SENG 401
Software Architecture

Lecture 4
Topic – Documenting Software Architecture
Sub-Topic – Module Styles

Gias Uddin
Assistant Professor, University of Calgary

http://giasuddin.ca/

Books Used/Cited in this Course

ID	Title
B1	Software Architecture: Foundations, Theory, and Practice. Taylor, Medvidovic, Dashofy. Wiley, 2009. *Available at UofC Library*
B2	The Architecture of Open-Source Applications. Amy Brown, Greg Wilson. Volume I. http://www.aosabook.org/en/index.html
B3	The Architecture of Open-Source Applications. Amy Brown, Greg Wilson. Volume II. http://www.aosabook.org/en/index.html
B4	**Documenting Software Architecture: Views and Beyond, 2nd Edition. Addison-Wesley, 2010. Clements et al.** ***Available at UofC Library***
B5	Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd Edition. Nick Rozanski. Addison-Wesley Professions, 2011. *Available at UofC Library*
B6	Software Architecture in Practice. Bass, Clements, Kazman. Addison-Wesley, 2012 *Available at UofC Library*

Documenting Software Architecture

Topic: Collection of Software Architectural Styles

Sub-Topic: Module Styles

References: B4 – C2

Outline

Module Styles in Architecture Documentation

Style	Description
Decomposition	used to show the structure of modules and submodules (that is, containment relations among modules)
Uses	used to indicate functional dependency relations among modules
Generalization	used to indicate specialization relations among modules
Layered	used to describe the *allowed-to-use* relation in a restricted fashion between groups of modules called layers
Aspects	used to describe particular modules called aspects that are responsible for crosscutting concerns
Data Model	used to show the relations among data entities

1. Decomposition Style
Overview

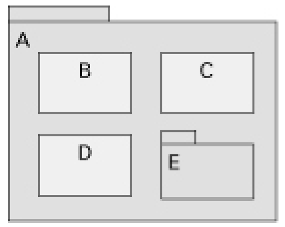
describes the organization of the code as modules and submodules and shows how system responsibilities are partitioned across them. Almost all architects begin with the decomposition style. Architects tend to attack a problem with divide-and-conquer techniques, and a decomposition view records their campaign.

Criteria Used for Decomposition and Explanation	
Achievement of quality attributes	to support modifiability, the information-hiding design principle calls for encapsulating changeable aspects of a system in separate modules, so that the change is localized.
Build-versus-buy decisions	Some modules may be reused from existing another software. These modules already have a set of responsibilities implemented. The remaining responsibilities then must be decomposed around those established modules.
Product line implementation	To support the efficient implementation of products of a product family, it is essential to distinguish between common modules, used in every or most products, and variable modules, which differ across products.
Team allocation	To allow implementation of different responsibilities in parallel, separate modules that can be allocated to different teams should be defined. The skills of developers also influence the decomposition. For example, if specialized Web developers are available, modules that handle the Web UI should be kept separate.

1. Decomposition Style
Summary of Key Points

Point	Description
Overview	Describes the organization of source code into a set of modules and sub-modules in a part-of hierarchical relationship
Elements	Module
Relations	A decomposition relation is a form of is-part-of relation. The documentation should specify the criteria used to define the decomposition relation.
Constraints	• No loops are allowed in the decomposition graph • A module can have only one parent
What it's for	• To reason about and communicate to newcomers the structure of software • To provide input for work assignment • To reason about the localization of changes

In a decomposition style, modules may be nested like below. In UML, module decomposition is shown by nesting, with the aggregate module shown as a package.



1. Decomposition Style
Notations. First an example real-world software system

The ATIA-M System

- Army Training Information Architecture-Migrated (ATIA-M) is a large Web-based, Java EE application that supports training in the U.S. Army. It has “thick clients”: Windows desktop applications developed using .NET (C#) that communicate with the server-side Java EE components using Web services technology.

The code is divided into three large modules:

- *Windowsapps* contains the code of the thick clients. The three submodules correspond to Training and Doctrine Development Tool (TDDT), Unit Training Management Configuration (UTMC), and a separate submodule with common code used by the different Windows applications. TDDT and UTMC were the two Windows applications originally planned, but others could be added.
- *ATIA server-side Web modules* contains all non-Java modules that would be deployed to server machines. The Web modules include JavaServer Pages (JSP) files, JavaScript and HTML code, and applets.
- *ATIA server-side Java modules* contains all Java source code in ATIA that would run on application servers. This module does not include JSP, JavaScript, HTML, applet, or thick-client code.

1. Decomposition Style
UML Notations

Figure 2.2 Top-level decomposition view for the ATIA system

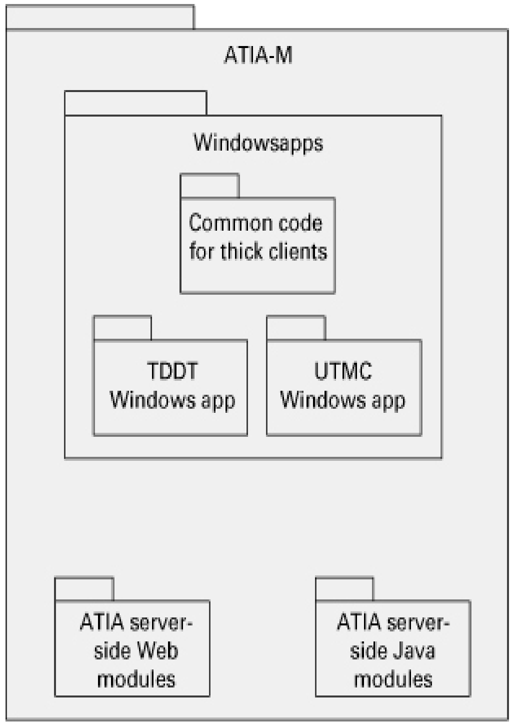
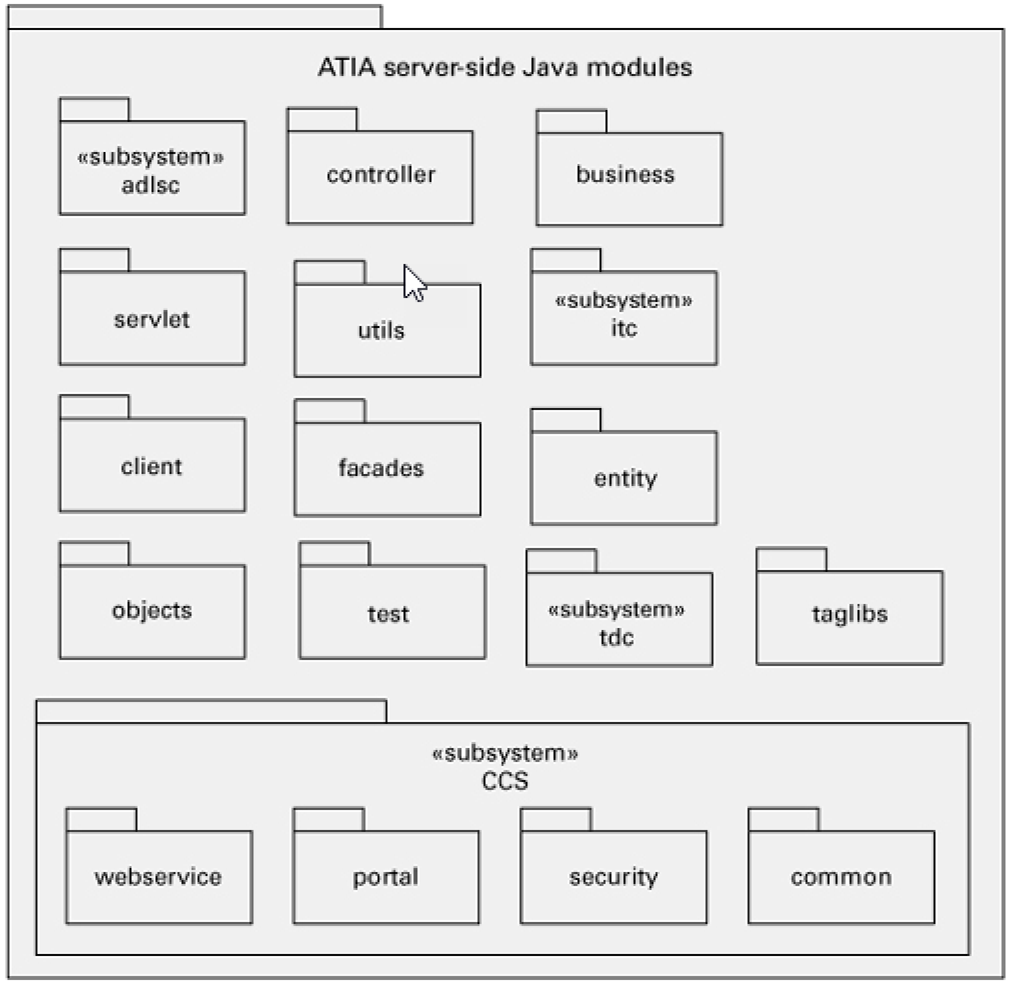

ATIA-M
Windowsapps
Common code for thick clients
TDDT Windows app
UTMC Windows app
ATIA server-side Web modules
ATIA server-side Java modules

Figure 2.3 Refinement of ATIA-M server-side Java modules showing how it is further decomposed into submodules

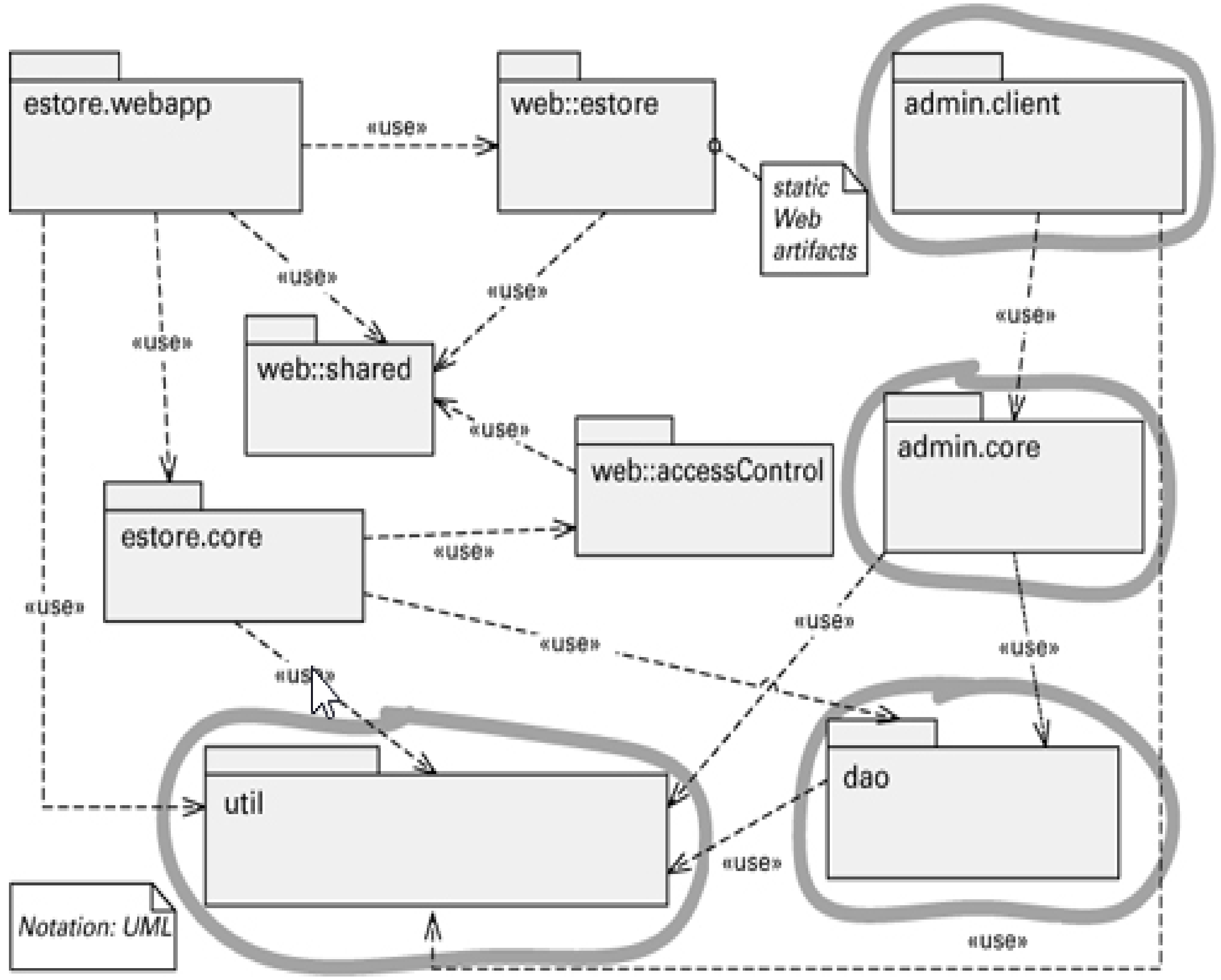


2. Uses Style

Overview

Useful for planning incremental development, system extensions and subsets, debugging and testing, and gauging the effects of specific changes. The uses view also helps in managing the dependencies of a system that is being built or maintained. The goal of this task is to keep complexity under control and avoid degradation in the modifiability of the system due to the addition of undesirable dependencies.

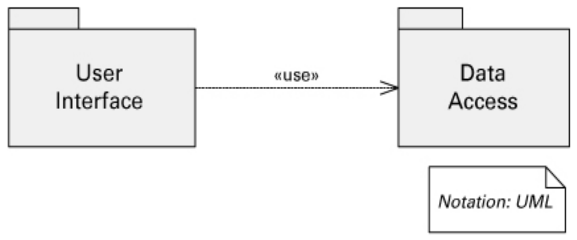
Figure shows the primary presentation of a uses view and how it can help with incremental development. To define incremental subsets, modules should be defined at the right level of granularity. In the example, admin.core may not need the entire dao package, only a submodule of it; the diagram should then show the submodules of dao.



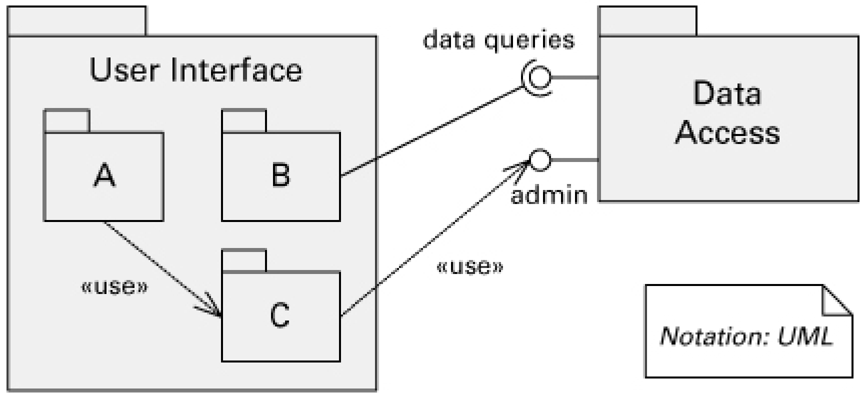
2. Uses Style

Notations Examples

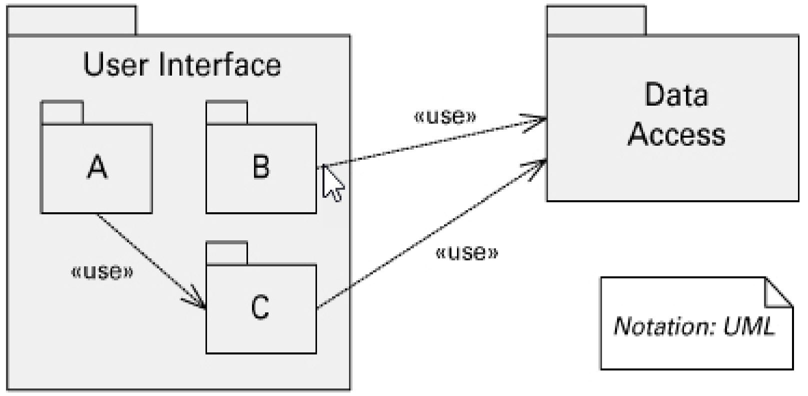
The User Interface module is an aggregate module with a *uses* dependency on the Data Access module. We use UML package notation to represent modules and the specialized form of *depends-on* arrow to indicate a *uses* relation.



Here is a variation of left figure in which the User Interface module has been decomposed into modules A, B, and C. At least one of the modules must depend on the Data Access module or the decomposition would not be consistent.



In UML we can represent the *uses* relations and also show interfaces explicitly. This version shows that the Data Access module has two interfaces, which are used by modules B and C, respectively. Both the socket lollipop connection and the <<use>> dependency connected to the lollipop indicate *uses* relations.



2. Uses Style

UML Package Diagram & DSM (Dependency Structure Matrix)

Figure 2.8 UML package diagram showing <<uses>> dependencies

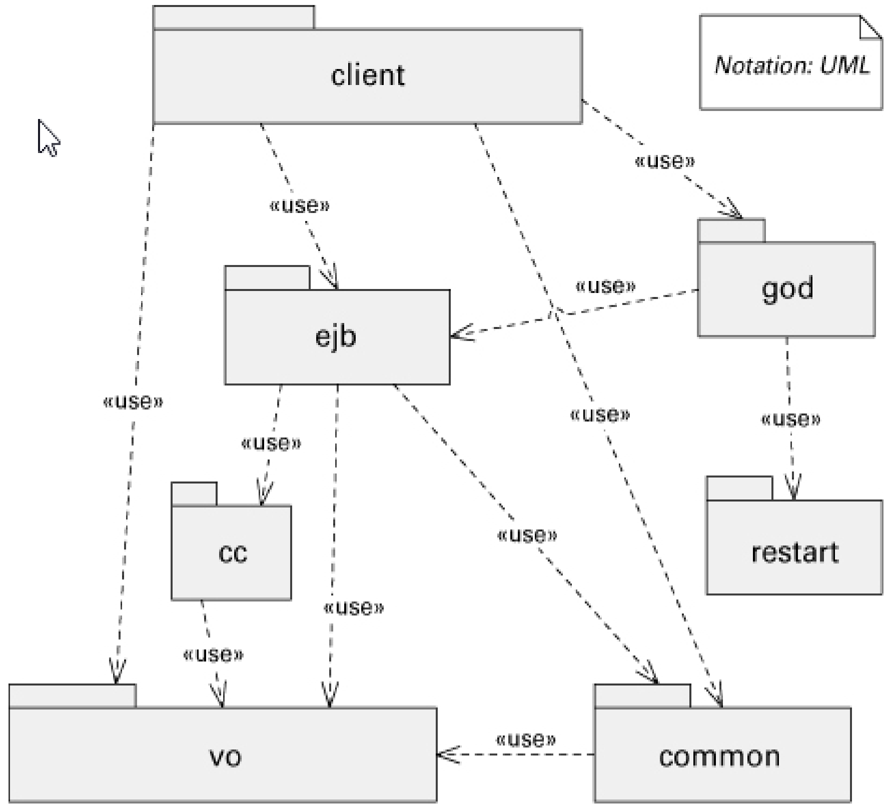
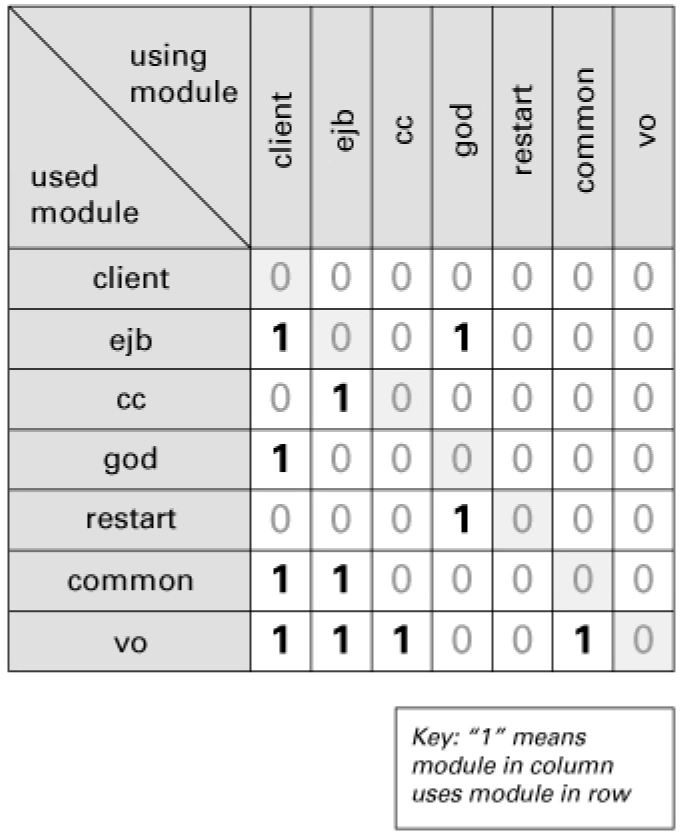


Figure 2.9 DSM for the UML diagram in Figure 2.8



using module / used module	client	ejb	cc	god	restart	common	vo
client	0	0	0	0	0	0	0
ejb	1	0	0	1	0	0	0
cc	0	1	0	0	0	0	0
god	1	0	0	0	0	0	0
restart	0	0	0	1	0	0	0
common	1	1	0	0	0	0	0
vo	1	1	1	0	0	1	0

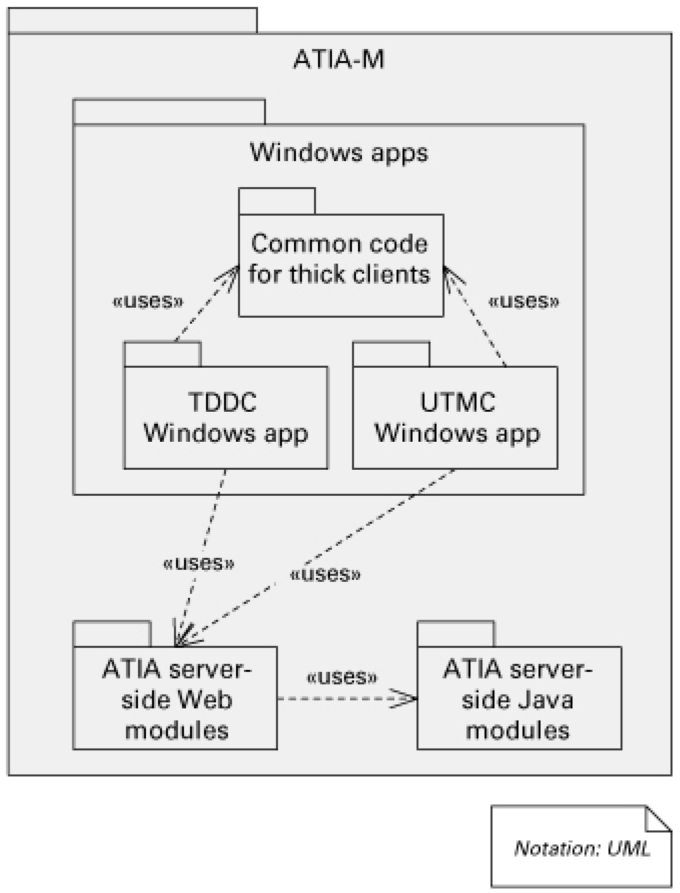
Key: "1" means module in column uses module in row

2. Uses Style

Top-level "uses" view for the ATIA-M system

The code is divided into three large modules:

- *Windowsapps* contains the code of the thick clients. The three submodules correspond to Training and Doctrine Development Tool (TDDT), Unit Training Management Configuration (UTMC), and a separate submodule with common code used by the different Windows applications. TDDT and UTMC were the two Windows applications originally planned, but others could be added.
- *ATIA server-side Web modules* contains all non-Java modules that would be deployed to server machines. The Web modules include JavaServer Pages (JSP) files, JavaScript and HTML code, and applets.
- *ATIA server-side Java modules* contains all Java source code in ATIA that would run on application servers. This module does not include JSP, JavaScript, HTML, applet, or thick-client code.



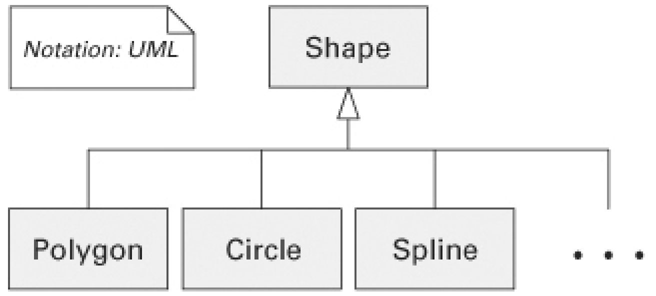
3. Generalizational Style

Overview

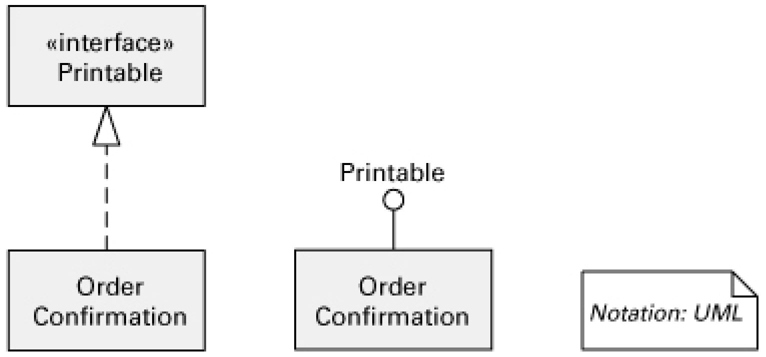
The generalization style employs the is-a relation to support extension and evolution of architectures and individual elements. Modules in this style are defined in a such a way that they capture commonalities and variations.	
Elements	Module. A module can have an “abstract” property to indicate it does not contain a complete implementation.
Relations	Generalization, which is a specialization of the is-a relation. The relation can be further specialized to indicate, for example, if it is a class inheritance, interface inheritance, or interface realization.
Constraints	• A module can have multiple parents, although multiple inheritance is often considered a dangerous design approach • Cycles in the generalization relation are not allowed, that is, child module cannot be a generalization of one or more its ancestor modules in a view
What it’s for	• Expressing inheritance in object-oriented design • Incrementally describing evolution and extension • Capturing commonalities, with variations as children • Supporting reuse

3. Generalizational Style
UML Notations

In UML, class or interface inheritance is represented by a solid line with a closed, hollow arrowhead. UML allows an ellipsis (. . .) in place of a submodule, indicating that a module can have more children than shown and that additional ones are likely. Module Shape is the parent of modules Polygon, Circle, and Spline, each of which is in turn a subclass, child, or descendant of Shape. Shape is more general; its children are specialized versions. The arrow points toward the more general entity.



Interface realization (sometimes called interface implementation) is also a kind of generalization. It can be expressed in UML in two ways: (1) a dashed line with a closed hollow arrowhead going from the module to the interface it realizes; (2) a lollipop symbol for the interface connected to the module that implements it. Thus the two notations in the figure are equivalent. However, the one on the left is more convenient when multiple modules realize the same interface.

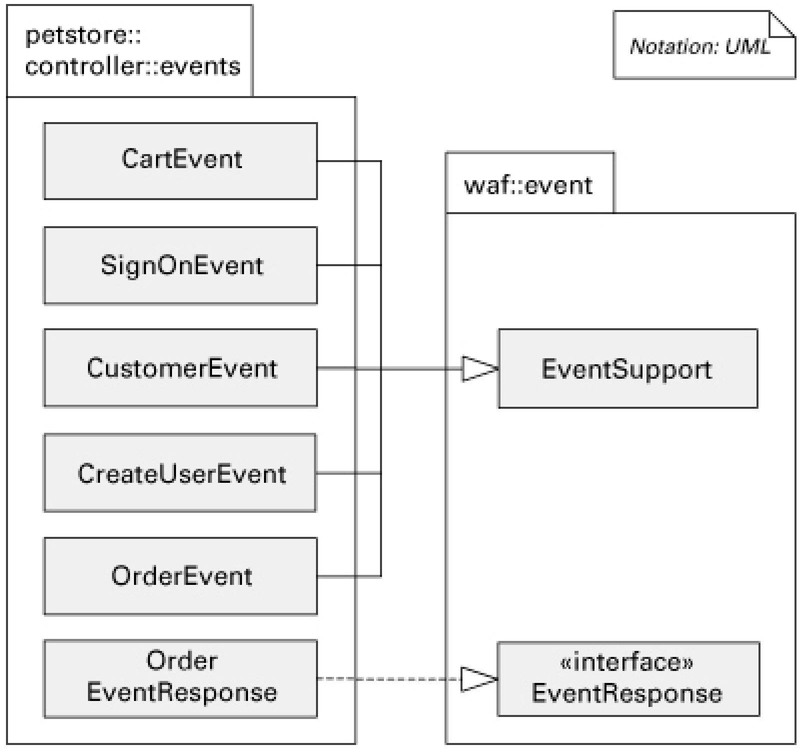


3. Generalizational Style
An Example using UML Notations

Online Pet Store:

This is a multi-tier, Web-based application that implements an online pet store. The generalization view shows several important hierarchies in the system

Part of the primary presentation of the generalization view for the PetStore application. It shows a hierarchy of classes that represent events in the system, and an interface realization. The package on the right is part of a Web application framework (waf), which offers an event-handling service. An application such as PetStore has to define the application-specific events. The events are used for the interaction of other modules in the system (not shown) following the model-view-controller pattern.



4. Layered Style

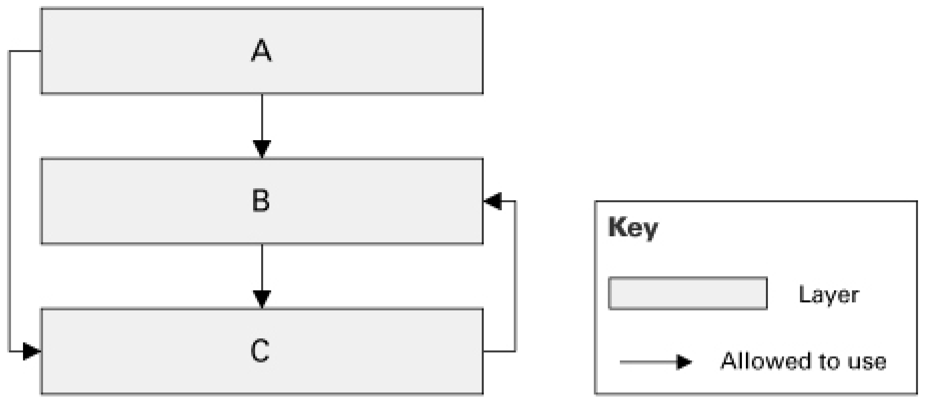
Overview

The layered style puts together layers (groupings of modules that offer a cohesive set of services) in a unidirectional allowed-to-use relation with each other	
Elements	Layer. The description of a layer should define what modules the layer contains Each layer represents a grouping of modules that offers a cohesive set of services.
Relations	Allowed to use, which is a specialization of the generic depends-on relation. The design should define the layer usage rules (for example, "A layer is allowed to use any lower layer") and any allowable exceptions.
Constraints	• Every piece of software is allocated to exactly one layer • There are at least two layers (typically three or more) • The allowed-to-use relations should not be circular, i.e., a lower layer cannot use its upper layer
What it's for	• Promoting modifiability and portability • Managing complexity and facilitating the communication of the code structure to developers • Promoting reuse • Achieving separation of concerns

4. Layered Style

Overview

There may be three layers here, but this is not a design in the layered style, which forbids upward uses.

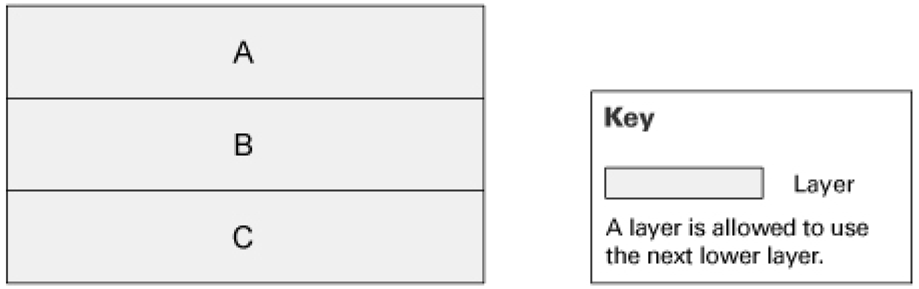


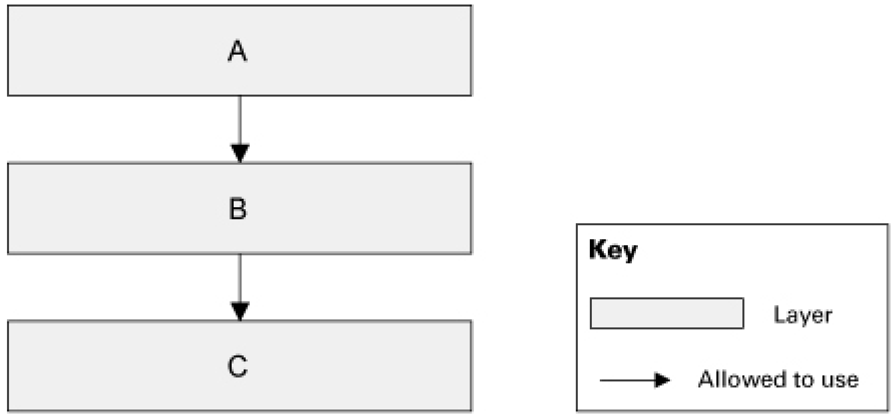
In some cases, modules in a very high layer might be required to directly use modules in a very low layer where normally only next-lower-layer uses are allowed. The layer diagram or an accompanying document will have to show these exceptions. The case of software in a higher layer using modules in a lower layer that is not just the next lower layer is called layer bridging. If many of these are present, the system is poorly structured, at least with respect to the portability and modifiability goals that layering helps to achieve. Systems with upward usages are not, strictly according to the definition, layered. However, in such cases, the layered style may represent a close approximation to reality and also conveys the ideal design that the architect was trying to achieve. Layers cannot be derived by examining source code. Layers are logical groupings that are wonderful aids in creating and communicating the architecture, but often they are not explicitly delimited in the source code. The source code may disclose what uses what, but the relation in layers is allowed to use.

4. Layered Style

Notations

- Layers are almost always drawn as a stack of boxes. The allowed-to-use relation is denoted by geometric adjacency and is read from the top down, as in Figure 2.18 (note that the key could have said, “A layer is allowed to use any lower layer”).

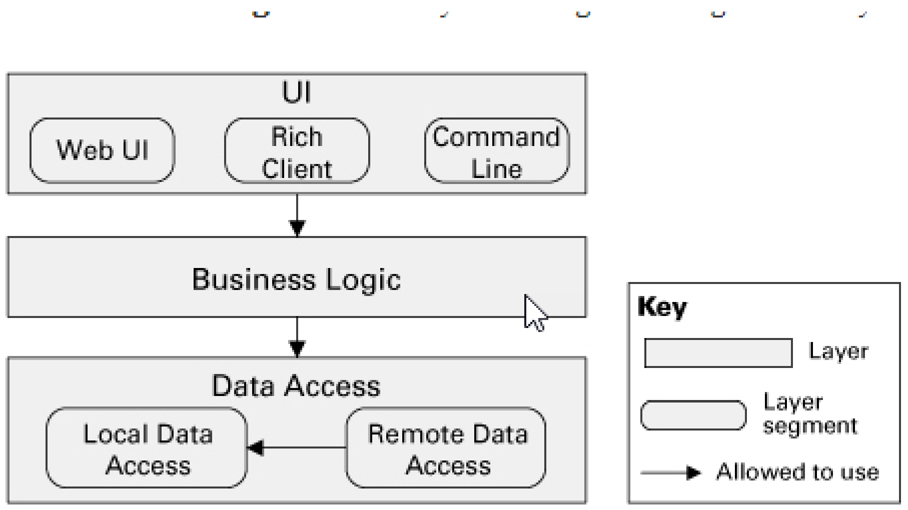




4. Layered Style

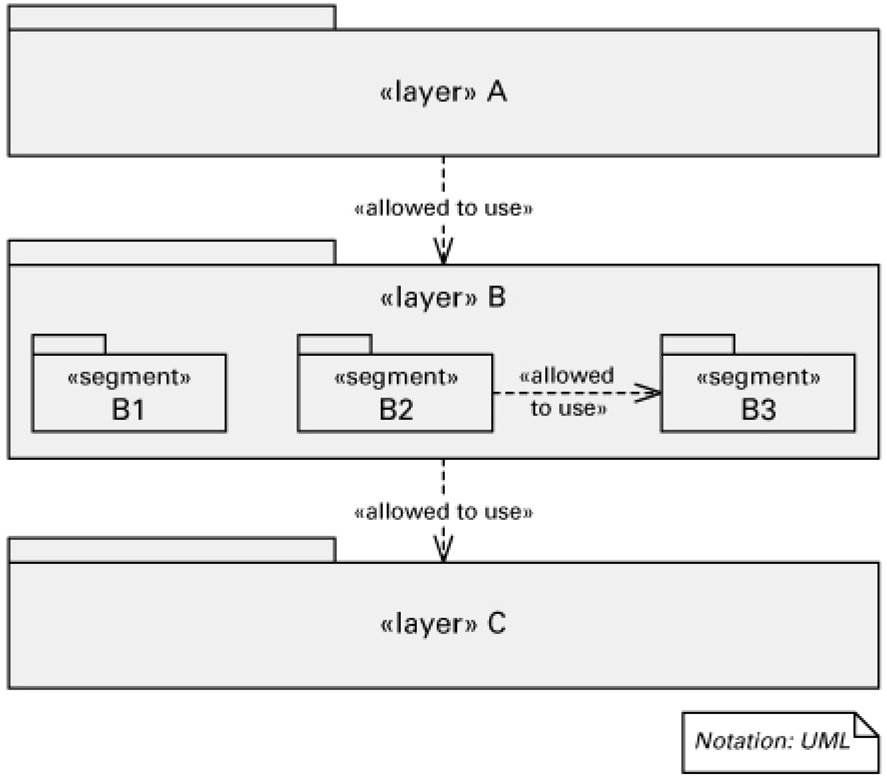
Notations – Segmented Layers

Sometimes layers are divided into segments denoting a finer-grained aggregation of the modules. Often, this occurs when a preexisting set of units, such as imported modules, share the same *allowed-to-use* relation. When this happens, the creator of the diagram must specify what usage rules are in effect among the segments. Many usage rules are possible, but they must be made explicit. Here, the top and the bottom layers are segmented. Segments of the top layer are not allowed to use each other, but segments of the bottom layer are. If you draw the same diagram without the arrows, it will be harder to differentiate the usage rules within segmented layers. Layered diagrams are often a source of ambiguity because the diagram does not make explicit the *allowed-to-use* relations.



4. Layered Style
UML Notations

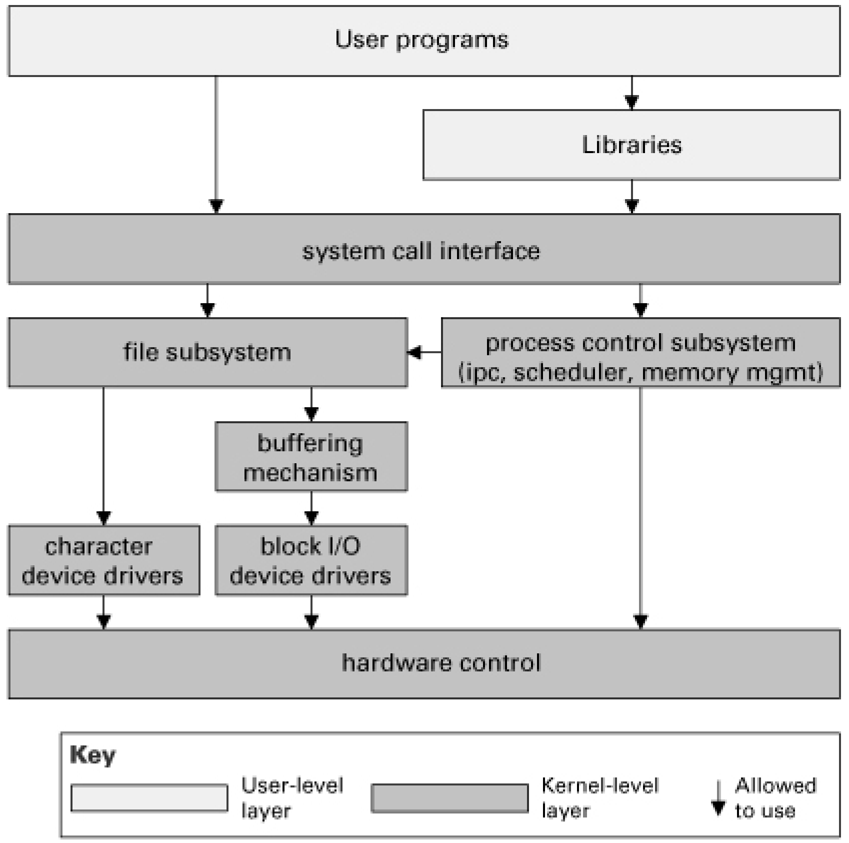
- UML has no built-in primitive corresponding to a layer. However, layers can be represented in UML as stereotyped packages, as shown in Figure 2.24. A package is a general-purpose mechanism for organizing elements into groups, and it suits the notion of layers. The *allowed-to-use* relation can be a stereotyped dependency between layer packages.
- Documenting segmented layers in UML. If segments in a layer are allowed to use each other, then <<allowed to use>> dependencies must be added among them as well.



4. Layered Style

Examples using layered style

A classic layered design is the UNIX System V operating system, as shown in Figure 2.26. The lower layers form the system kernel; top layers are user programs or libraries that access the kernel through system calls. The system call interface layer isolates the kernel implementation details and provides a virtual machine to user programs. The file subsystem is responsible for managing files (devices are treated as files), administering free space, controlling access, and reading/writing data. The process control subsystem is responsible for process scheduling, interprocess communication, process synchronization, and memory management. The hardware control layer is responsible for handling interrupts and communicating with the machine.

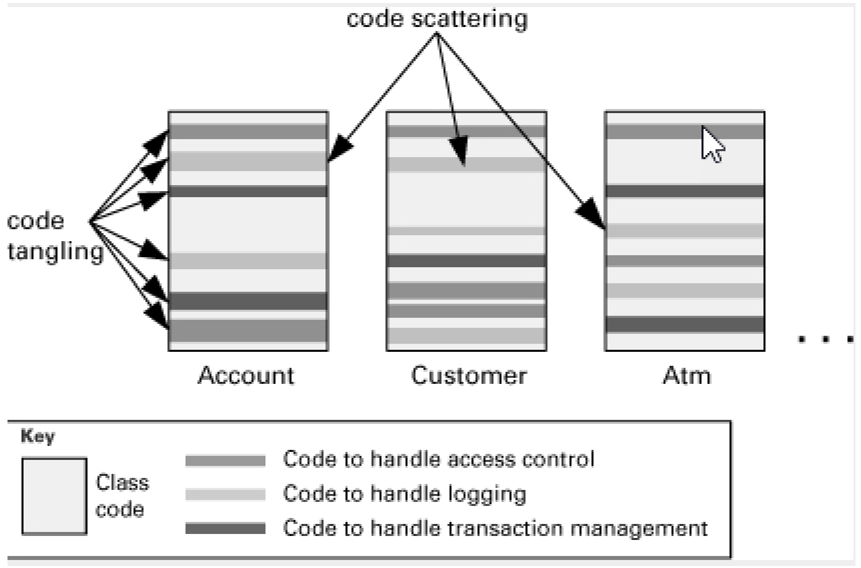


5. Aspects Style

Design a system based on the principles of Aspect-Oriented Programming (AOP)

Aspect-oriented programming is an evolutionary implementation paradigm that complements object-oriented programming and facilitates the implementation of crosscutting concerns. AspectJ is probably the most widely known AOP package. Other implementations include Spring AOP, JBoss AOP, AspectC++, and Aspect#.

The traditional object-oriented implementation of a bank automation system would have several classes where the business logic is tangled with code that handles crosscutting concerns, such as access control, logging, and transaction management. In addition, the code that handles a particular crosscutting concern is repeated and scattered across several classes.

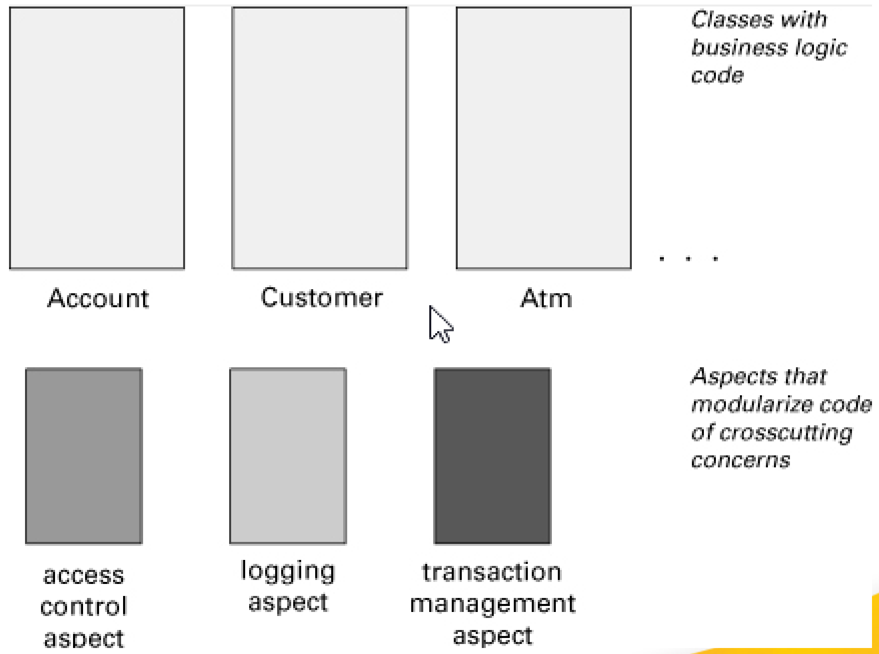


5. Aspects Style

Design a system based on the principles of Aspect-Oriented Programming (AOP)

AOP brings an ingenious solution to improve modularity and resolve the code tangling and code scattering problems. The crosscutting code is factored out from the classes and placed in a special module called aspect, as represented in previous figure. An aspect has two important parts: advices and pointcut specifications. Advices contain the code for the crosscutting concerns. Such code will be injected at certain points (called join points) of the classes through a process called weaving, carried on by the AOP compiler. The pointcut specifications contain declarations that map to specific sets of join points in the target classes. In the aspect code, advices are associated to pointcut specifications to let the AOP compiler know where exactly each advice code will be injected in the target classes.

In the aspect-oriented implementation of the same bank automation system, the classes don't contain code for logging, access control, transaction management, and other crosscutting concerns. The code to handle these concerns is now inside aspect modules. Classes such as Account, Customer and Atm contain the business logic only. The AOP compiler will use the weaving process to insert the code inside aspects at the locations in the classes where it's needed.


Classes with business logic code
Account
Customer
Atm
. . .
Aspects that modularize code of crosscutting concerns
access control aspect
logging aspect
transaction management aspect

5. Aspects Style
AOP and Aspects Styles

- The aspects style is a module style used to isolate in the architecture the modules responsible for crosscutting concerns.
- When we implement software modules in general, the business logic code ends up intermixed with code that deals with crosscutting concerns. For example, if you’re writing a bank automation system, there may be modules such as Account, Customer, and ATM. The Account module ideally would contain only the code to deal with the bank account business logic (open/close account, deposit, withdraw, transfer, and so on). But in practice we have to add code to handle crosscutting concerns, such as access control, transaction management, and logging.
- The aspects style prescribes that the modules responsible for the crosscutting functionality should be placed in one or more aspect views. These modules are called aspects, based on the terminology introduced by aspect-oriented programming (AOP). The aspect views should contain information to bind each aspect module to the other modules that require the crosscutting functionality.

5. Aspects Style

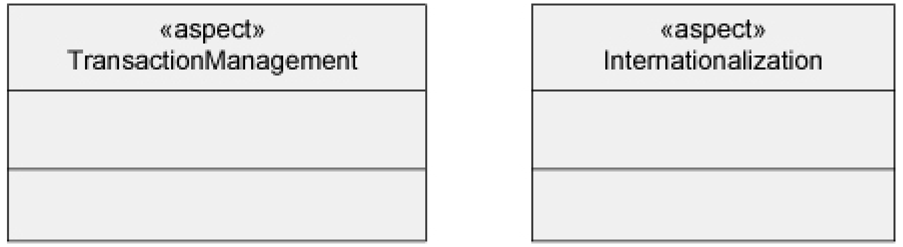
Overview

The aspects style shows aspect modules that implement crosscutting concerns and how they are bound to other modules of the system.	
Elements	Aspect, which is a specialized module that contains the implementation of a crosscutting concern
Relations	Crosscuts, which is a specialized module that contains the implementation of a crosscutting concern
Constraints	• An aspect can crosscut one or more regular modules as well as aspect modules • An aspect that crosscuts itself may cause infinite recursion, depending on the implementation
What it's for	• Modeling crosscutting concerns in object-oriented designs • It promotes modifiability by increasing modularity and by avoiding the tangling of crosscutting functionality and business domain functionality

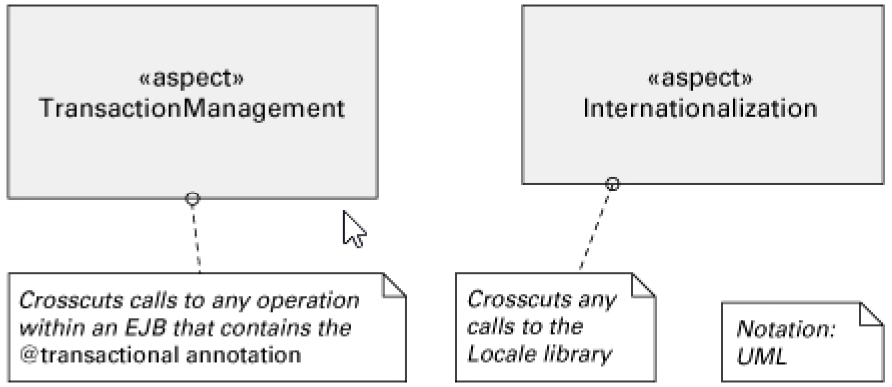
5. Aspects Style

UML Notations

Aspect modules are often represented in UML as classes with stereotype <<aspect>>

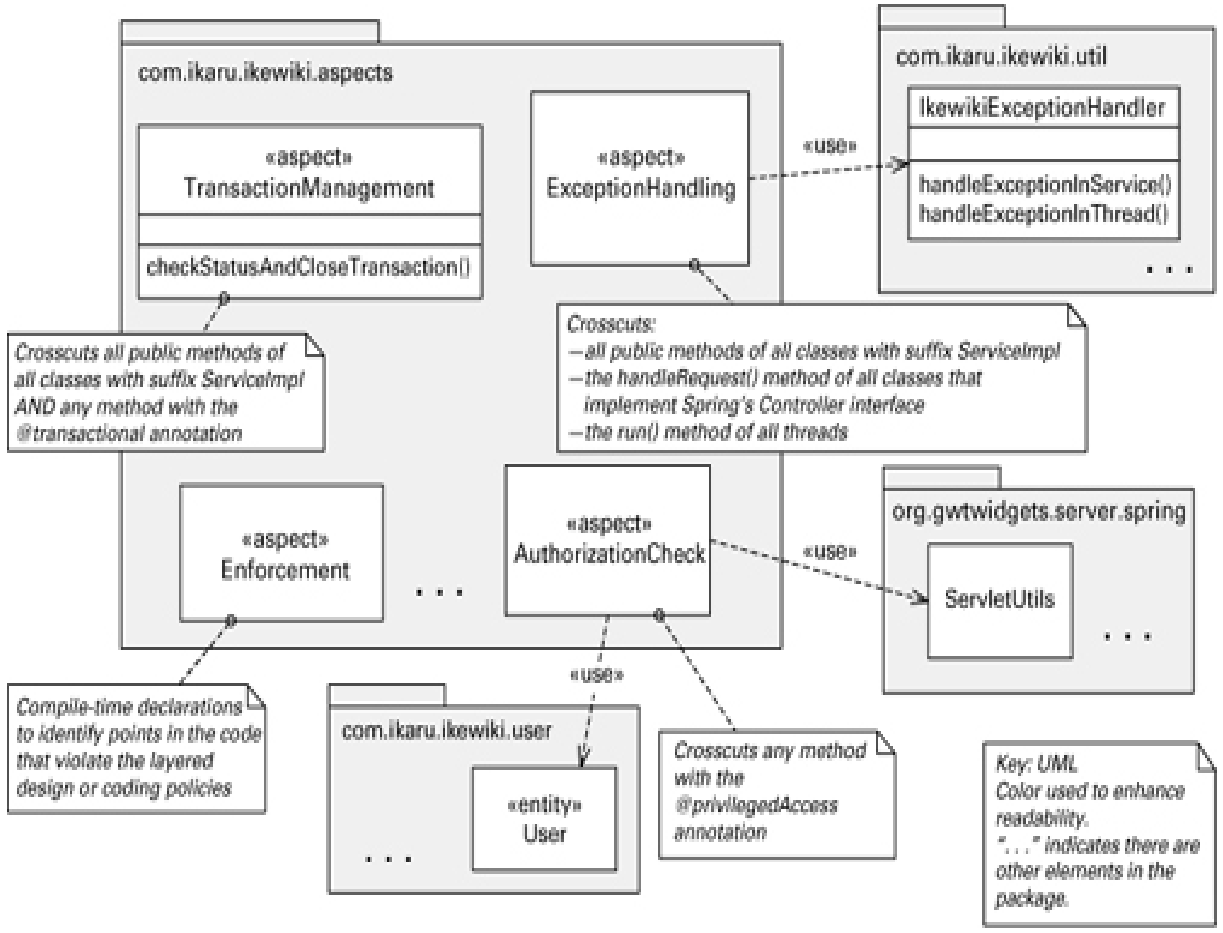


Instead of trying to draw a line from each aspect to every module it crosscuts, we simply add a comment box that characterizes what modules will be crosscut.


«aspect»
TransactionManagement
«aspect»
Internationalization
Crosscuts calls to any operation within an EJB that contains the @transactional annotation
Crosscuts any calls to the Locale library
Notation: UML

5. Aspects Style
UML Notations for IkeWiki Application

Primary presentation for the aspects view of the IkeWiki application. This Java EE application implemented with the Spring framework and Google Web Toolkit uses aspects for some crosscutting concerns. The Transaction Management aspect makes sure all requests received by the server will close the transaction and release database resources properly, performing a rollback when an exception occurs. The ExceptionHandling aspect has code to log the error to the database, send e-mail notification if applicable, and wrap the exception with a proper user message to be displayed by the client application. This aspect is woven into server-side classes that are either threads or entry points to process HTTP requests. The AuthorizationCheck aspect is used to check if the current user has permission to execute a specific method. The Enforcement aspect is different from the others. It doesn't exactly implement a crosscutting concern, but rather it scans the source code at compile time looking for violations of the layered design, as well as violations of several coding policies.


com.ikaru.ikewiki.aspects
«aspect» TransactionManagement
checkStatusAndCloseTransaction()
«aspect» ExceptionHandling
«use»
com.ikaru.ikewiki.util
IkewikiExceptionHandler
handleExceptionInService()
handleExceptionInThread()
. . .
Crosscuts all public methods of all classes with suffix ServiceImpl AND any method with the @transactional annotation
Crosscuts:
—all public methods of all classes with suffix ServiceImpl
—the handleRequest() method of all classes that implement Spring's Controller interface
—the run() method of all threads
«aspect» Enforcement
. . .
«aspect» AuthorizationCheck
«use»
org.gwtwidgets.server.spring
ServletUtils
. . .
«use»
Compile-time declarations to identify points in the code that violate the layered design or coding policies
com.ikaru.ikewiki.user
«entity» User
. . .
Crosscuts any method with the @privilegedAccess annotation
Key: UML
Color used to enhance readability.
". . ." indicates there are other elements in the package.

6. Data Model

Overview

The data model describes the structure of the data entities and their relationships.
For example, in a banking system, entities typically include Account, Customer and Loan. Account has several attributes, such as account number, type (savings or checking), status, and current balance. A relationship may dictate that one customer can have one or more accounts, and one account is associated to one or two customers. The data model is often represented graphically in entity-relationship diagrams (ERDs) or UML class diagrams.

Elements	Data entity, which is an object that holds the information that needs to be stored or somehow represented in a system. Properties include name, data attributes, primary key, and rules to grant users permission to access the entity
Relations	• 1-to-1, 1-to-n, n-to-n relationships - logical associations between data entities • Generalization/specialization, which indicate an is-a relation between entities. Car Insurance and House Insurance are specializations of entity Insurance. • Aggregation, which turns a relationship into an aggregated entity. For example, a relationship between a patient, a physician, and a date can be abstracted as an aggregate entity called Appointment.
Constraints	Functional dependencies should be avoided
What it's for	• Describing the structure of the data used in the system • Performing impact analysis of changes of the data model; extensibility analysis • Enforcing data quality by avoiding redundancy and inconsistency • Guiding implementation of modules that access the data

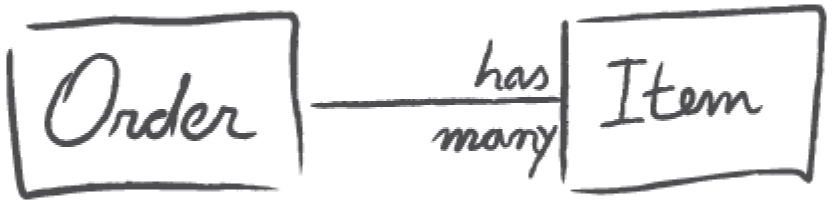
6. Data Model
Types of Data Model

Type	Description
Conceptual	The conceptual data model abstracts implementation details and focuses on the entities and their relationships as perceived in the problem domain.
Logical	The logical data model is an evolution of the conceptual data model toward a data management technology (such as relational databases). It is typically the subject of normalization
Physical	The physical data model is concerned with the implementation of the data entities. It incorporates optimizations that may include partitioning or merging entities, duplicating data, and creating identification keys and indexes. For example, in Figure 2.40 a column named total-Price was likely added to the entity Order as a performance optimization, since the total price could also be obtained by reading all order items and adding up their prices. In an early stage, the architecture documentation may contain the data model with the key entities and important relationships. Later on, this initial model is superseded by the detailed model approved by the data administrators.

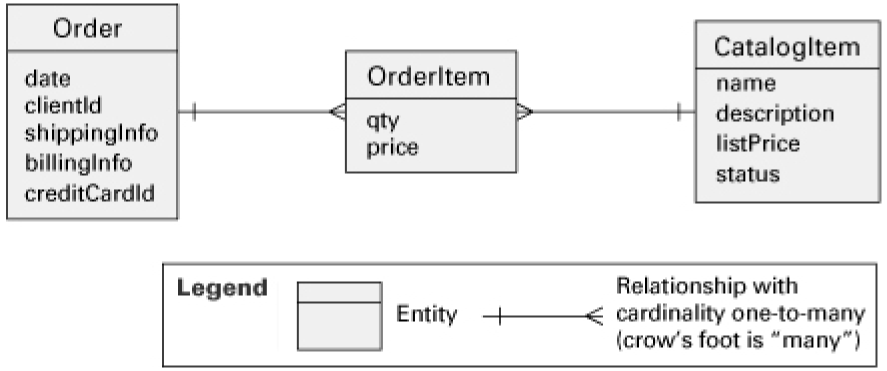
6. Data Model

Types of Data Model

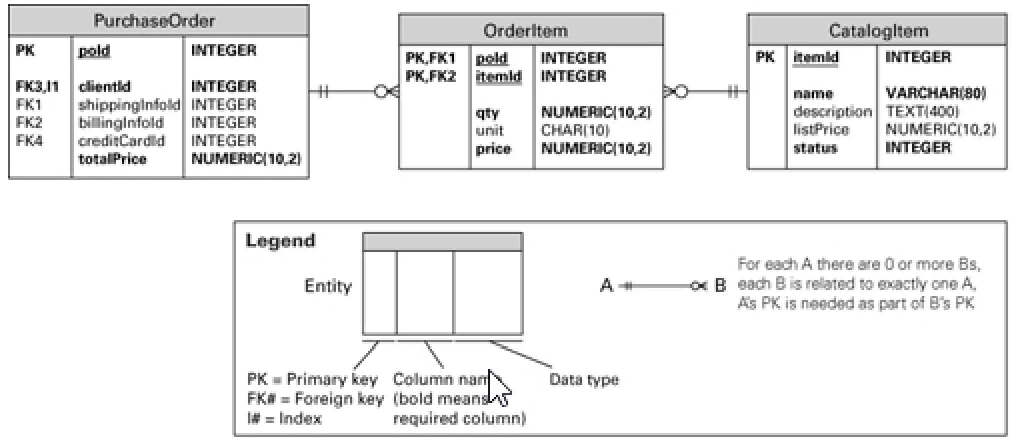
First draft of a conceptual data model. Fragments of an online order-processing system at different stages.



Logical data model that has evolved from the conceptual data model above



Physical data model that was created by adding implementation details and optimizations to the logical data model

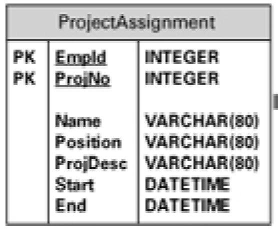


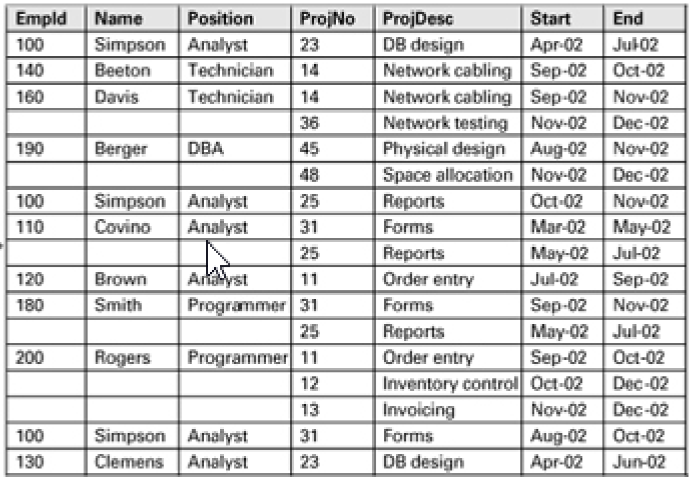
6. Data Model

Normalization is necessary to avoid repetition in data modeling

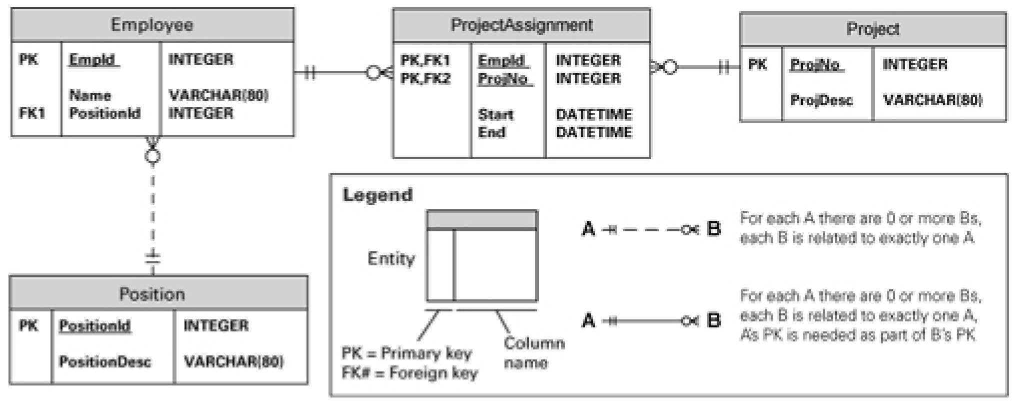
Entity ProjectAssignment before normalization, along with sample data (adapted from Ponniah 2007). The attributes that uniquely identify a project assignment (that is, the primary key) are EmpId and ProjNo.

Data model for ProjectAssignment after normalization. One of the rules of normalization is that non-key attributes should have functional dependencies to the whole primary key only. Attribute ProjDesc has a functional dependency to ProjNo, which is not the whole primary key. After this and other violations of the normalization rules were fixed, this is the resulting data model diagram.





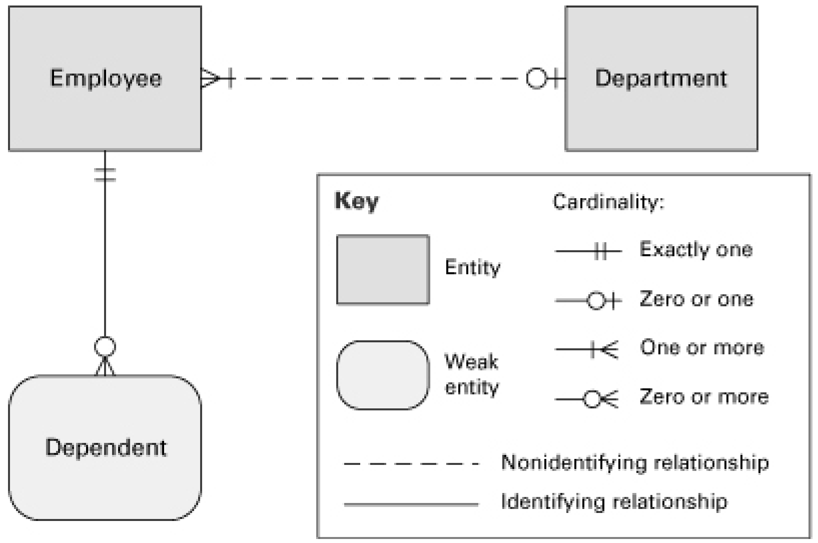
EmpId	Name	Position	ProjNo	ProjDesc	Start	End
100	Simpson	Analyst	23	DB design	Apr-02	Jul-02
140	Beeton	Technician	14	Network cabling	Sep-02	Oct-02
160	Davis	Technician	14	Network cabling	Sep-02	Nov-02
			36	Network testing	Nov-02	Dec-02
190	Berger	DBA	45	Physical design	Aug-02	Nov-02
			48	Space allocation	Nov-02	Dec-02
100	Simpson	Analyst	25	Reports	Oct-02	Nov-02
110	Covino	Analyst	31	Forms	Mar-02	May-02
			25	Reports	May-02	Jul-02
120	Brown	Analyst	11	Order entry	Jul-02	Sep-02
180	Smith	Programmer	31	Forms	Sep-02	Nov-02
			25	Reports	May-02	Jul-02
200	Rogers	Programmer	11	Order entry	Sep-02	Oct-02
			12	Inventory control	Oct-02	Dec-02
			13	Invoicing	Nov-02	Dec-02
100	Simpson	Analyst	31	Forms	Aug-02	Oct-02
130	Clemens	Analyst	23	DB design	Apr-02	Jun-02



6. Data Model
ERD and UML Notations

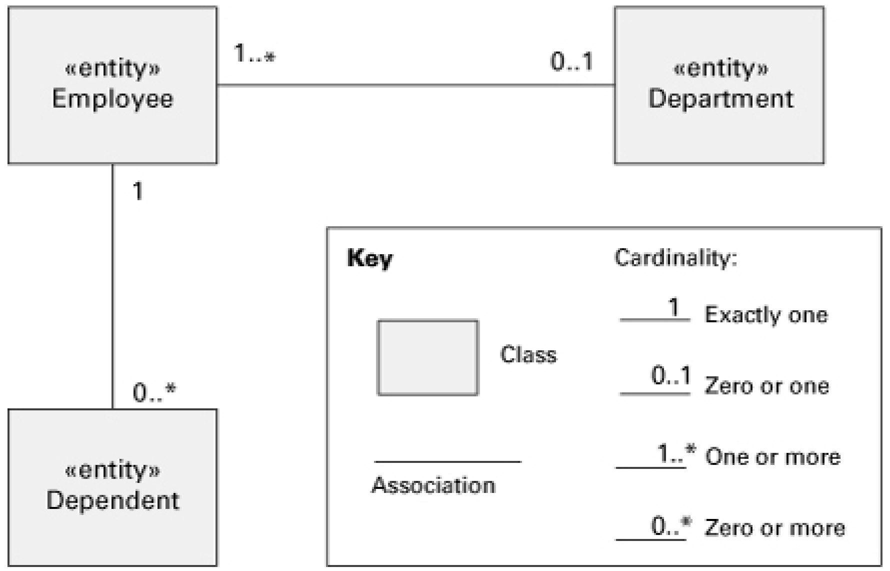
One of the most popular ERD notations for relationships uses lines with special symbols at each end to indicate cardinality. These symbols include a dash (indicating one), a ring (indicating zero), and a crow's foot (indicating many).

ERD: Data model (simplified) of a human resource system using crow's foot ERD notation



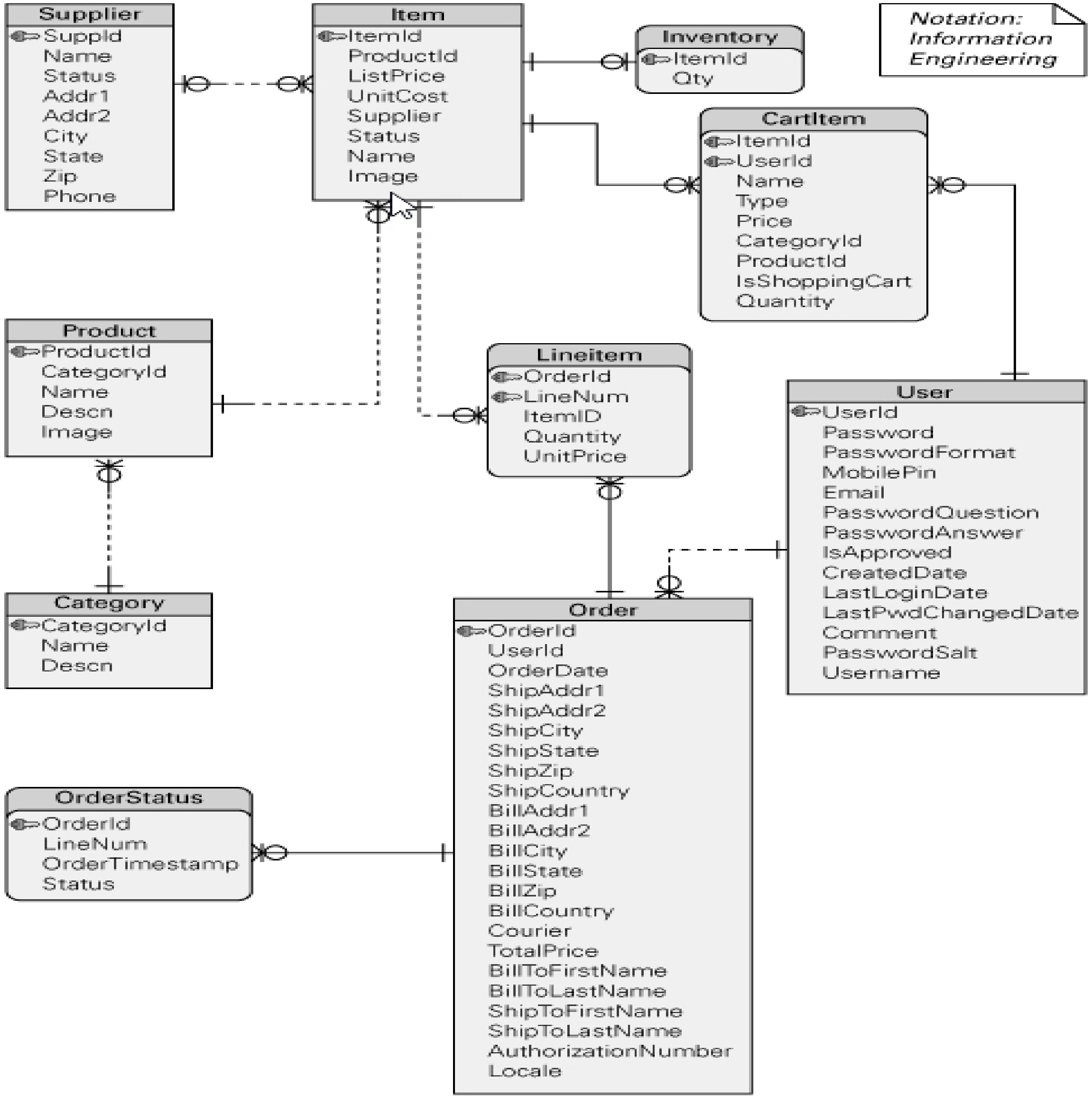
The data model can be represented as a UML class diagram, where the classes correspond to data entities. The attribute compartment lists the entity attributes, and the operation compartment is empty. UML associations represent the relationships between entities and the multiplicity intervals shown at both ends of the association lines (for example, "1..*") indicate the cardinality of the relationship.

UML: Data model (simplified) of a human resource system shown as a UML class diagram



6. Data Model
ERD Notations

Data model for the Pet Shop application using Information Engineering crow's foot ERD notation



Acknowledgment

Lecture contents and pictures are taken from B4 Chapter 2 (Module Styles)