SS ZG653 (RL 7.1): Software Architecture

Introduction to OODesign

Instructor: Prof. Santonu Sarkar

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Online shopping application- Pet Store

by Sun Microsystems (Oracle)

- Storefront has the main user interface in a Web front-end. Customers use the Storefront to place orders for pets
  - Register User
  - Login user
  - Browse catalog of products
  - Place order to OPC (asynchronous messaging)
- Order Processing Center (OPC) receives orders from the Storefront.
- Administrators
  - Examine pending orders
  - Approve or deny a pending order
- Supplier
  - View and edit the inventory
  - Fulfills orders from the OPC from inventory and invoices the OPC.

System quality

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability
- Interoperability

- Show orders in multiple languages
- Help in browsing products
- Easy checkout
- Assurance to customer, supplier and bank
- Guaranteed delivery
- Shopping cart is only modified by the customer
- Order never fails
- System is always ready to sell
- Order is always processed in 2sec.
- System always optimally use the hardware infrastructure and performs load-balancing
- Uses standard protocol to communicate with Suppliers
- Uses secure electronic transaction protocol (SET) for credit card processing
- Quite easy to add new supplier
- Little change necessary to add a third party vendor
- Quite easy to sell books in addition to Pets
- Logging is extensive to trace the root cause of the fault

Example

Programming for this Application

- Everything is an object
  - Pet, Customer, Supplier, Credit card, Customer's address, order processing center
- Pets can be sold, credit card can be charged, clients can be authenticated, order can be fulfilled!
  - Have behavior, properties
- Interact with each other
  - Storefront places order to OPC

What's cool?

- Very close to the problem domain… domain experts can pitch in easily.
- Could group features and related operations together
- It's possible to add new types of Pets (e.g., add Birds in addition to Cat, Dog and Fish)

What is a class

- Class represents an abstract, user-defined type
  - Structure – definition of properties/attributes
    - Commonly known as member variables
  - Behavior – operation specification
    - Set of methods or member functions

- An object is an instance of a class. It can be instantiated (or created) w/o a class

Object State

- **Properties/Attribute Values at a particular moment represents the state of an object**
- **Object may or may not change state in response to an outside stimuli.**
- **Objects whose state can not be altered after creation are known as immutable objects and class as immutable class [String class in Java]**
- **Objects whose state can be altered after creation are known as mutable objects and class as mutable class [StringBuffer class in Java]**

States of Three Different INSTANCES of "Dog" class

Labrador	***Golden Retriever***	***Pug***
Age: 1 yr	***Age: 6months***	***Age: 1.5yr***
Price : 3500	***Price : 2000***	***Price : 1500***
Sex:Male	***Sex:Female***	***Sex: Female***

Object-Oriented Programming

- A program is a bunch of objects telling each other what to do, by sending messages
  - In Java, C++, C# one object (say o1) invokes a method of another object (o2), which performs operations on o2
  - You can create any type of objects you want
- OO different from procedural?
  - No difference at all: Every reasonable language is ultimate the same!!
  - Very different if you want to build a large system
    - Increases understandability
    - Less chance of committing errors
    - Makes modifications faster
    - Compilers can perform stronger error detections

Type System

- Classes are user-defined data-types
- Primitive types
  - int, float, char, boolean in Java (bool in C#), double, short, long, string in C#
- Unified type
  - C# keyword object – mother of all types (root)
    - Everything including primitive types are objects
  - Java JDK gives java.lang.Object– not a part of the language
    - Primitive types are not objects

References and Values

Value

- Primitive types are accessed by value
- C++ allow a variable to have object as its value
- C# uses struct to define types whose variables are values
- No explicit object creation/deletion required
  - Faster, space decided during compilation

Reference

- Java allows a variable to have reference to an object only
- C++ uses pointers for references
- Needs explicit object creation
  - Slower, space allocated during runtime from heap
- Java performs escape analysis for faster allocation

Modules

- You need an organization when you deal with large body of software – 20MLOC, 30000 classes or files!
- Notion of module introduced in 1970
  - Group similar functionality together
  - Hide implementation and expose interface
  - Earlier languages like Modula, Ada introduced this notion
  - In OO language “class” was synonymous to a module
- But they all faced the problem of managing 20M lines of C or C++ code
- ANSI C++, Java, Haskell, C#, Perl, Python, PHP all support modules
  - Namespace (C++, C#)
  - Package (Java, Perl)

Module- aka namespace, package

- A set of classes grouped into a module
  - A module is decomposed into sub-modules
  - Containment hierarchy of modules – forms a tree
- A fully qualified, unique name= module+name of the class
  - namespace
  - package
- Import- A class can selectively use one or more classes in a module or import the entire module

Inheritance

- Parent (called Base class) and children classes (Derived classes)
  - A Derived class inherits the methods and member variables of the Base -- also called ISA relationship
  - A child can have multiple parents – Multiple inheritance
  - Hierarchy of inheritance (DAG)

- The Derived class can
  - Use the inherited variables and methods – reuse
  - Add new methods – extends the functionality
  - Modify derived method- called **overriding** the base class member function

Introducing Interface

What is it?

- A published declaration of a set of services
  - An interface is a collection of constants and method declarations
  - No implementation, a separate class needs to implement an interface
- A class can implement more than one interfaces

Why?

- Provide capability for unrelated classes to implement a set of common methods
- Standardize interaction
- Extension- let's the designer to defer the design
- User does not know who implemented it
  - It is easy to change implementation without impacting the user

Interface Definition

- An interface in C++, C# is a class that has at least one pure virtual method
  - A pure virtual method only has specification, but no body
  - This is called abstract base class
  - One can have an abstract class where some methods are concrete (with implementation) and some as pure virtual which could be clumsy

- Java uses keyword “interface” and is more clean
  - Interface only provides method declarations

- Java also allows to define an abstract base class just like C++

Creating Objects

- With built-in types like **int** or **char**, we just say

  int i; char c; ---   and we get them

- When we define a class A-- user-defined type
  - We need to explicitly
    1. tell that we want a new object of type A (operator new)
    2. Initialize the object (you need to decide) after creation -- constructor method
       - Special method that compiler understands. The constructor method name must be same as the class name
       - C++ allows constructor method for struct also
- Constructor methods can be overloaded

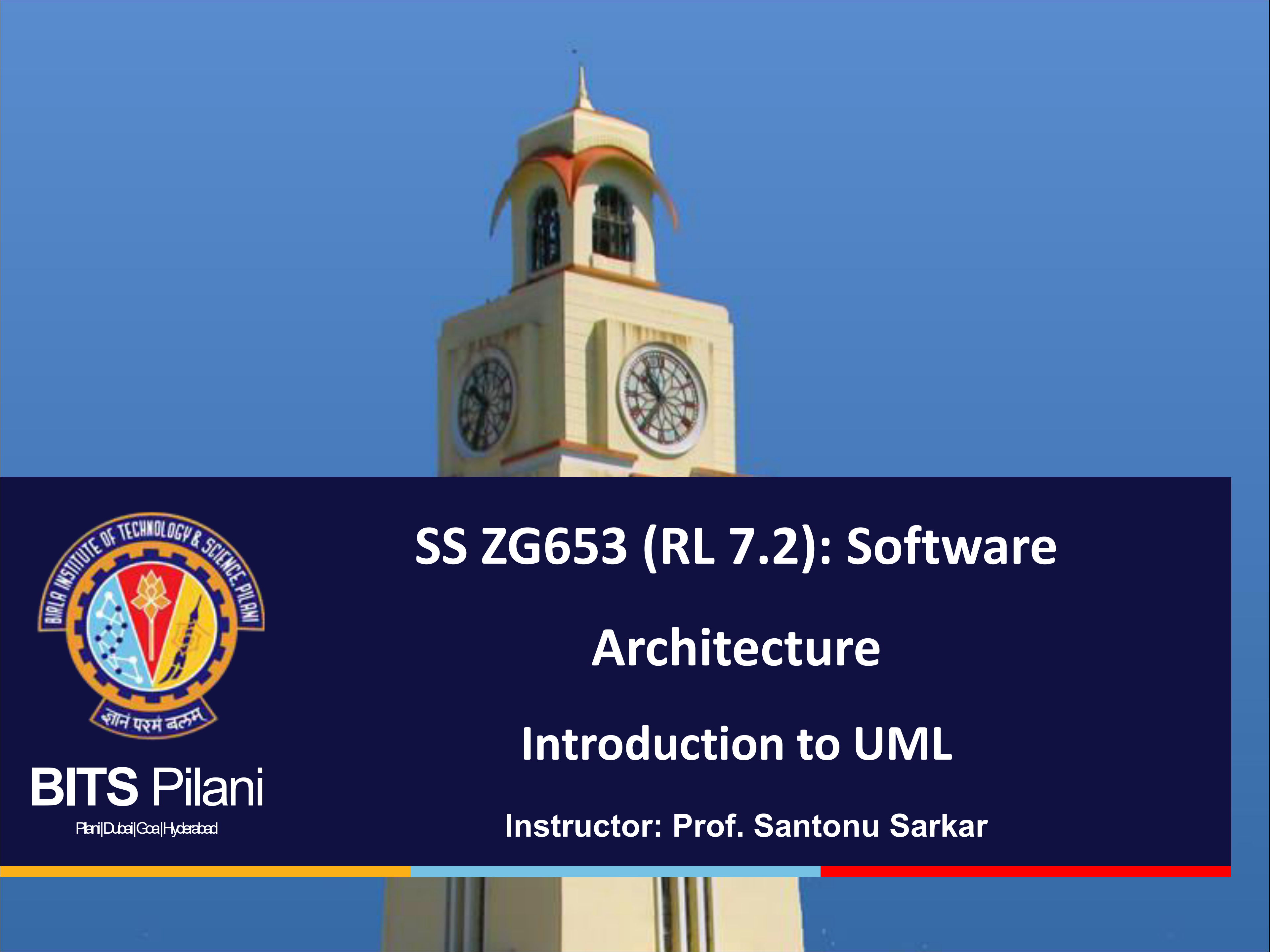
Destroying Objects

- If an object goes “out of scope,” it can no longer be used (its name is no longer known) it is necessary to free the memory occupied by the object
  - Otherwise there will be a “memory leak”

1. Just before freeing the memory
   - It is necessary to perform clean-up tasks (you need to decide) just before getting deleted
     - » Destructor method describes these tasks
     - » Special name for destructor method ~<ClassName> in C++
     - » finalize() method in Java
     - » does not have any return value
2. Then free the memory
   - In C++, we need to explicitly delete this object (delete operator)
   - Java uses references and “garbage collection” automatically.

Thank You

Next class - UML



BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 7.2): Software Architecture

Introduction to UML

Instructor: Prof. Santonu Sarkar

Unified Modeling Language (Introduction)

- Modeling Language for specifying, Constructing, Visualizing and documenting software system and its components
- Model -> Abstract Representation of the system [Simplified Representation of Reality]
- UML supports two types of models:

  - Static
  - Dynamic

UML

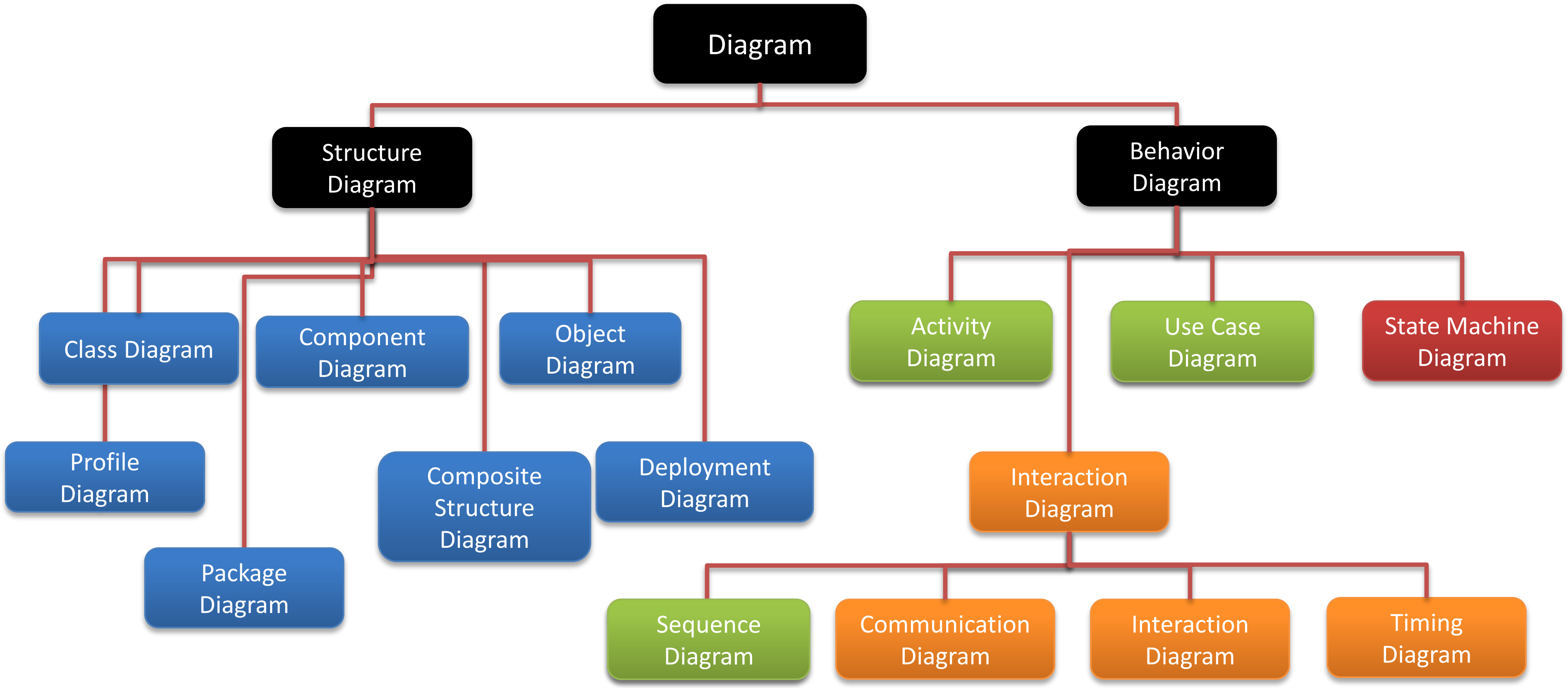
- Unified Modeling Language is a standardized general purpose modeling language in the field of object oriented software engineering

- The standard is managed, and was created by, the Object Management Group.

- UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems

UML Diagrams overview

- Structure diagrams emphasize the things that must be present in the system being modeled- extensively used for documenting software architecture
- Behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.



Structural Diagrams

- **Class diagram**: system's classes, their attributes, and the relationships
- Component diagram: A system, comprising of components and their dependencies
- Composite structure diagram: decomposition of a class into more finer elements and their interactions
- **Deployment diagram**: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of an example modeled system at a specific time.
- Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
- Profile diagram: operates at the metamodel level

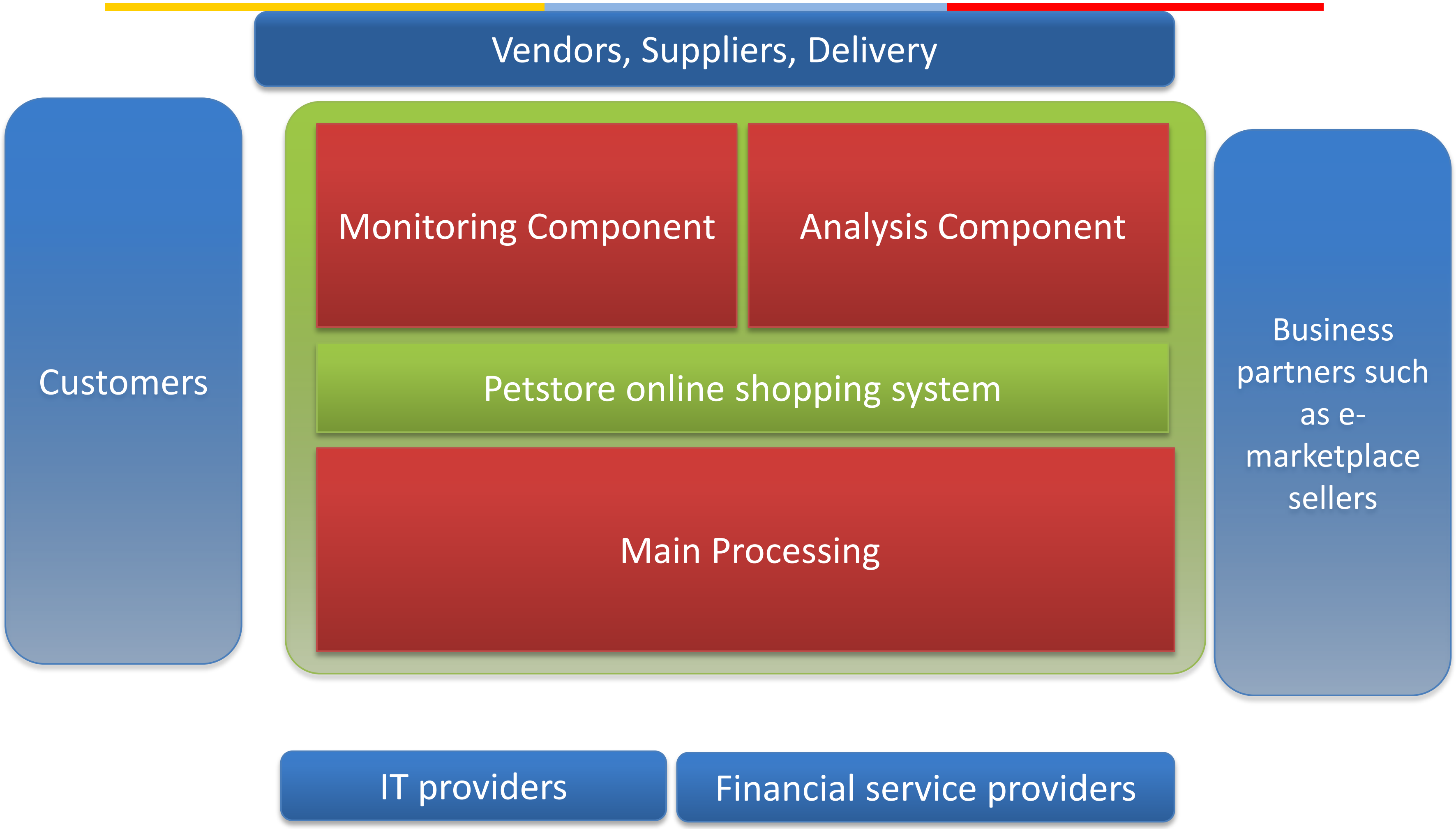
Behavioral Diagrams

- **Activity diagram**: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- **State machine diagram**: describes the states and state transitions of part of the system.
- **Use case diagram**: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases

Behavioral Model- Interactions

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- **Sequence diagram**: Interaction among objects through a sequence of messages. Also indicates the lifespans of objects relative to those messages
  - Timing diagrams: a specific type of sequence diagram where the focus is on timing constraints.

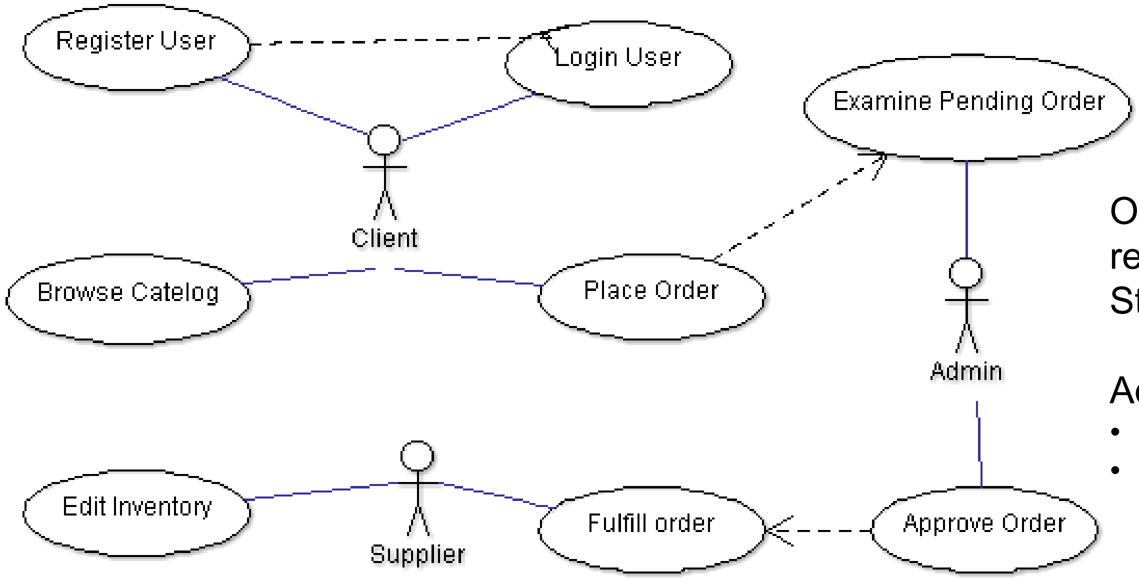
Petstore Shopping System


Vendors, Suppliers, Delivery
Customers
Monitoring Component
Analysis Component
Petstore online shopping system
Main Processing
Business partners such as e-marketplace sellers
IT providers
Financial service providers

Use Case

Storefront has the main user interface in a Web front-end. Customers use the Storefront to place orders for pets

- Register User
- Login user
- Browse catalog of products
- Place order to OPC (asynchronous messaging)



Order Processing Center (OPC) receives orders from the Storefront.

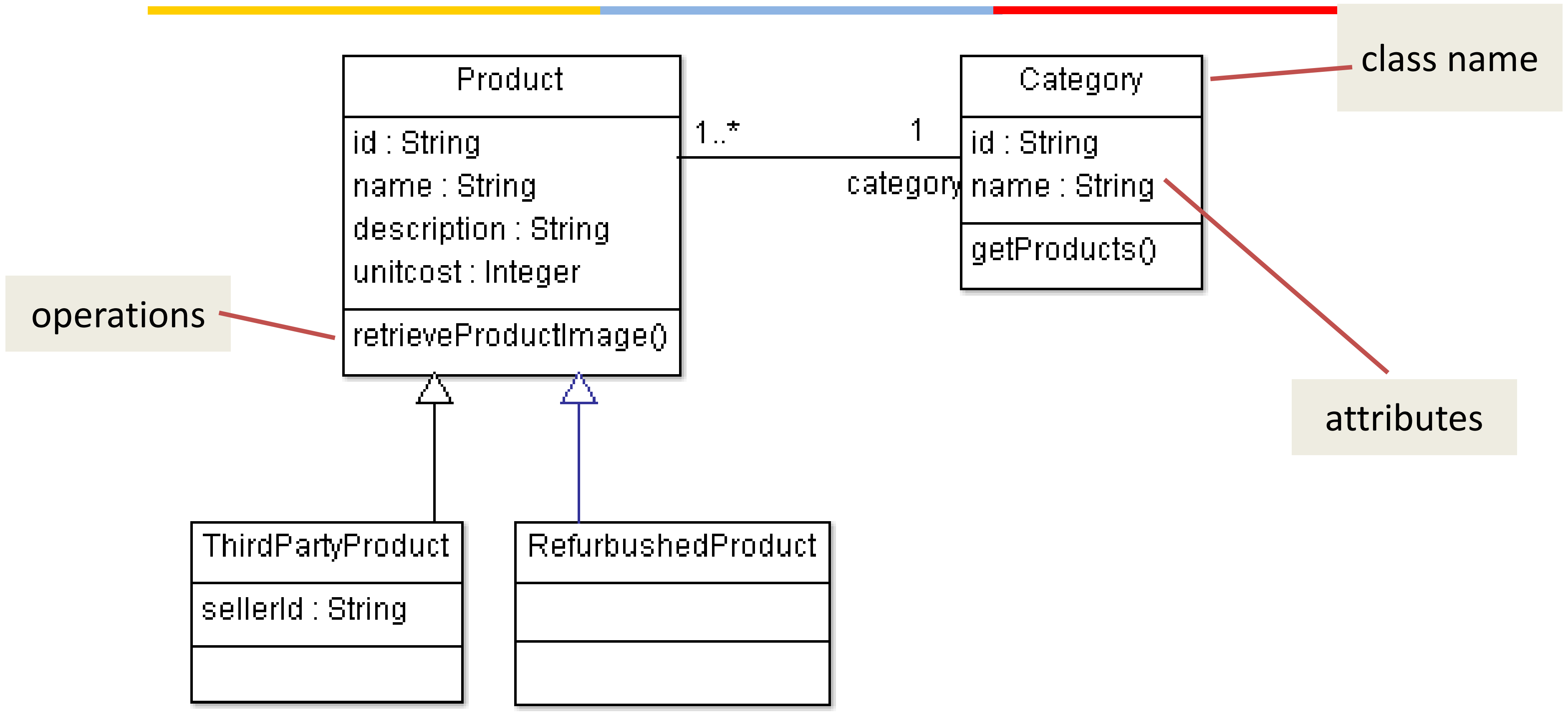
Administrator

- Examine pending orders
- Approve or deny a pending order

Supplier

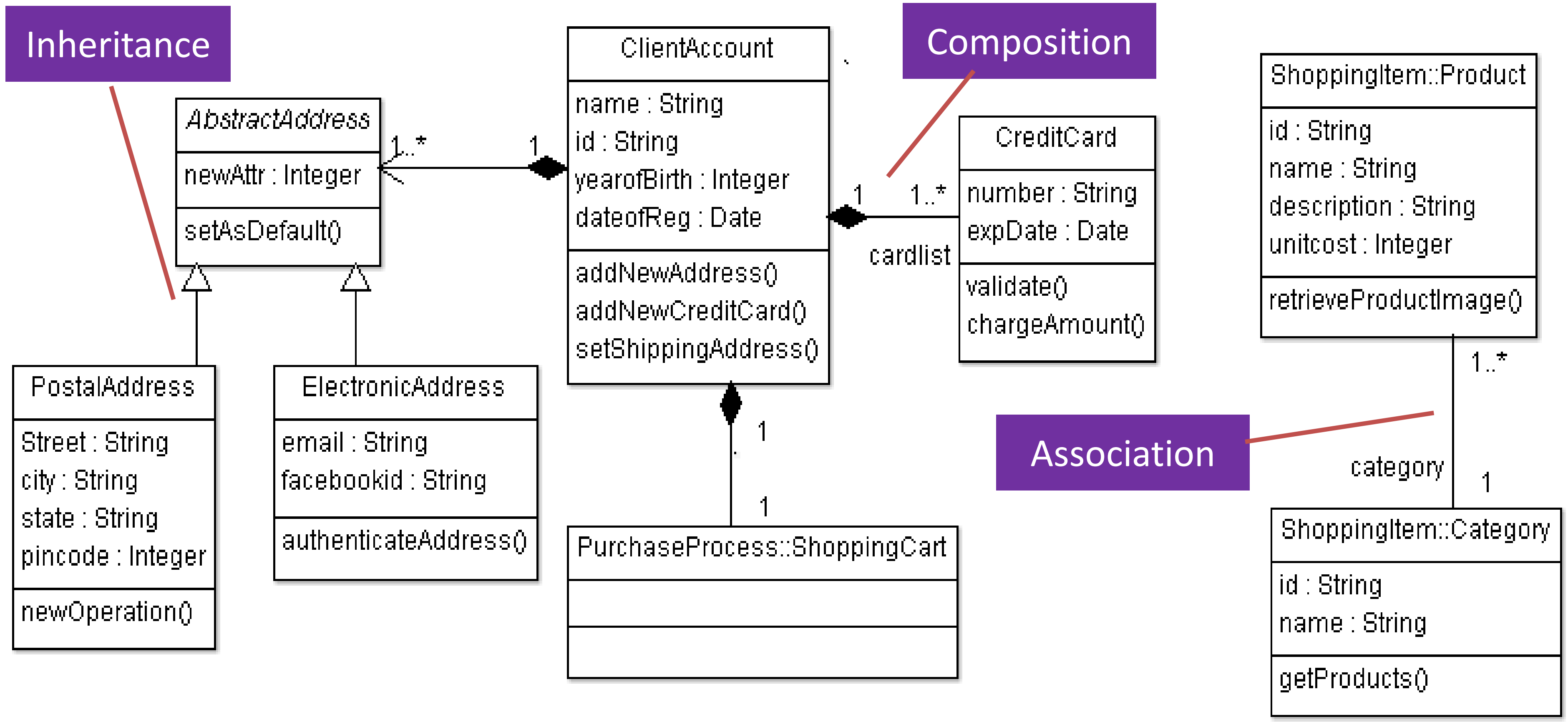
- View and edit the inventory
- Fulfills orders from the OPC from inventory and invoices the OPC.

Class Notation



Visibility Modes : Private(-), Protected (#) , Public (+) and Package Private()

Class Relationships



Generalization Example

- isA, is-a-type of relation ship
- A PostalAddress, or an EmailAddress is an Address
- There can be ThirdPartyProduct or a Refurbished product in the online store

```
public class PostalAddress extends AbstractAddress  {

 private String Street;
 private String city;
 private String state;
 private Integer pincode;
}
```

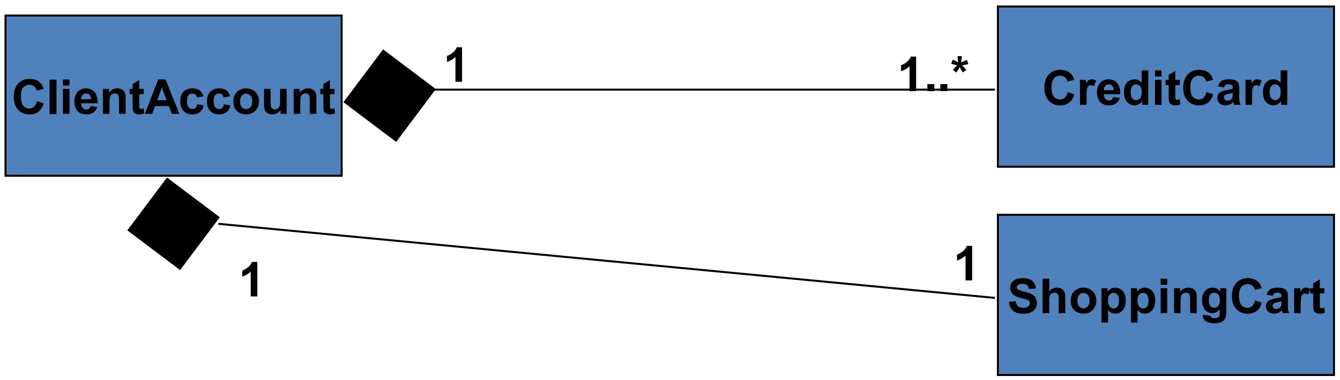
```
public class ElectronicAddress
extends AbstractAddress  {

  private String email;
  private String facebookid;

  public void authenticateAddress() {
  }
}
```

Different forms of association

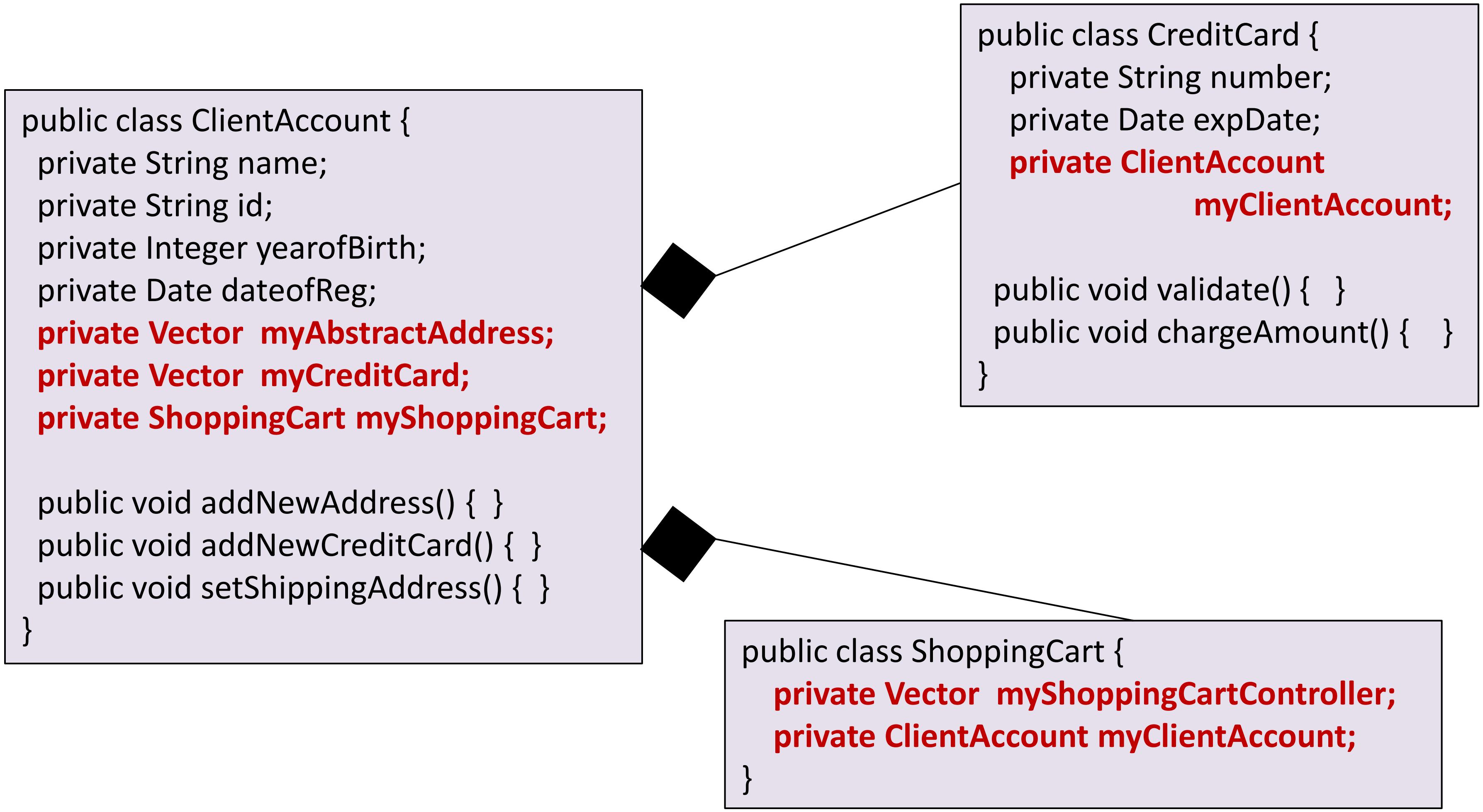
1. Strongest (Composition)
   - Implies total ownership, if the owner is destroyed, the parts are also destroyed
   - Inner classes will certainly be a composition
2. Aggregation
   - Implies has-a part ownership
   - If the owner is destroyed, the parts still exist
3. Weakest (Association)
   - General form of dependency based on the usage of features

Composition

- Total ownership
- A CreditCard exclusively belongs to one Client, and one client can have many credit cards.
- A client exclusively owns her shopping cart.
- The shopping cart and the credit card will no longer exist if the client is removed



Composition Code snippet



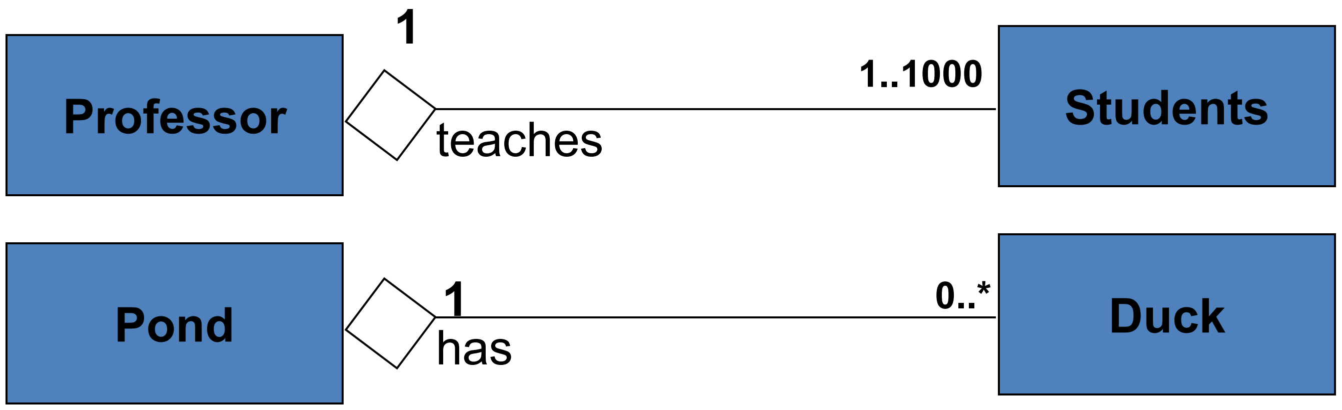
```
public class ClientAccount {
  private String name;
  private String id;
  private Integer yearofBirth;
  private Date dateofReg;
  private Vector  myAbstractAddress;
  private Vector  myCreditCard;
  private ShoppingCart myShoppingCart;

  public void addNewAddress() {  }
  public void addNewCreditCard() {  }
  public void setShippingAddress() {  }
}
```

```
public class CreditCard {
    private String number;
    private Date expDate;
    private ClientAccount
                  myClientAccount;

  public void validate() {   }
  public void chargeAmount() {    }
}
```

```
public class ShoppingCart {
    private Vector  myShoppingCartController;
    private ClientAccount myClientAccount;
}
```

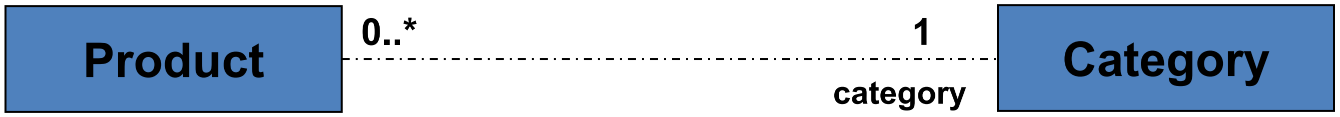
Aggregation Relationship

- Relatively weaker composition
- Students will exist even when the Professor stops taking the class
- Ducks will exist without the Pond



Association or Dependency Relation

- **A product category in the online shop can have many products**
- **However, a product belongs to only one category.**
- **Both of them independently exist.**

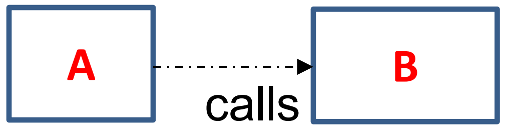


```
public class Product {
   private String id;
   private String name;
   private String description;
   private Integer unitcost;
   private Category category;

  public void retrieveProductImage() {
  }
}
```

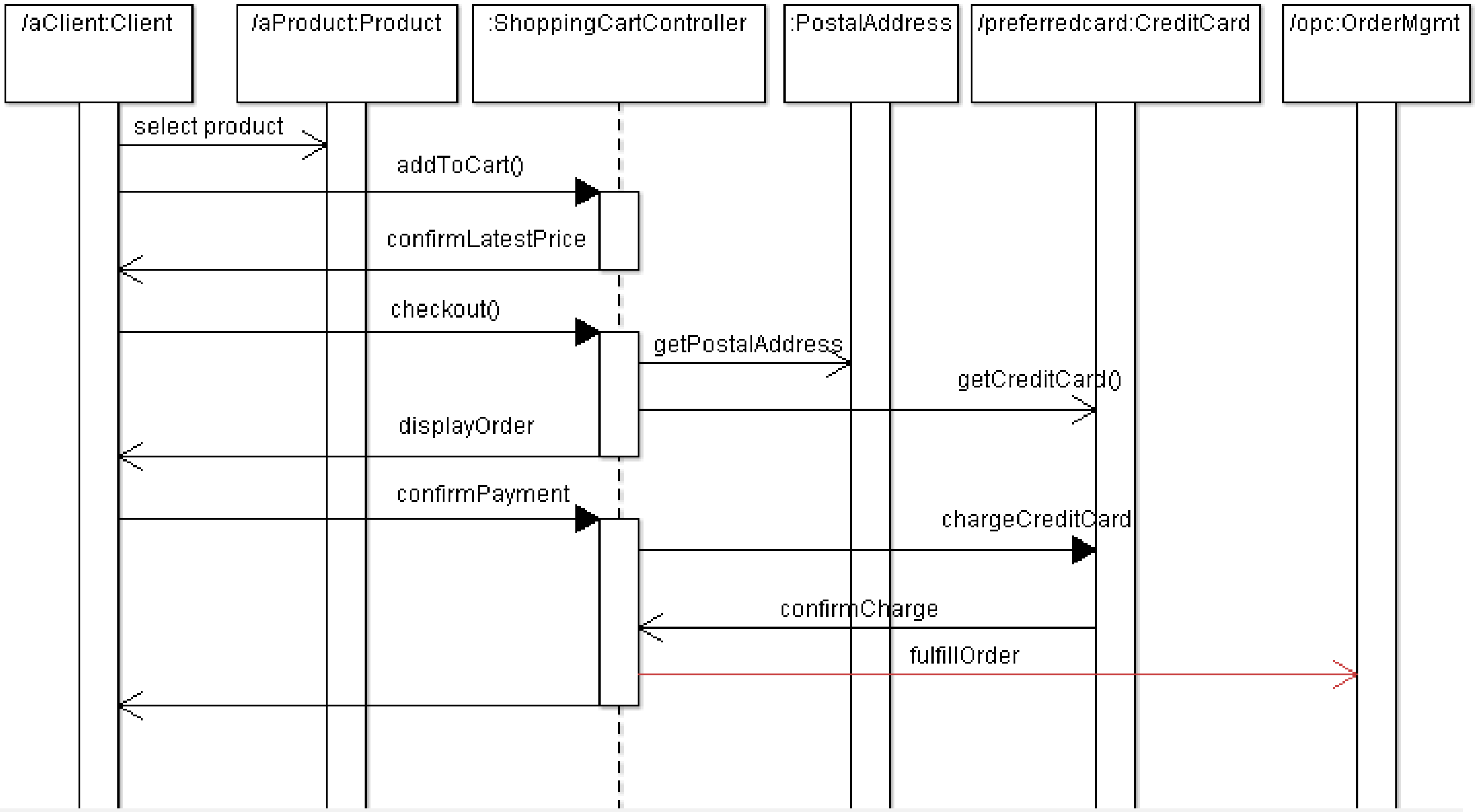
```
public class Category {
   private String id;
   private String name;
   private Vector<Product> myProduct;

   public void getProducts() {
   }
}
```

Dependency Example 2

```
class B
{
  public void doB()
  {
    System.out.println("Hello");
  }
}
class A
{
  public void doS()
  {
    B b1 = new B();
    b1.doB();
  }
} //End of class Test
```



Sequence Diagrams



Sequence diagram is drawn to represent (i) objects participating in an interaction and (ii) what messages have exchanged among those objects

Thank You

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SS ZG653 (RL 8.1): Software Architecture

Documenting Architecture with UML

Instructor: Prof. Santonu Sarkar

September 23, 2022

Unified Modeling Language (Introduction)

- **Modeling Language for specifying, Constructing, Visualizing and documenting software system and its components**
- **Model -> Abstract Representation of the system [Simplified Representation of Reality]**
- **UML supports two types of models:**

  - Static - Dynamic

UML

- Unified Modeling Language is a standardized general purpose modeling language in the field of object oriented software engineering

- The standard is managed, and was created by, the Object Management Group.

- UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems

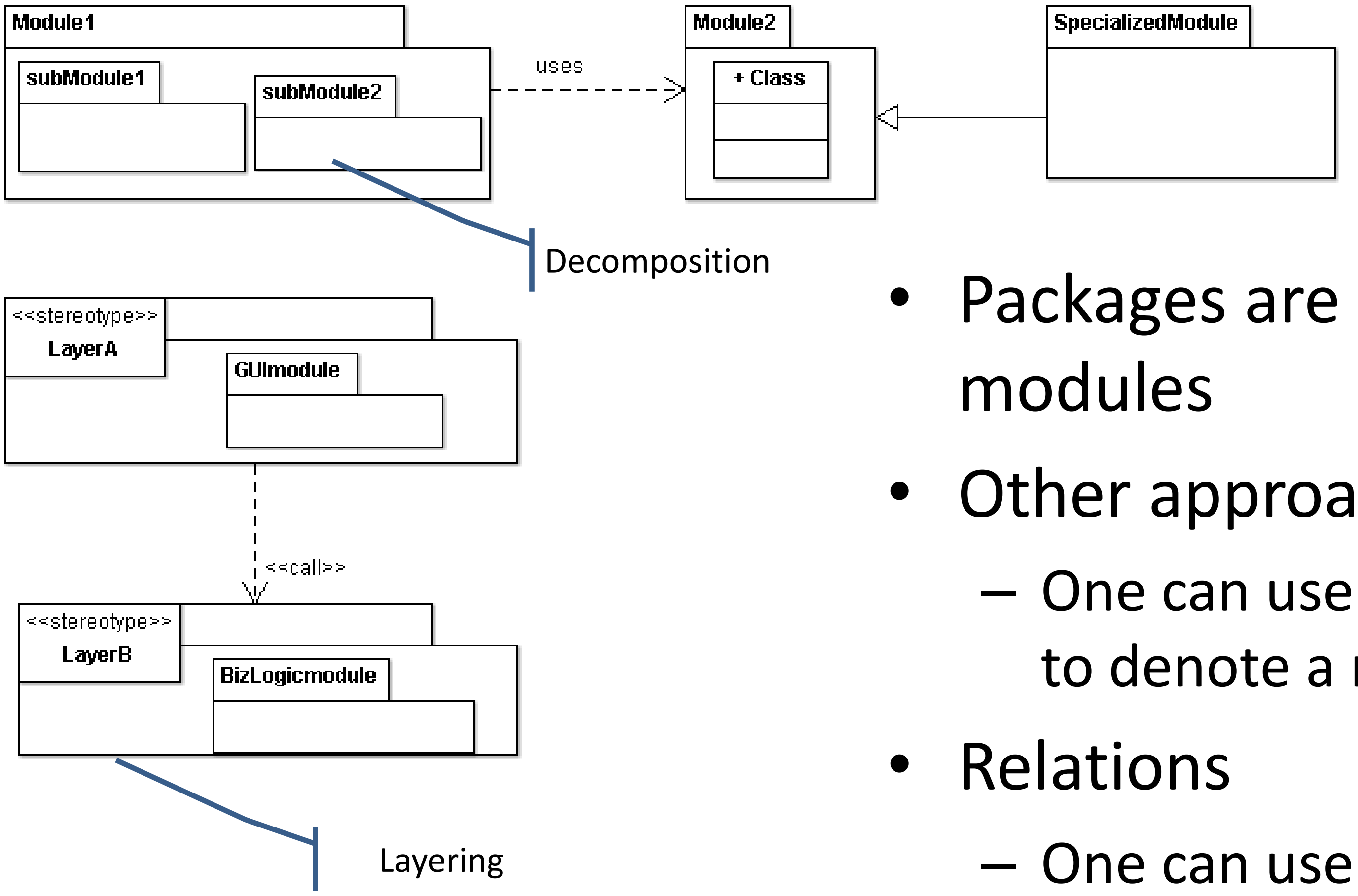
Documenting Architecture

- UML has not been designed specifically for architecture, though practitioners use UML for architecture description
  - It is upto the architect to augment UML for architecture
  - UML provides no direct support for module-structure, component-connector structure or allocation structure

Three Structures- Recap

- Module Structure
  - Code units grouped into modules
  - Decomposition
    - Larger modules decomposed into smaller modules
  - Use
    - One module uses functionality of another module
    - Layered
      - Careful control of uses relation

Illustration- Module Views

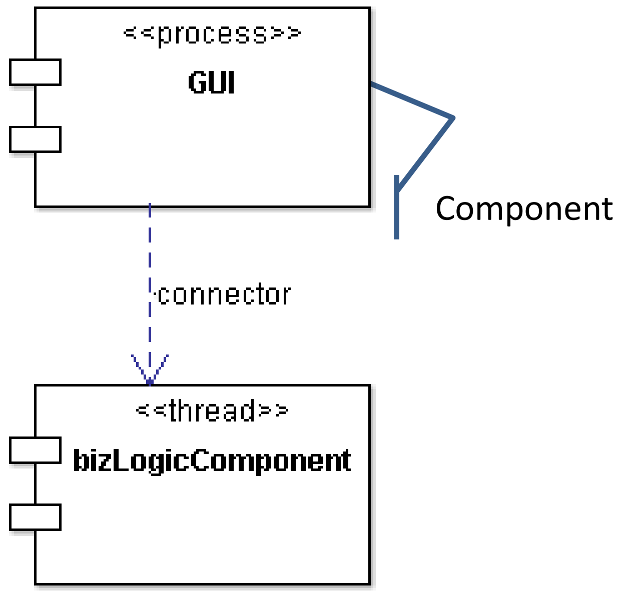


- Packages are used as modules
- Other approaches
  - One can use class, or interface to denote a module
- Relations
  - One can use various other UML relations to denote uses, layering or generalization

Three Structures- Recap

- Component-Connector Structure
  - Processes
    - Components are processes and relations are communications among them
  - Concurrency
    - Relationships between components- control flow dependency, and parallelism
  - Client-Server
    - Components are clients or servers, connectors are protocols
  - Shared data
    - Components have data store, and connectors describe how data is created, stored, retrieved

Illustration- CNC Views

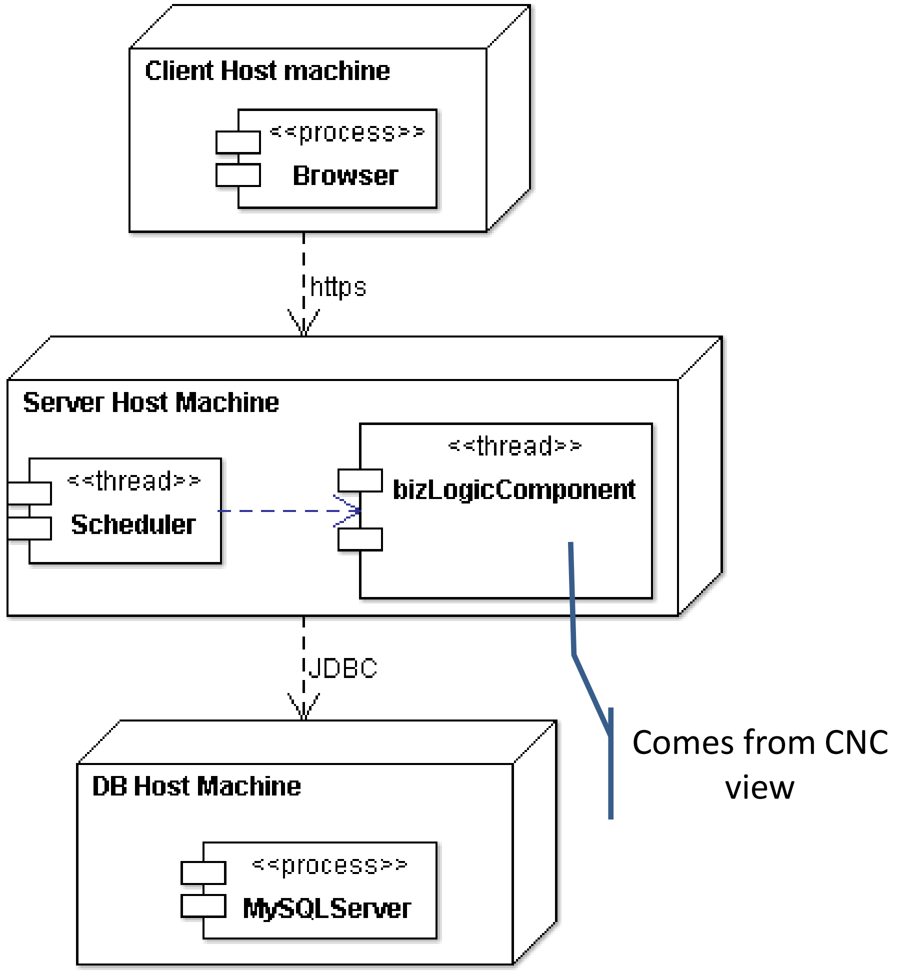


- No standard representation exists
- UML components are used with stereotypes
- Other approaches
  - One can use class, interface, or package to denote a component
- Relations
  - One can use various other UML relations such as association class

Three Structures- Recap

- Allocation Structure
  - Deployment
    - Units are software (processes from component-connector) and hardware processors
    - Relation means how a software is allocated or migrated to a hardware
  - Implementation
    - Units are modules (from module view) and connectors denote how they are mapped to files, folders
  - Work assignment
    - Assigns responsibility for implementing and integrating the modules to people or team

Illustration-Allocation Views



- UML Deployment diagram is a good option for deployment structure
- No specific recommendation for work assignment and implementation

Thank You

SS ZG653 (RL 8.3): Software Architecture

Introduction to Unified Process

Instructor: Prof. Santonu Sarkar

BITS Pilani

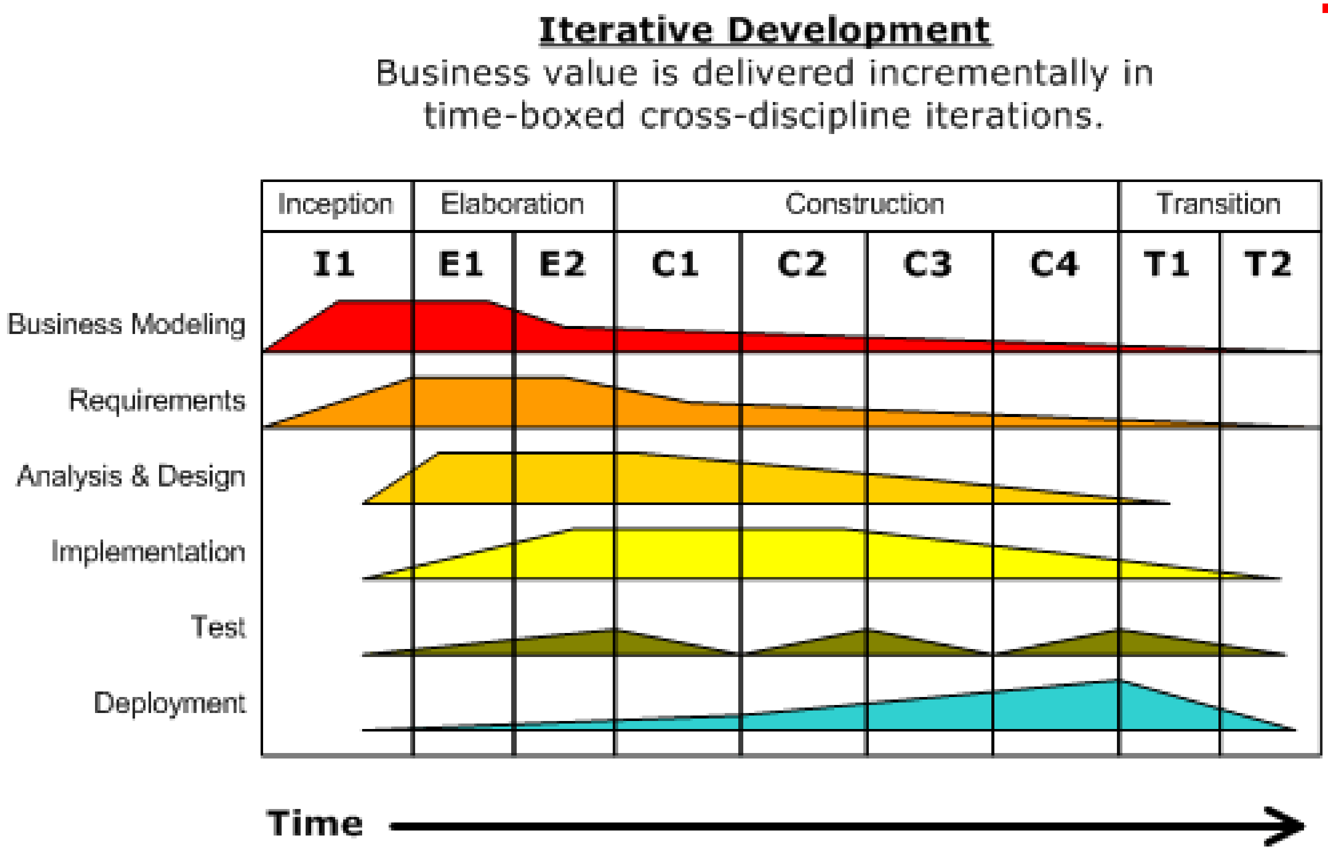
Pilani|Dubai|Goa|Hyderabad

What is (Rational) Unified Process

- While UML provided the necessary technology for OO software design
- Unified process gives a framework to build the software using UML
- Iterative approach
- Five main phases
  - Inception
    - Establish a justification and define project scope
    - Outline the use cases and key requirements that will drive the design tradeoffs Outline one or more candidate architectures
    - Identify risks and prepare a preliminary project schedule alongwith cost estimate
  - Elaboration (Architecture and Design)
  - Construction (Actual implementation)
  - Transition (initial release)
  - Production (Actual deployment)

– Rumbaugh, Booch, Jacobson

Architecture Activities



- Architecture Focused all the time
- Starts during inception and described in detail during the elaboration phase

"Development-iterative" by Dutchguilder - Own work. Licensed under Public Domain via Commons – https://commons.wikimedia.org/wiki/File:Development-iterative.png#/media/File:Development-iterative.png

Feature Driven Design

- Starts during Inception-Elaboration Phase
- Mostly used in the context of Agile development
- The requirement is modeled as a set of features
  - A feature is a client-valued functionality that can be implemented and demonstrated quickly
- ‘Feature’ template <*action*> **the** <*result*> (**by**|**for**|**of**|**to**) **a(n)** <*object*>
  - Add a product to a shopping-cart
  - Store the shipping-information for the customer

From Feature to Architecture

- Once a set of features are collected
- Similar features are grouped together into a module
- Set of clusters created out of the features, forms a set of modules
- This becomes the basic Module Structure
  – Recall the software architecture and views…

Use-case Model

- Use-cases are used to describe an usage scenario from the user's point of view
  - Create a basic use-case diagram
  - Elaborate each use-case
- To create Use-Case
  - Define actors (who will use this use-case)
  - Describe the scenario
    - Preconditions
    - Main tasks
    - Exceptions
    - Variation in the actors interaction
    - What system information will the actor acquire, produce or change?
    - Will the actor have to inform about the changes?
    - Does the actor wish to be informed about any unexpected changes?

First step towards Module Views

- Analysis Classes are not the final implementation level classes. They are more coarse grained. They are typically modules. They manifest as
  - External Entities
    - Other systems, devices that produce or consume information related to this system
  - Things
    - Report, letters, signals
  - Structure
    - Sensors, four-wheeled car,… that define a class of objects

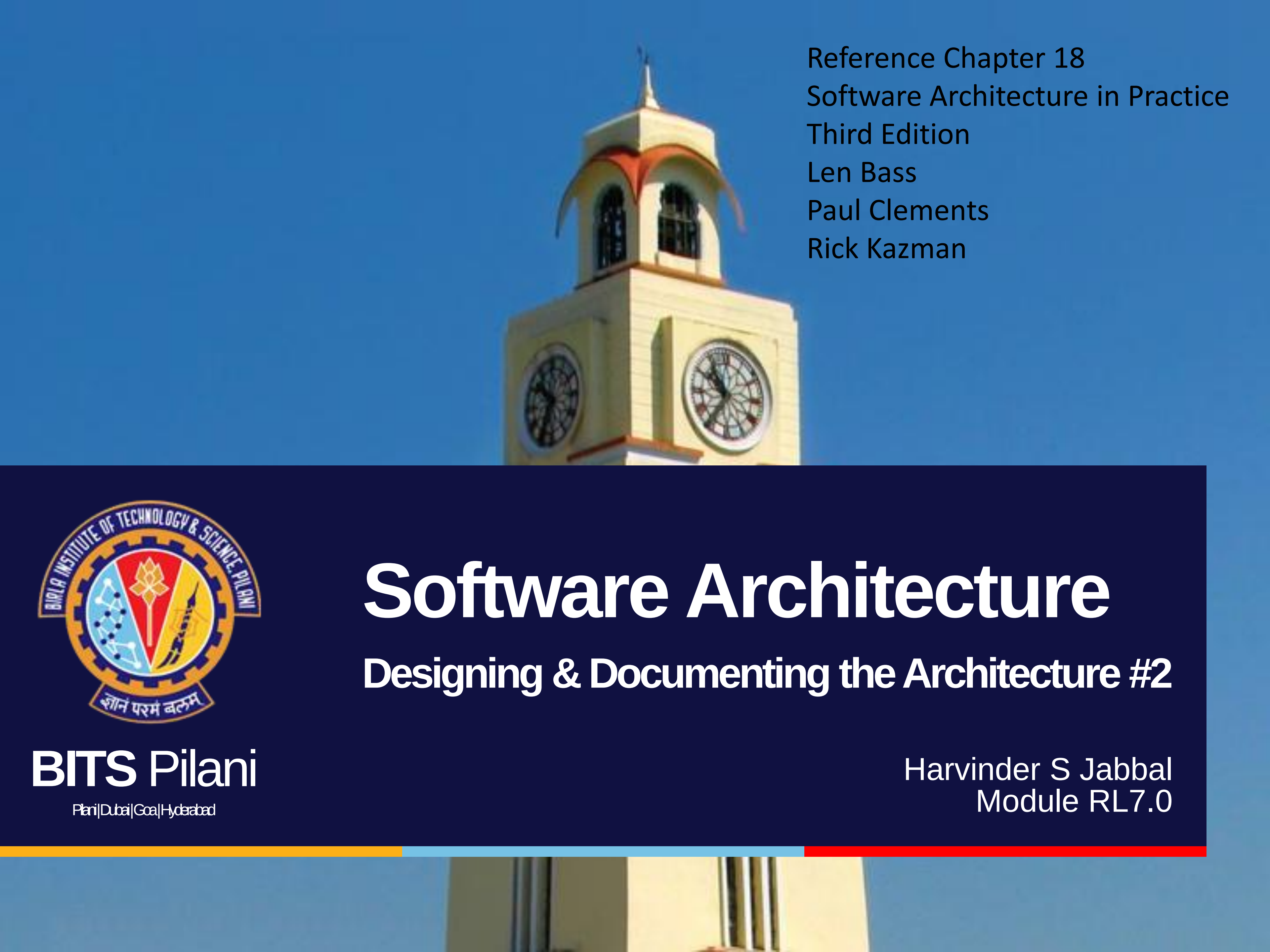
Analysis Classes manifest as…

- Event Occurrences
  - Transfer of fund, Completion of Job
- Roles
  - People who interact with the systems (Manager, engineer, etc.)-- Actors
- Organizational Units
  - Divisions or groups that are important for this system
- Places
  - Location that establish the context of the overall functionality of the system

How to get these classes?

- Get the noun phrases. They are the potential objects.
- Apply the following heuristics to get a legitimate analysis class
  - It should retain information for processing
  - It should have a set of identifiable operations
  - Multiple attributes should be there for a class
  - Operations should be applicable for all instances of this class
  - Attributes should be meaningful for all instances of this class

Thank You



Reference Chapter 18
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Software Architecture

Designing & Documenting the Architecture #2

Harvinder S Jabbal
Module RL7.0

Designing & Documenting the Architecture #2

Documenting the Architecture with relevant views

- Module,
- C&C,
- Allocation &
- Quality

Documenting beyond views,

- Documenting behaviours with sequence diagram

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Documenting the Architecture with relevant views – Module, C&C, Allocation & Quality

Architecture Documentation

Even the best architecture will be useless if the people who need it

- do not know what it is;
- cannot understand it well enough to use, build, or modify it;
- misunderstand it and apply it incorrectly.

All of the effort, analysis, hard work, and insightful design on the part of the architecture team will have been wasted.

Chapter Outline

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

1. Uses and Audience for Architecture Documentation

1. Uses and Audience for Architecture Documentation

Architecture documentation must

- be sufficiently transparent and accessible to be quickly understood by new employees
- be sufficiently concrete to serve as a blueprint for construction
- have enough information to serve as a basis for analysis.

Architecture documentation is both prescriptive and descriptive.

- For some audiences, it prescribes what *should* be true, placing constraints on decisions yet to be made.
- For other audiences, it describes what *is* true, recounting decisions already made about a system’s design.

Understanding stakeholder uses of architecture documentation is essential

- Those uses determine the information to capture.

Three Uses for Architecture Documentation

Education

- Introducing people to the system
  - New members of the team
  - External analysts or evaluators
  - New architect

Primary vehicle for communication among stakeholders

- Especially architect to developers
- Especially architect to future architect!

Basis for system analysis and construction

- documentation serves as the basis for architecture evaluation.

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

2. Notations

2. Notations

Informal notations

- Views are depicted (often graphically) using general-purpose diagramming and editing tools
- The semantics of the description are characterized in natural language
- They cannot be formally analyzed

Semiformal notations

- Standardized notation that prescribes graphical elements and rules of construction
- Lacks a complete semantic treatment of the meaning of those elements
- Rudimentary analysis can be applied
- UML is a semiformal notation in this sense.

Formal notations

- Views are described in a notation that has a precise (usually mathematically based) semantics.
- Formal analysis of both syntax and semantics is possible.
- Architecture description languages (ADLs)
- Support automation through associated tools.

Choosing a Notation

Tradeoffs

- Typically, more formal notations take more time and effort to create and understand, but offer reduced ambiguity and more opportunities for analysis.
- Conversely, more informal notations are easier to create, but they provide fewer guarantees.

Different notations are better (or worse) for expressing different kinds of information.

- UML class diagram will not help you reason about schedulability, nor will a sequence chart tell you very much about the system’s likelihood of being delivered on time.
- Choose your notations and representation languages knowing the important issues you need to capture and reason about.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

3. Views

3. Views

Views let us divide a software architecture into a number of (we hope) interesting and manageable representations of the system.

Principle of architecture documentation:

- *Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.*

Which Views? The Ones You Need!

Different views support different goals and uses.

We do not advocate a particular view or collection of views.

The views you should document depend on the uses you expect to make of the documentation.

Each view has a cost and a benefit; you should ensure that the benefits of maintaining a view outweigh its costs.

Overview of Module Views

Elements

- Modules, which are implementation units of software that provide a coherent set of responsibilities.

Relations

- *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole.
- *Depends on*, which defines a dependency relationship between two modules. Specific module views elaborate what dependency is meant.
- *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.

Overview of Module Views

Constraints

- Different module views may impose specific topological constraints, such as limitations on the visibility between modules.

Usage

- Blueprint for construction of the code
- Change-impact analysis
- Planning incremental development
- Requirements traceability analysis
- Communicating the functionality of a system and the structure of its code base
- Supporting the definition of work assignments, implementation schedules, and budget information
- Showing the structure of information that the system needs to manage

Module Views

- It is unlikely that the documentation of any software architecture can be complete without at least one module view.

Overview of C&C Views

Elements

- *Components.* Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors).
- *Connectors*. Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components may use a connector in interactions.

Relations

- *Attachments*. Component ports are associated with connector roles to yield a graph of components and connectors.
- *Interface delegation.* In some situations component ports are associated with one or more ports in an “internal” subarchitecture. The case is similar for the roles of a connector

Overview of C&C Views

Constraints

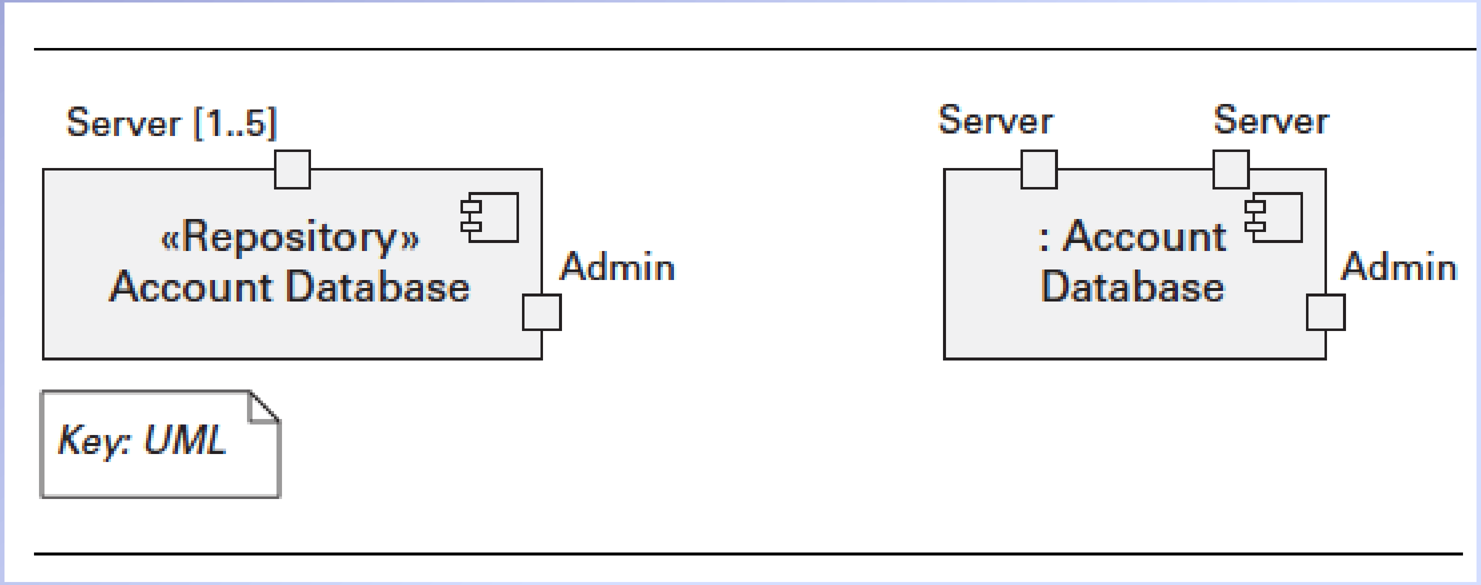
- Components can only be attached to connectors, not directly to other components.
- Connectors can only be attached to components, not directly to other connectors.
- Attachments can only be made between compatible ports and roles.
- Interface delegation can only be defined between two compatible ports (or two compatible roles).
- Connectors cannot appear in isolation; a connector must be attached to a component.

Usage

- Show how the system works.
- Guide development by specifying structure and behavior of runtime elements.
- Help reason about runtime system quality attributes, such as performance and availability.

Notations for C&C Views

UML components are good match for C&C components.



Suggested Reading:

http://agilemodeling.com/artifacts/componentDiagram.htm

Notations for C&C Views

UML connectors are not rich enough to represent many C&C connectors.

- UML connectors cannot have substructure, attributes, or behavioral descriptions.

Represent a "simple" C&C connector using a UML connector—a line.

- Many commonly used C&C connectors have well-known, application-independent semantics and implementations, such as function calls or data read operations.
- You can use a stereotype to denote the type of connector.

Connector roles cannot be explicitly represented with a UML connector.

- The UML connector element does not allow the inclusion of interfaces.
- Label the connector ends and use these labels to identify role descriptions that must be documented elsewhere.

Represent a "rich" C&C connector

- using a UML component, or by annotating a line UML connector with a tag that explains the meaning of the complex connector.

Overview of Allocation Views

Elements

- *Software element* and *environmental element.*
- A software element has properties that are *required* of the environment.
- An environmental element has properties that are *provided* to the software.

Relations

- *Allocated to*. A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular view.

Overview of Allocation Views

Constraints

- Varies by view

Usage

- Reasoning about performance, availability, security, and safety.
- Reasoning about distributed development and allocation of work to teams.
- Reasoning about concurrent access to software versions.
- Reasoning about the form and mechanisms of system installation.

Quality Views

A *quality view* can be tailored

- for specific stakeholders or to address specific concerns.

A quality views is formed

- by extracting the relevant pieces of structural views and packaging them together.

Quality Views: Examples

- Show the components that have some security role or responsibility, how those components communicate, any data repositories for security information, and repositories that are of security interest.
- The view's context information would show other security measures (such as physical security) in the system's environment.
- The behavior part of a security view
  - Show how the operation of security protocols and where and how humans interact with the security elements.
  - Capture how the system would respond to

Security view

- Especially helpful for systems that are globally dispersed and heterogeneous.
- Show all of the component-to-component channels, the various network channels, quality-of-service parameter values, and areas of concurrency.
- Used to analyze certain kinds of performance and reliability (such as deadlock or race condition detection).
- The behavior part of this view could show (for example) how network bandwidth is dynamically allocated.

Communications view

Quality Views: Examples

- Could help illuminate and draw attention to error reporting and resolution mechanisms.
- Show how components detect, report, and resolve faults or errors.
- It would help identify the sources of errors

Exception or *error-handling view*

- Models mechanisms such as replication and switchover.
- Depicts timing issues and transaction integrity.

Reliability view

- Shows those aspects of the architecture useful for inferring the system's performance.
- Show network traffic models, maximum latencies for operations, and so forth.

Performance view

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Sample separator slide for the presentation

4. Choosing the Views

You can determine which views are required, when to create them, and how much detail to include if you know the following:

What people, and with what skills, are available	Which standards you have to comply with	What budget is on hand	What the schedule is	What the information needs of the important stakeholders are	What the driving quality attribute requirements are	What the approximate size of the system is

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

5. Combining Views

5. Combining Views

At a minimum, expect to have

- at least one module view,
- at least one C&C view,
- and for larger systems, at least one allocation view in your architecture document.

Method for Choosing the Views

Step 1. Build a stakeholder/view table.

- Rows: List the stakeholders for your project's software architecture documentation
- Columns: Enumerate the views that apply to your system.
  - Use the structures discussed in Chapter 1, the views discussed in this chapter, and the views that your design work in ADD has suggested as a starting list of candidates.
  - Include the views or view sketches you have as a result of your design work so far.
- Some views (such as decomposition, uses, and work assignment) apply to every system, while others (various C&C views, the layered view) only apply to some systems.
- Fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

Method for Choosing the Views

Step 2. Combine views to reduce their number

- Look for marginal views in the table; those that require only an overview, or that serve very few stakeholders.
- Combine each marginal view with another view that has a stronger constituency.
- These views often combine naturally:
  - *Various C&C views.* Because C&C views all show runtime relations among components and connectors of various types, they tend to combine well.
  - *Deployment view with either SOA or communicating-processes views.* An SOA view shows services, and a communicating-processes view shows processes. In both cases, these are components that are deployed onto processors.
  - *Decomposition view and any of work assignment, implementation, uses, or layered views.* The decomposed modules form the units of work, development, and uses. In addition, these modules populate layers.

Method for Choosing the Views

Step 3. Prioritize and stage.

- The decomposition view (one of the module views) is a particularly helpful view to release early.
  - High-level decompositions are often easy to design
  - The project manager can start to staff development teams, put training in place, determine which parts to outsource, and start producing budgets and schedules.
- You don't have to satisfy all the information needs of all the stakeholders to the fullest extent.
  - Providing 80 percent of the information goes a long way, and this might be "good enough" so that the stakeholders can do their job.
  - Check with the stakeholder if a subset of information would be sufficient.
- You don't have to complete one view before starting another.
  - People can make progress with overview-level information
  - A breadth-first approach is often the best.

6. Building the Documentation Package

6. Building the Documentation Package

Documentation package consists of

- Views
- Documentation beyond views

Documenting a View

Section 1: The Primary Presentation.

- The *primary presentation* shows the elements and relations of the view.
- The primary presentation should contain the information you wish to convey about the system—in the vocabulary of that view.
- The primary presentation is most often graphical.
  - It might be a diagram you've drawn in an informal notation using a simple drawing tool, or it might be a diagram in a semiformal or formal notation imported from a design or modeling tool that you're using.
  - If your primary presentation is graphical, make sure to include a key that explains the notation.
  - Lack of a key is the most common mistake that we see in documentation in practice.
- Occasionally the primary presentation will be textual, such as a table or a list.
  - If that text is presented according to certain stylistic rules, these rules should be stated or incorporated by reference, as the analog to the graphical notation key.

Documenting a View

Section 2: The Element Catalog.

- The *element catalog* details at least those elements depicted in the primary presentation.
  - For instance, if a diagram shows elements A, B, and C, then the element catalog needs to explain what A, B, and C are.
  - If elements or relations relevant to this view were omitted from the primary presentation, they should be introduced and explained in the catalog.
- Parts of the catalog:
  - *Elements and their properties*. This section names each element in the view and lists the properties of that element. Each view introduced in Chapter 1 listed a set of suggested properties associated with that view.
  - *Relations and their properties*. Each view has specific relation types that it depicts among the elements in that view.
  - *Element interfaces*. This section documents element interfaces.
  - *Element behavior*. This section documents element behavior that is not obvious from the primary presentation.

Documenting a View

Section 3: Context Diagram.

- A *context diagram* shows how the system or portion of the system depicted in this view relates to its environment.
- The purpose of a context diagram is to depict the scope of a view.
- Entities in the environment may be humans, other computer systems, or physical objects, such as sensors or controlled devices.

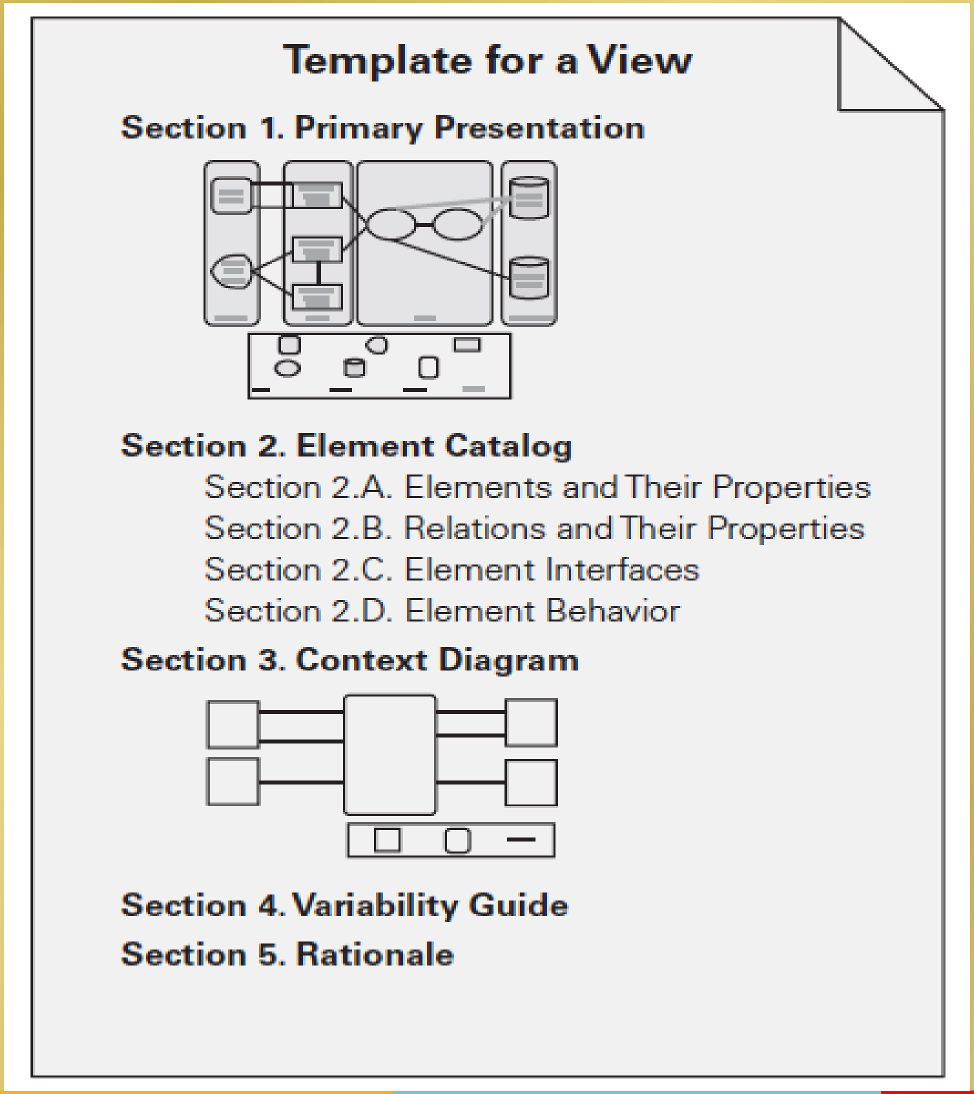
Section 4: Variability Guide.

- A *variability guide* shows how to exercise any variation points that are a part of the architecture shown in this view.

Section 5: Rationale.

- *Rationale* explains why the design reflected in the view came to be.
- The goal of this section is to explain why the design is as it is and to provide a convincing argument that it is sound.
- The choice of a pattern in this view should be justified here by describing the architectural problem that the chosen pattern solves and the rationale for choosing it over another.

View Template



Template for a View

Section 1. Primary Presentation

Section 2. Element Catalog

Section 2.A. Elements and Their Properties

Section 2.B. Relations and Their Properties

Section 2.C. Element Interfaces

Section 2.D. Element Behavior

Section 3. Context Diagram

Section 4. Variability Guide

Section 5. Rationale

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Documenting beyond views,

Documenting Information Beyond Views

innovate achieve lead

Document control information.

List the

- issuing organization,
- the current version number,
- date of issue and
- status,
- a change history, and
- the procedure for submitting change requests to the document.

Usually captured in the front matter

Documenting Information Beyond Views

Section 1: Documentation Roadmap. The documentation map tells the reader what information is in the documentation and where to find it.

- *Scope and summary*. Explain the purpose of the document and briefly summarize what is covered.
- *How the documentation is organized*. For each section in the documentation, give a short synopsis of the information that can be found there.
- *View overview*. Describes the views that the architect has included in the package. For each view::
  - The name of the view and what pattern it instantiates, if any.
  - A description of the view's element types, relation types, and property types.
  - A description of language, modeling techniques, or analytical methods used in constructing the view.
- *How stakeholders can use the documentation*.
  - This section shows how various stakeholders might use the documentation to help address their concerns.
  - Include short scenarios, such as "A maintainer wishes to know the units of software that are likely to be changed by a proposed modification."
  - To be compliant with ISO/IEC 42010-2007, you must consider the concerns of at least users, acquirers, developers, and maintainers.

Documenting Information Beyond Views

Section 2: How a View Is Documented.

- Explain the standard organization you're using to document views—either the one described in this chapter or one of your own.

Section 3: System Overview.

- Short prose description of the system's function, its users, and any important background or constraints.
- Provides your readers with a consistent mental model of the system and its purpose.
- This might be a pointer to your project's concept-of-operations document for the system.

Documenting Information Beyond Views

Section 4: Mapping Between Views.

- Helping a reader understand the associations between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole.
- The associations between elements across views in an architecture are, in general, many-to-many.
- View-to-view associations can be captured as tables.
  - The table should name the correspondence between the elements across the two views.
  - Examples
    - "is implemented by" for mapping from a component-and-connector view to a module view
    - "implements" for mapping from a module view to a component-and-connector view
    - "included in" for mapping from a decomposition view to a layered view

Documenting Information Beyond Views

Section 5: Rationale.

- Documents the architectural decisions that apply to more than one view.
  - Documentation of background or organizational constraints or major requirements that led to decisions of system-wide import.
  - Decisions about which fundamental architecture patterns are used.

Section 6: Directory.

- Set of reference material that helps readers find more information quickly.
  - Index of terms
  - Glossary
  - Acronym list.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

7 Documenting behaviours

7. Documenting Behavior

Behavior documentation complements each views by describing how architecture elements in that view interact with each other.

Behavior documentation enables reasoning about

- a system’s potential to deadlock
- a system’s ability to complete a task in the desired amount of time
- maximum memory consumption
- and more

Behavior has its own section in our view template’s element catalog.

Notations for Documenting Behavior

Trace-oriented languages

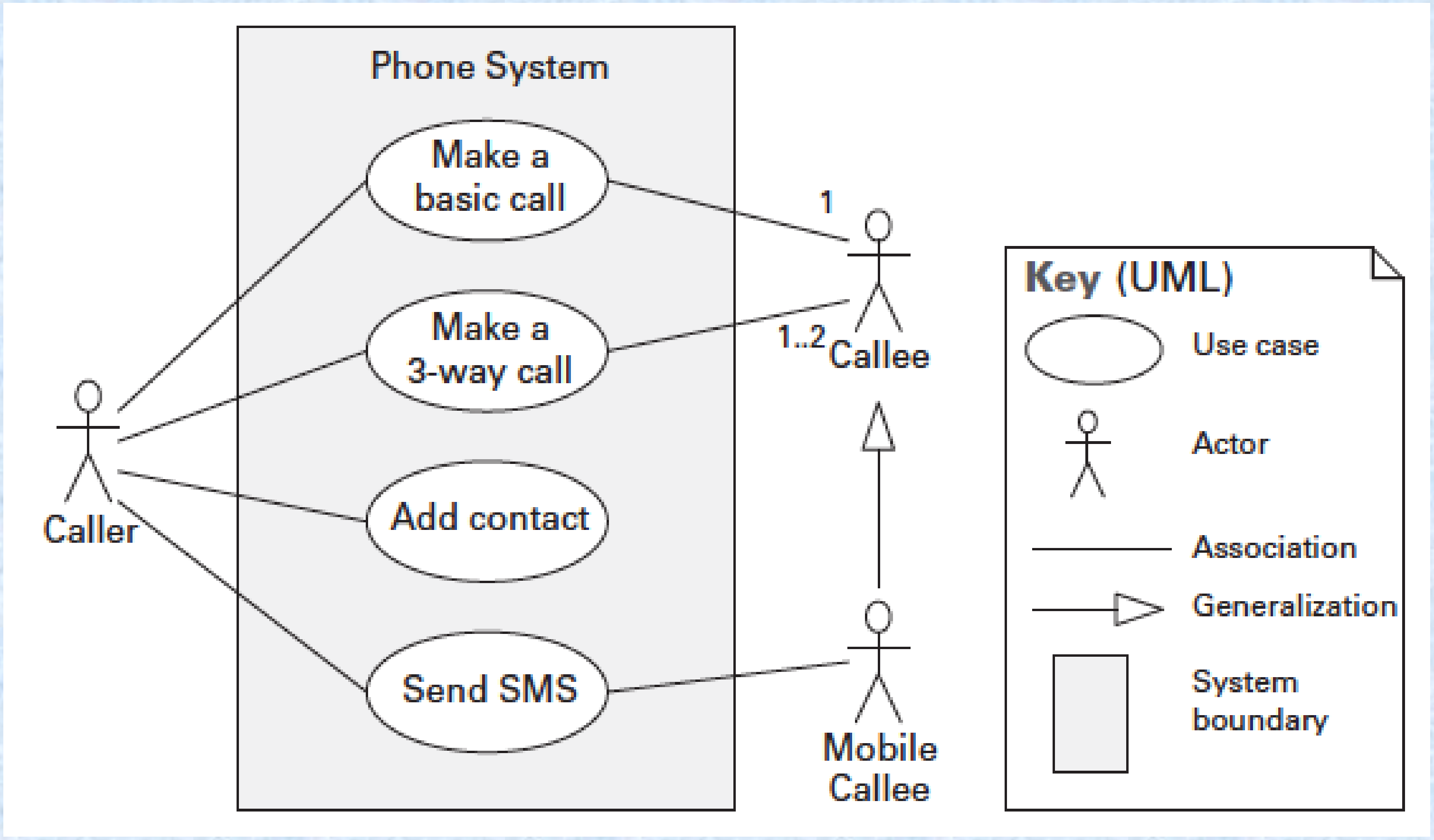
Traces are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state.

A trace describes a particular sequence of activities or interactions between structural elements of the system.

Examples

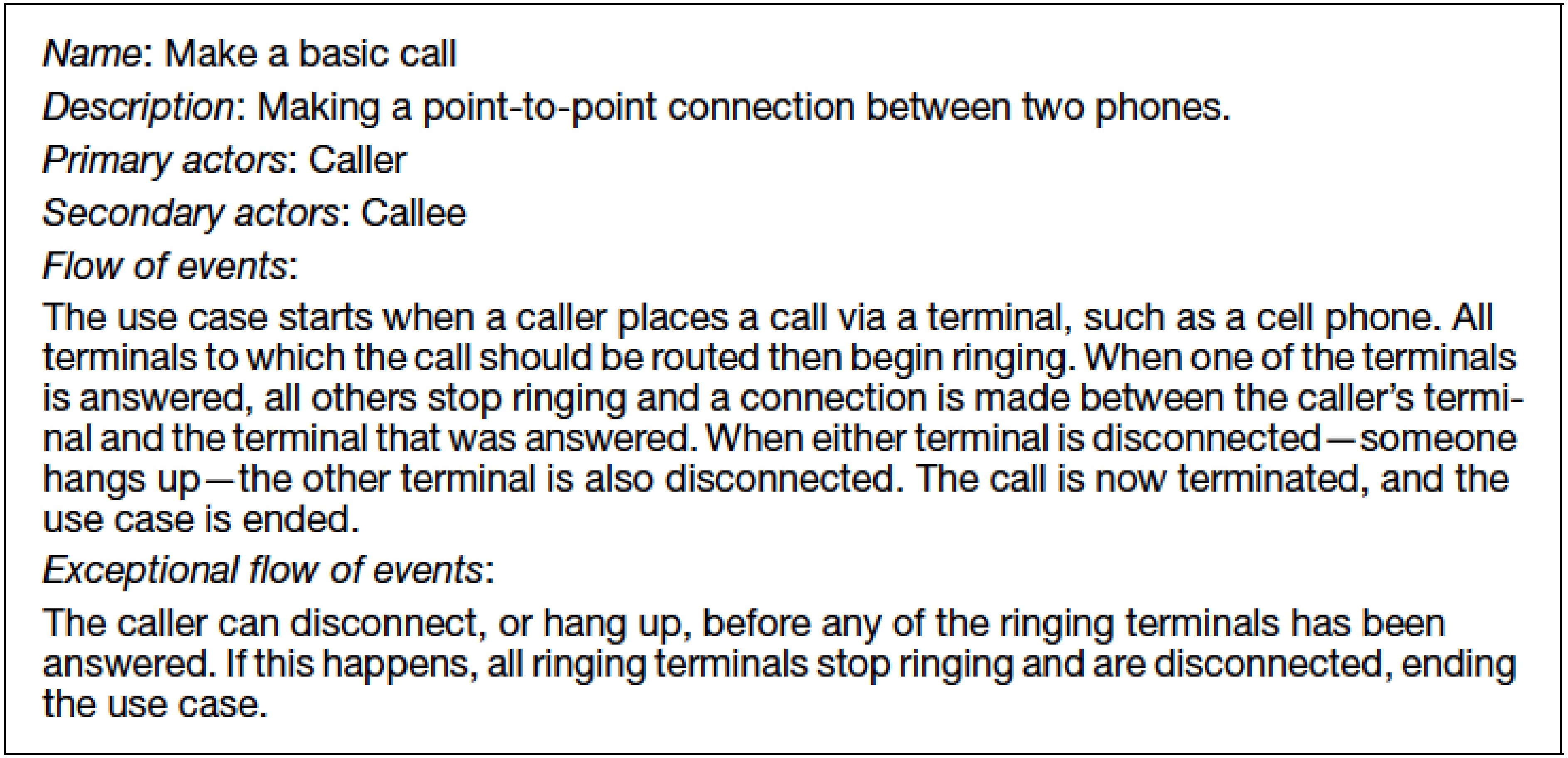
- use cases
- sequence diagrams
- communication diagrams
- activity diagrams
- message sequence charts
- timing diagrams
- Business Process Execution Language

Use Case Diagram



From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Use Case Description



Name: Make a basic call

Description: Making a point-to-point connection between two phones.

Primary actors: Caller

Secondary actors: Callee

Flow of events:

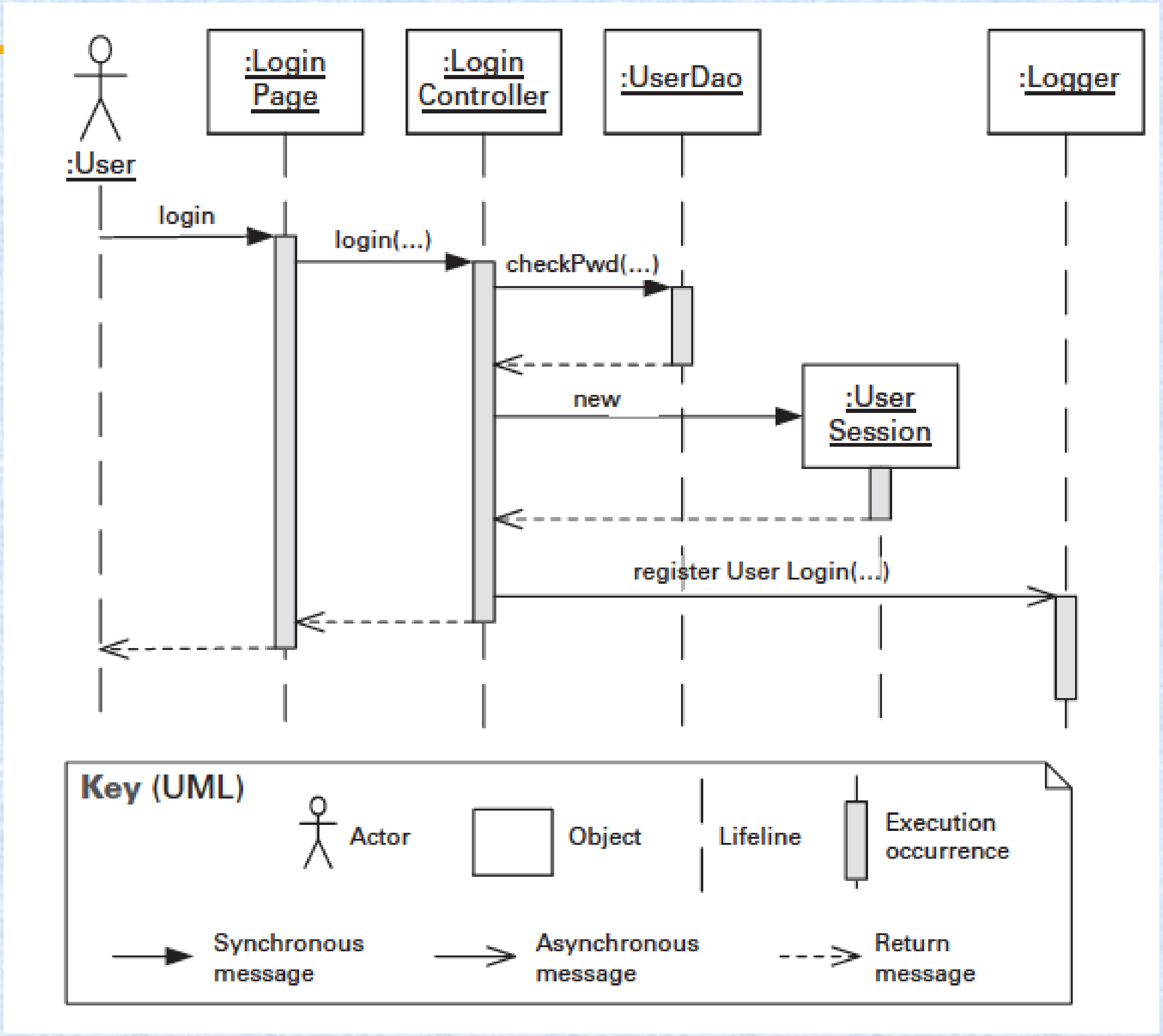
The use case starts when a caller places a call via a terminal, such as a cell phone. All terminals to which the call should be routed then begin ringing. When one of the terminals is answered, all others stop ringing and a connection is made between the caller's terminal and the terminal that was answered. When either terminal is disconnected—someone hangs up—the other terminal is also disconnected. The call is now terminated, and the use case is ended.

Exceptional flow of events:

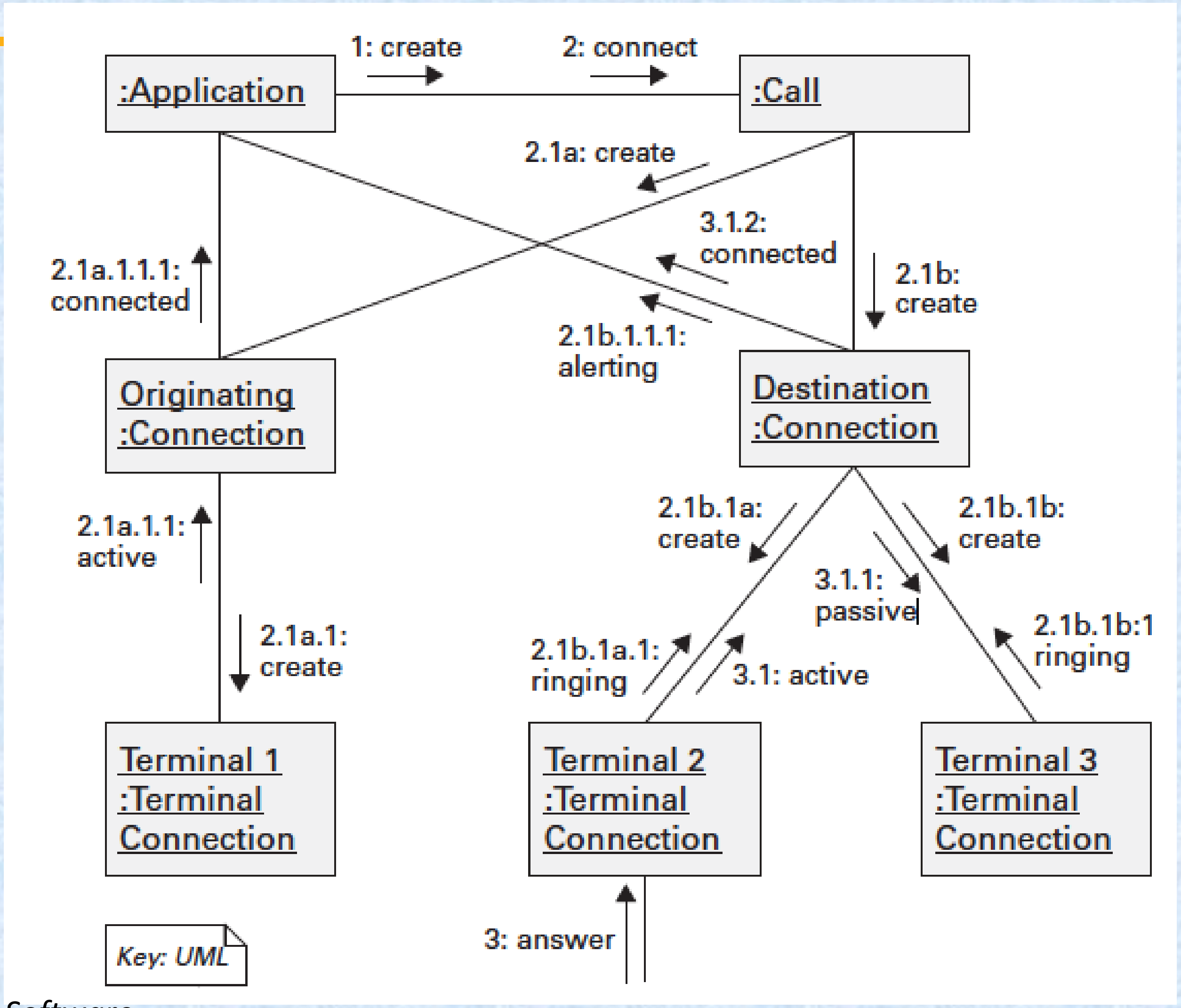
The caller can disconnect, or hang up, before any of the ringing terminals has been answered. If this happens, all ringing terminals stop ringing and are disconnected, ending the use case.

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Sequence Diagram

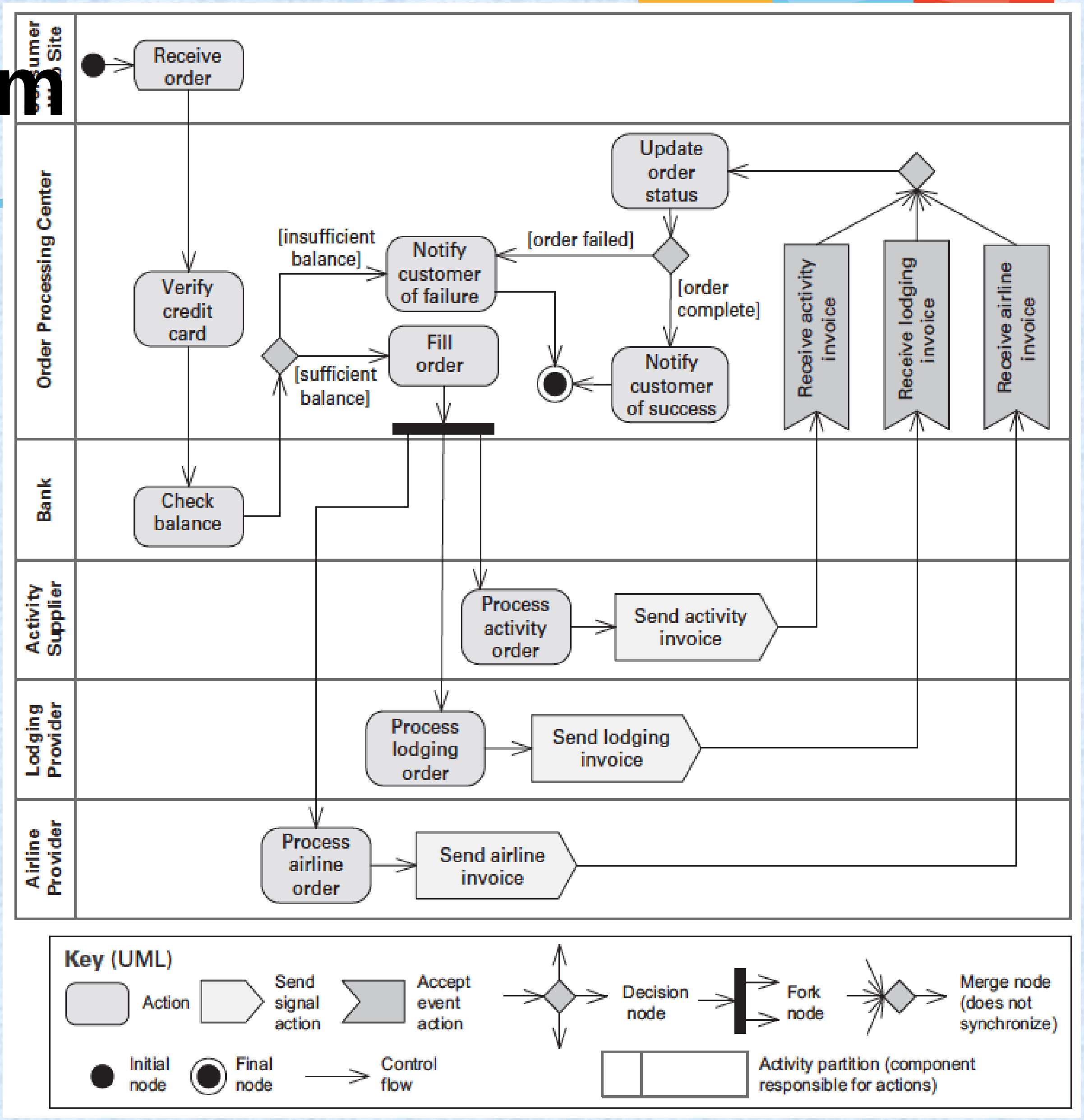


Communication Diagram



From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Activity Diagram



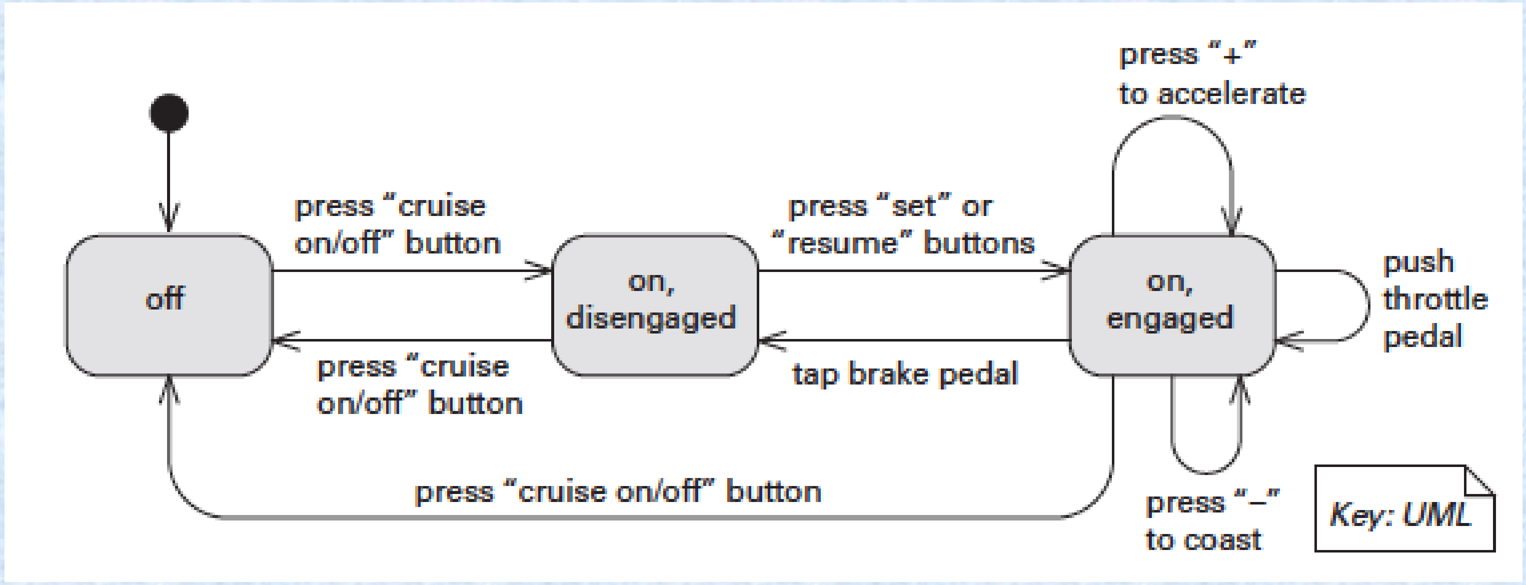
From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Notations for Documenting Behavior

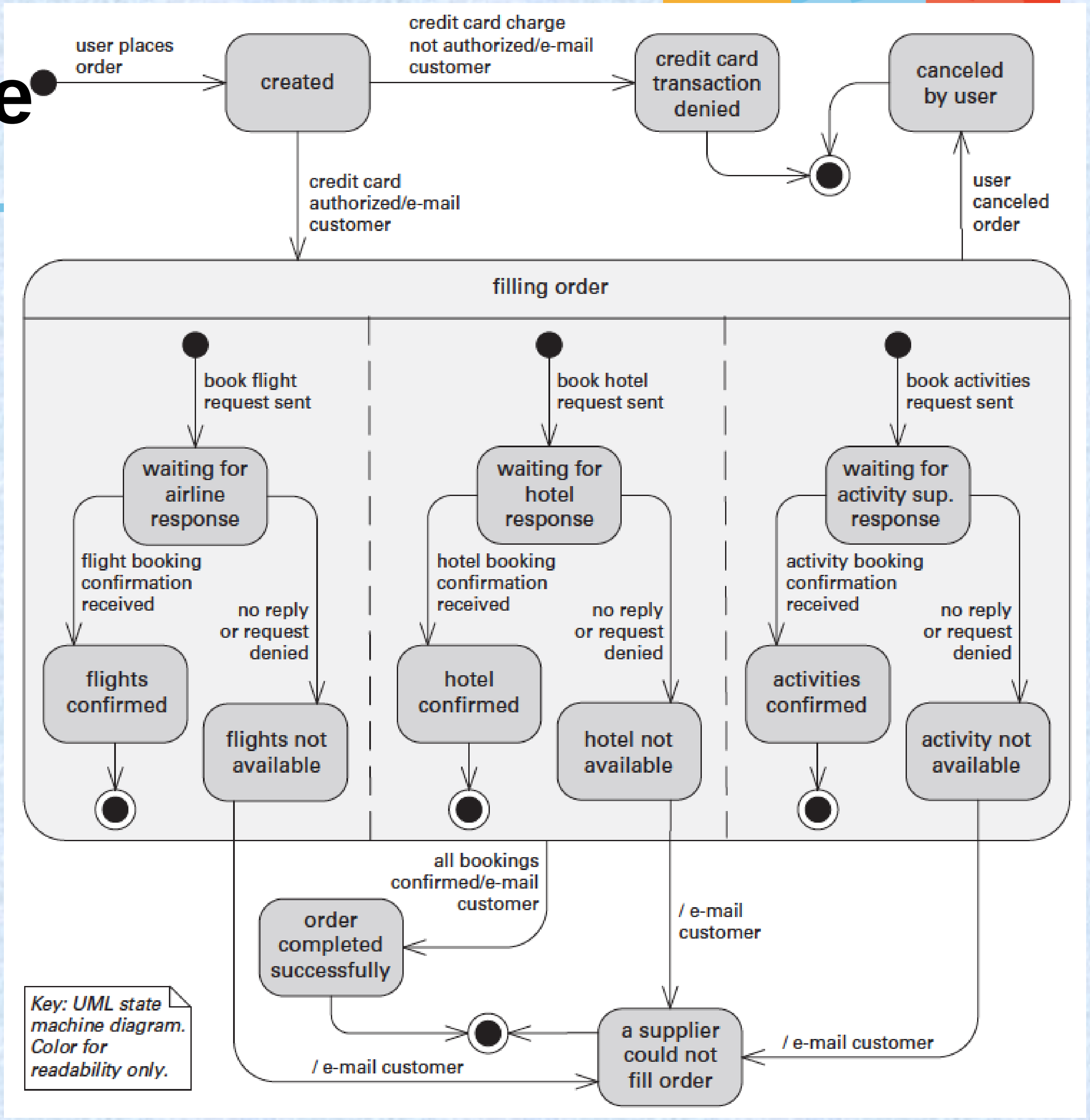
Comprehensive languages

- *Comprehensive models* show the complete behavior of structural elements.
- Given this type of documentation, it is possible to infer all possible paths from initial state to final state.
- The state machine formalism represents the behavior of architecture elements because each state is an abstraction of all possible histories that could lead to that state.
- State machine languages allow you to complement a structural description of the elements of the system with constraints on interactions and timed reactions to both internal and environmental stimuli.

State Machine


off
on, disengaged
on, engaged
press "cruise on/off" button
press "cruise on/off" button
press "set" or "resume" buttons
tap brake pedal
press "+" to accelerate
push throttle pedal
press "–" to coast
press "cruise on/off" button
Key: UML

State Machine



user places order
created
credit card charge not authorized/e-mail customer
credit card transaction denied
canceled by user
credit card authorized/e-mail customer
user canceled order
filling order
book flight request sent
waiting for airline response
flight booking confirmation received
no reply or request denied
flights confirmed
flights not available
book hotel request sent
waiting for hotel response
hotel booking confirmation received
no reply or request denied
hotel confirmed
hotel not available
book activities request sent
waiting for activity sup. response
activity booking confirmation received
no reply or request denied
activities confirmed
activity not available
all bookings confirmed/e-mail customer
order completed successfully
/ e-mail customer
a supplier could not fill order
/ e-mail customer
/ e-mail customer
Key: UML state machine diagram. Color for readability only.

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

8. Documenting Quality Attributes

8. Documenting Quality Attributes

Where do quality attributes show up in the documentation? There are five major ways:

Rationale that explains the choice of design approach should include a discussion about the quality attribute requirements and tradeoffs.	Architectural elements providing a service often have quality attribute bounds assigned to them, defined in the interface documentation for the elements, or recorded as *properties* that the elements exhibit.	Quality attributes often impart a “language” of things that you would look for. Someone fluent in the “language” of a quality attribute can search for the kinds of architectural elements) put in place to satisfy that quality attribute requirement.	Architecture documentation often contains a *mapping to requirements* that shows how requirements (including quality attribute requirements) are satisfied.	Every quality attribute requirement will have a constituency of stakeholders who want to know that it is going to be satisfied. For these stakeholders, the roadmap tells the stakeholder where in the document to find it.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

9. Documenting Architectures That Change Faster Than You Can Document Them

9. Documenting Architectures That Change Faster Than You Can Document Them

An architecture that changes at runtime, or as a result of a high-frequency release-and-deploy cycle, change much faster than the documentation cycle.

Nobody will wait until a new architecture document is produced, reviewed, and released.

In this case:

- *Document what is true about all versions of your system.* Record those invariants as you would for any architecture. This may make your documented architecture more a description of constraints or guidelines that any compliant version of the system must follow.
- *Document the ways the architecture is allowed to change.* This will usually mean adding new components and replacing components with new implementations. The place to do this is called the variability guide.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

10. Documenting Architecture in an Agile Development Project

10. Documenting Architecture in an Agile Development Project

Adopt a template or standard organization to capture your design decisions.

Plan to document a view if (but only if) it has a strongly identified stakeholder constituency.

Fill in the sections of the template for a view, and for information beyond views, when (and in whatever order) the information becomes available. But only do this if writing down this information will make it easier (or cheaper or make success more likely) for someone downstream doing their job.

Don’t worry about creating an architectural design document and then a finer-grained design document. Produce just enough design information to allow you to move on to code.

Don’t feel obliged to fill up all sections of the template, and certainly not all at once. Write “N/A” for the sections for which you don’t need to record the information (perhaps because you will convey it orally).

Agile teams sometimes make models in brief discussions by the whiteboard. Take a picture and use it as the primary presentation.

Summary

- You must understand the uses to which the writing is to be put and the audience for the writing.
- Architectural documentation serves as a means for communication among various stakeholders, not only up the management chain and down to the developers but also across to peers.
- An architecture is a complicated artifact, best expressed by focusing on views.
- You must choose the views to document, must choose the notation to document these views, and must choose a set of views that is both minimal and adequate.
- You must document not only the structure of the architecture but also the behavior.

Thank you……..

Credits

- **Chapter Reference from Text T1: 16, 17, 18**
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
  - Len Bass
  - Paul Clements
  - Rick Kazman