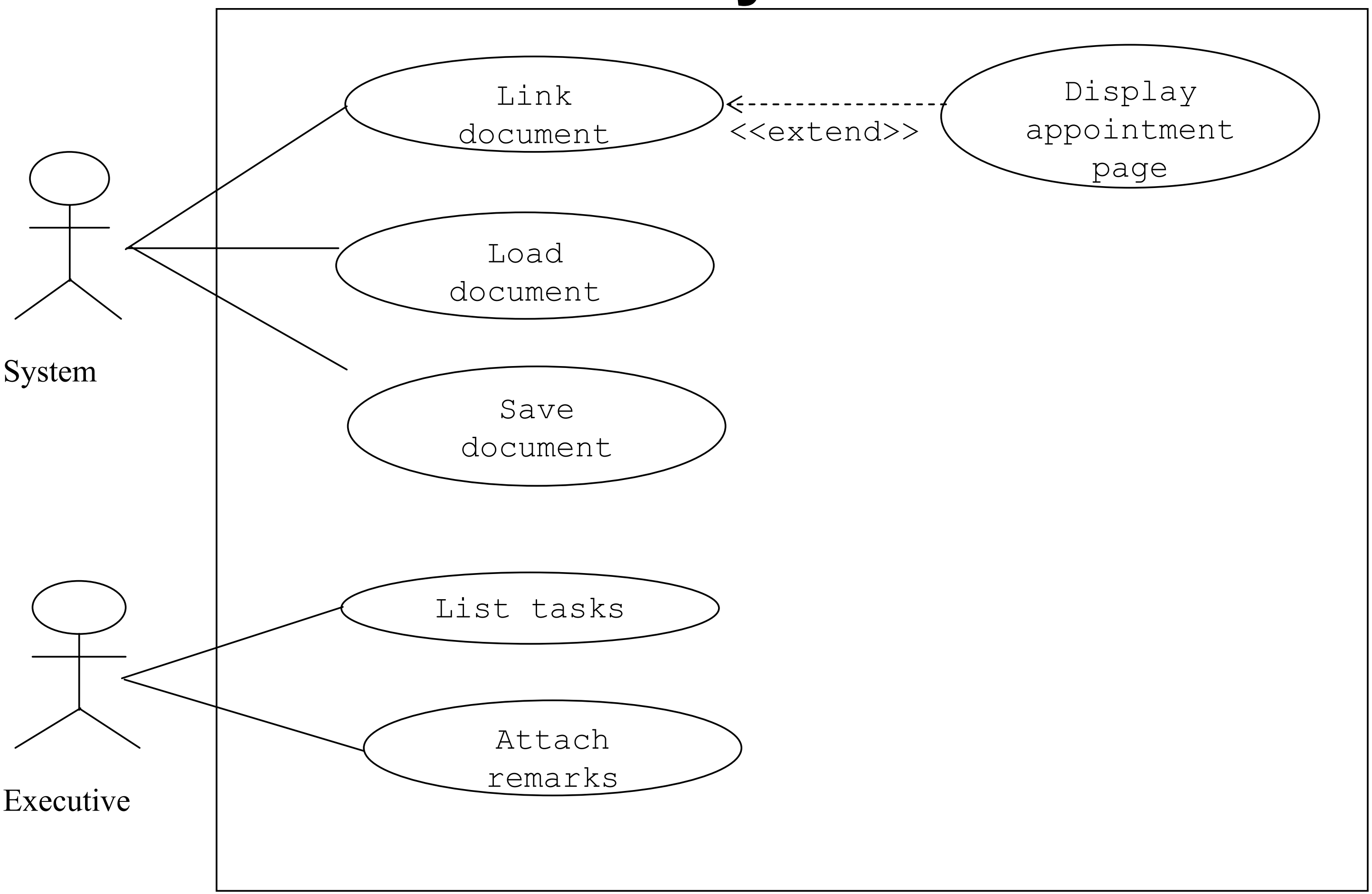
Swimlanes


Student
Mess Account Manager
Display Utility
Check monthly statement
Process request
Get statement
Display Student
Display Statement

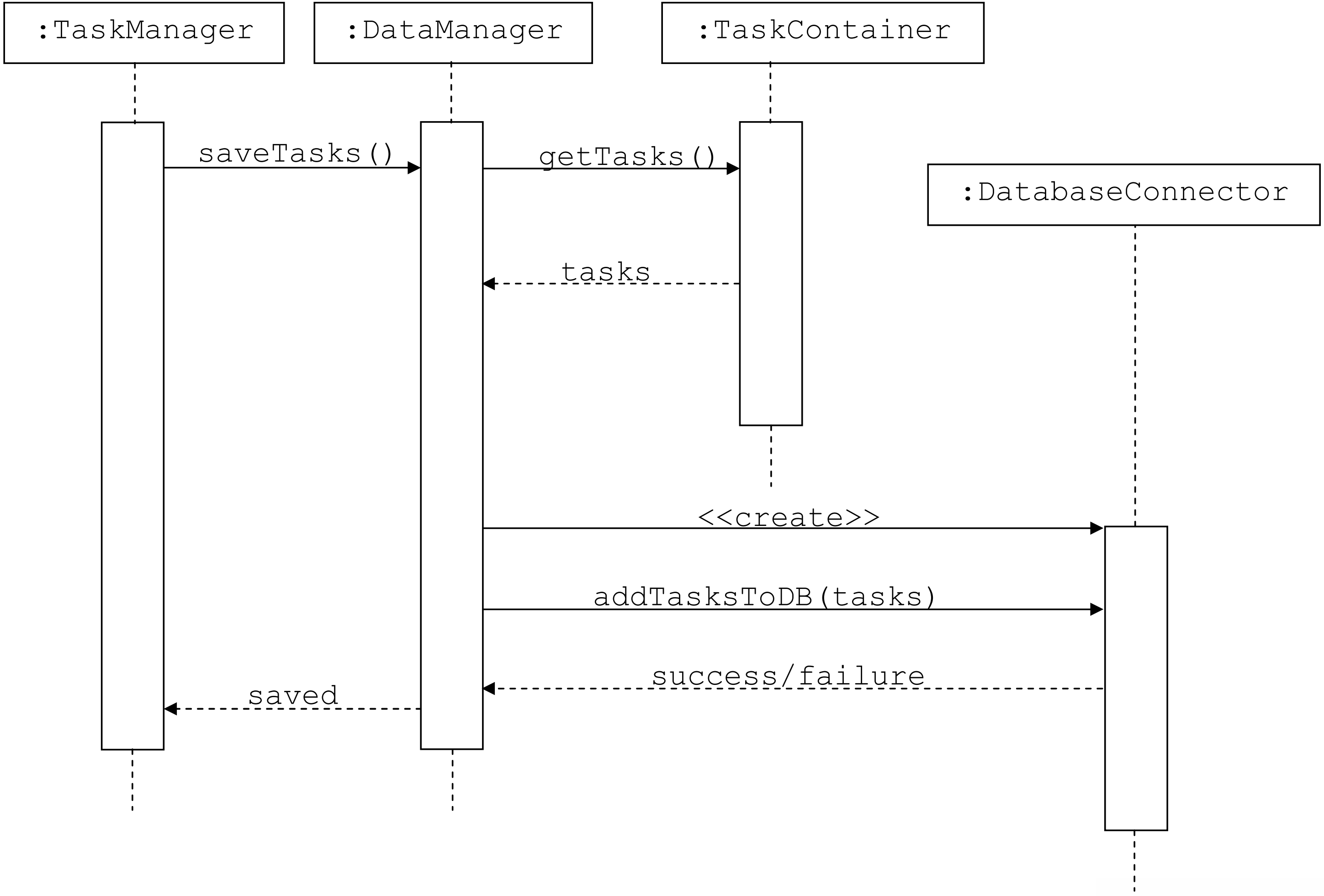
Use Cases for the Electronic Diary



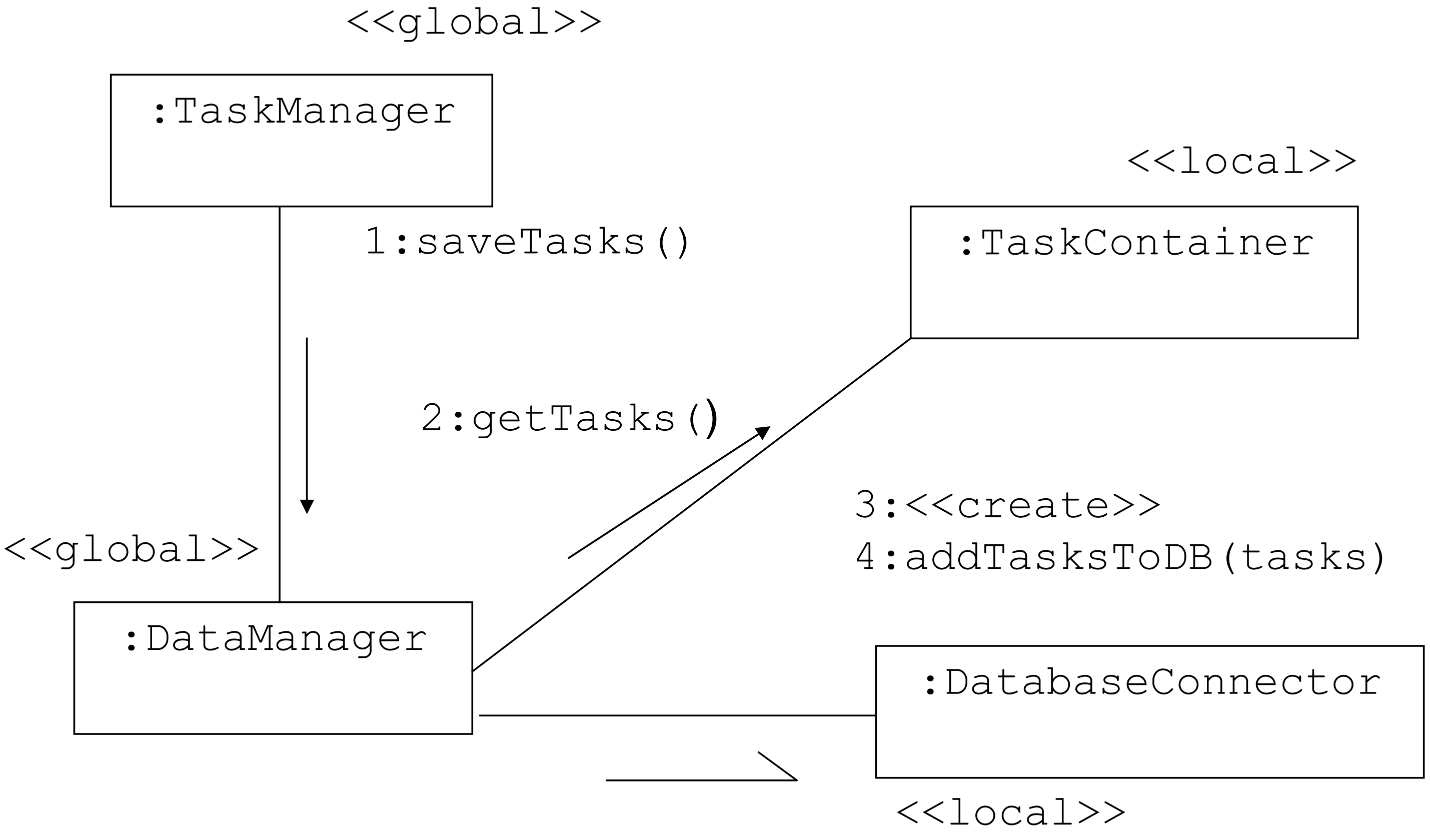
Use Case Diagram for the Electronic Diary



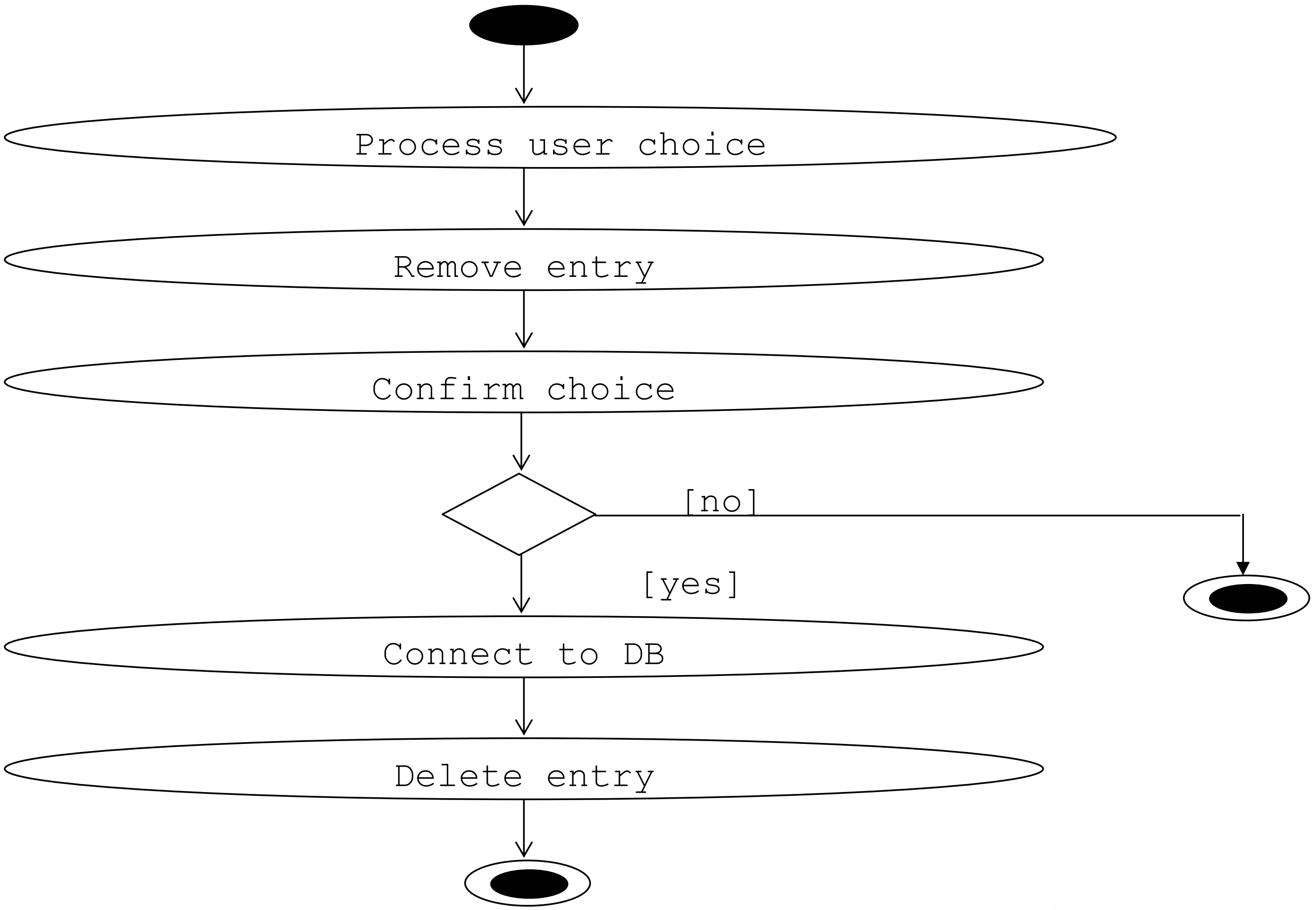
Sequence Diagram for the Electronic Diary



Collaboration Diagram for the Electronic Diary



Activity Diagram for the Electronic Diary



Summary

- We understood the concept of dynamic modeling
- We learnt about interactions, use cases and activity
- We described interaction diagrams
- We gained knowledge in the usage of case diagrams
- We comprehended the need for activity diagrams
- We learnt about events and signals
- We understood state machines
- We studied the time and space concept
- We described statechart diagrams

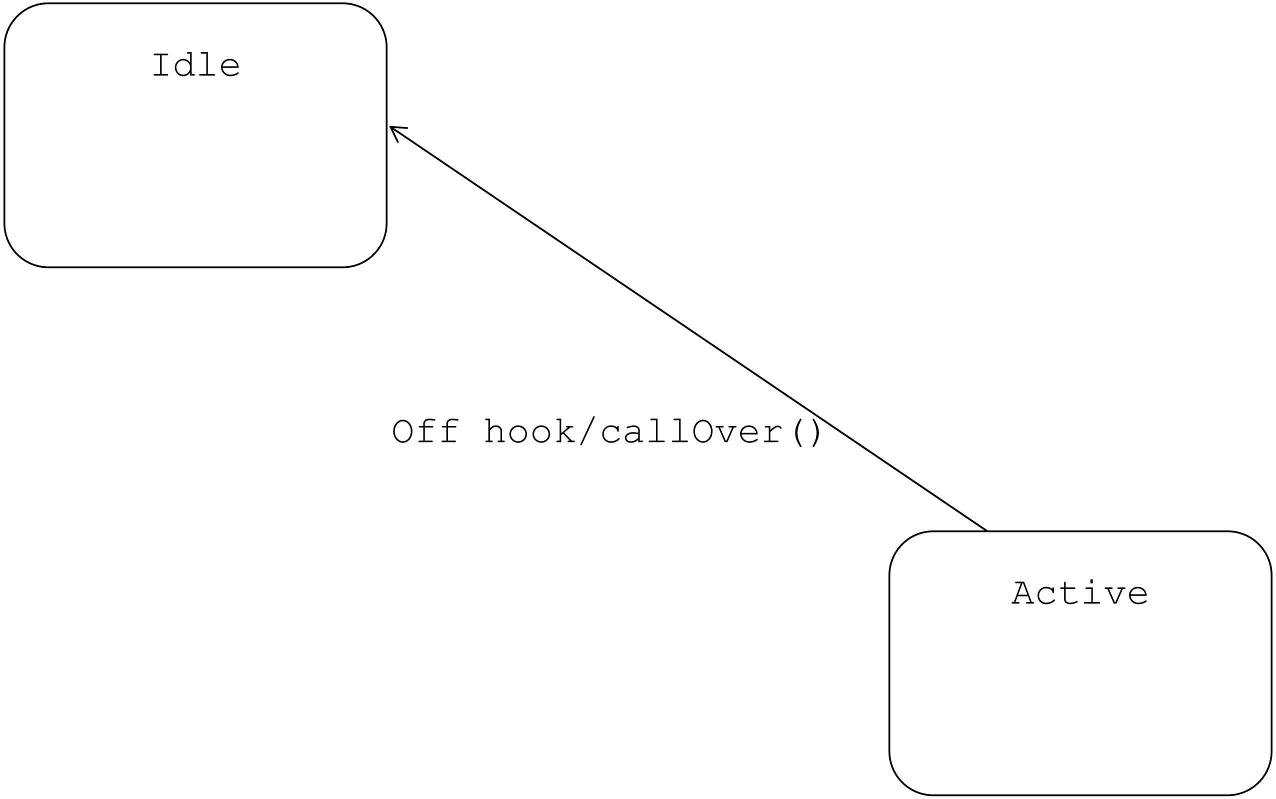
Unit 2

Advanced Behavioral Modeling

Learning Objectives

- To discuss events and signals
- To learn what are state machines
- To learn to construct statechart diagrams
- To enumerate the differences and similarities between processes and threads
- To state how time and space constrains affect system modeling

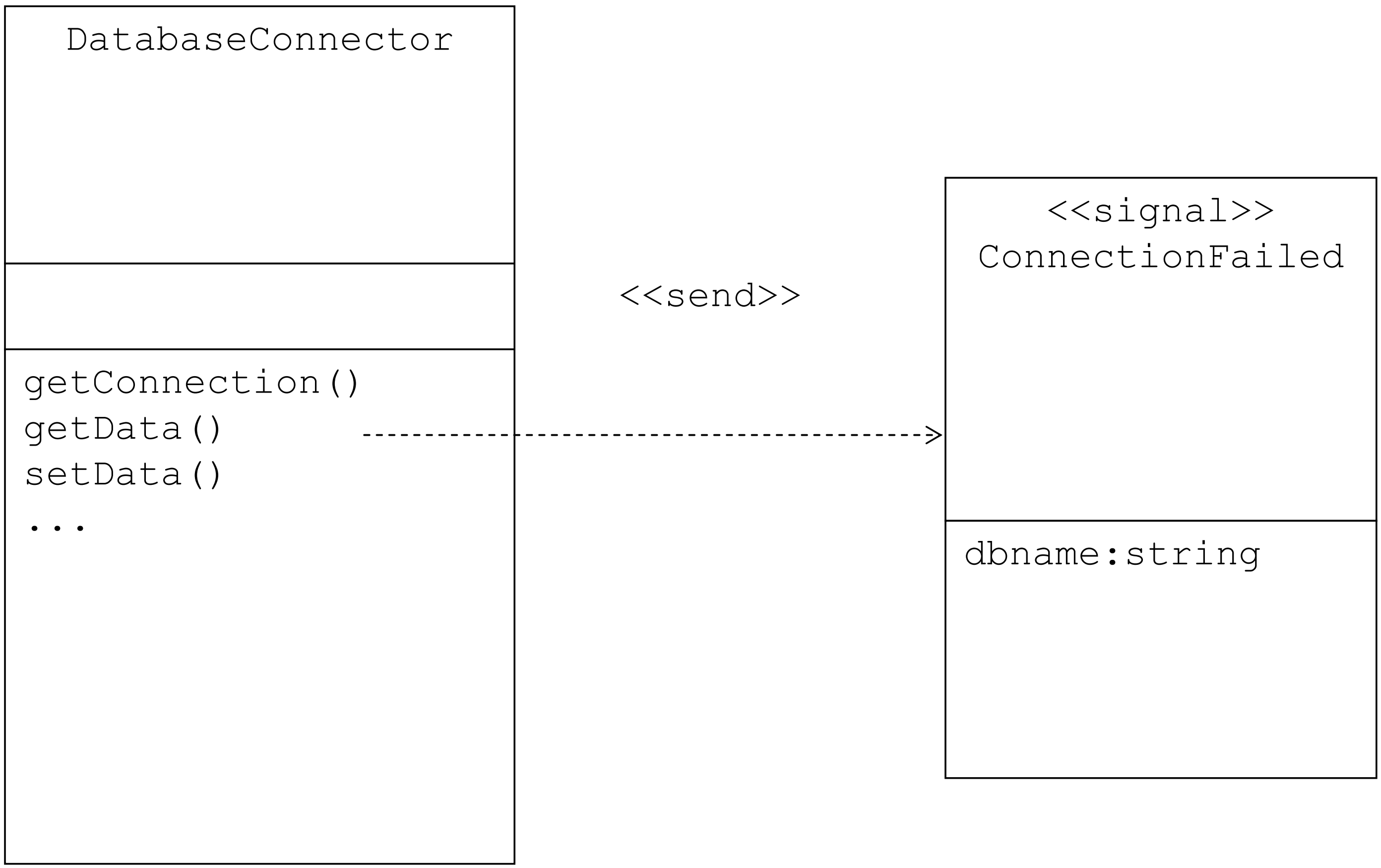
Events



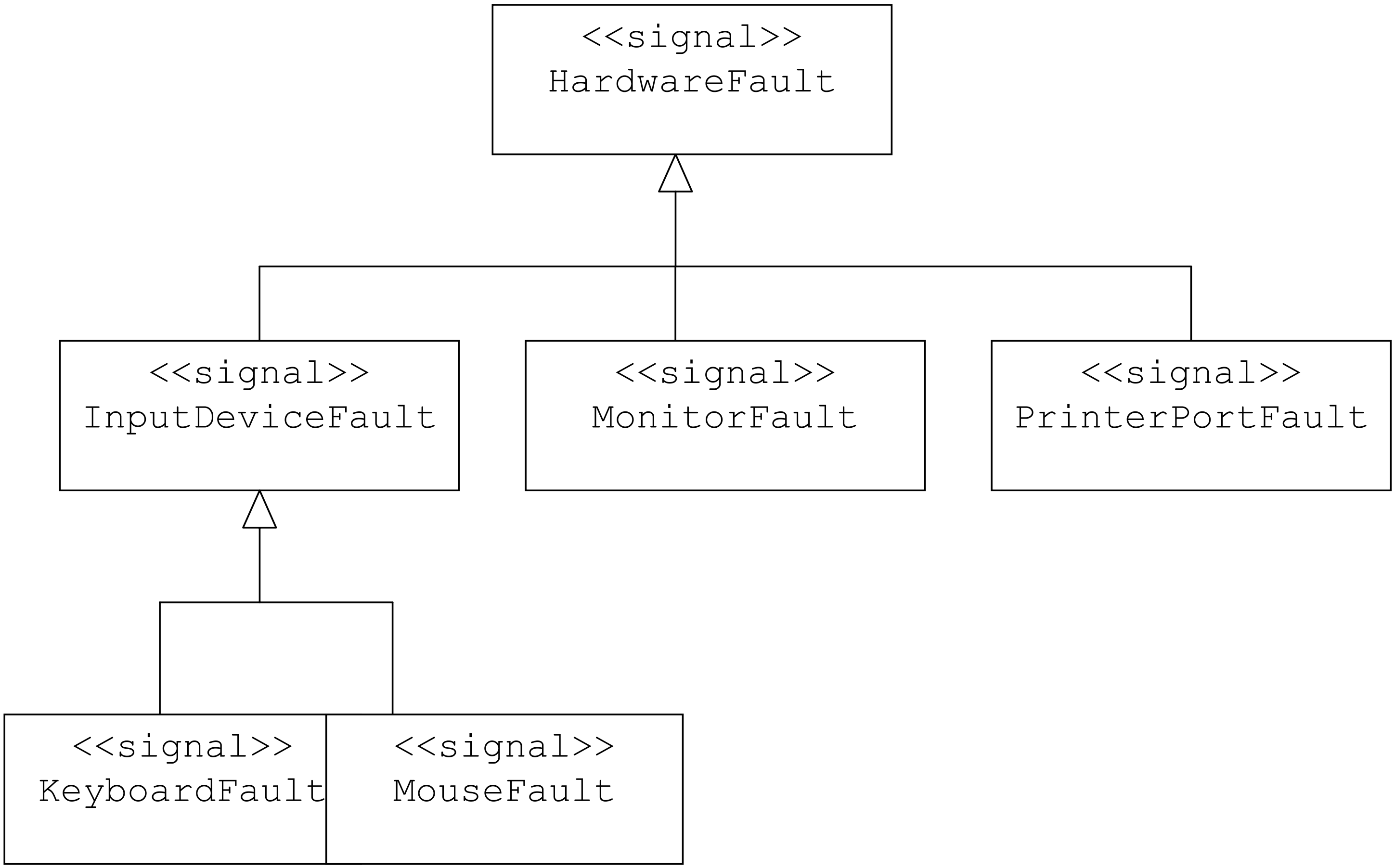
Events … continued

- *External events*
  - Events happening between the actors and the system are called external events
  - Inserting a card in an automated system is an example of an external event
- *Internal events*
  - Those events that pass between objects residing in the system are referred to as internal events
  - A floating point underflow exception is an example of an internal event
- The four kinds of events that can be modeled in the UML are signals, call events, time event and change event

Signals

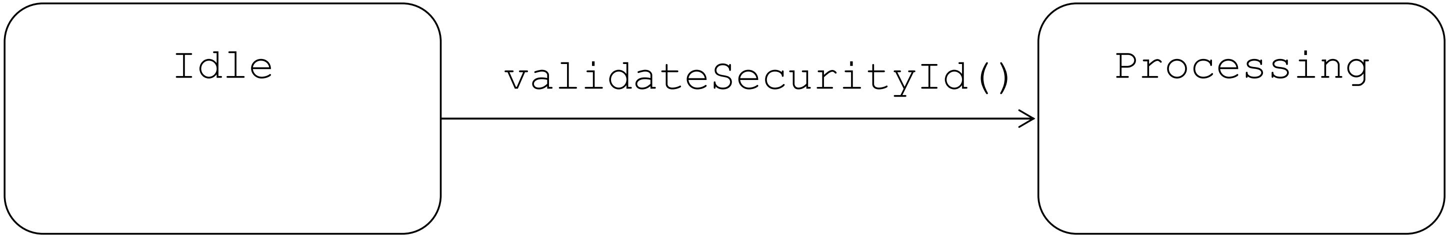

DatabaseConnector
getConnection()
getData()
setData()
...
<<send>>
<<signal>>
ConnectionFailed
dbname:string

Hierarchy of Signals

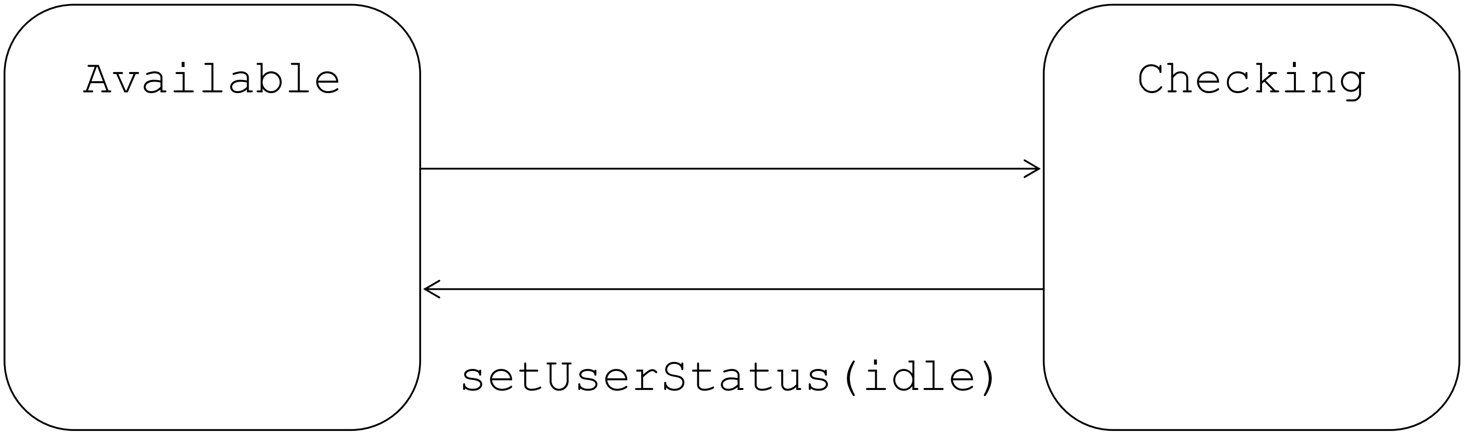


Call Events

- Call events represent the dispatch of an operation
- The dispatch results in a state transition
- Signals are asynchronous while call events typically are synchronous
- An object invokes an operation on another object that has a state
- The receiver object completes the operation and transition to a new state
- The control is returned back to the sender
- The sender waits till the receiver finishes the operation



Time Events



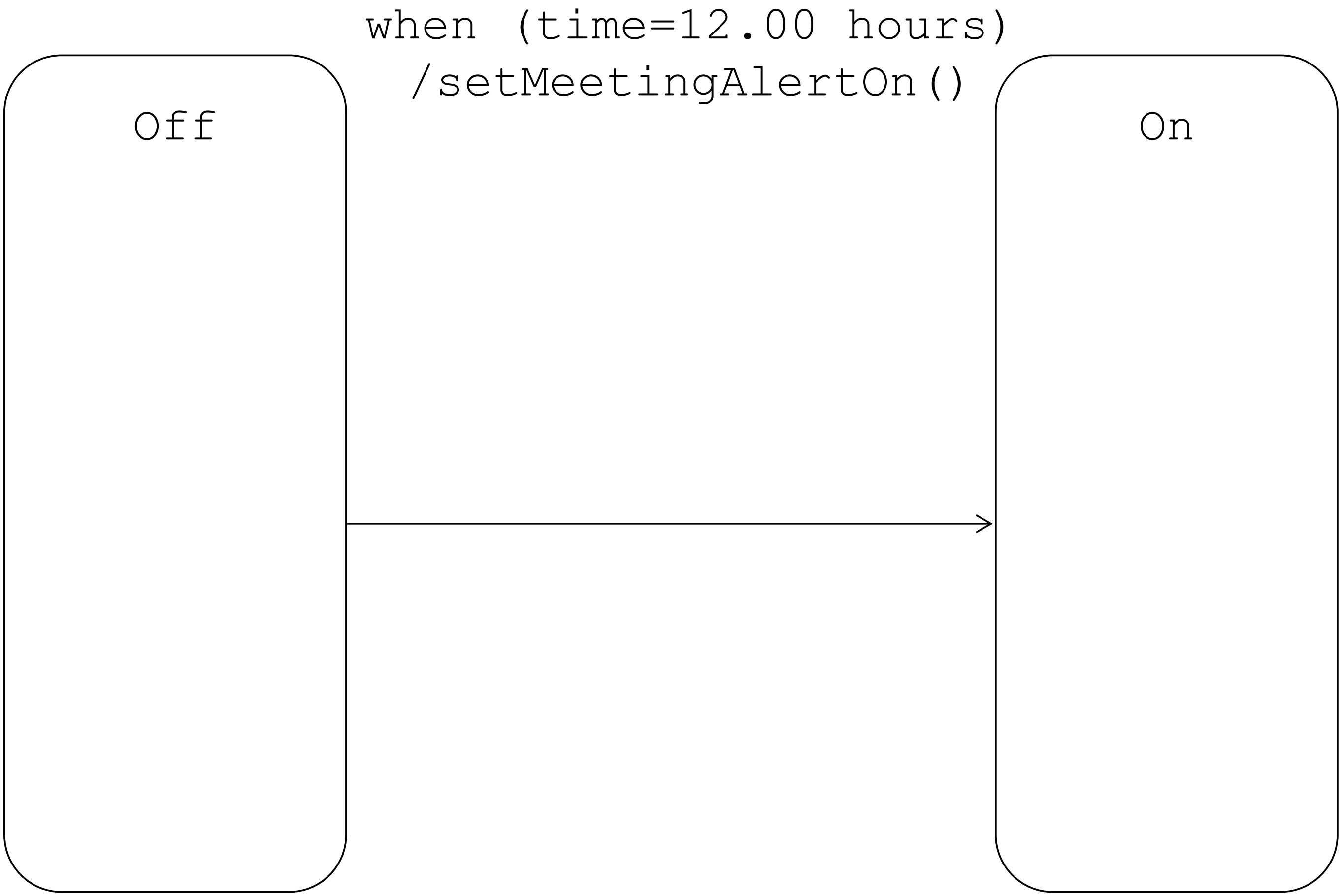
after(5 minutes
since no keystroke)
/checkUserStatus()

Available

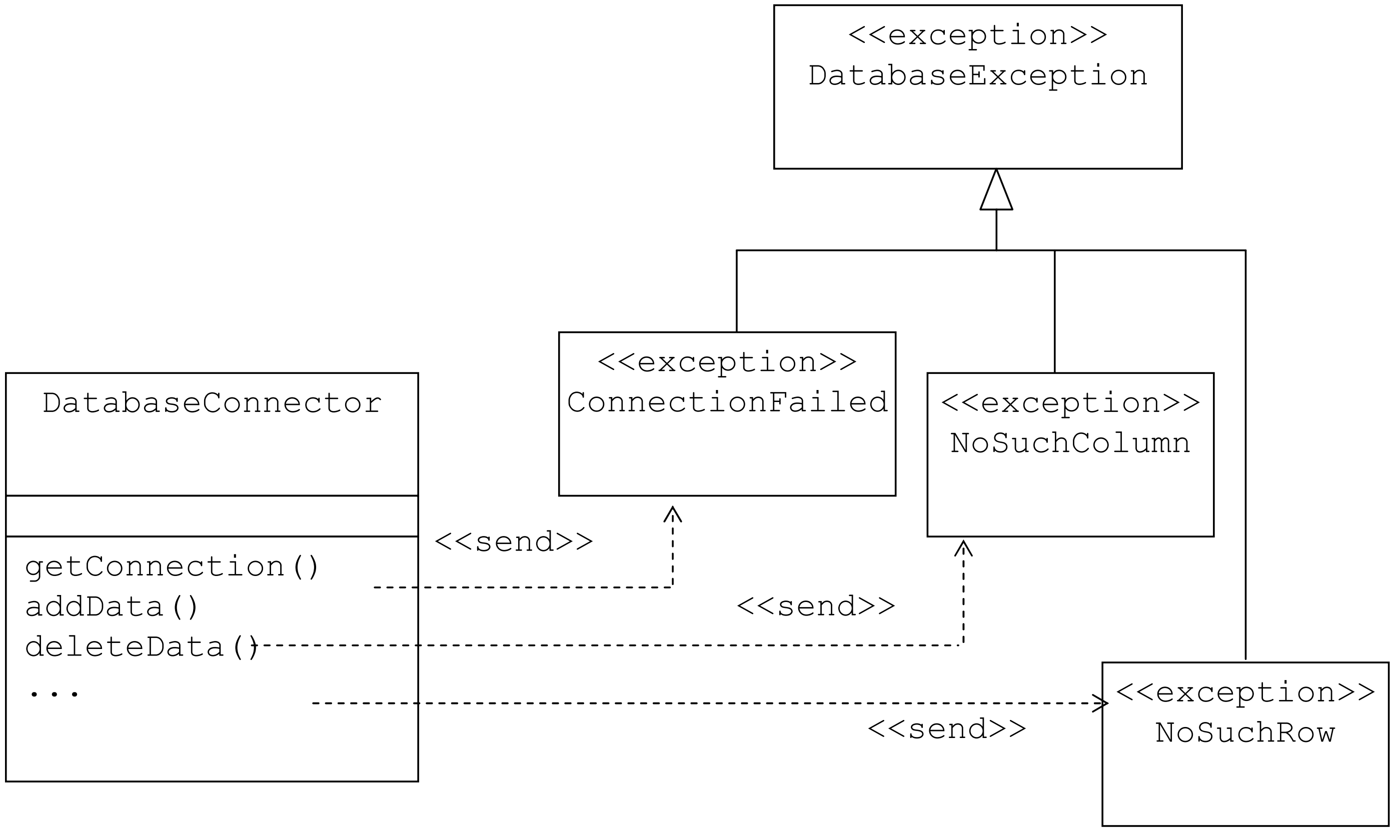
Checking

setUserStatus(idle)

Change Events



Exceptions can be Modeled With Signals



State Machines

- State machines allow us to model the behavior of an individual object
- State machines specify the sequence of states an object goes through during its lifetime as a result of events occurring
- This also includes the object's reaction to those events
- State machines can be used to model the behavior of an instance of a class or a use case
- Sometimes state machines are used to model the entire system
- Every object has a lifetime
- It comes into existence at some point in time and ceases to exist at some other point in time
- During its existence, it can undergo a number of changes
- State machines help us model these changes in a simple manner

State Machines ... continued

- The instances being modeled using state machines respond to the four kinds of events we discussed earlier
- When an event occurs, some activity will take place, and result in the instance transiting from one state to another
- An activity is a non-atomic execution within a state machine
- State machines can be visualized in two ways, as mentioned below
  - *Using activity diagrams* – In this the flow of control is emphasized from one activity to another
  - *Using statechart diagrams* – These diagrams show the various states the object goes through during its lifetime along with the transitions among those states

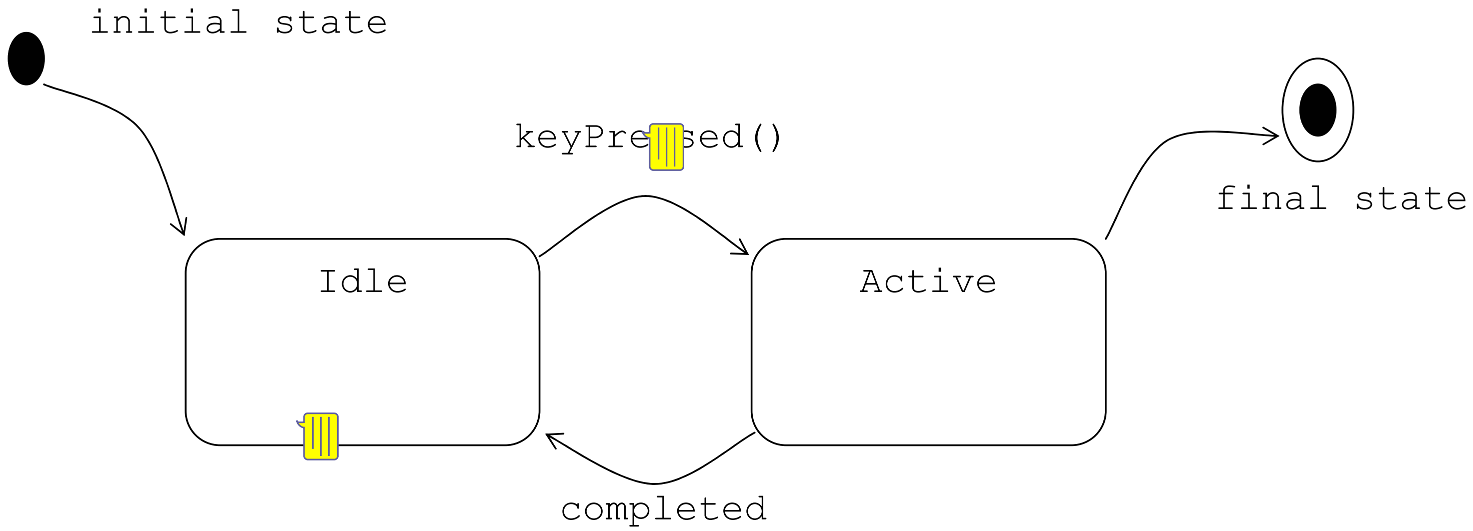
States

- A state is a specific condition during the lifetime of an object when it satisfies some condition, does some specific activity or waits for some events to occur
- A state of an object has various parts
  - *Name* – Every state has a name associated with it. If no name has been provided, it is an anonymous state
  - *Entry actions* – We can specify an action when an object is entering a state. For instance, we can set the action `setAlarm()` on entering a state
  - *Exit actions* – We can also specify an action when an object exits a state. For instance, we can set the action `clearAlarm()` on exiting a state
  - *Internal transitions* – Transitions that result in no distinct state change are called internal transitions
  - *Substates* – We can have states nested within other states. These nested states can be either sequential or concurrent substates
  - *Deferred events* – An object can defer the handling of events to another state

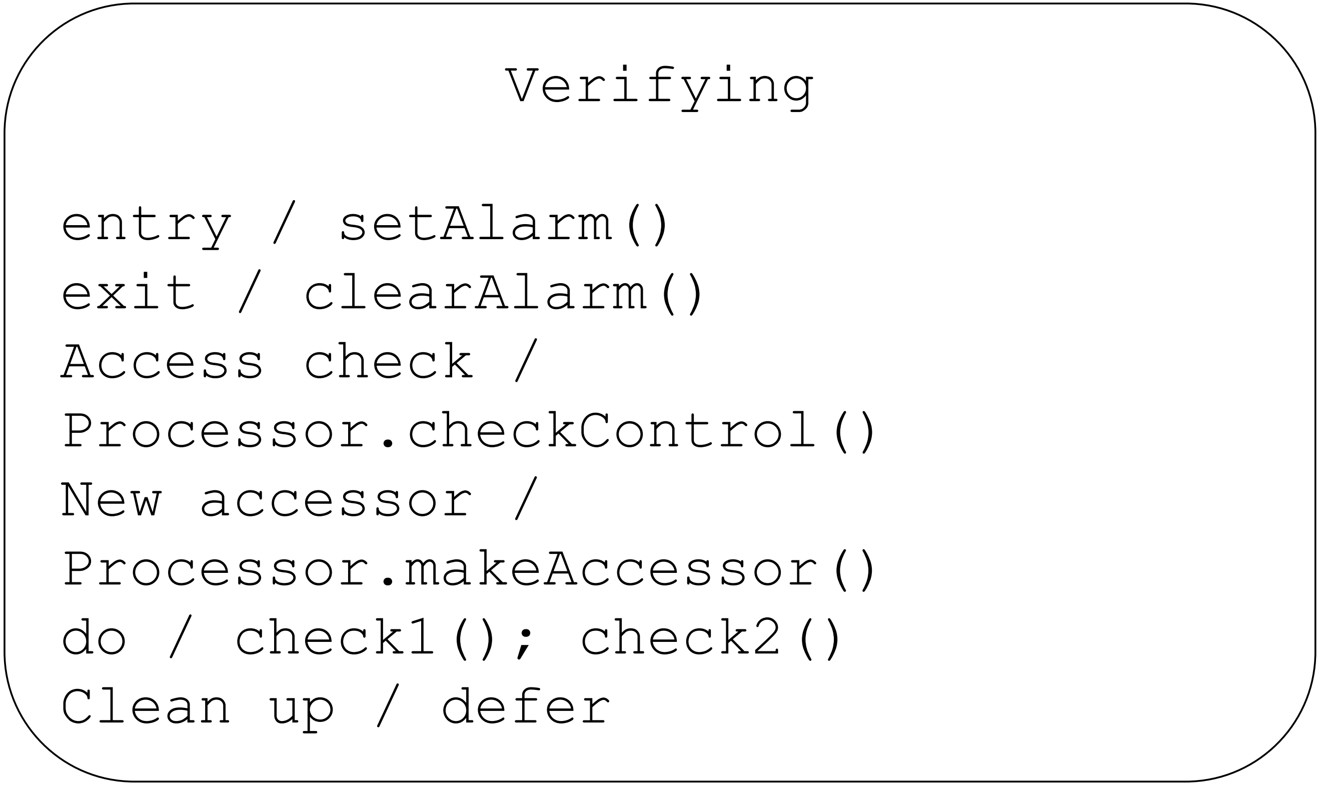
Transitions

- A transition is a relationship between two states. On the occurrence of an event, the object enters the next state in the relationship
- The occurrence of the events results in some specified action
- The transition is said to *fire* when an object moves from one state to another
- A transition has five parts
  - Source state
  - Event trigger
  - Guard condition
  - Action
  - Target state

States and Transitions



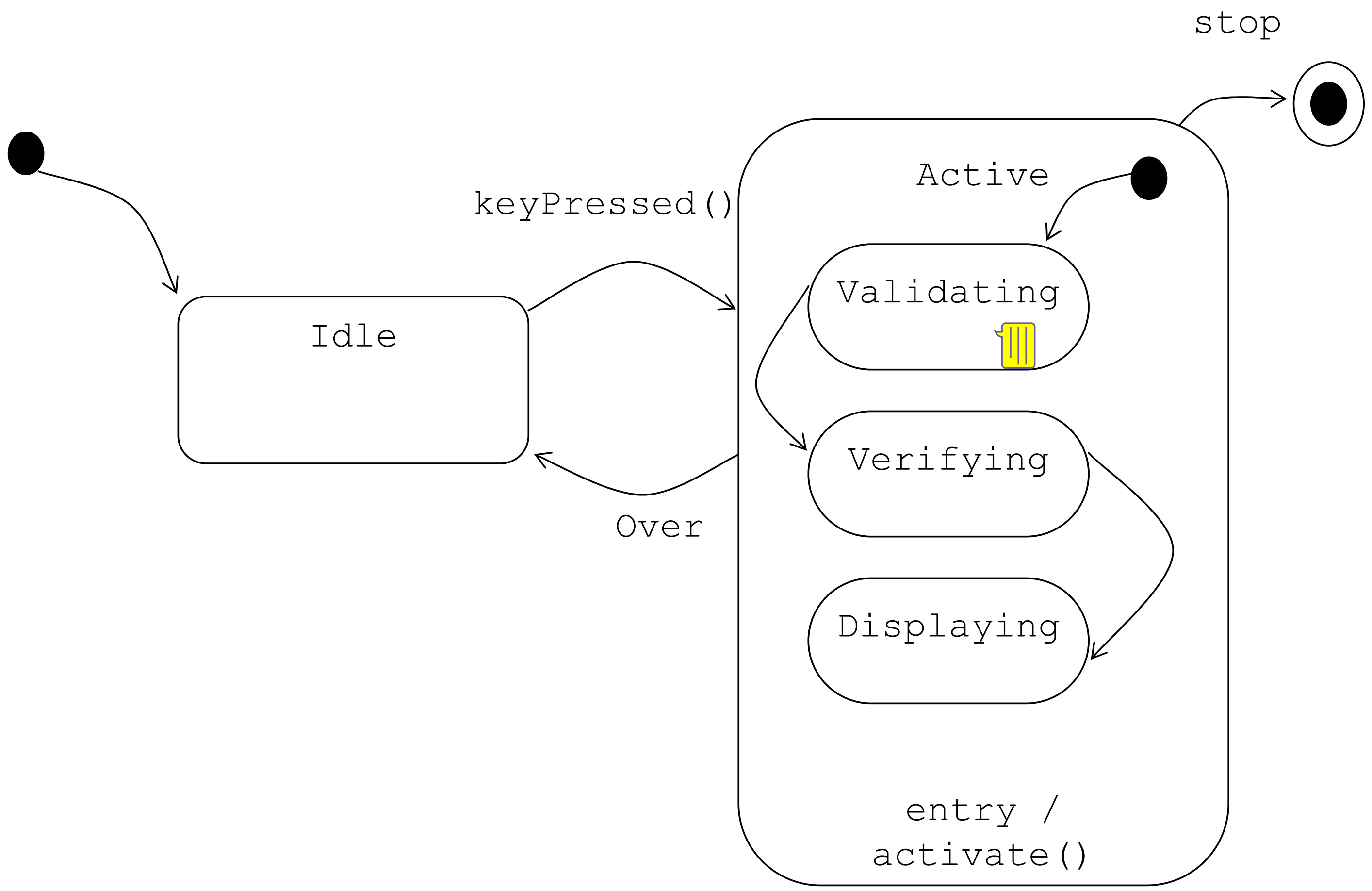
States and Transitions ... continued



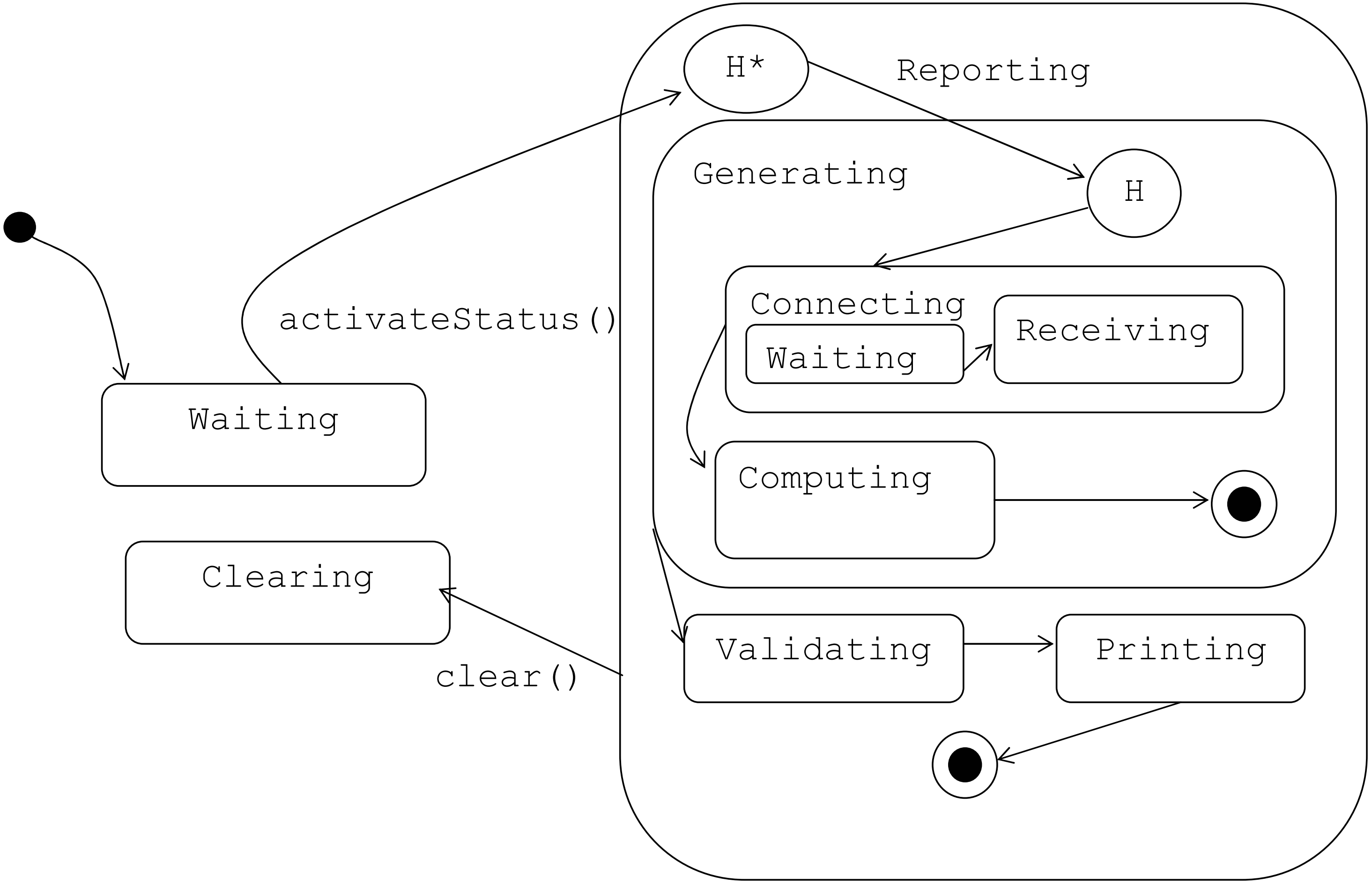
Verifying

entry / setAlarm()
exit / clearAlarm()
Access check /
Processor.checkControl()
New accessor /
Processor.makeAccessor()
do / check1(); check2()
Clean up / defer

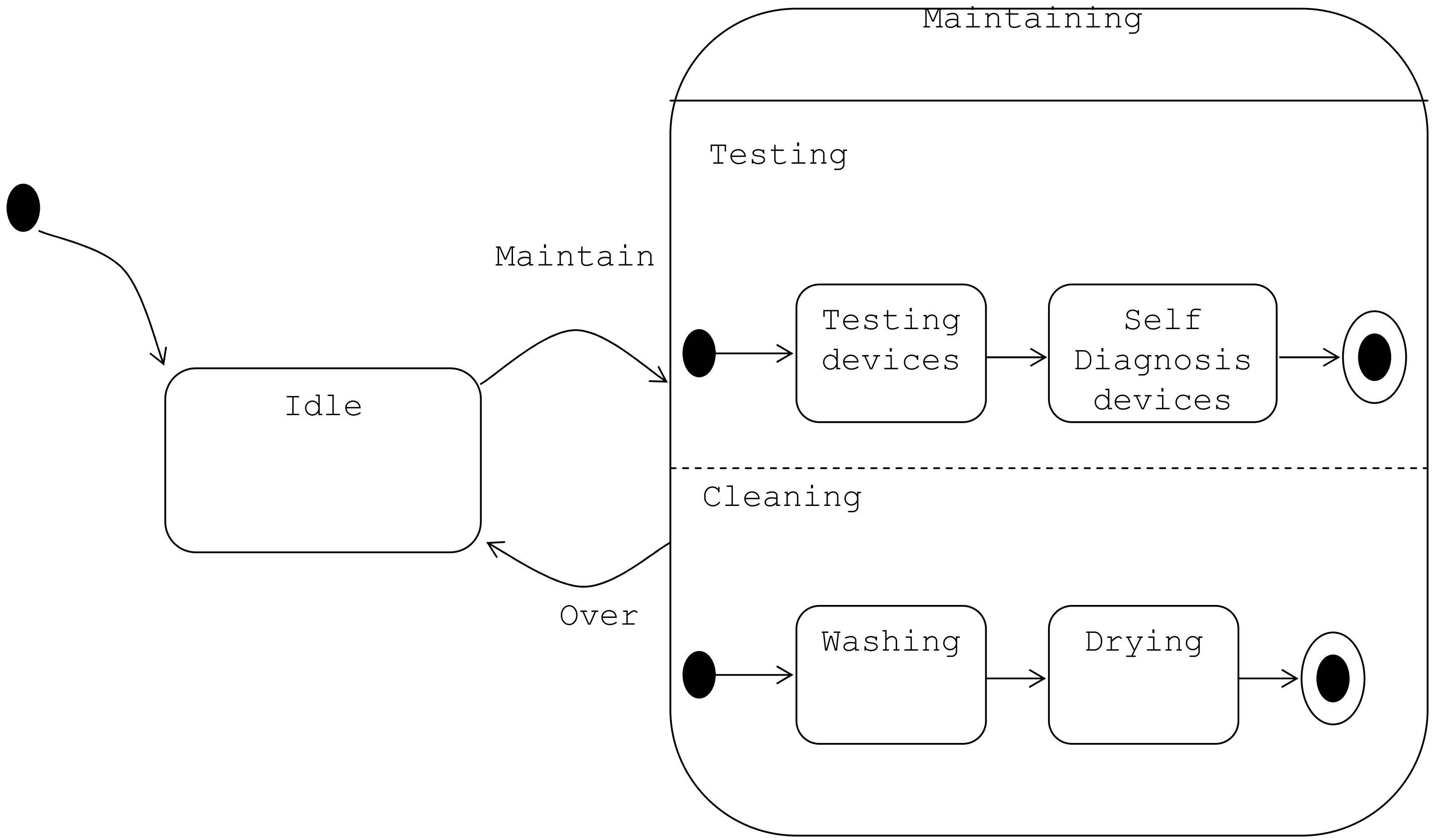
Sequential Substates



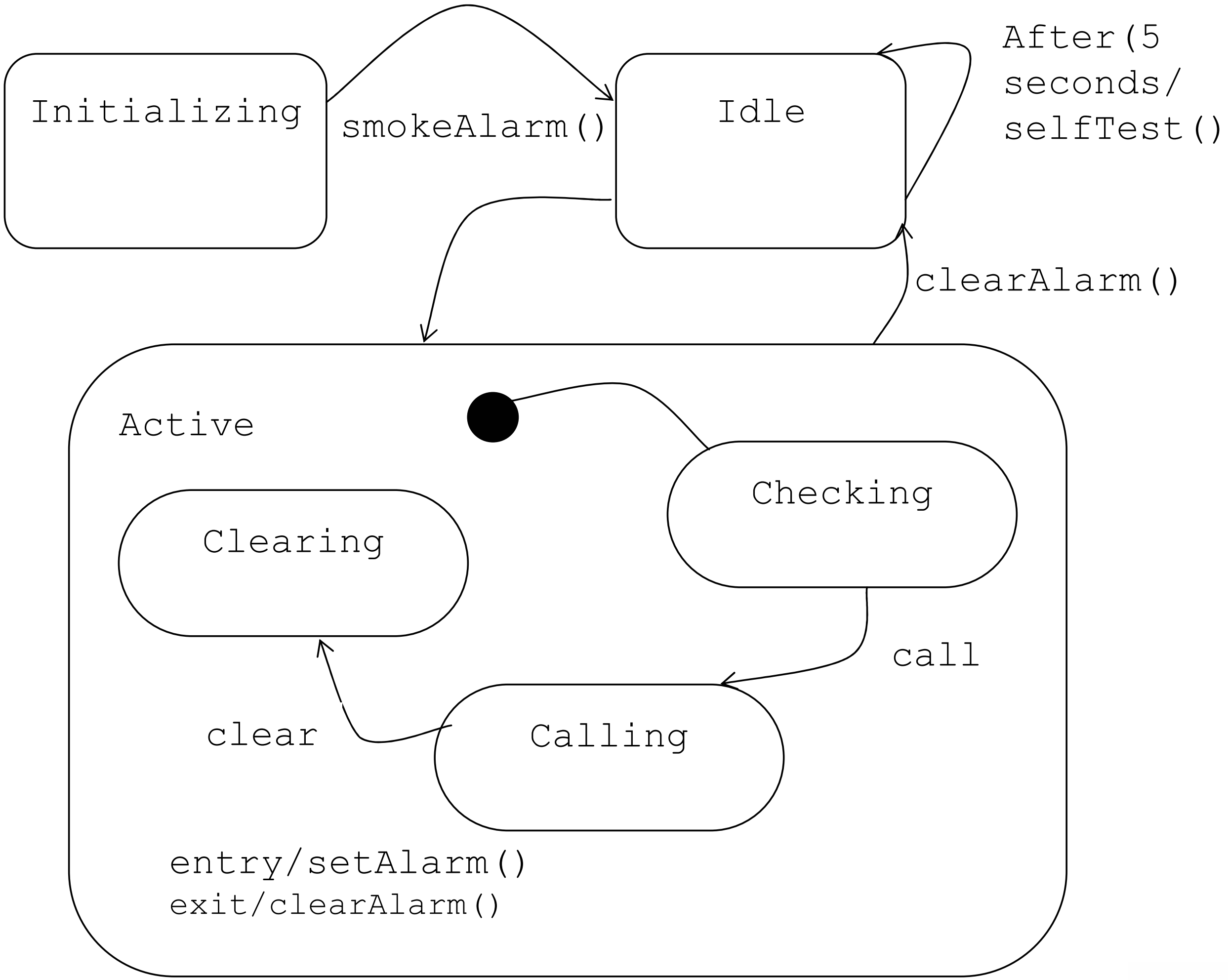
History States



Concurrent Substates

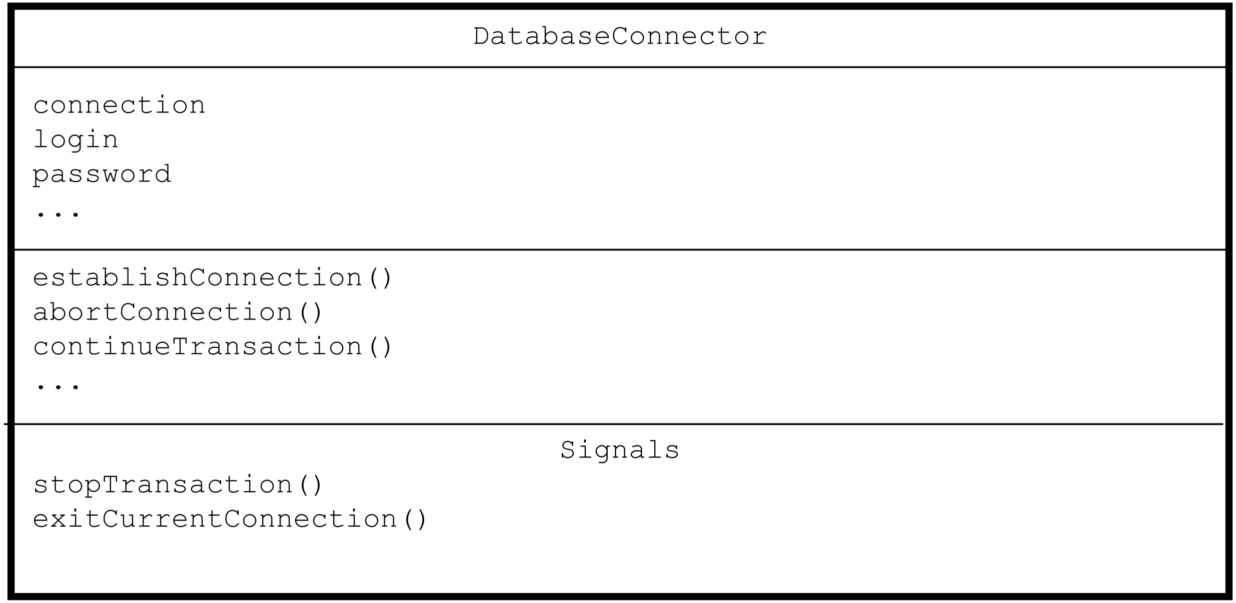


Statechart Diagrams



Active Classes

- A class is referred to as an active class when it is capable of initiating control activity
- An instance of an active class is an active object
- Typically an active object is either a process or a thread as they are capable of initiating control activity



DatabaseConnector

connection
login
password
...

establishConnection()
abortConnection()
continueTransaction()
...

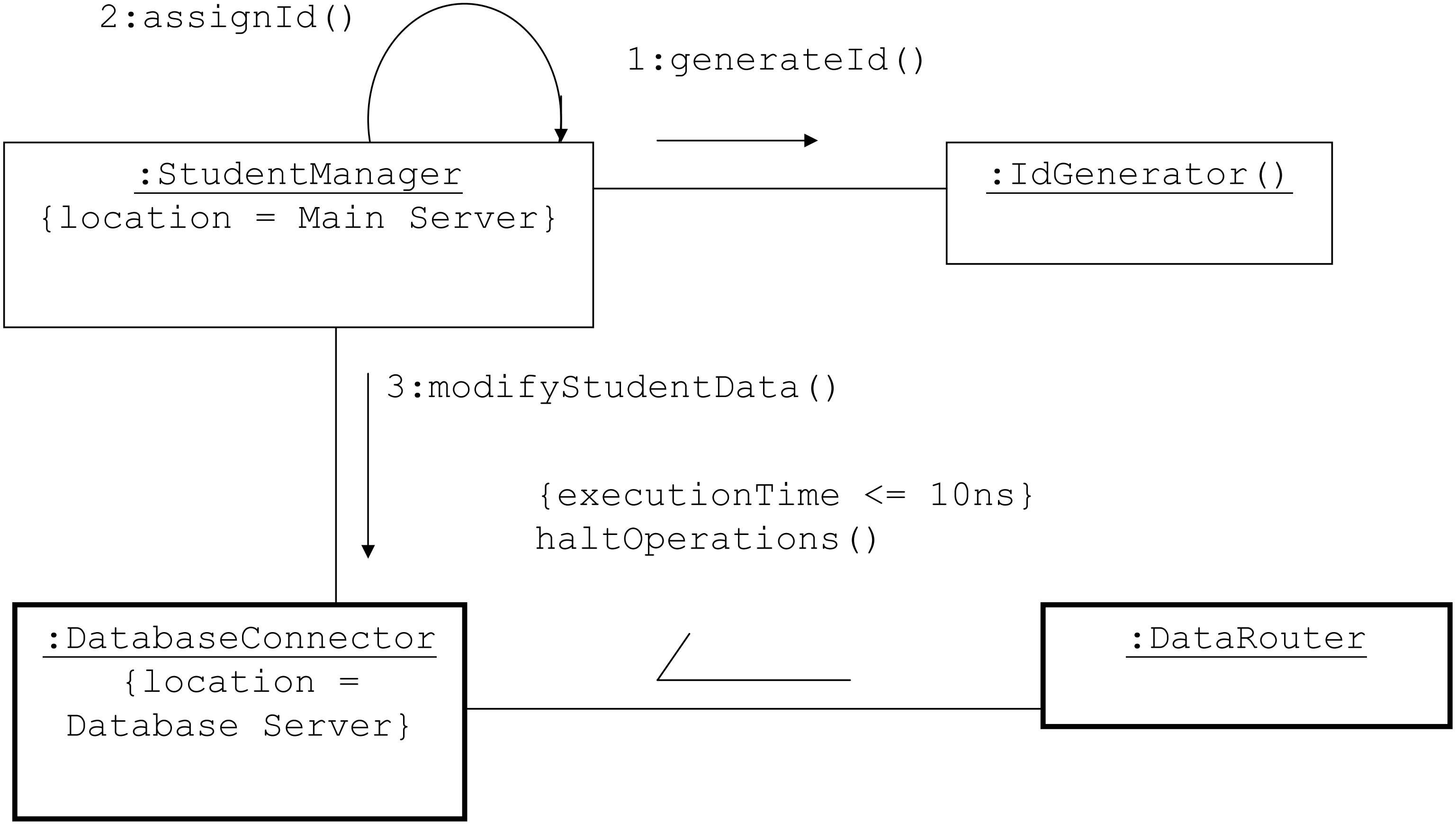
Signals

stopTransaction()
exitCurrentConnection()

Synchronization

- When an object is the receiver of more than one flow of control at the same instance, there arises the difficulty of handling those multiple flows of control
- The state of the object needs to be maintained. More than one flow can interfere with another and can cause corruption to the state of the object
- This is referred to as the *mutual exclusion* problem
- There are three ways in which the problem of mutual exclusion can be handled
  - Sequential
  - Guarded
  - Concurrent

Time and Space



2:assignId()

1:generateId()

:StudentManager
{location = Main Server}

:IdGenerator()

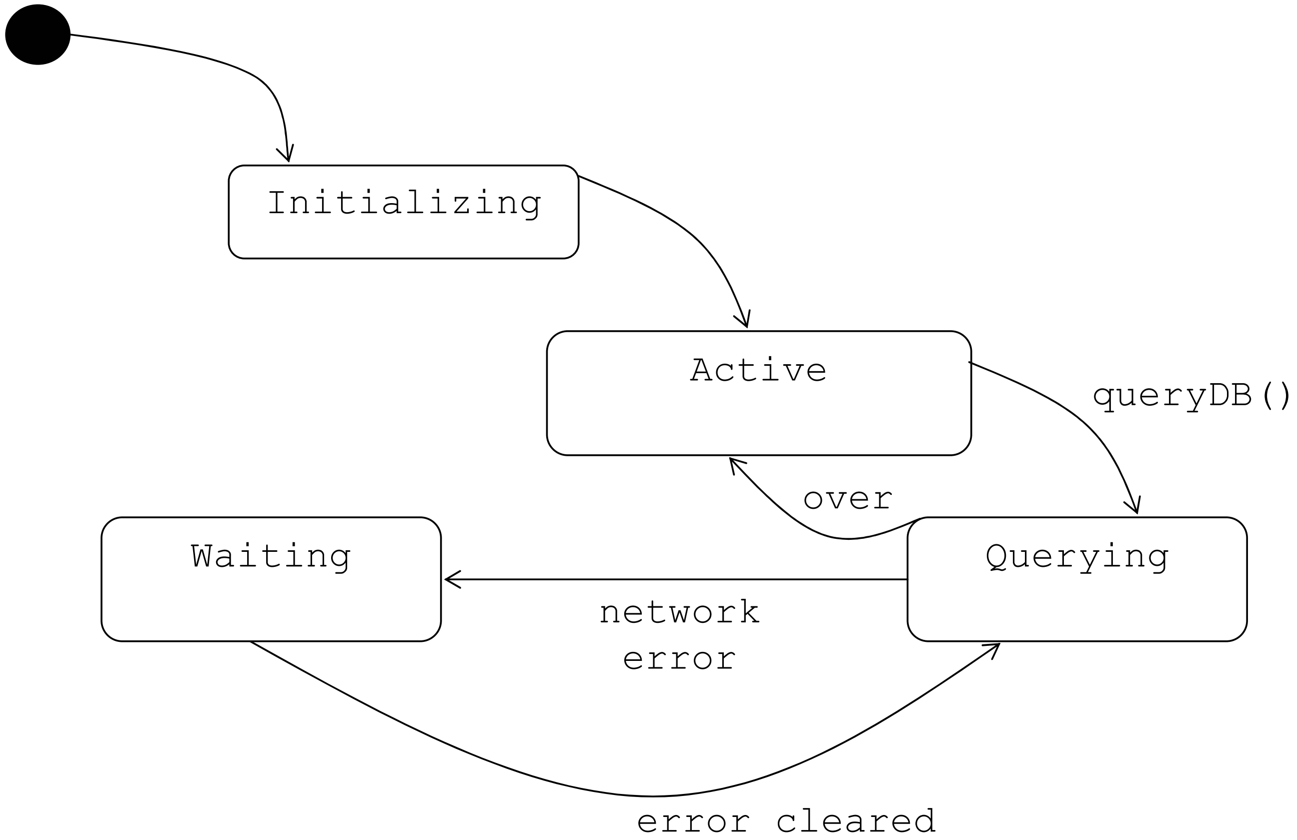
3:modifyStudentData()

{executionTime <= 10ns}
haltOperations()

:DatabaseConnector
{location =
Database Server}

:DataRouter

Statechart Diagram for the Class Databaseconnector



Summary

- We discussed events and signals
- We learnt what are state machines
- We learnt to construct statechart diagrams
- We enumerated the difference and similarities between processes and threads
- We stated how time and space constrains affect system modeling

Unit 3

Behavioral Modeling Lab.

Lab. Exercises

Unit 4

Architectural Modeling

Learning Objectives

- To learn about components, deployment and collaborations
- To discuss about component diagrams
- To enumerate the need for deployment diagrams
- To understand the difference between systems and subsystems

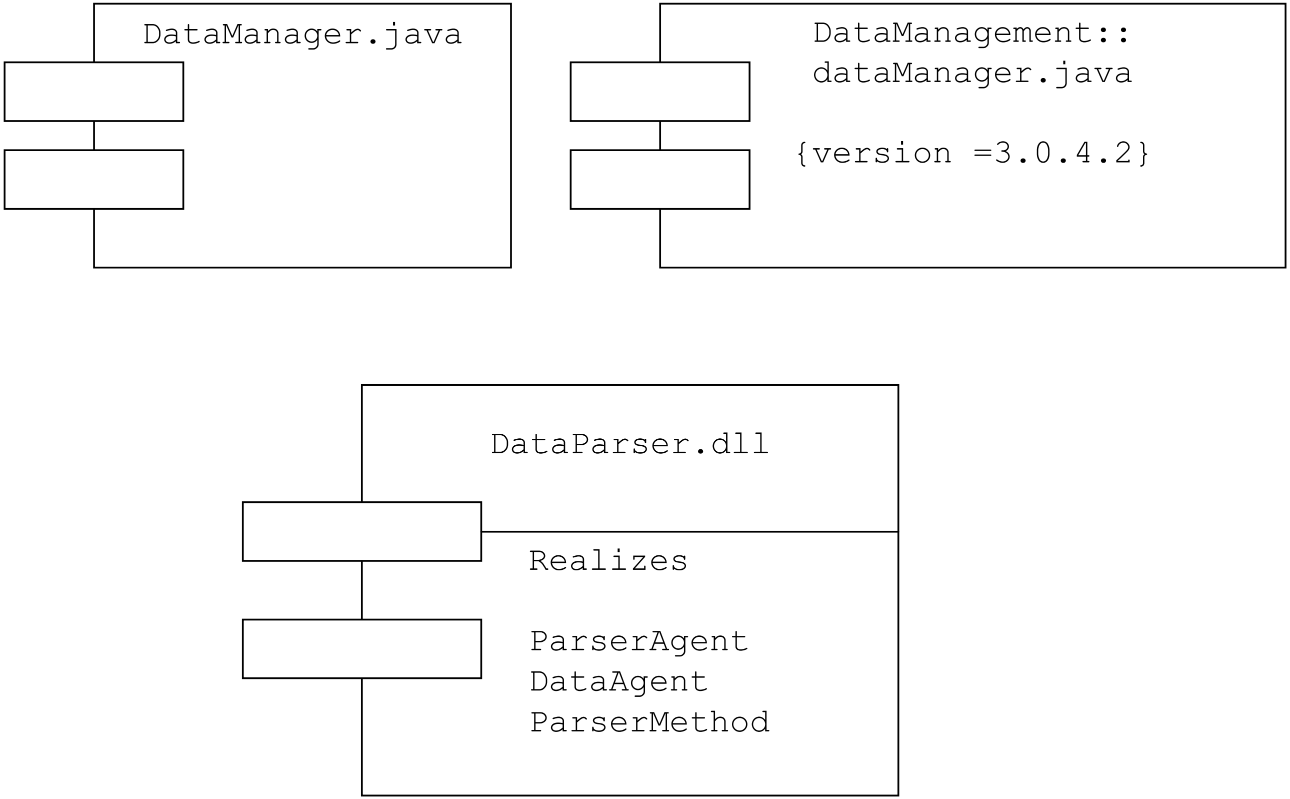
An Introduction

- Structural modeling deals with abstractions, instances of those abstractions and their relationships
- Behavioral modeling helps us in understanding the flows of control between two or more objects modeled using different diagrams
- Architectural diagrams enable us to model executables, libraries, tables, files, source code, physical nodes etc
- The two diagrams that enable us to provide architectural perspective to a system are
  - component diagrams
  - deployment diagrams

Components

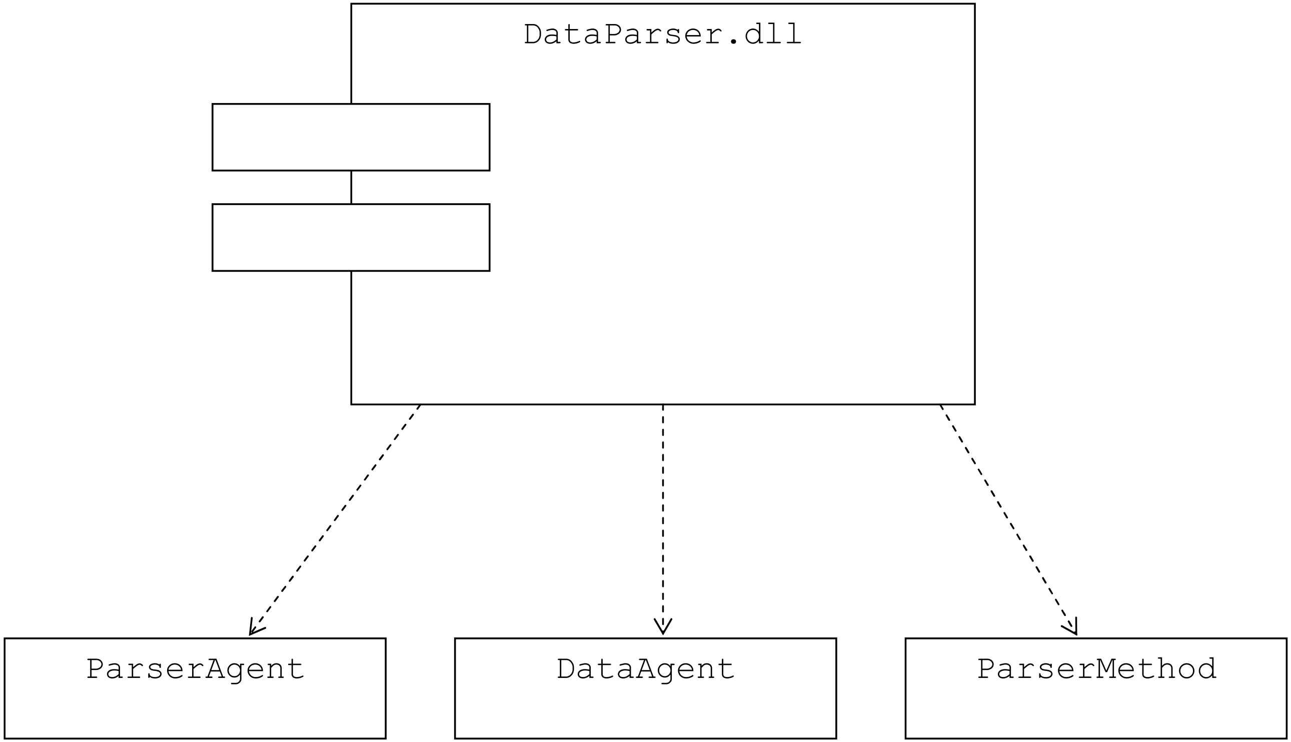
- A component is a physical part of a system
- There are high-level physical components that may or may not be equivalent to the other smaller ones that we create for our applications
- A component in the UML is depicted as a rectangular box with tabs
- Every component has a name, which can either be a simple name or a path name
- We can have compartments and tagged values as we did for classes

Components ... continued



Components and Classes

- Classes represent logical abstractions
- Components represent the physical parts of a system

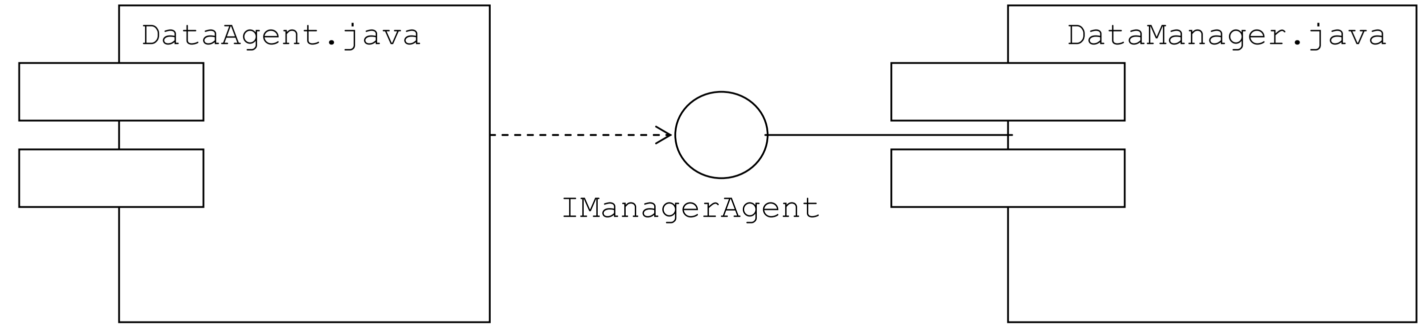


Components and Interfaces

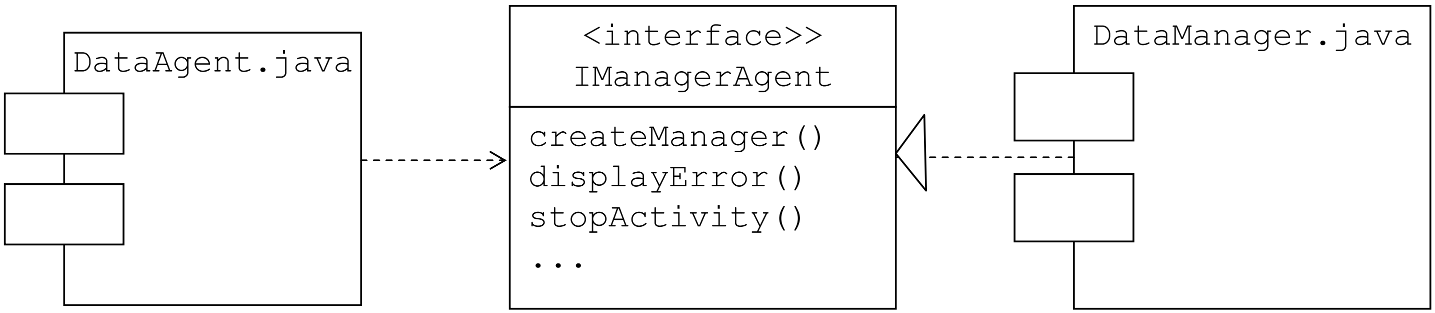
- An interface is a set of operations that specifies the services offered by a class or a component
- Interfaces bridge components and classes
- An interface can be realized both by a component and a class
- The relationship between a component and an interface can be shown in two ways
  - Iconic form
  - Expanded form
- The interface a component realizes is called an *export* interface
- The interface that a component uses is called an *import* interface
- A component may both import and export interfaces

Components and Interfaces … continued

- Iconic Form



- Expanded form



Kinds of Components

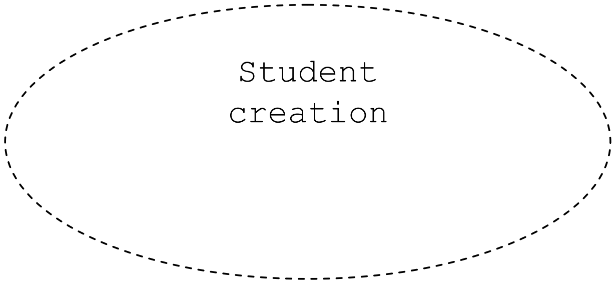
- There are three kinds of components
  - Deployment: The components used to form an executable system
  - Work products: These are the left over parts of a development process
  - Execution: These components are created as a consequence of an executing system

Components and Stereotypes

- `executable` – To specify that a component can be executed on a node
- `library` – To specify a library, both static and dynamic
- `table` –To specify that the component is a database table
- `file` – To specify either a source code or data file
- `document` – To specify that the component represents a document

Collaborations

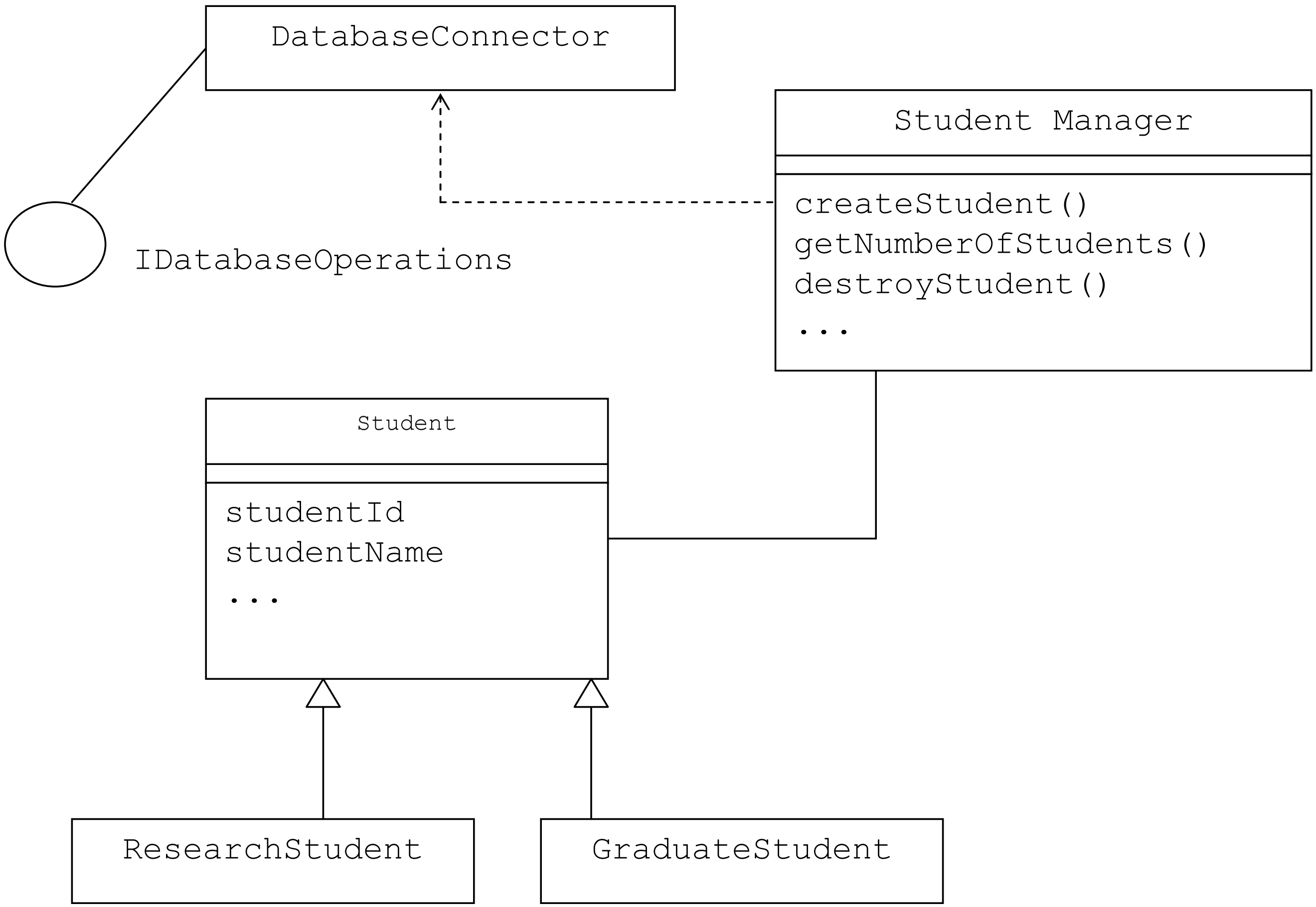
- Collaborations are a set of classes, interfaces, nodes, components and use cases
- These elements together provide a behavior that is bigger than the sum of all the parts
- A collaboration also specifies how classifiers and associations realize an element



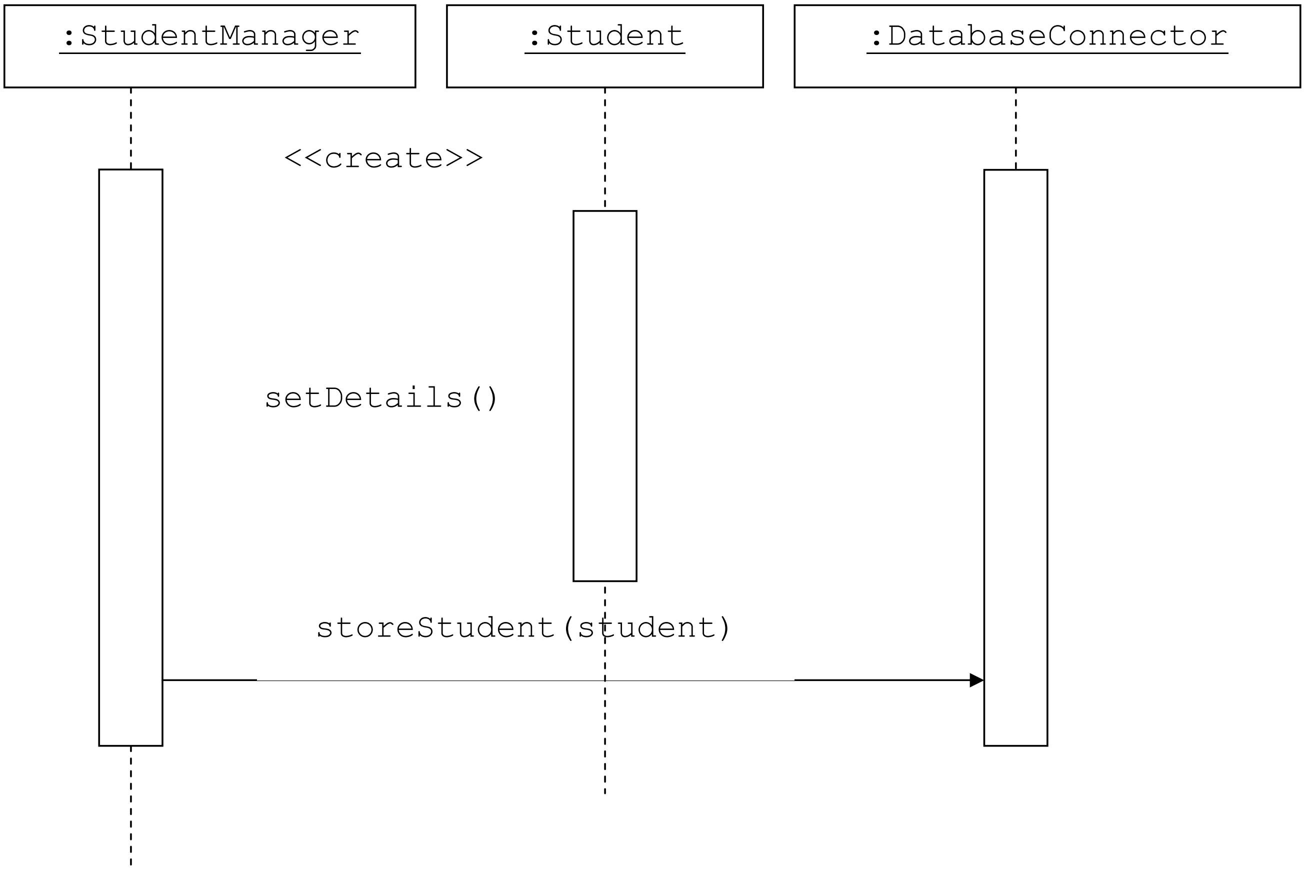
Collaborations … continued

- Collaborations deal with two aspects
- *Structural Aspects:* These include
  - Classes
  - Interfaces
  - Nodes
  - Components
  - use cases
- *Behavioral Aspects*: These include
  - the dynamic aspects of how the structural elements interact with one another
  - the behavioral aspects are rendered as an interaction diagram

Collaborations – Structural Aspect

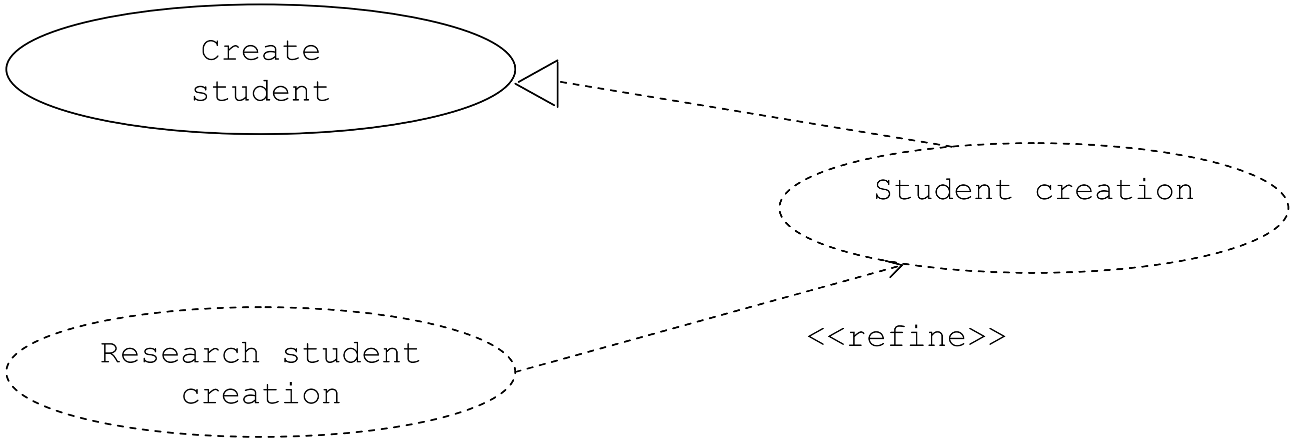


Collaborations – Behavioral Aspect



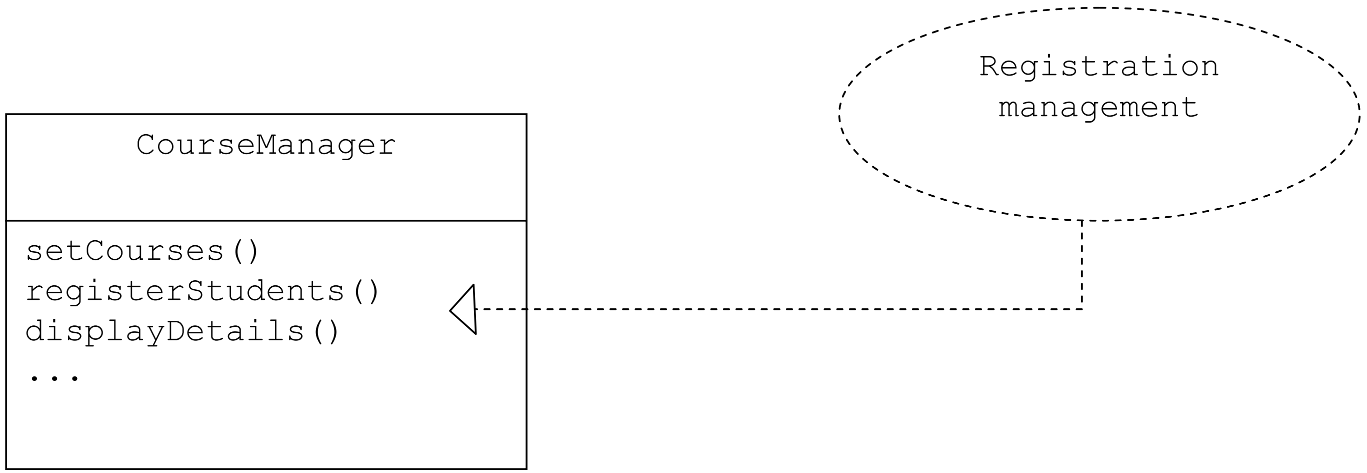
Collaborations and Use Cases

- Use cases are derived and then designed during the process of understanding the system
- During the final implementation of the system, the use cases are realized as collaborations
- Collaborations realized through use cases can further be depicted using the structural and behavioral modeling



Collaborations and Implementation of an Operation

- An operation can be realized by a collaboration using the same relationship



Patterns

- We understand patterns to mean a provider of a common solution to a common problem for a given context
- The three ingredients of patterns are
  - Context
  - Problem
  - Solution
- For any given context, we do not tackle the solution to the problem by inventing a new solution that is distinct from existing ones
- That such a problem existed earlier to which a solution has been provided would be known
- The essence of the existing solution would be reused to provide a new solution

Patterns … continued

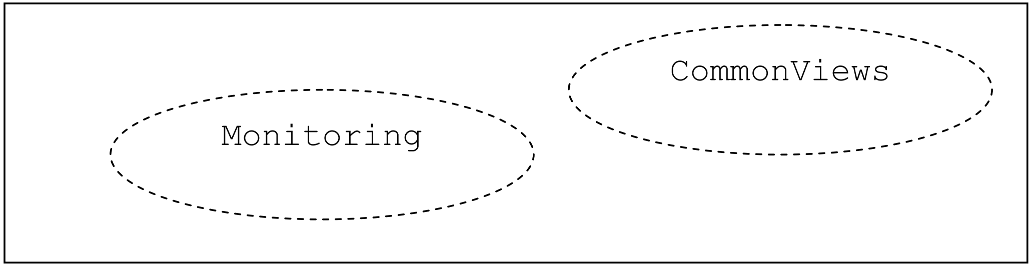
- The example on construction of a building provides us two views
  - The external view of the building could represent a Victorian-style house, a Roman-style house or a Greco-Roman-style house
  - The internal view of the building shows the efforts involved in designing the house which includes how the load bearing is done for the walls and how the roof are supported
- Patterns help us in visualizing, specifying, constructing and documenting the pieces of a software system
- We can perform both forward and reverse engineering using patterns.
- Patterns are of two kinds, viz. architectural and design patterns

Design Patterns

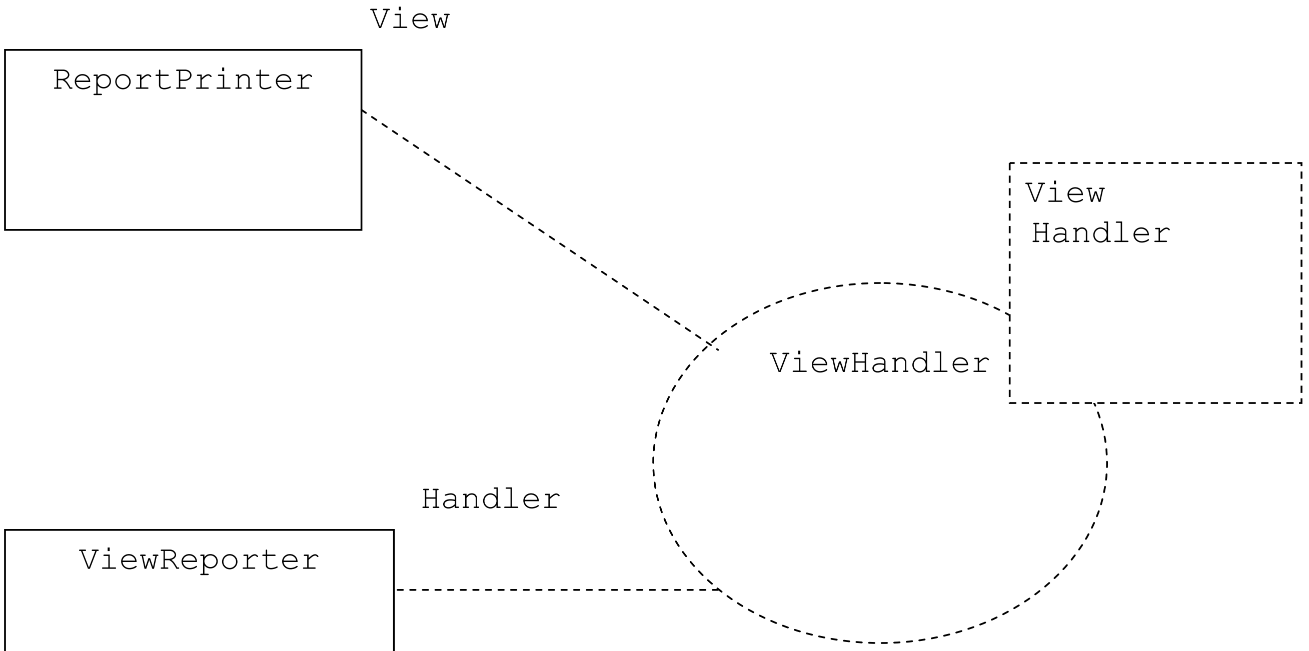
- A design pattern describes a commonly recurring structure of the components in communication
- It solves a general design problem for a particular context
- Design patterns are smaller in scale than architectural patterns but higher than programming language-specific idioms
- Some common examples of design patterns are
  - Whole-Part
  - Master-Slave
  - Proxy
  - Command Processor
  - View Handler
  - Forwarder-Receiver
  - Client-Dispatcher-Server
  - Publisher-Subscriber

Mechanisms

- Mechanism is a design pattern that applies to a society of classes
- It either
  - names a set of abstractions, in the form of collaborations that work together to carry out some common behavior
  - name a template for a set of abstractions, in the form of parameterized collaborations that work together to carry out some common behavior



Mechanisms ... continued

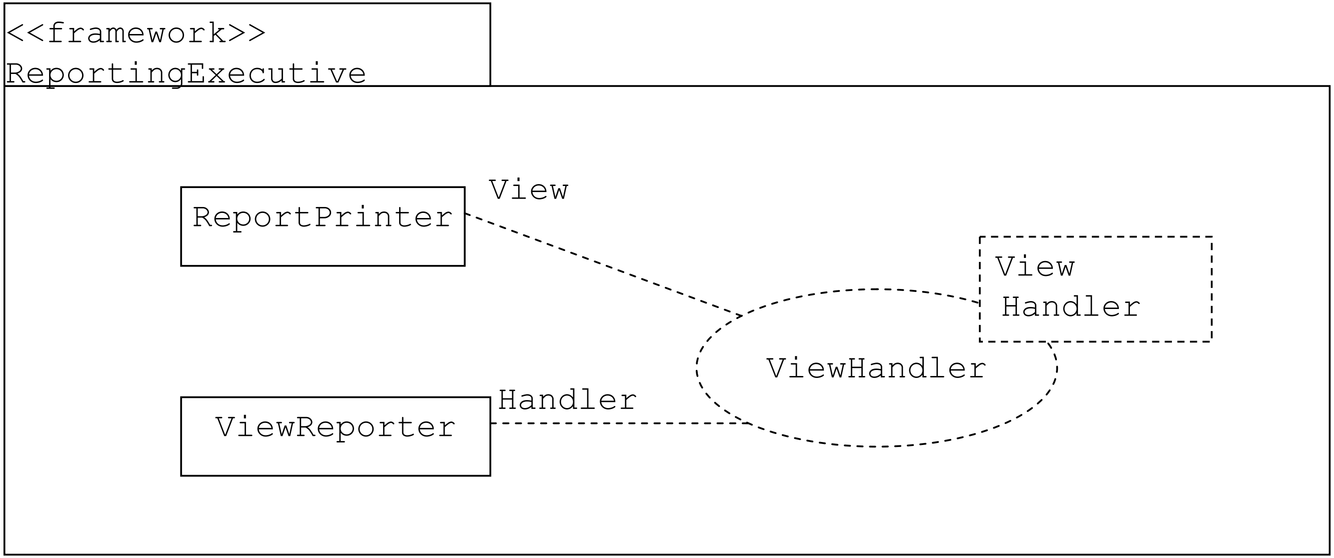


Architectural Patterns

- Architectural patterns express fundamental structural organization in larger chunks than design patterns
- Some of the architectural patterns are
  - Layers
  - Pipes and Filters
  - Blackboard
  - Broker
  - Model-View-Controller
  - Presentation-Abstraction-Control
  - Microkernel
  - Reflection

Frameworks

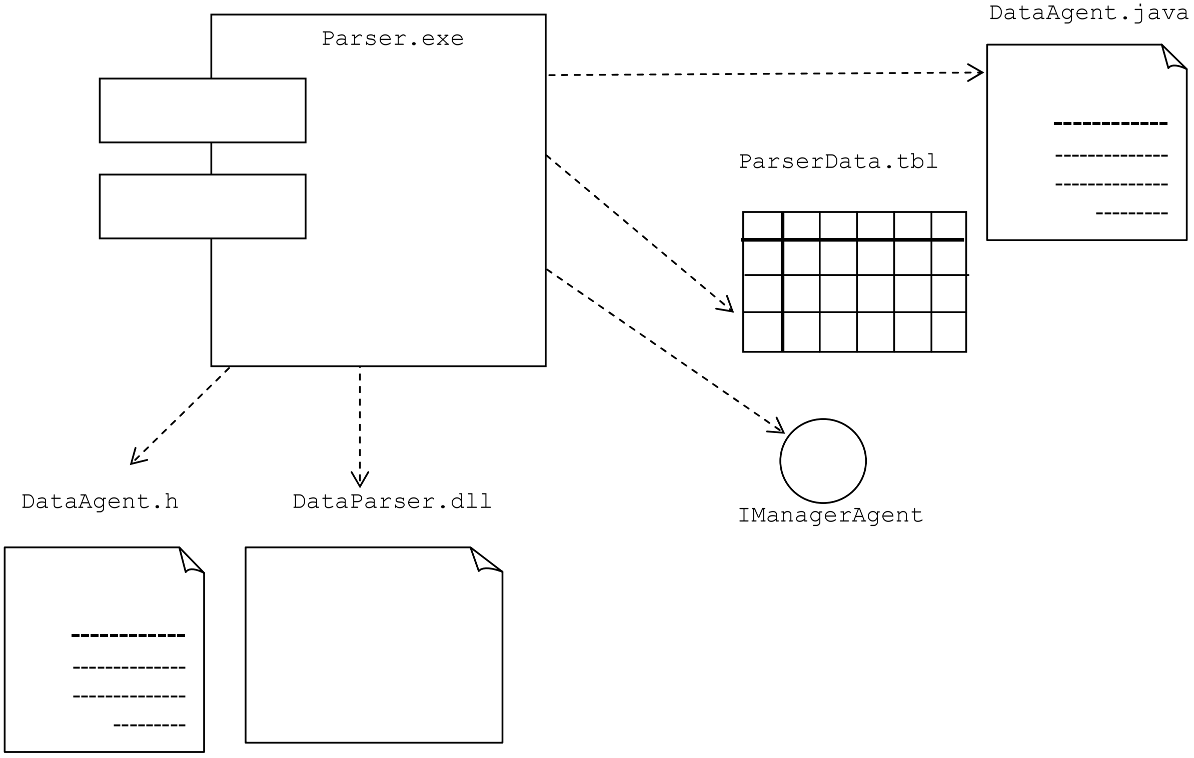
- A framework provides a template that is reusable and extensible for an application within a domain
- A framework is a kind of micro-architecture encompassing a set of mechanisms
- A framework is at a higher level than a mechanism
- In the UML, a framework is modeled as a stereotyped package. The package reveals mechanisms that live within the framework



Component Diagrams

- The two diagrams to represent the physical aspects of a system are
  - Component diagrams
  - Deployment diagrams
- Component diagrams are used to model the static implementation of physical elements residing on a node, such as executables, libraries, tables, files, etc.
- Component diagrams are also useful for building executable systems both through forward and reverse engineering

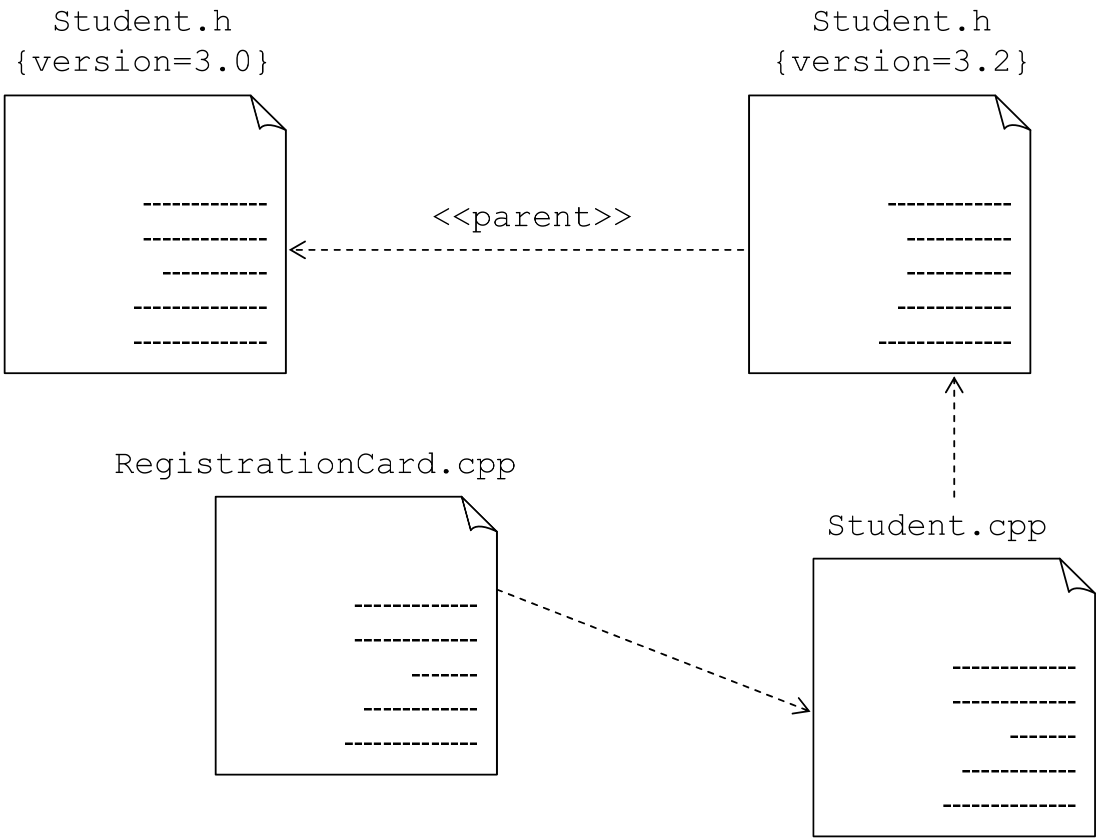
Component Diagrams



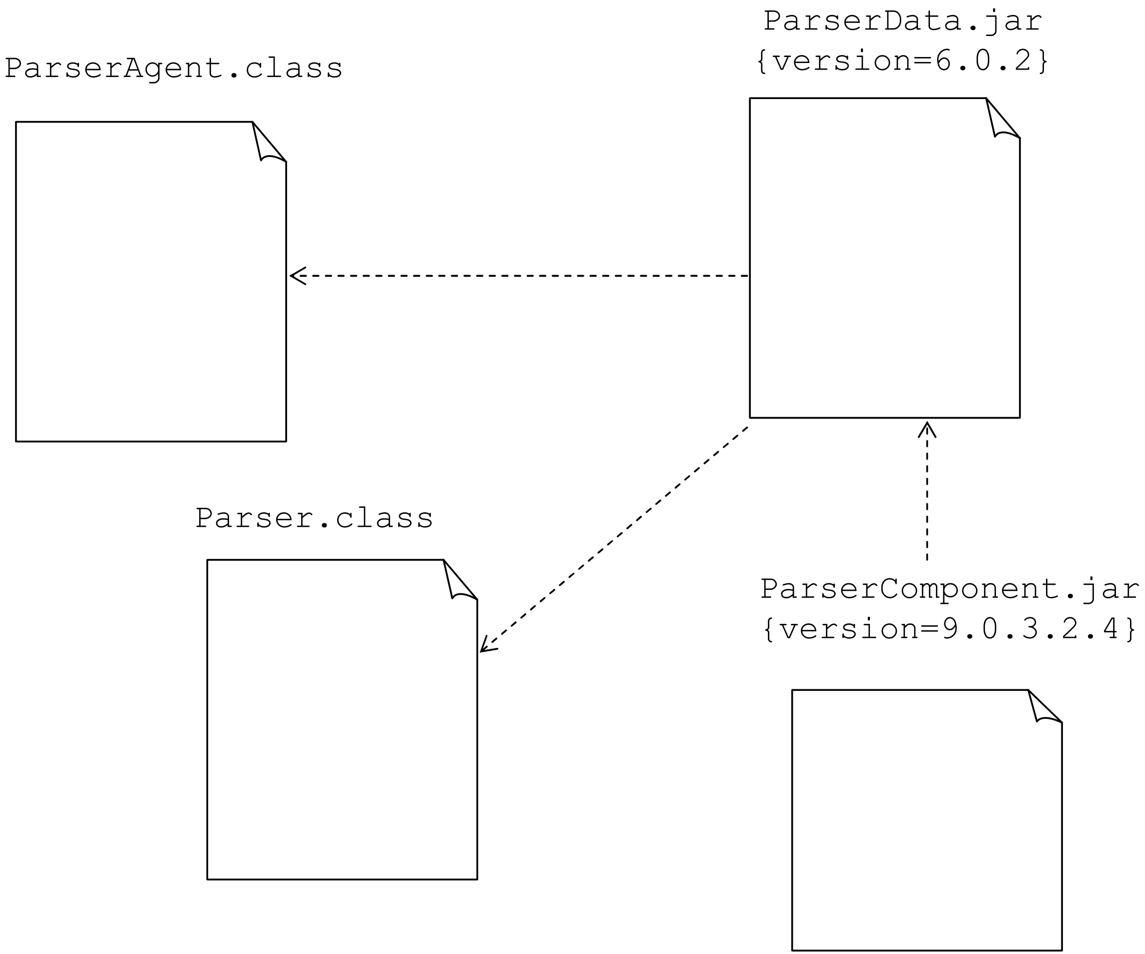
Component Diagrams … continued

- Component diagrams are used typically to model the following elements
  - Source Code
  - Executable Releases
  - Physical Databases
  - Adaptable Systems
  - Forward and Reverse Engineering

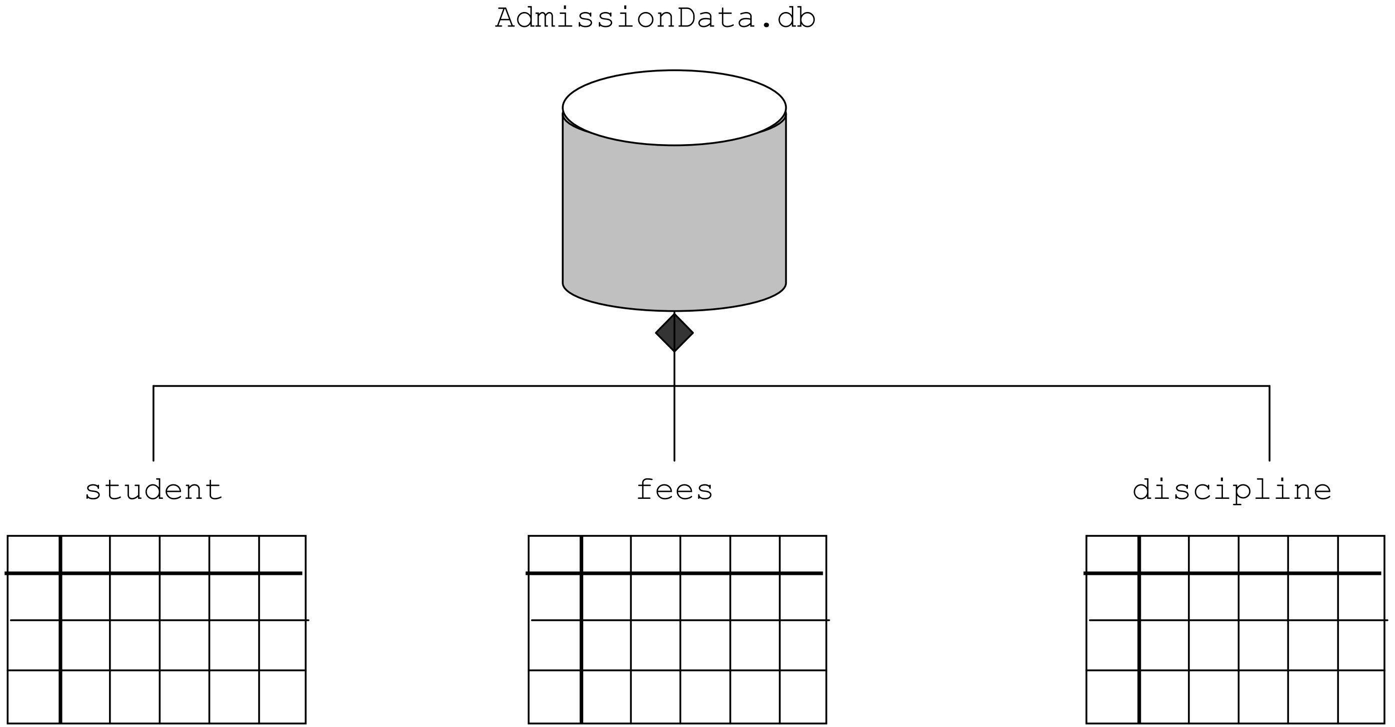
Modeling Source Code



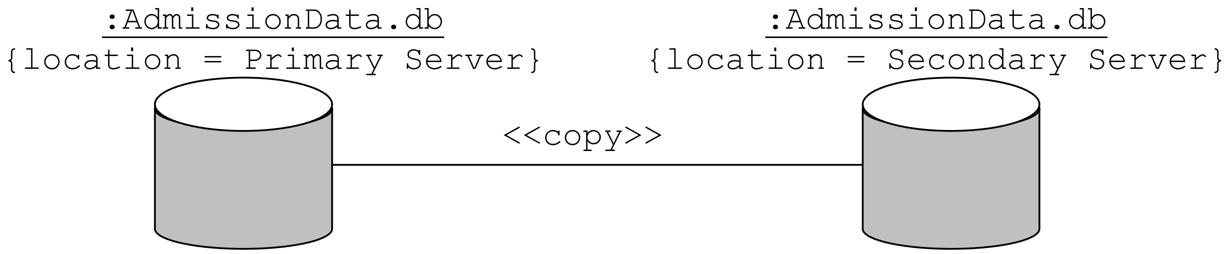
Modeling Executable Releases

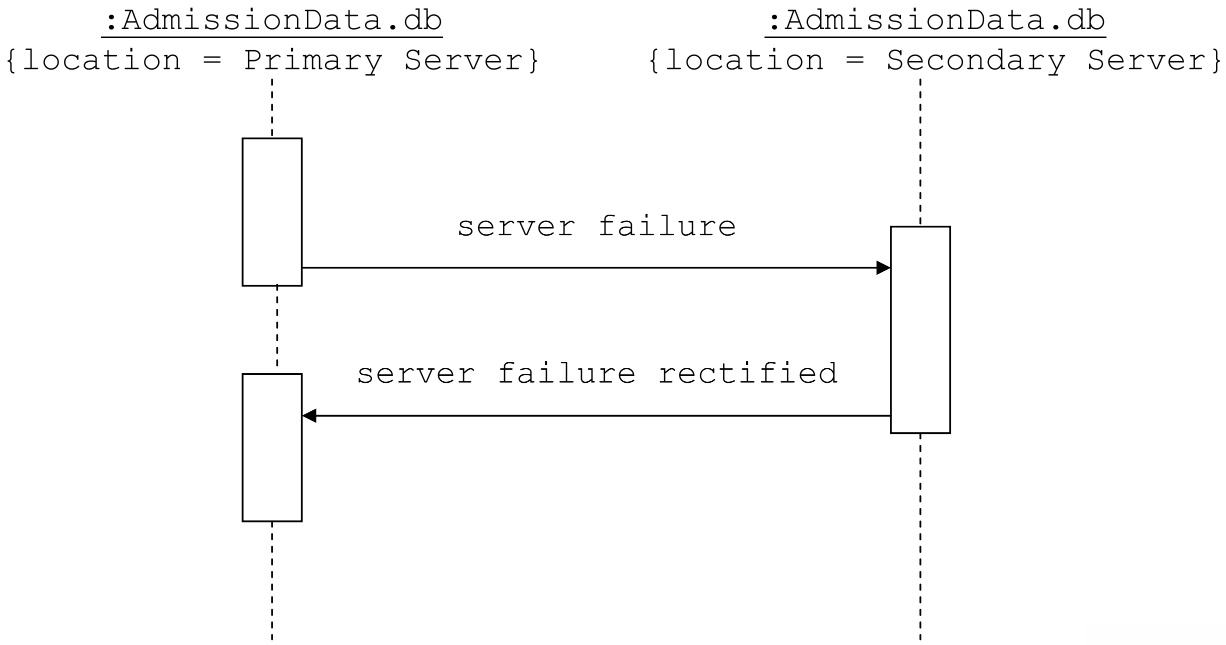


Modeling Physical Databases

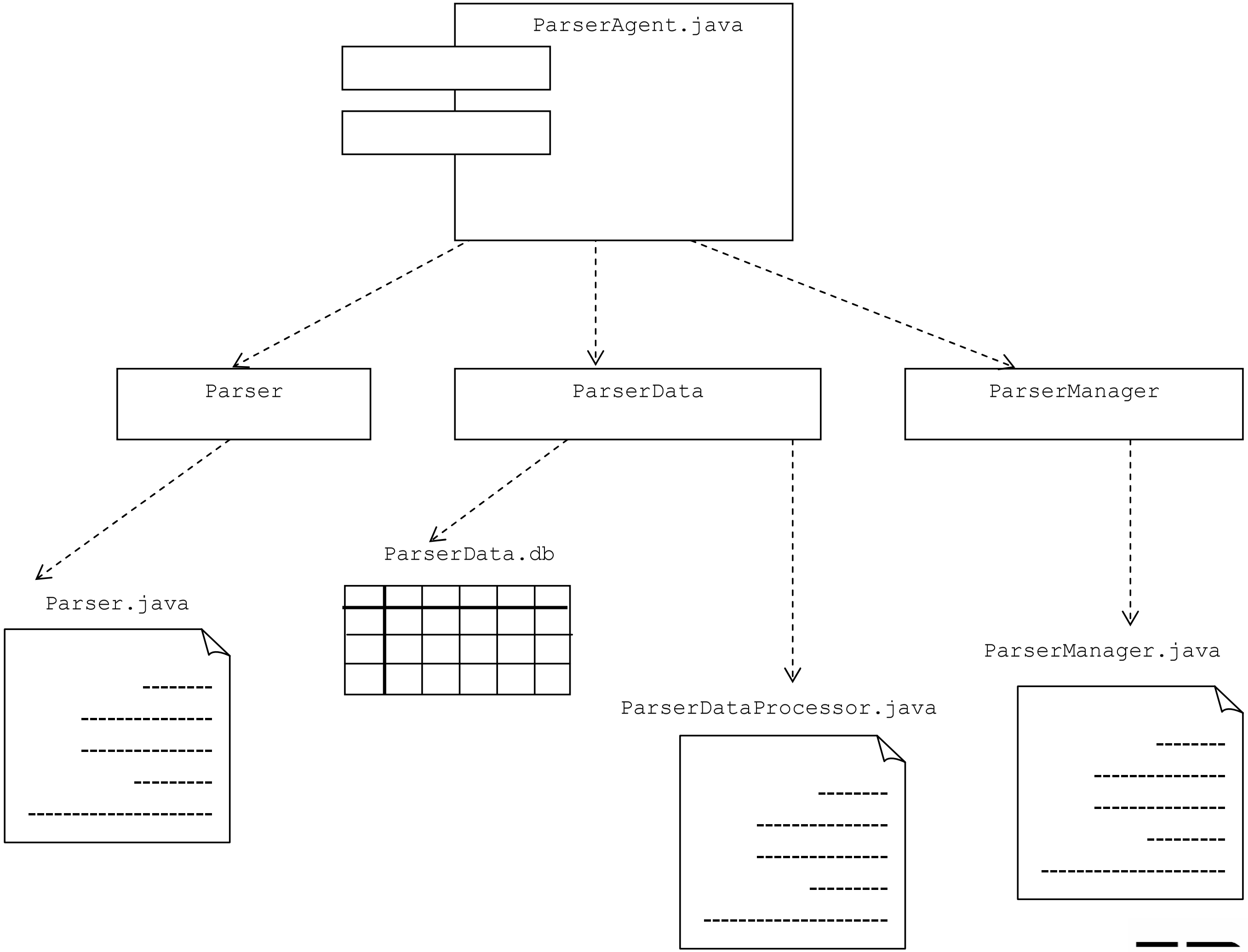


Modeling Adaptable Systems





Forward and Reverse Engineering

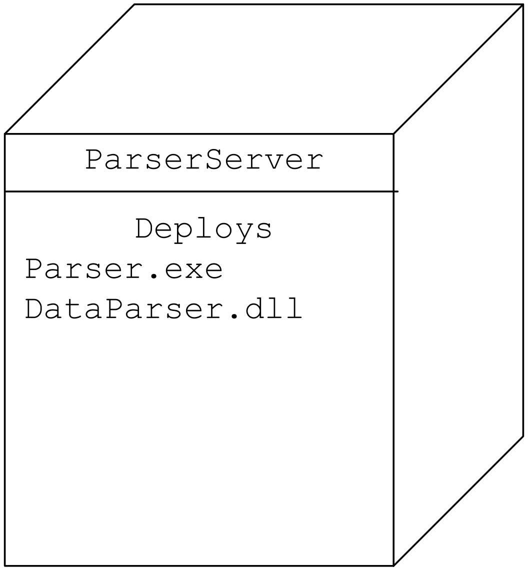


Deployment Diagrams

- Deployment diagrams show the arrangement of run-time processing elements
- These elements contain
  - components
  - processes
  - objects
- These run-time processing elements are referred to as *nodes* representing a computational resource having memory and processing capability
- A deployment diagram is a graph of nodes
- Each node is connected to depict communication association
- The relationship between component and deployment diagrams is strong
- They are collectively called *implementation diagrams*

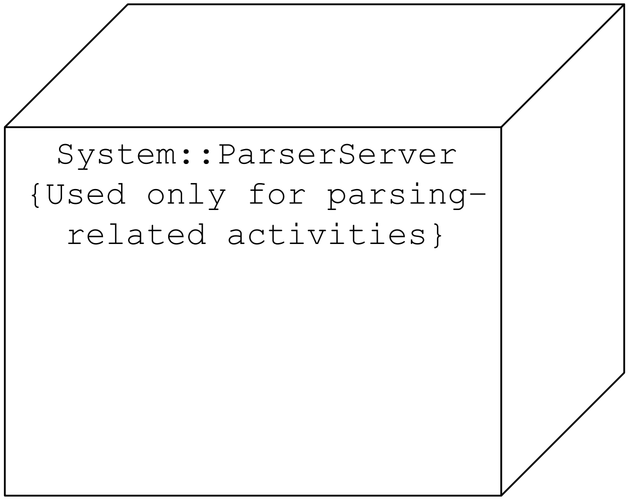
Nodes

- Every node has a name, either a simple or a path name



ParserServer

Deploys
Parser.exe
DataParser.dll

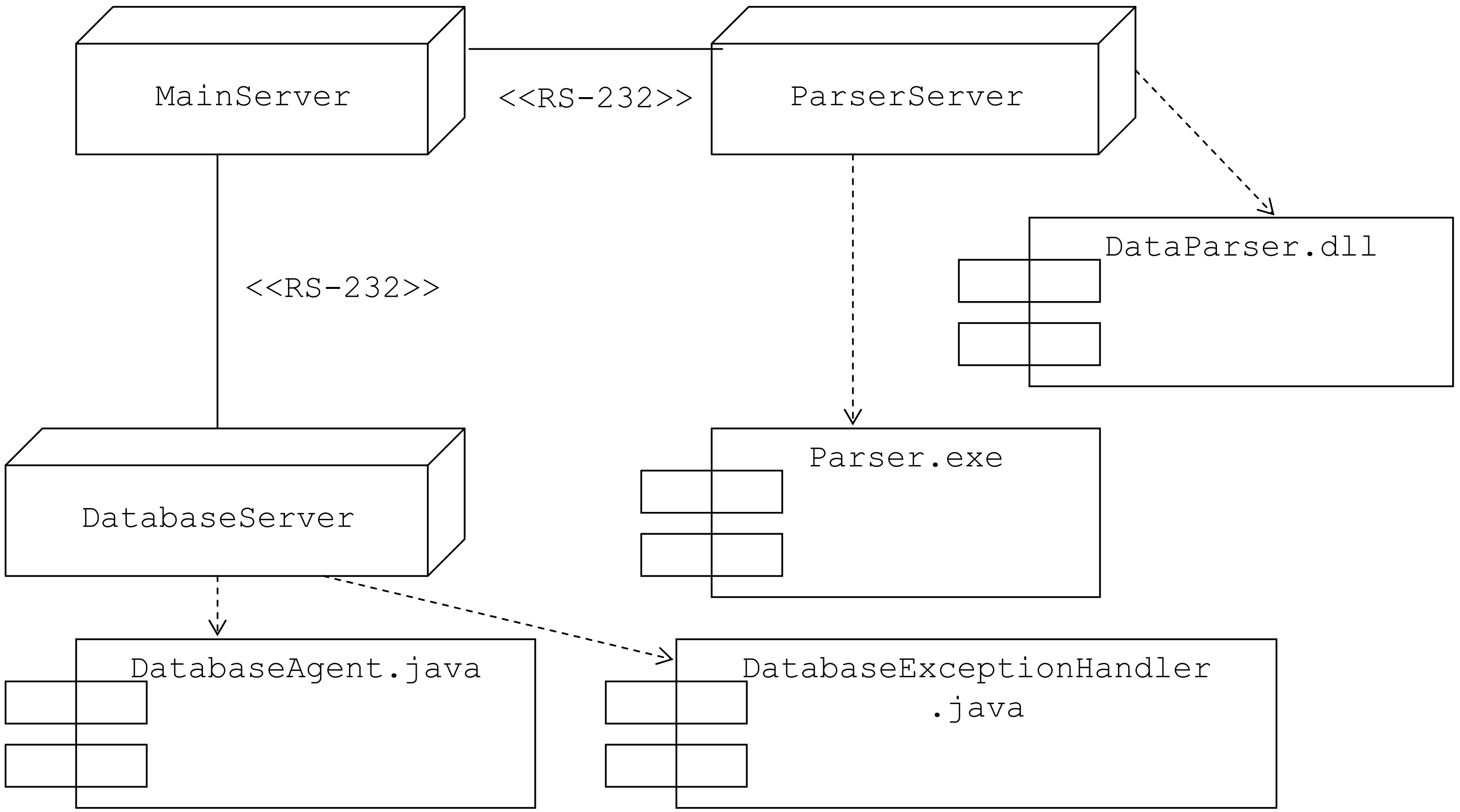


System::ParserServer
{Used only for parsing-
related activities}

Nodes and Components

- Nodes and components participate in dependency, generalization and association relationships
- They can participate in interaction diagrams and we can create instances of nodes and components
- The differences between them are
  - Components take part in the execution of a system, while nodes execute components
  - Components characterize the packaging of logical elements such as collaborations and classes, while nodes represent physically deployed components
- When a set of components is deployed on a node, the group is referred to as a *distribution unit*

Nodes and Components: An Example

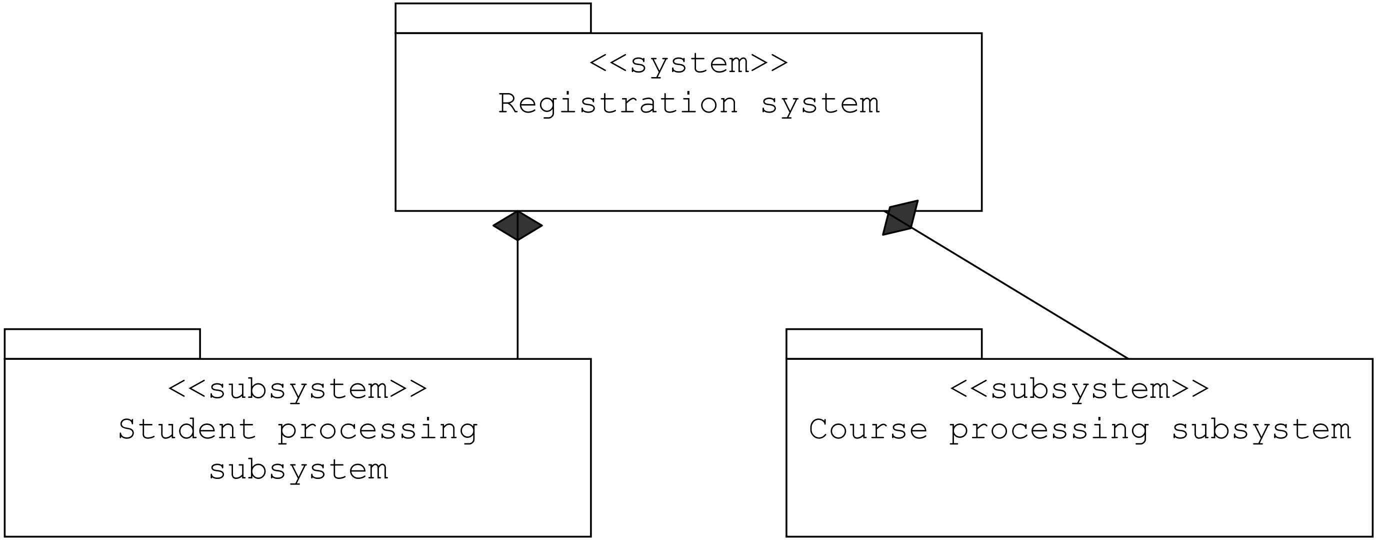


Use of Deployment Diagrams

- Deployment diagrams are used to model the following
  - Embedded systems
  - Client-server systems
  - Distributed systems
- For the client-server system modeling, the client and server can be shown as stereotyped package icons, with the nodes embedded within them
- For the other two, the various parts of the system can be shown as a deployment diagram

Systems and Subsystems

- A system comprises a set of elements used to accomplish a task
- A system can be categorized into subsystems, each subsystem being a grouping of elements
- The relationship between a system and a subsystem is the aggregation relationship



Model and View

- A model is an abstraction of a system
- It is a simplification of reality
- A model typically possesses elements such as classes, interactions and relationships
- A view is a projection of a model
- A view shows only the relevant part of the system as applicable to the view and cuts out the unnecessary
- A view may not cross model boundaries

Summary

- We learnt about components, deployment and collaborations
- We discussed about component diagrams
- We enumerated the need for deployment diagrams
- We understood the difference between systems and subsystems

Unit 5

Object Oriented Testing Methodologies

Learning Objectives

- To describe the ways in which testing in object oriented system is different as compared to those for conventional, procedural systems
- To examine the basic unit of testing in object oriented systems
- To describe the complications that arise in testing object-oriented systems in the context of encapsulation, inherited classes, polymorphic behavior, etc.
- To briefly describe state-based testing
- To explain scenario-based testing
- To state what constitutes integration testing and system testing in the context of object oriented software
- To describe testing in the Unified Development Process
- To provide an overview of object oriented software metrics

An Introduction

- Testing methodologies have evolved over nearly four decades for software developed using structured methodologies and predominantly, the procedural paradigm to coding
- Object oriented analysis and design does bring about a certain discipline in software development that presents several advantages over the conventional methodologies
- Object oriented software development poses new challenges in the area of testing
- There are significant differences between testing methodologies for procedural and object oriented paradigm

Basic Unit of Testing

- The *class* is the natural unit for testing
- The functions (methods in OO) are not the basic units for testing
- Testing the class might throw up different test requirements based on whether the intended use of the class is specific to an application or whether it is for a general purpose
- Test requirements would also vary based on whether the class is an abstract class or a parameterized class
- The entire system has to be tested as a whole when such tested classes are used to create objects that constitute the application

Implications of Inheritance on Testing

- Inheritance can be used in development on the basis on many motivations
- Inheritance can be used to implement specialization relationships or merely as a programming convenience
- The implementation specialization, when used as motivation for inheritance, should correspond to problem domain specialization
- In such a case, we can expect that the test cases used for testing the super class may be reusable while testing the class that inherits
- If we use inheritance merely as a programming convenience (and many programmers indeed do that), it is not likely that such subclasses will reflect a true specialization relationship

Implications of Inheritance on Testing

... continued

- It is dangerous to think that the test cases used in the super class may be reused in the inherited class
- It is more likely that such inherited classes may have to be tested all over again
- Consider the case when we have the following

  class `Base` contains functions `inheritedFunction()` and `redefinedFunction()`

  class `Derived` redefines `redefinedFunction()`
- It is clear that `Derived::redefinedFunction` has to be tested again since it is new code

Implications of Inheritance on Testing

... continued

- The two methods `Base::redefinedFunction()` and `Derived::redefinedFunction()` are two different functions with different specifications and implementations
- They would each have a set of test requirements derived from the specification and implementation
- The test requirements probe for plausible faults, integration faults, condition faults, boundary faults, etc.
- The functions are likely to be similar and their sets of test requirements will overlap
- The better the object oriented design, the greater the overlap

Implications of Encapsulation

- Encapsulation is one of the desirable features of object-oriented systems
- Testing requires reporting on the concrete and abstract state of an object
- The fact is that to some extent, encapsulation may make it hard to provide such reporting
- Built-in or inherited state reporting methods can be provided
- We can have low-level probes or debug tools that can be used to manually inspect an object
- It is possible to make use of some formal “proof of correctness” techniques
- If we are able to do this, then a proved method could be excused from testing
- A formal proof of correctness is similar to exhaustive testing
- It can turn out to be difficult and time consuming

Implications of Polymorphism

- A polymorphic object would have multiple bindings
- We must note that each possible binding of a polymorphic object requires a separate test
- It may be difficult to find all the bindings
- The typical problems faced would be
  - the special requirement to separately test multiple bindings of polymorphic objects
  - complication integration planning

White Box Testing

- White box testing involves source code examination to develop test cases
- As part of this, we have *interface-based testing* and *method-based testing*
- Most conventional white box approaches are based on analysis of either control or data flow
- These techniques can be adapted to method testing, but it must be noted that they are not sufficient for class testing
- This applicability of conventional flow graph approaches to object oriented software appears to be severely limited

White Box Testing … continued

- A minimum white box test strategy for testing OO systems needs to consider the following:
  - The allocation of resources to perform class and method analysis and to document and review the same
  - Developing a test harness made up of stubs, drivers and test object libraries
  - Development and use of standard procedures, naming conventions and libraries
  - Establishment and maintenance of regression test suites and procedures
  - Allocation of resources to design, document and manage a test history library
  - The means to develop or acquire tool support for automation of capture/replay/compare, test suite execution, results verification and documentation capabilities

Black Box Testing

- A minimum black box test strategy for testing OO systems needs to consider the following
  - Allocation of resources to analyze application domain relationships (use of directed graphs to analyze/represent object relationships is very helpful)
  - The ability to create and review test scenarios during requirements analysis and design of system
  - Extensive use of prototyping, and walkthroughs (reviews) with users
  - Purchasing of tools and training early in the development effort
  - Automation of regression testing as much as possible
  - The amount of flexibility and abstraction to be designed into the system; the more there is, the greater the need for some of the techniques outlined above

State-Based Testing

- We can model a class as a state machine
- This enables the use of state based testing to derive test cases
- The state model defines allowable transition sequences ensuring an instance must be created before it can be updated or deleted
- Test cases may be devised to exercise each transition
- When moderately complex classes are involved, the state model based testing becomes usually too large to be handled easily

Integration Strategies and Testing

- There are two main strategies for integration of classes into an application
  - thread based integration strategy
  - uses based integration strategy
- A thread in the context of classes consists of all the classes needed to respond to a single external input
- Each class in the thread set is subjected to unit test
- After clearing this, the thread set itself is exercised. In the case of uses based integration, we begin by testing only such classes that use few or no server classes
- After this, we proceed with the testing of classes of a first group that uses classes of a second group
- This process continues until all groups of classes that use another group are tested

OO Programming Effects on Testing Methods

- In object orientation, when we invoke a function (pass a message), it may be hard to tell exactly what code gets exercised
- We cannot make any major generalizations
- Testing of object-oriented software can be affected on account of two things
  - surface structure changes
  - deep structure changes
- Important variants that should be tested may not be tested
- Particular subsystem interactions may not be probed

Scenario-Based Testing

- This is also called *use case-based testing*
- This type of testing concentrates on what the customer does, not what the product does
- It means capturing the tasks (use cases) the customer has to perform, then using them and their variants as tests
- This is an offshoot of a careful attempt at requirements elicitation
- These scenarios will also tend to locate interaction errors
- They are more complex and more realistic than fault based tests
- Scenario-based tests tend to exercise multiple subsystems in a single test

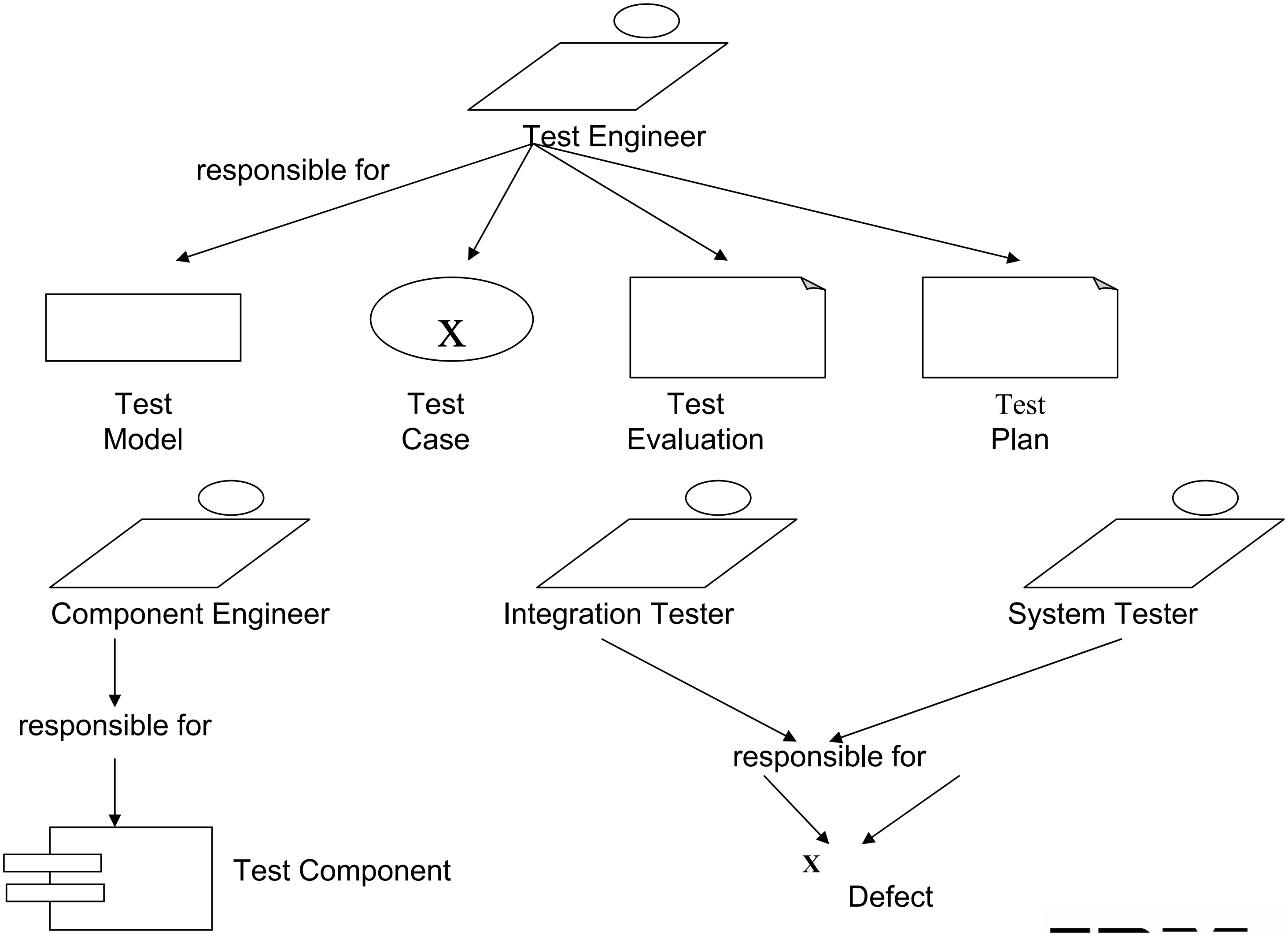
Levels of Object Oriented Testing

- The different levels of object oriented testing are
  - a method
  - message quiescence
  - event quiescence
  - thread testing
  - thread interaction testing
- The object oriented testing units are the object method
- Traditional functional and structural testing techniques are fully applicable to the unit level
- For the system level, they consider the notion of a thread, a thread being interpreted as a sequence of method executions linked by messages in the object network

Testing in the Unified Software Development Process

- In the unified software development process, we have a test workflow in which we need to test each build
- A build is a particular manifestation of a product being developed using the unified development process at a stage
- The build that has to be tested includes internal and intermediate builds
- The final version of the system to be released also needs to be tested

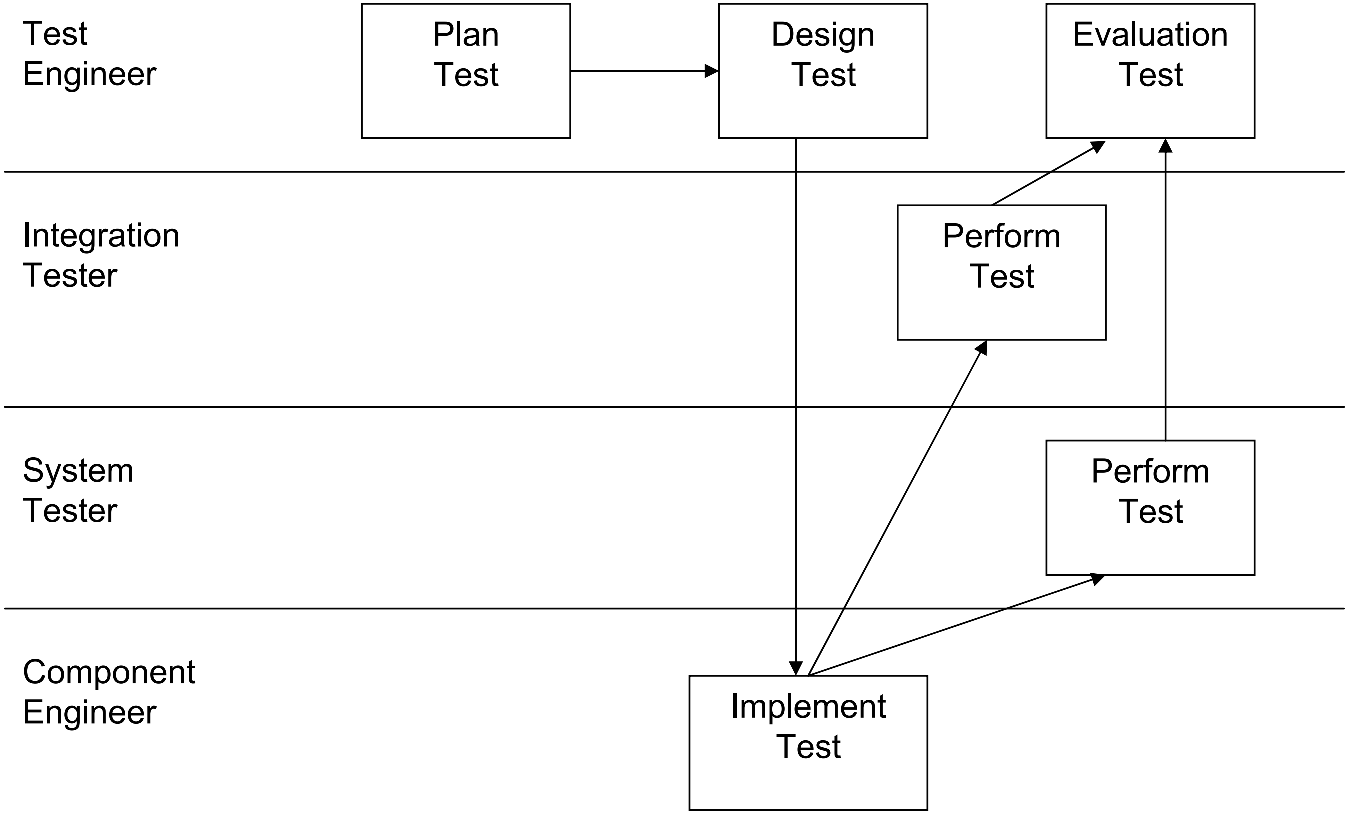
Workers and Artifacts



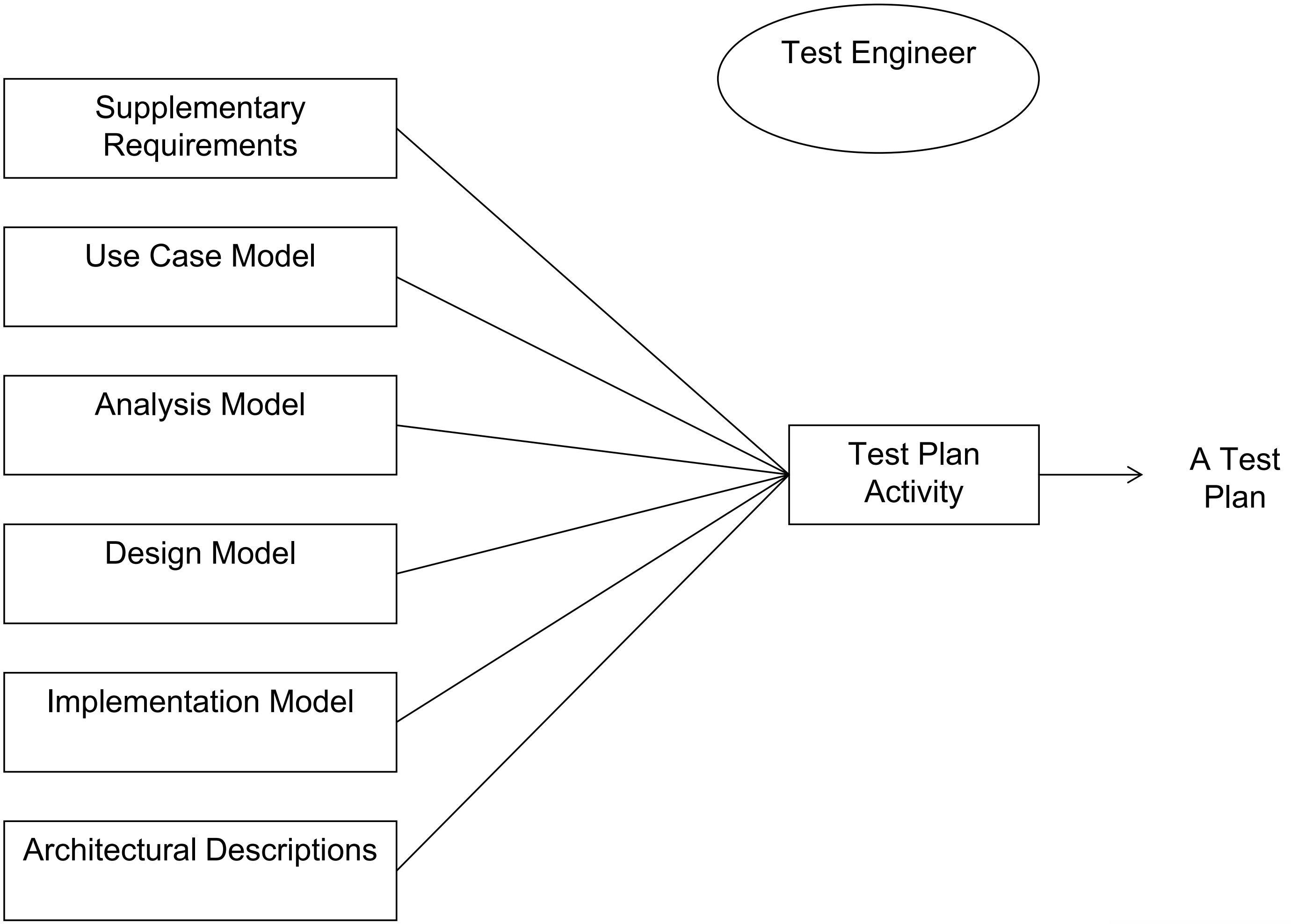
Workers and Artifacts … continued

- Test Model in the Context of Artifacts
- Test Case in the Context of Artifacts
- Test Procedure in the Context of Artifacts
- Test Component in the Context of Artifacts
- Test Plan in the Context of Artifacts
- Defects in the Context of Artifacts
- Evaluation of a Test in the Context of Artifacts
- Test Designer in the Context of the Worker
- Component Engineer in the Context of the Worker
- Integration Tester in the Context of the Worker
- System Tester in the Context of the Worker
- Testing Workflow

Testing Workflow



Testing Workflow … continued



Testing Workflow … continued

- The main purposes of test planning are the following
  - Describing the test strategy
  - Estimating the requirements for the testing effort
  - Scheduling the testing effort
- The design test activity is mainly to
  - identify and describe test cases for each build
  - identify and structure test procedures specifying how to perform test cases

Object Oriented Metrics

- The following are some of the metrics proposed by Morris in his Master's degree thesis in the year 1989
  - Methods Per Class

    Average number of methods per object class = Total number of methods/Total number of object classes

  - Inheritance Dependencies

    Inheritance tree depth = max (inheritance tree path length)

  - Degree of Coupling Between Objects

    Average number of uses dependencies per object = total number of arcs/total number of objects

    arcs = max (number of uses arcs) - in an object uses network arcs - attached to any single object in a uses network

Object Oriented Metrics ... continued

- Degree of Cohesion of Objects

Degree of Cohesion of Objects = Total Fan-in for All Objects / Total No. of Objects

- Object Library Effectiveness

Average number = Total Number of Object Reuses / Total Number of Library Objects

- Factoring Effectiveness

Factoring Effectiveness = Number of Unique Methods / Total Number of Methods

- Degree of Reuse of Inheritance Methods

Percent of Potential Method Uses Actually Reused (PP):

PP = (Total Number of Actual Method Uses / Total Number of Potential Method Uses) x 100

Percent of Potential Method Uses Overridden (PM):

PM = (Total Number of Methods overridden / Total Number of Potential Method Uses) x 100

Object Oriented Metrics … continued

- Average Method Complexity
- Application Granularity
- Weighted Methods Per Class (WMC)
- Depth of an Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling Between Object Classes (CBO)
- Response for a Class (RFC)
- Lack of Cohesion in Methods (LCOM)
- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Polymorphism Factor (PF)
- Coupling Factor (CF)

Summary

- We described the ways in which testing in object oriented system are different as compared to those for conventional, procedural systems
- We examined the basic unit of testing in object oriented systems
- We described the complications that arise in testing object-oriented systems in the context of encapsulation, inherited classes, polymorphic behavior, etc.
- We briefly described state based testing
- We explained scenario based testing
- We stated what constitutes integration testing and system testing in the context of object oriented software
- We described testing in the Unified Development Process
- We provided an overview of object oriented software metrics