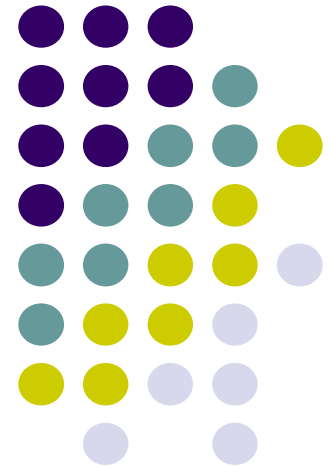
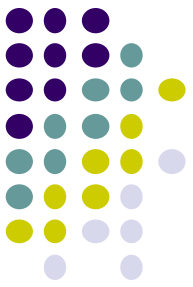


Designing the Software Architecture

Layers
Subsystems
Variants
Mapping the Data Model
Examples

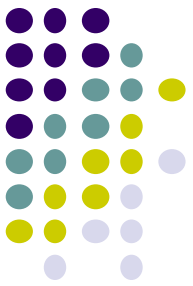




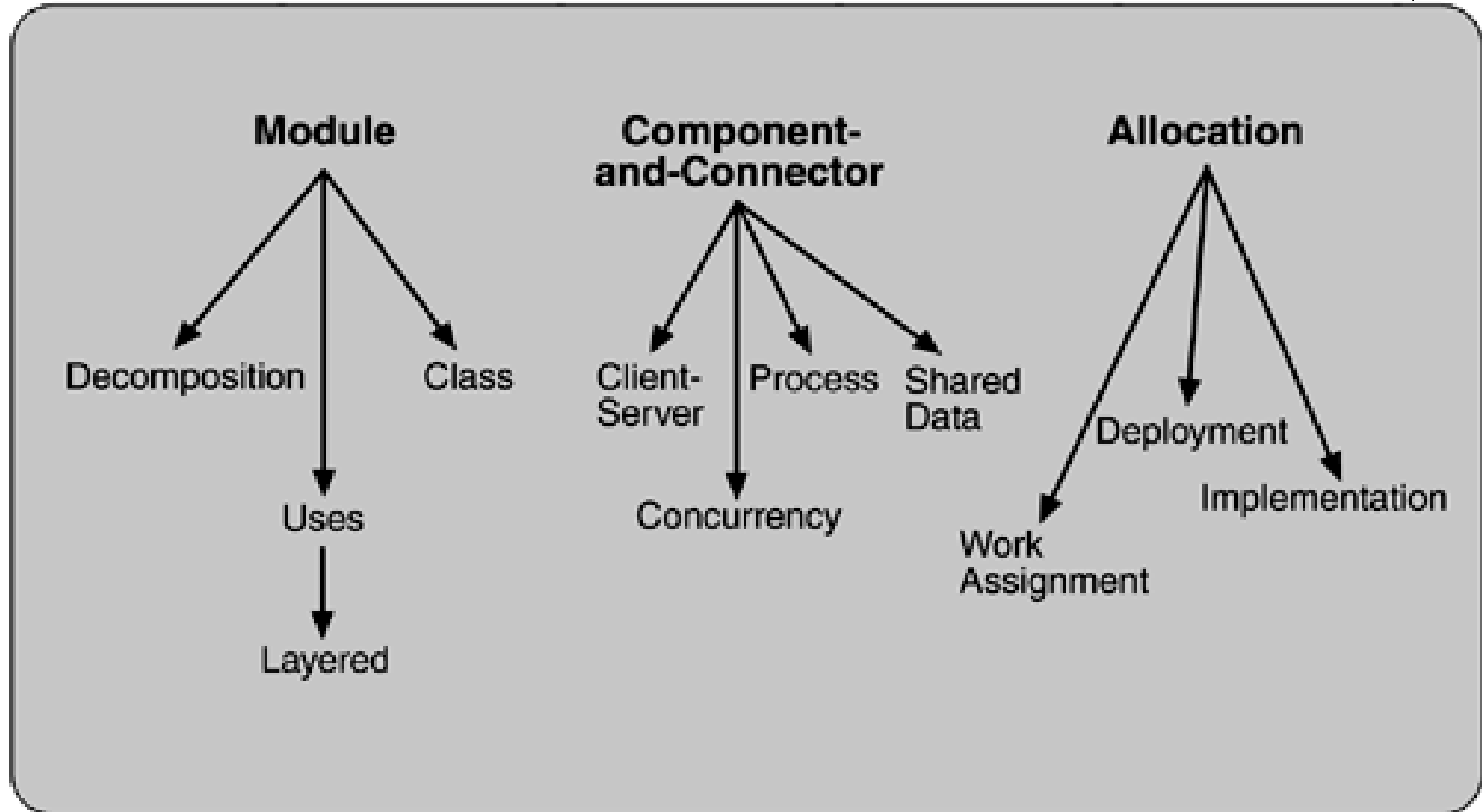
Software Architecture

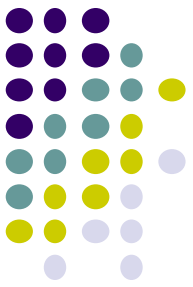
Software architecture encompasses:

- the significant decisions about the organization of a software system,
- the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements,
- the composition of the structural and behavioral elements into progressively larger subsystems,
- the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition.



Types of Architectural Structures

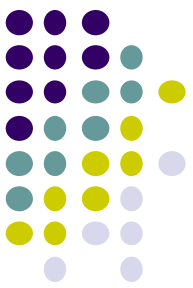




Systems and Subsystems

System - an instance, an executable configuration of a software application or of a software application family.

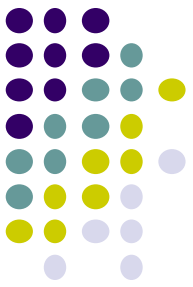
Subsystem - a model element which has the semantics of a *package*, such that it can contain other model elements, and class/classes, thus having its own behavior.



System Coupling

- **Decomposable system** - One or more of the components of a system have no interactions or other interrelationships with any of the other components at the same level of abstraction within the system
- **A nearly decomposable system** - Every component of the system has a direct *or indirect* interaction or other interrelationship with every other component at the same level of abstraction within the same system
- **Design Goal** - The interaction or other interrelationship between any two components at the same level of abstraction within the system be as weak as possible

Measure of the modular interdependence



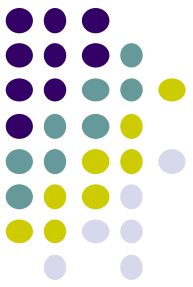
- Unnecessary object coupling:
 - needlessly decreases the **reusability** of the coupled objects
 - increases the chances of **system corruption** when changes are made to one or more of the coupled objects

Types of Modular Coupling

In order of desirability

Cure:
Decompose the
operation into
multiple primitive
operations

- Data Coupling (weakest most desirable) - output from one module is the input to another
- Control Coupling - passing control flags between modules so that one module controls the sequencing of the processing steps in another module.
- Global Data Coupling - two or more modules share the same global data structures
- Internal Data Coupling (strongest least desirable) - One module directly modifies local data of another module (like C++ Friends)
- Content Coupling (unrated)- some or all of the contents of one module are included in the contents of another (like C/C++ header files)



System Cohesion

- Cohesion – degree of functional relatedness between (sub) systems
- If there are many objects related to each other and performing similar tasks – high cohesion
- If there are many objects not related to each other – low cohesion

Cohesion

Source: 1) Object Coupling and Object Cohesion, chapter 7 of Essays on Object-Oriented Software Engineering, Vol 1, Berard, Prentice-Hall, 1993;

2) SDSU & Roger Whitney;

- "Cohesion is the degree to which the tasks performed by a single module are functionally related." *IEEE, 1983*
- "A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of functional relatedness. This means that each element in the program unit should be essential for that unit to achieve its purpose." *Sommerville, 1989*
- Types of Module Cohesion
 - Coincidental (worst)
 - Logical
 - Temporal
 - Procedural
 - Communication
 - Sequential
 - Functional (best)



Low cohesion example – Bruegge & Dutoit'2004

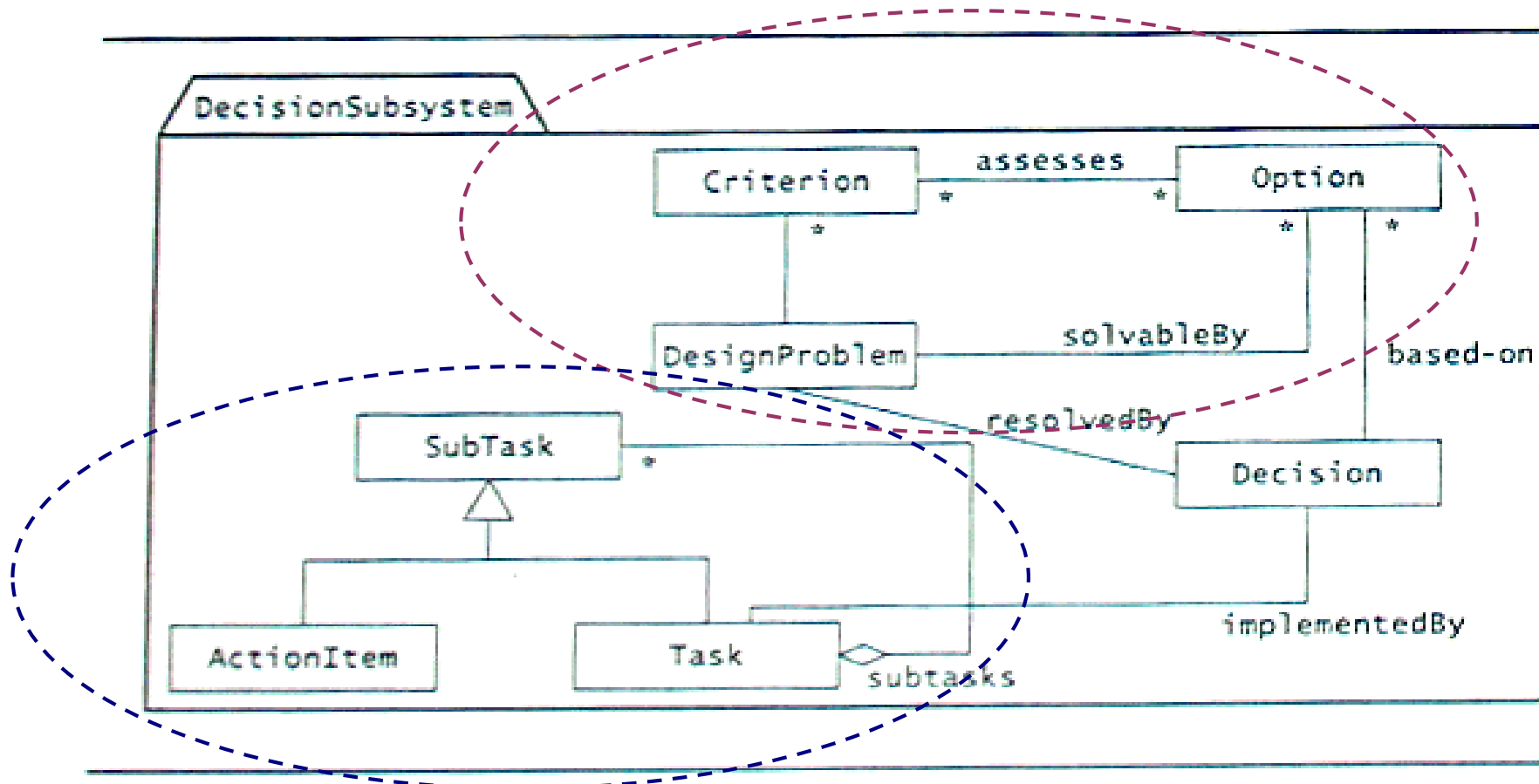
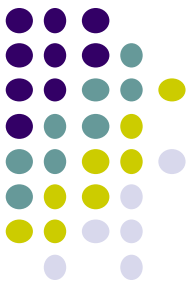
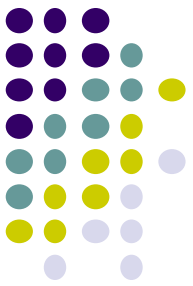


Figure 6-6 Decision tracking system (UML class diagram). The DecisionSubsystem has a low cohesion: The classes Criterion, Option, and DesignProblem have no relationships with Subtask, ActionItem, and Task.



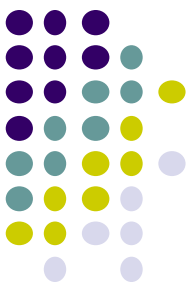
Architectural Layers

Layering represents an ordered grouping of functionality:

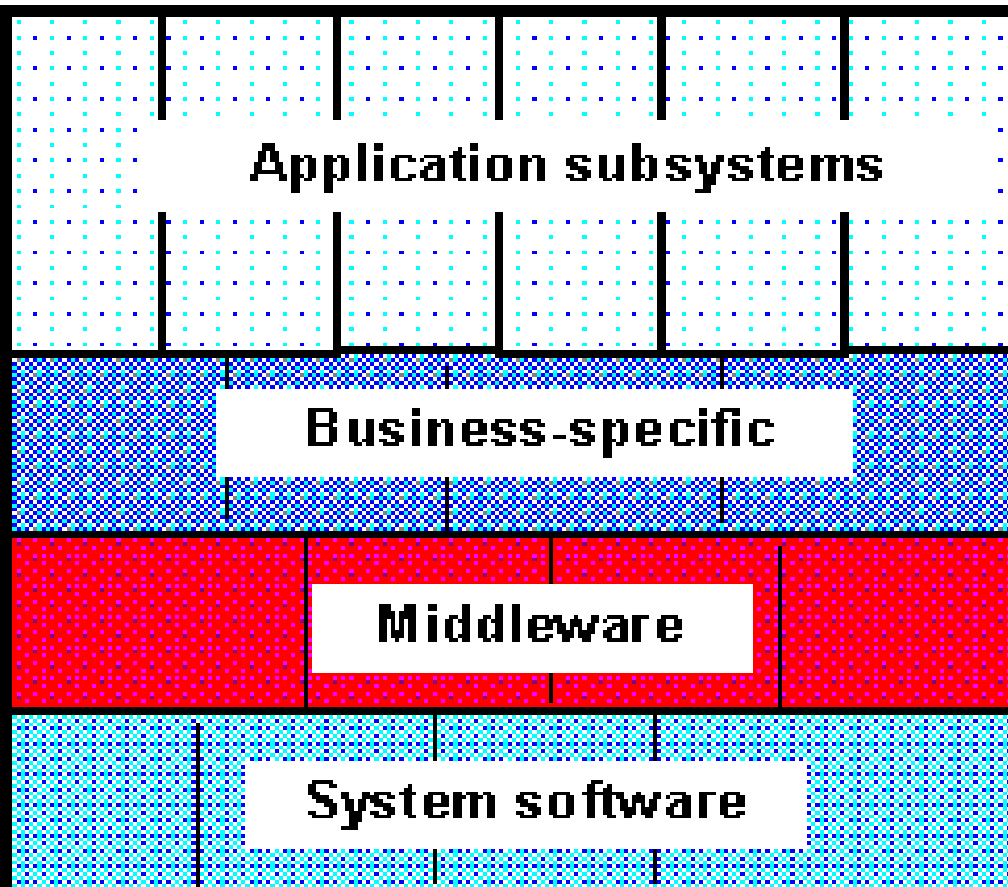
- with the ***application-specific*** located in the upper layers,
- functionality that spans ***application domains*** in the middle layers, and
- functionality specific to the ***deployment environment*** at the lower layers.

A layered structure:

- starts at the most general level of functionality, and
- grows towards more specific levels of functionality.



Architectural Layers Structure



Distinct application subsystem that make up an application - contains the value adding software developed by the organization.

Business specific - contains a number of reusable subsystems specific to the type of business.

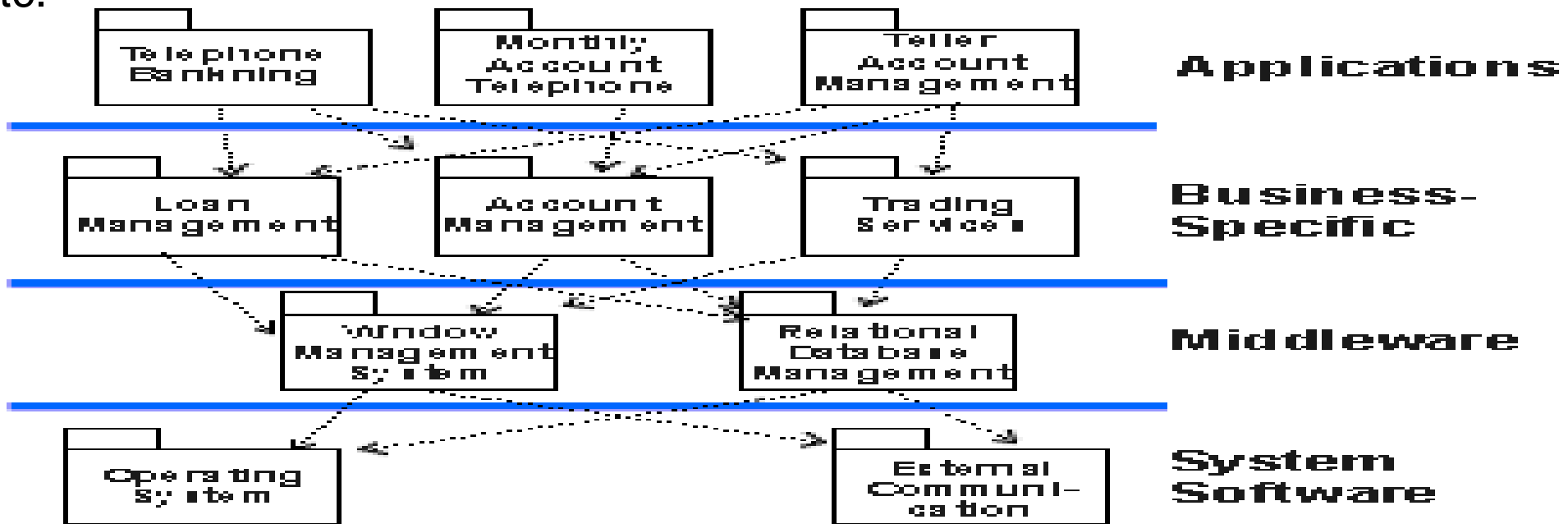
Middleware - offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

Subsystems Can Be Organized in Layers



- The top layer, **application layer**, contains the *application specific services*.
- The **business-specific layer**, contains *business specific components*.
- The **middleware layer** contains components such as GUI-builders, interfaces to DB, platform-independent operating system services, and OLE-components.
- The **system software layer**, contains components such as OS, HW interfaces, etc.

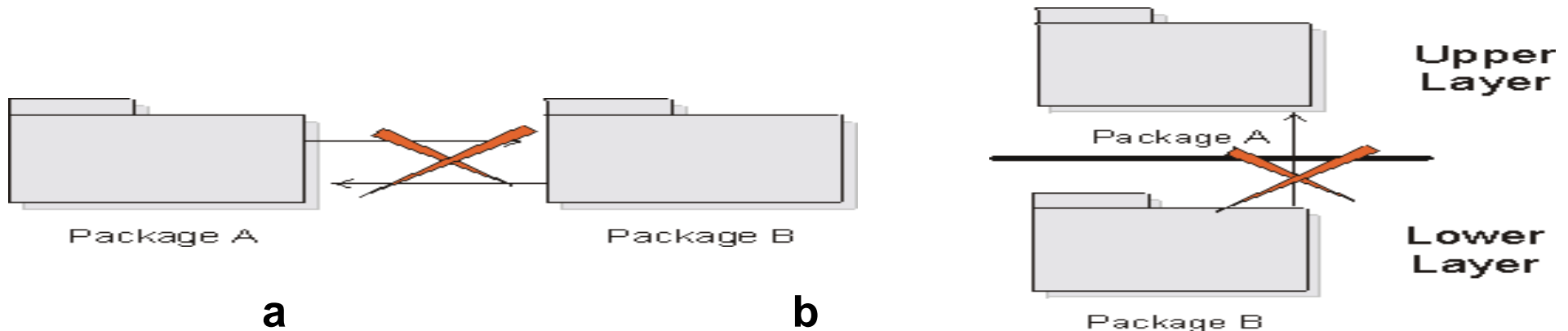


An example of a layered implementation model for a banking system. The arrows shows *top-down import dependencies* between subsystems.

Packages



A **package** is a collection of use cases & their diagrams (*use case packages*), of classes/relationships/ diagrams (*design packages*), of components (*implementation packages*) and of other packages; it is used to structure the design model by dividing it into smaller parts. Packages are used primarily for model organization and typically serve as a unit of configuration management.



- **Packages should not be cross-coupled (i.e. co-dependent)**
- **Packages should only be dependent upon packages in the same layer or next lower layer**

Hierarchical Decomposition



A **hierarchical decomposition** of a system yields an ordered set of layers. A **layer** is a grouping of subsystems providing related services, possibly realized using services from another layer. Layers are ordered in that each layer can depend only on lower level layers and has no knowledge of the layers above it. The layer that does not depend on any other layer is called the **bottom layer**, and the layer that is not used by any other is called the **top layer** (Figure 6-8). In a **closed architecture**, each layer can access only the layer immediately below it. In an **open architecture**,¹ a layer can also access layers at deeper levels.

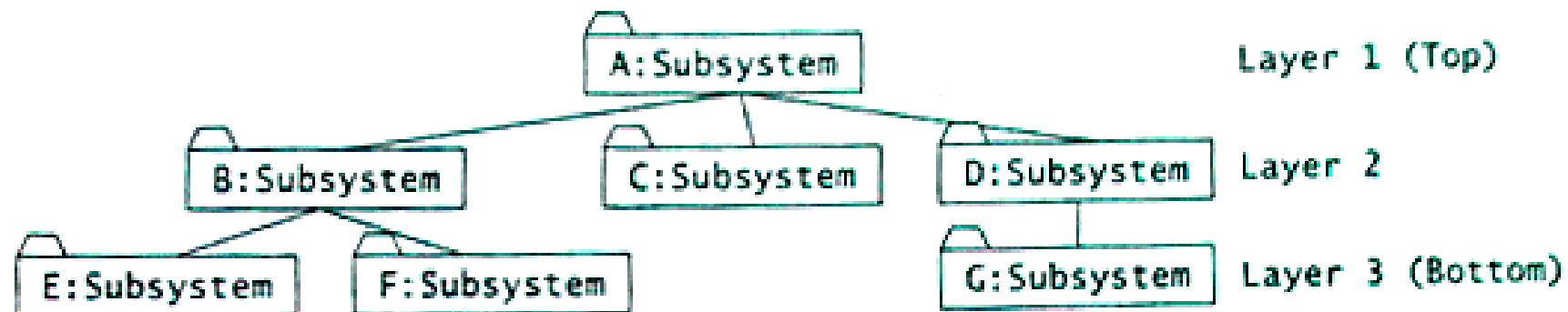


Figure 6-8 Subsystem decomposition of a system into three layers (UML object diagram). A subset from a layered decomposition that includes at least one subsystem from each layer is called a vertical slice. For example, the subsystems A, B, and E constitute a vertical slice, whereas the subsystems D and G do not.

Example of closed system [Bruegge & Dutoit'2004]

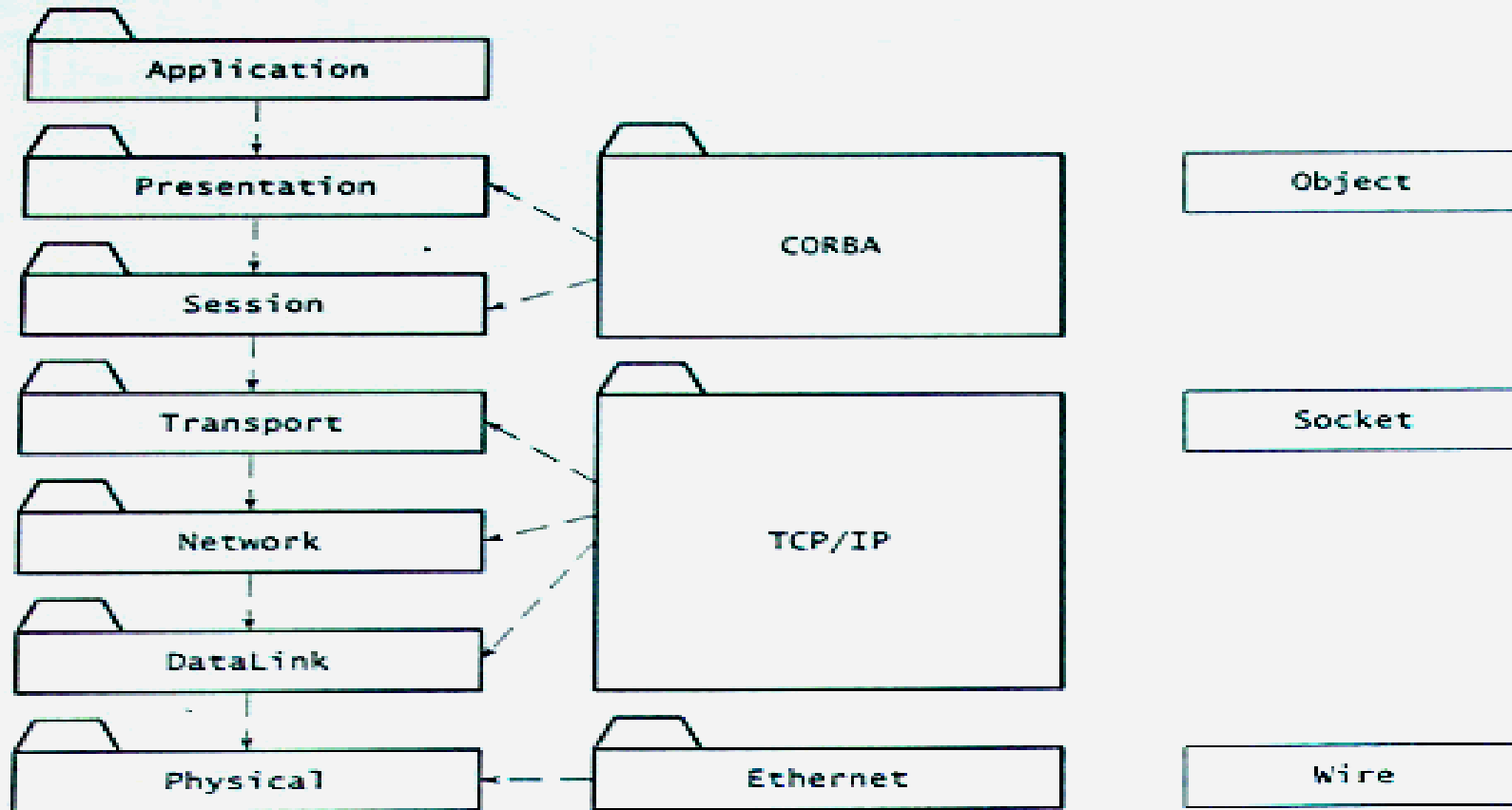
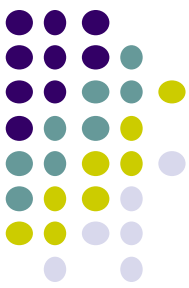


Figure 6-10 An example of closed architecture (UML class diagram). CORBA enables the access of objects implemented in different languages on different hosts. CORBA effectively implements the Presentation and Session layers of the OSI stack.

Example of open system [Bruegge & Dutoit'2004]

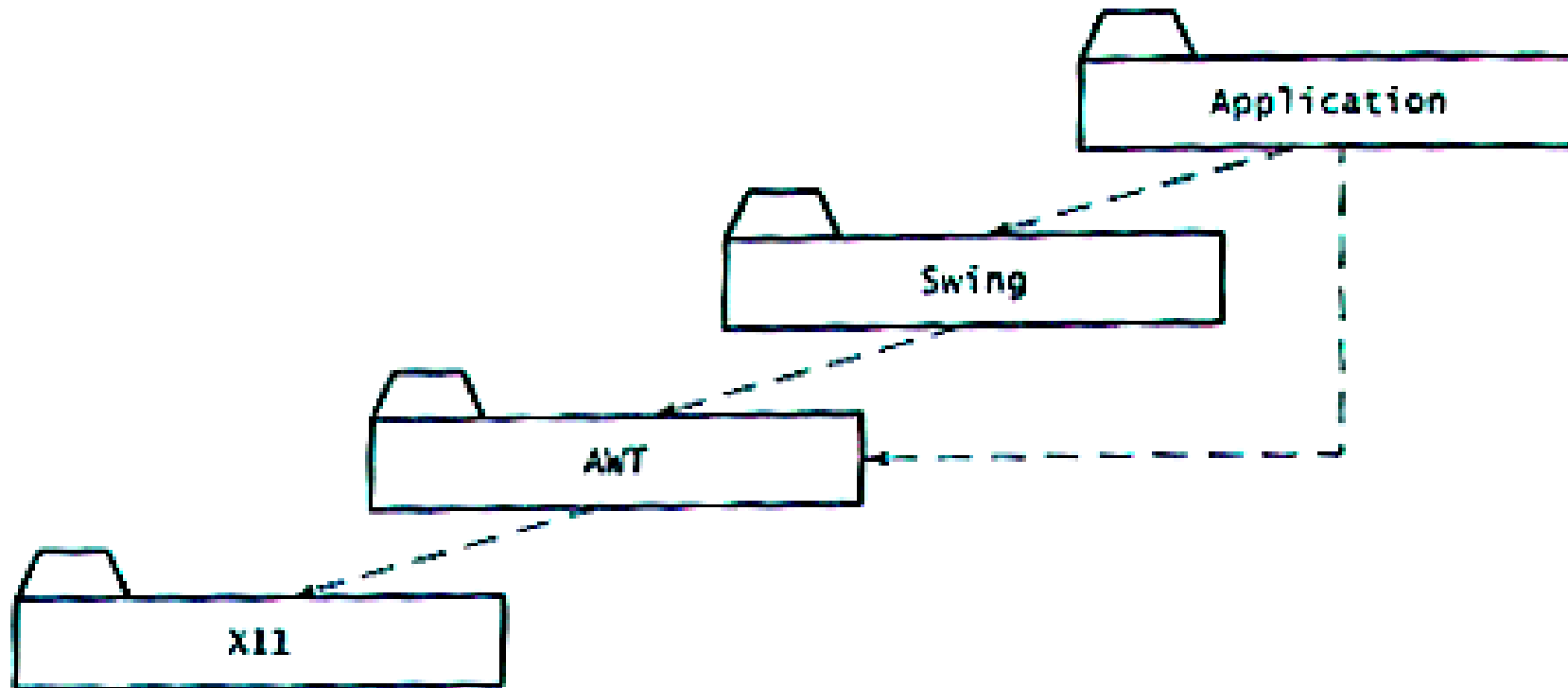
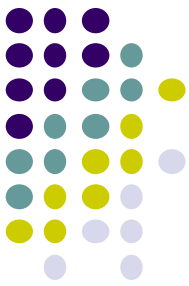
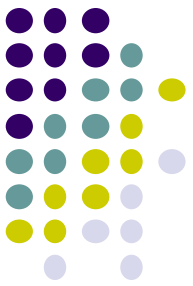


Figure 6-11 An example of open architecture: the Swing user interface library on an X11 platform (UML class diagram, packages collapsed). X11 provides low-level drawing facilities. AWT is the low-level interface provided by Java to shield programmers from the window system. Swing provides a large number of sophisticated user interface objects. Some Applications often bypass the Swing layer.



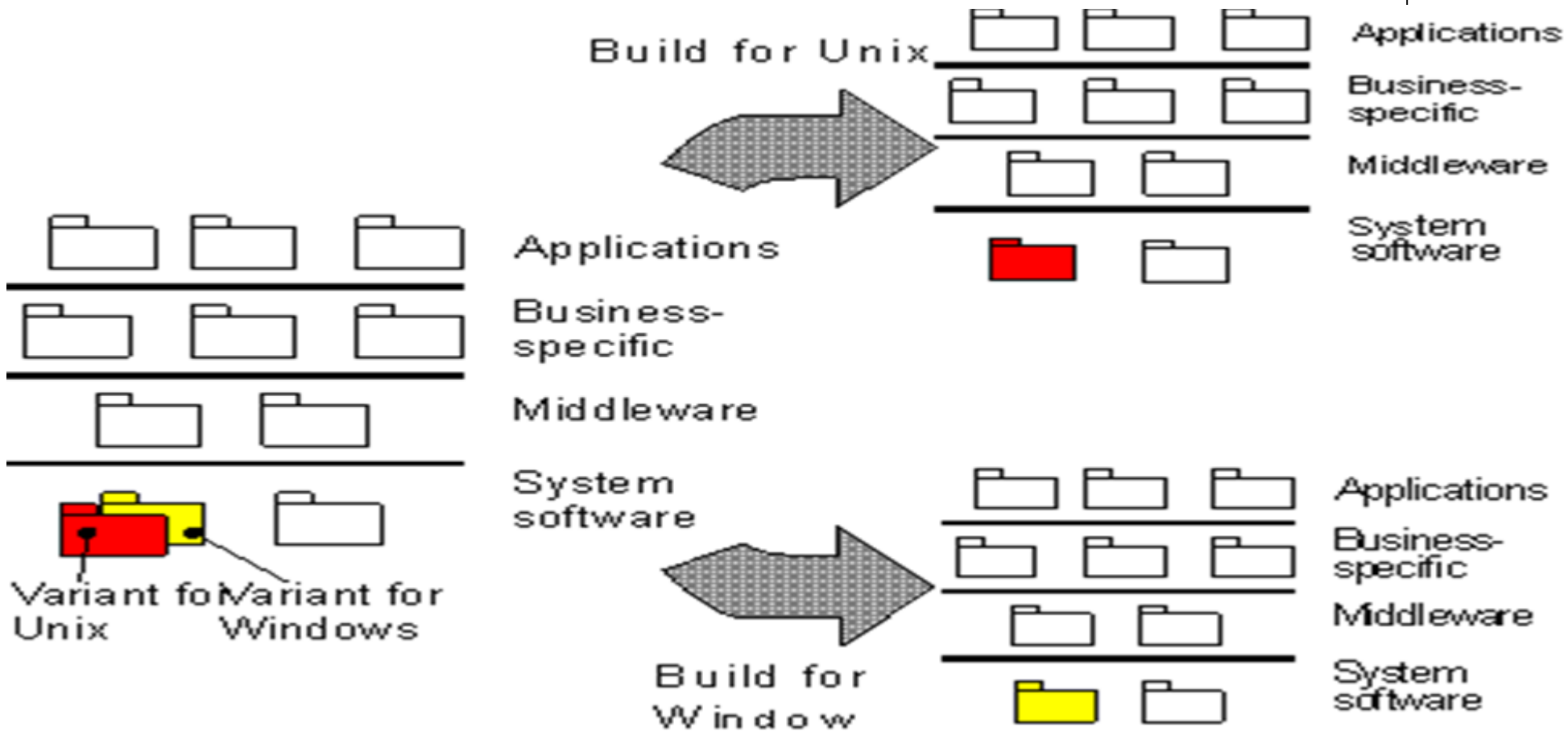
System Variants

Many systems are delivered in more than one variant. This means that the system is configured, packaged and installed differently for different (classes of) customers.

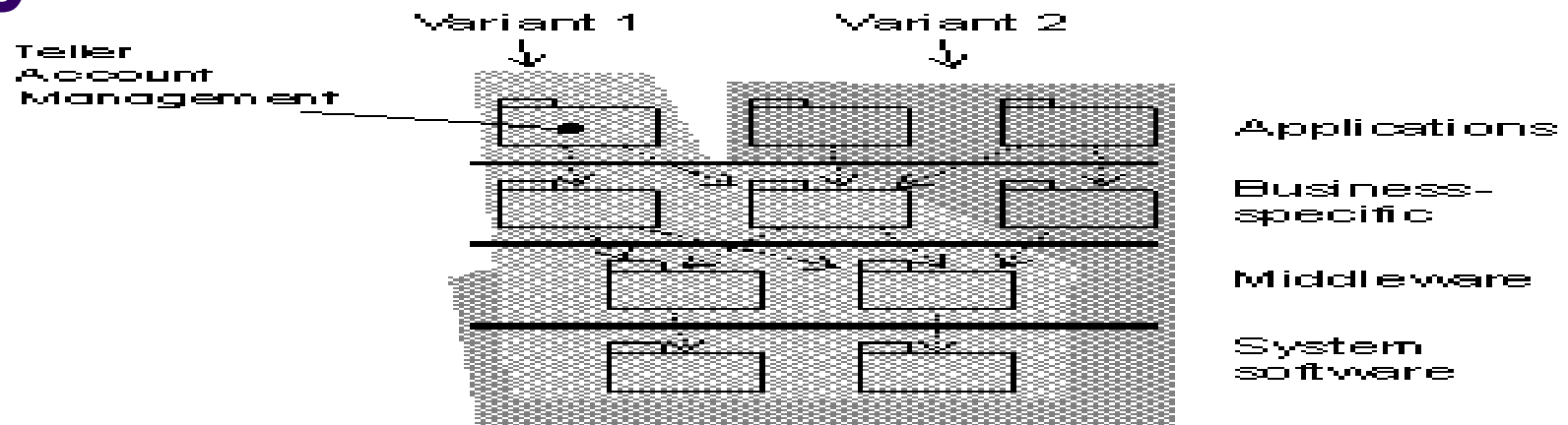
a) **different languages**

b) **different platforms:** in the example below, the platform-specific code is located in one subsystem. A compilation file (a 'makefile') specifies which version of each source code file should be compiled together.

System Variants for Different Platforms

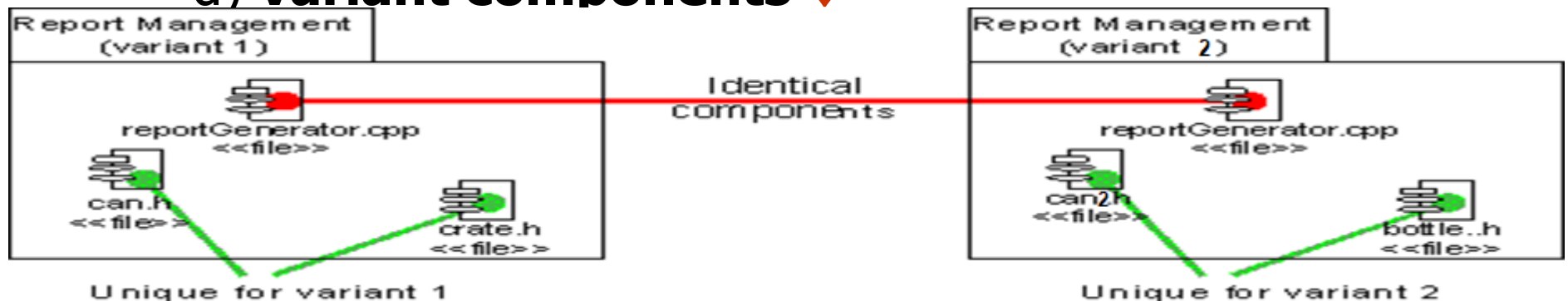


System Variants – cont.

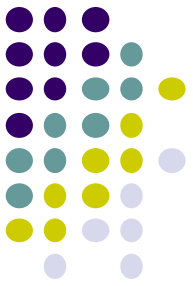


c) **different parts of the system** - for example, a banking system is delivered as two different products. Variant 1 of the system, contains everything about telephone banking; and variant 2, contains everything about teller account management ↑

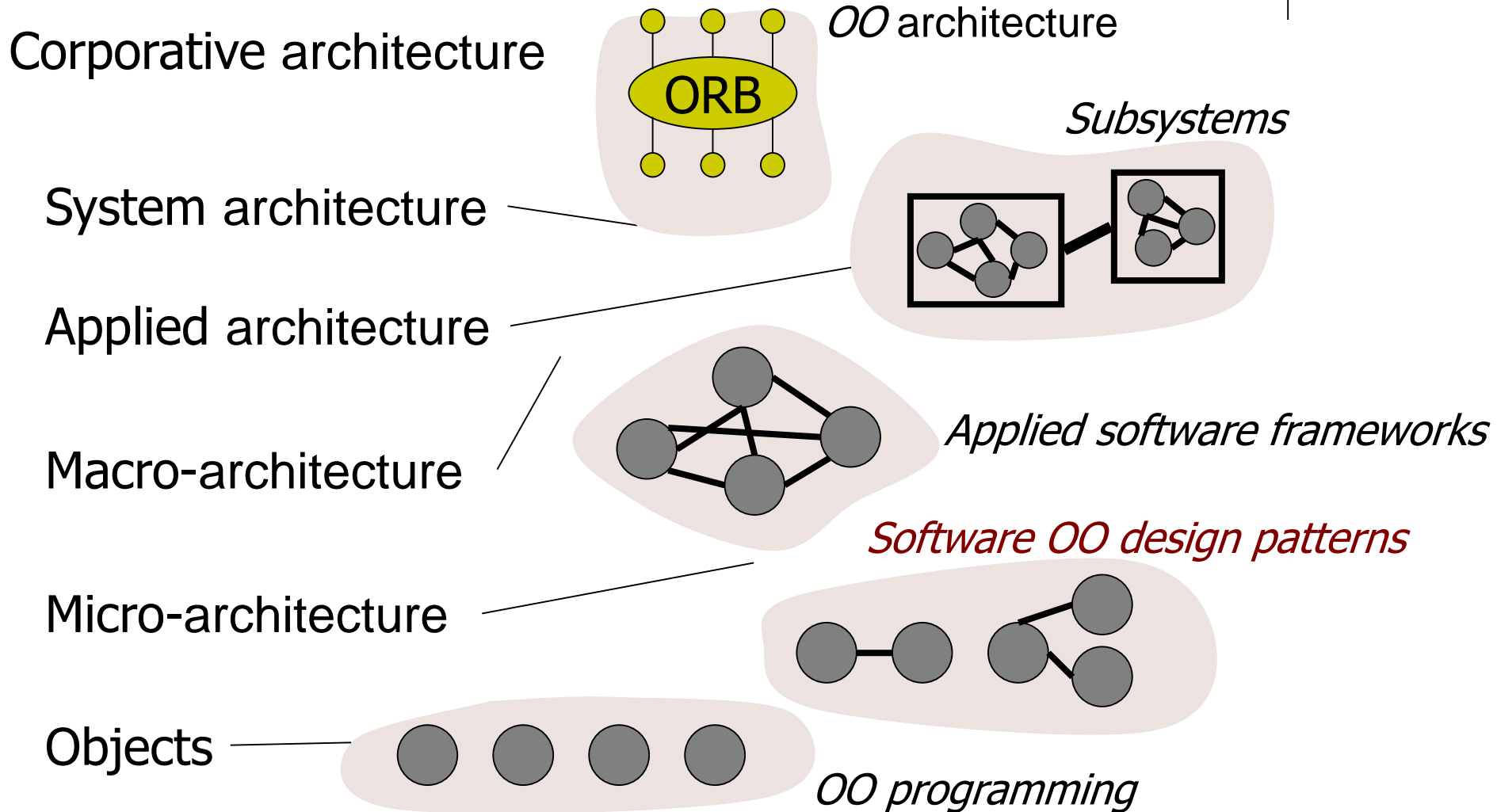
d) **variant components** ↓



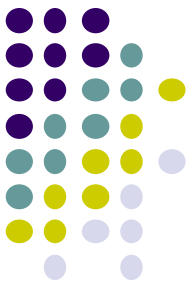
The seven levels of software architecture*



Global architecture



* Mowbray and Malveau, 1997



Architectural Style

- *Defines a family of systems by means of pattern for structural organization. In other words, it defines:*
 - Component dictionary and types of connecting elements
 - Set of restrictions and how we can combine them
 - One or more semantic models specifying how to determine common system properties based on the properties of its building blocks.

Repository Style [Bruegge & Dutoit'2004]

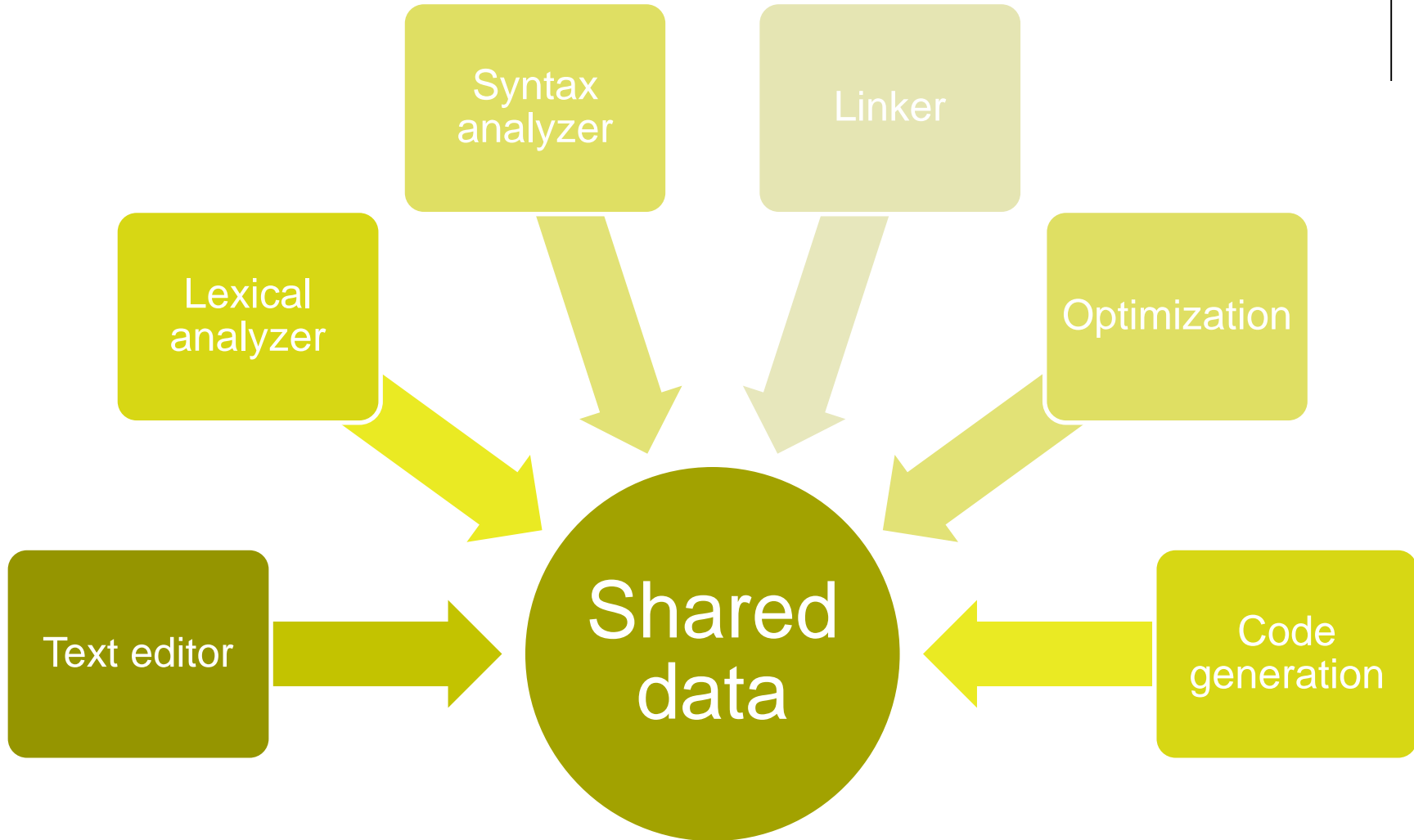
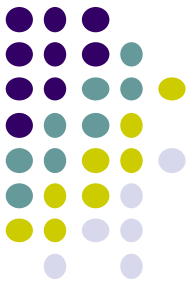


In the repository architectural style (see Figure 6-12), subsystems access and modify a single data structure called the central repository. Subsystems are relatively independent and interact only through the repository. Control flow can be dictated either by the central repository (e.g., triggers on the data invoke peripheral systems) or by the subsystems (e.g., independent flow of control and synchronization through locks in the repository).

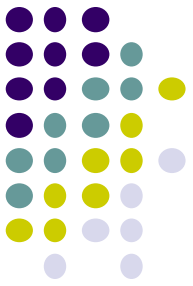


Figure 6-12 Repository architectural style (UML class diagram). Every Subsystem depends only on a central data structure called the Repository. The Repository has no knowledge of the other Subsystems.

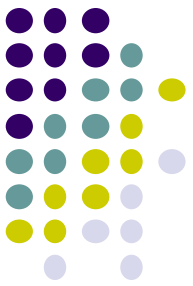
Repository Style for a Compiler



Benefits of the shared-data style



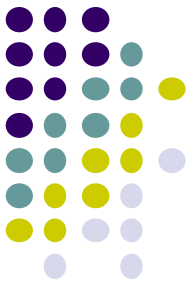
- Scalability - new components can be easily added
- Concurrency - all components can work in parallel
- Highly efficient when exchanging large amounts of data
- Centralized data management:
 - Better security, backup, etc.
 - The components are independent of the data manufacturer



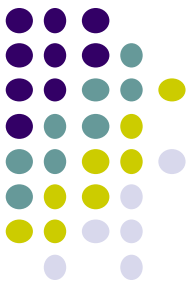
Drawbacks of shared-data

- It is difficult to apply in a distributed environment
- Shared data must support a single data model
- Changes to the model can lead to unnecessary costs
- Close relationship between the repository and data sources
- It can become a bottleneck in case of too many customers

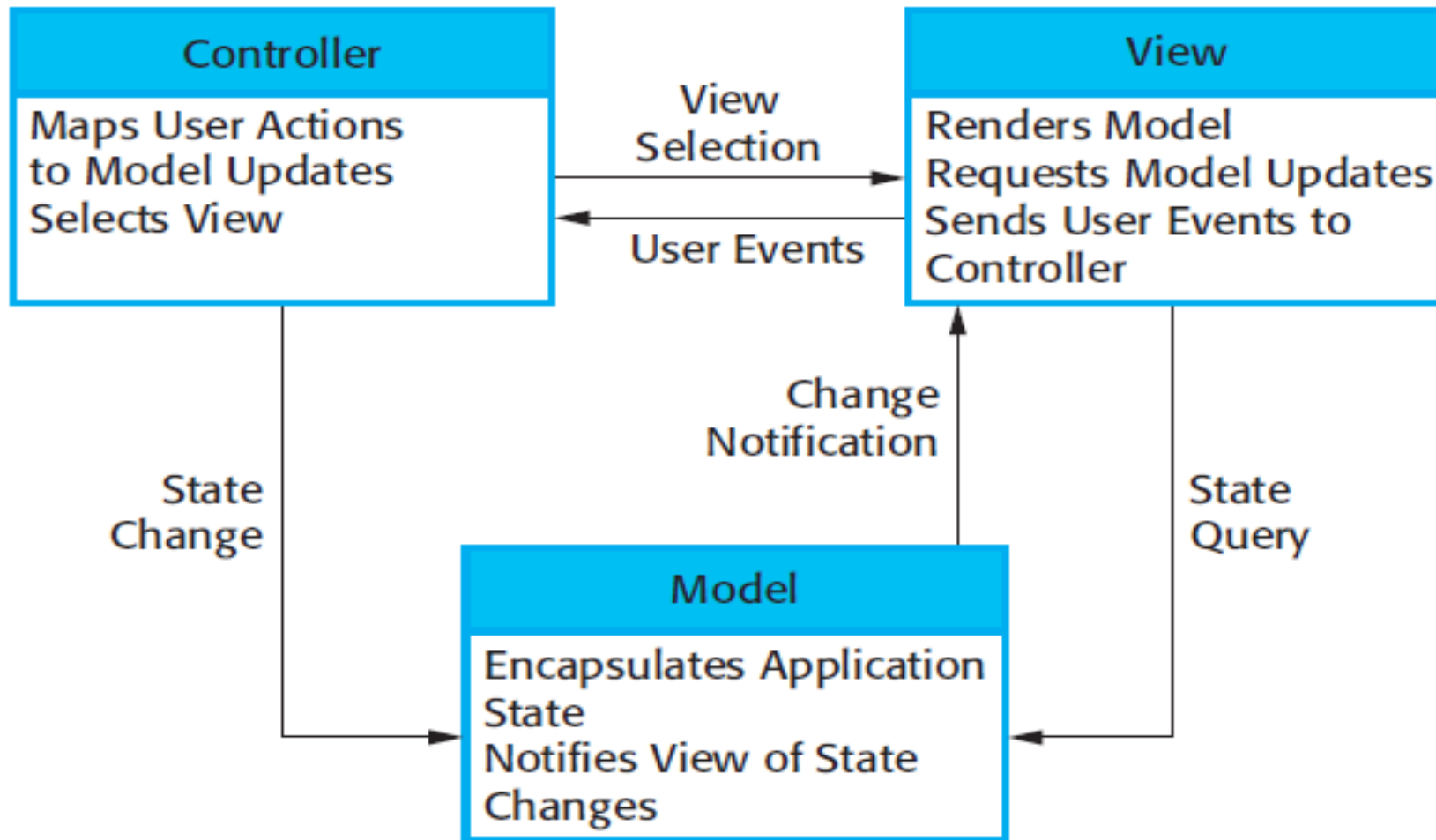
Model-View-Controller (MVC) Style

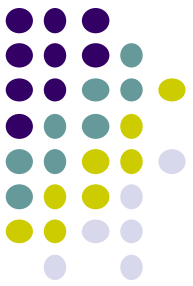


- Allows independence between data, data presentation and user control
- The **Model** component represents knowledge. It manages the behavior and data of the application's domain, sends information about its state (to the view) and responds to instructions to change the state (usually from the controller)
- **View** has the obligation to manage the presentation of information to users
- The **Controller** controls the interaction with the user (eg mouse clicks, keystrokes, etc.) and informs the model or view to take appropriate action



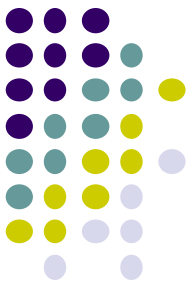
MVC





MVC - benefits

- Great flexibility
 - Easy to maintain and implement future improvements
 - Clear separation between presentation logic and business logic
- Easier update - e.g. when supporting new types of users
- The appearance is separate and in most systems undergoes many changes
- The development of the application is fast - many developers can easily collaborate and work together
- Easier to debug - we have several levels in the application



MVC - drawbacks

- Even if the data model is simple, this style can be complex and require a lot of additional code
- Not suitable for small applications - the architecture is relatively complex
- Performance issue with frequent updates to the model
- There must be strict rules on methods

Client-Server Style [Bruegge & Dutoit'2004]

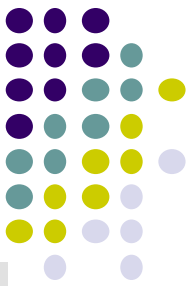
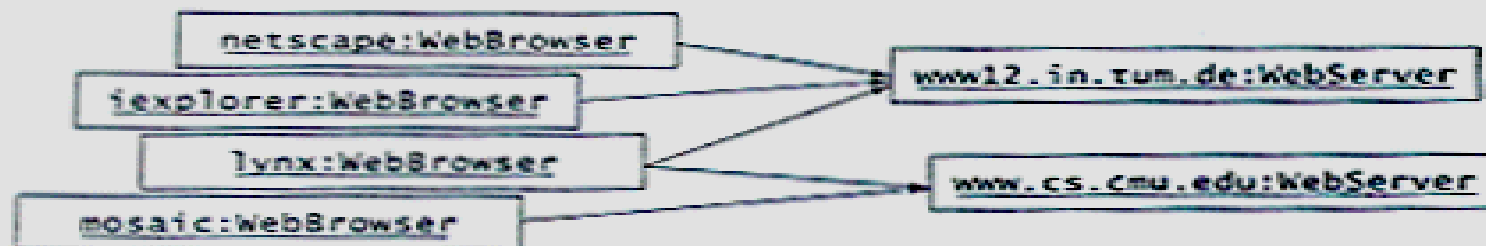
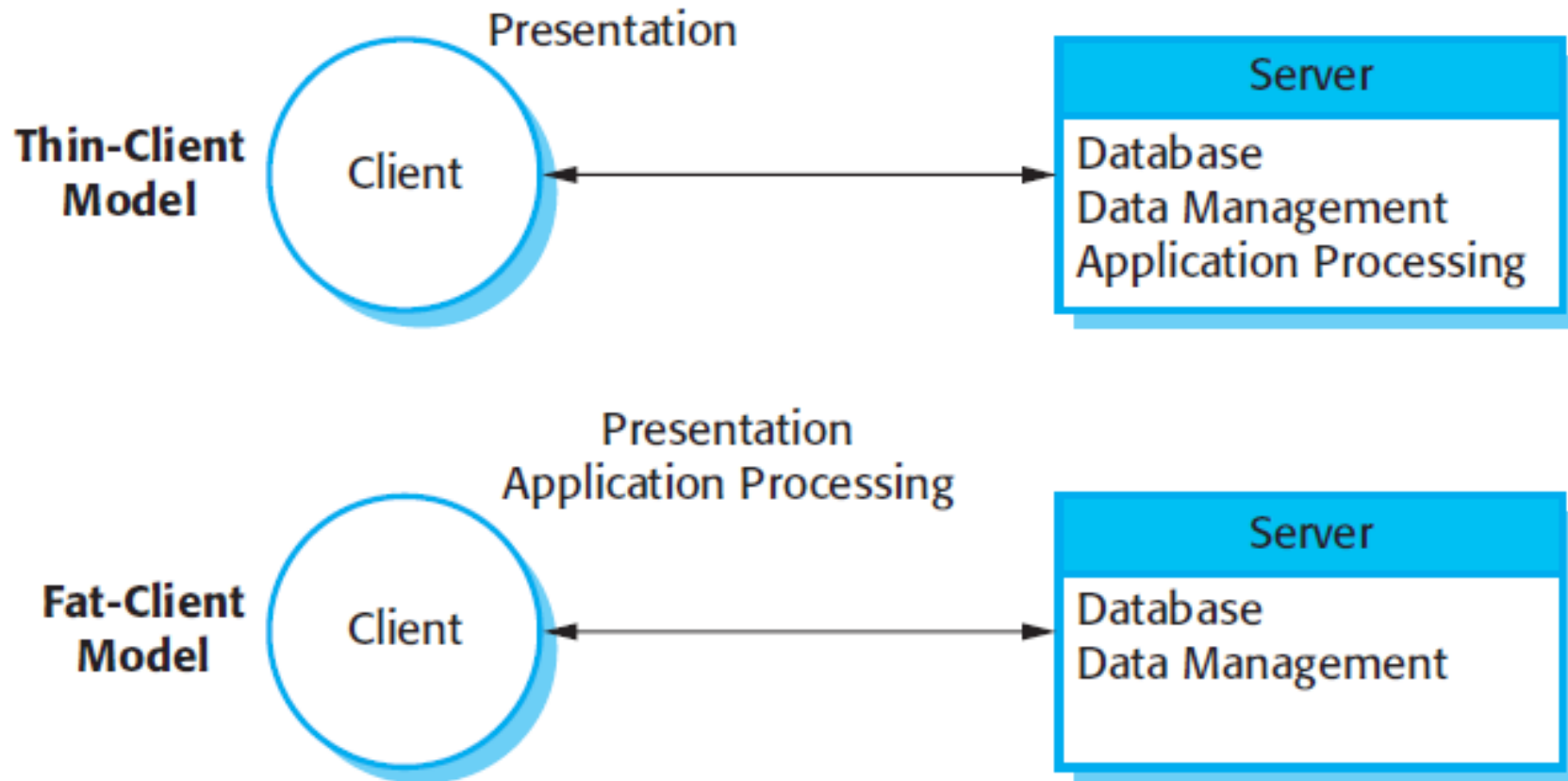
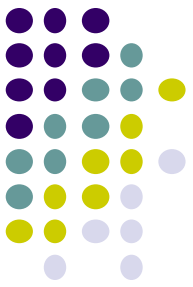


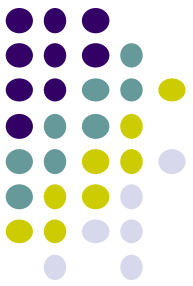
Figure 6-17 Client/server architectural style (UML class diagram). Clients request services from one or more Servers. The Server has no knowledge of the Client. The client/server architectural style is a specialization of the repository architectural style.

An information system with a central database is an example of a client/server architectural style. The clients are responsible for receiving inputs from the user, performing range checks, and initiating database transactions when all necessary data are collected. The server is then responsible for performing the transaction and guaranteeing the integrity of the data. In this case, a client/server architectural style is a special case of the repository architectural style in which the central data structure is managed by a process. Client/server systems, however, are not restricted to a single server. On the World Wide Web, a single client can easily access data from thousands of different servers (Figure 6-18).

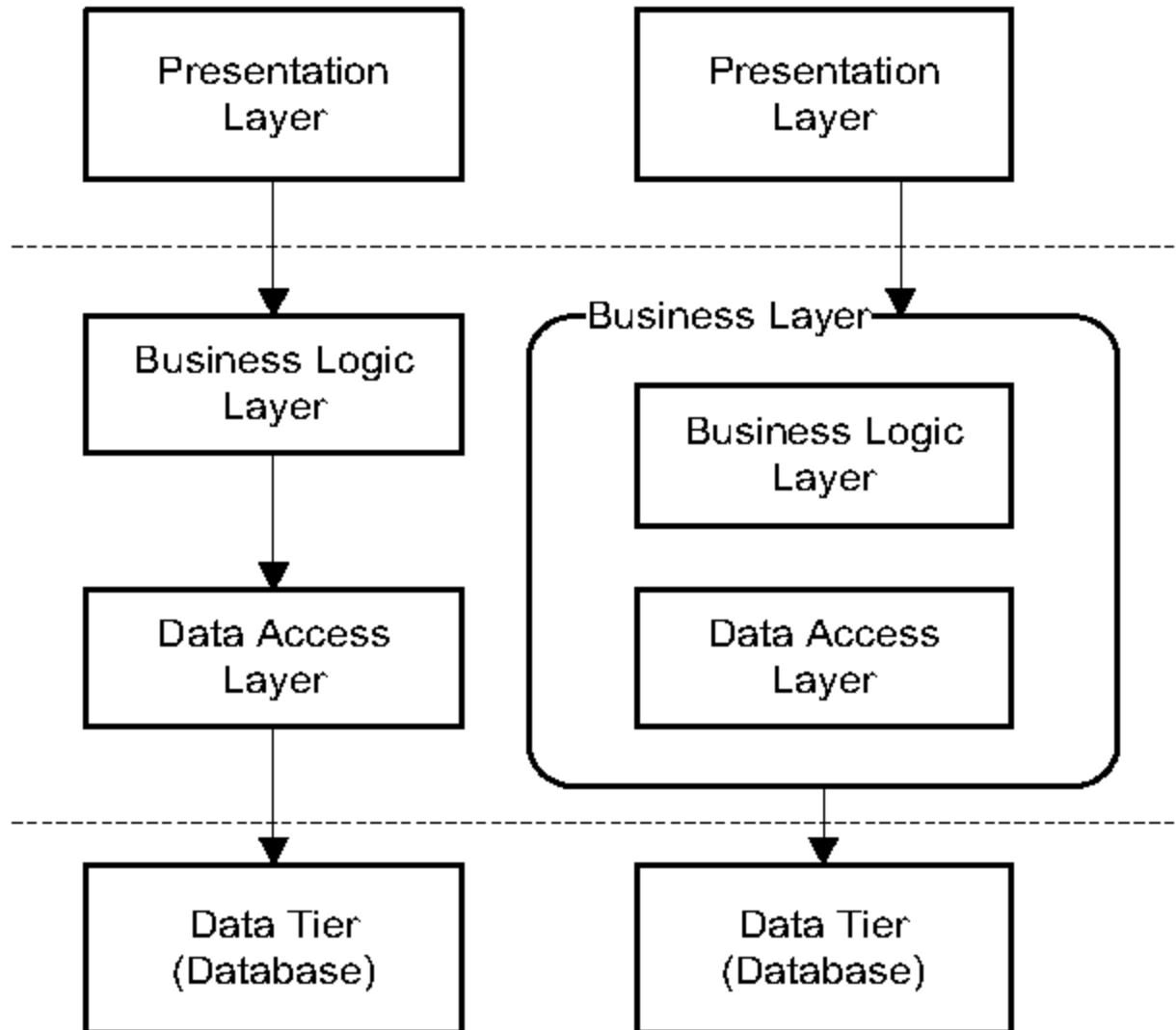


Thin or Fat Client at the Client-Server Style?

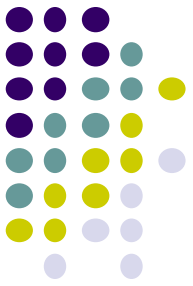




Three (Four) Tier Style



Pros's & Con's of the Client-Server Style



- Pro's:
 - Centralization of data – reusability, portability, modularity + abstractness
 - Security - at the client and at the servers
 - Easy implementation of backup and recovery
- Con's:
 - Server workload can increase too much on a large number of clients
 - What do we do in case of server failure - need for redundancy / fault-tolerance

Peer-to-Peer Style [Bruegge & Dutoit'2004]

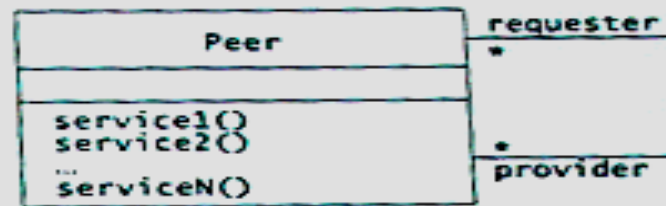
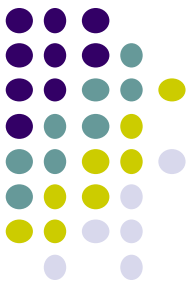
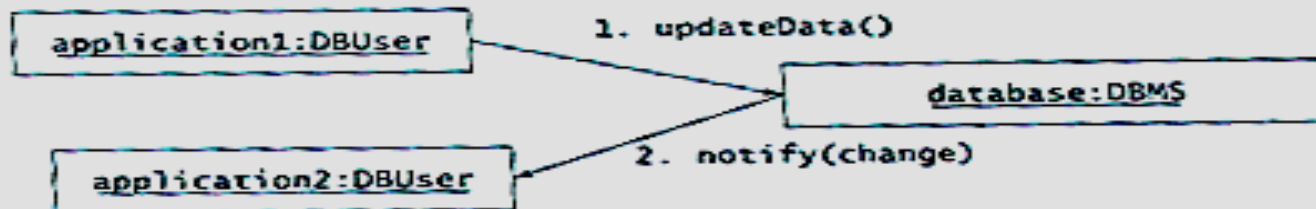
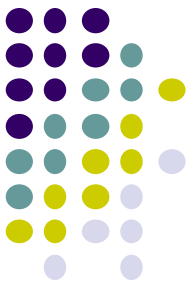


Figure 6-19 Peer-to-peer architectural style (UML class diagram). Peers can request services from and provide services to other peers.

An example of a peer-to-peer architectural style is a database that both accepts requests from the application and notifies to the application whenever certain data are changed (Figure 6-20). Peer-to-peer systems are more difficult to design than client/server systems because they introduce the possibility of deadlocks and complicate the control flow.

Callbacks are operations that are temporary and customized for a specific purpose. For example, a DBUser peer in Figure 6-20 can tell the DBMS peer which operation to invoke upon a change notification. The DBUser then uses the callback operation specified by each DBUser for notification when a change occurs. Peer-to-peer systems in which a "server" peer invokes "client" peers only through callbacks are often referred to as client/server systems, even though this is inaccurate since the "server" can also initiate the control flow.



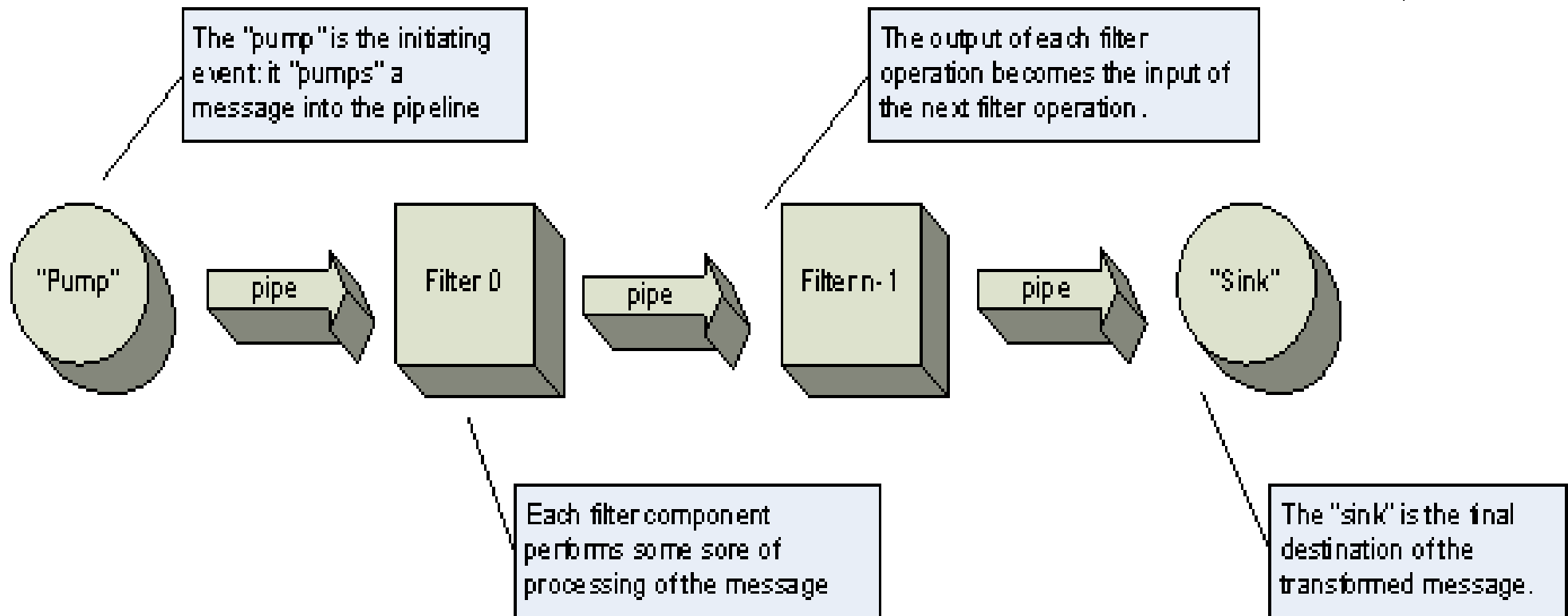
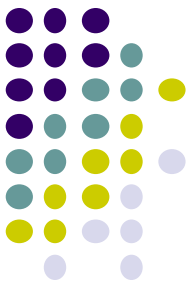


Pipe and Filter Style 1/2

- Each component (filter) in the system transfers the data in sequential order to the next component
- The connectors (pipes) between the filters are the actual data transmission mechanisms

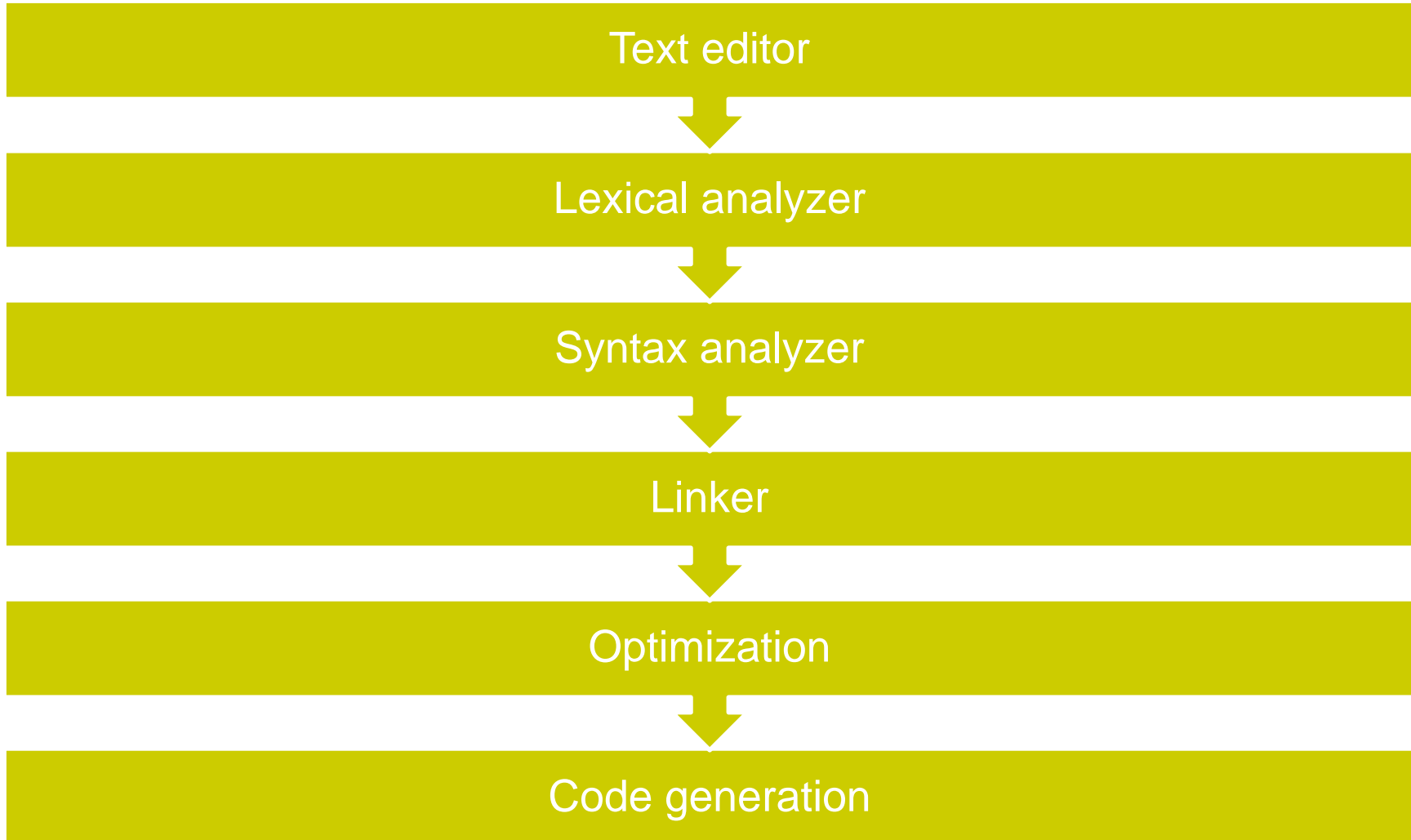
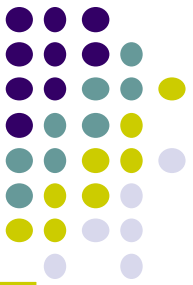


Pipe and Filter Style 2/2

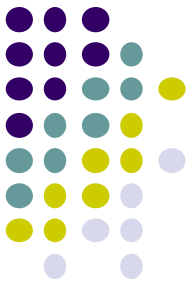


Subsystems (***filters***) process data received from other subsystems and send them via ***pipes*** (associations b/n subsystems).

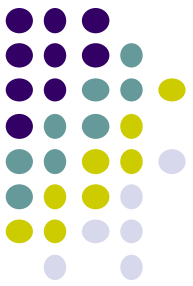
A Pipeline Architecture of a Compiler



Pipe-and-Filter style – FEATURES



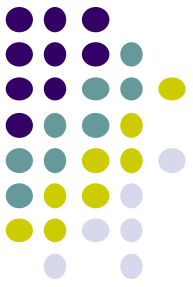
- **Complexity** - in a distributed environment, filters can run on different servers
- **Reliability** - uses an infrastructure that ensures that data will not be lost when the data passes between the filters in the pipeline
- **Idempotency** - detects and removes duplicate messages
- **Context and state** - each filter must be provided with sufficient context to perform its work, which may require a significant amount of status information



Benefits of pipe-and-filter style

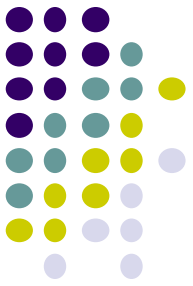
- Intuitive and easy to understand
- The filters are self-contained and can be treated as black boxes, which leads to flexibility in terms of maintenance and reuse
- Easy to implement parallelism (not in batch sequence where computations start after receiving all the packages)
- It is directly applicable to the structures of many business processes
- Easy to use when the processing required by the application can be easily decomposed into a set of discrete, independent steps

Drawbacks of the pipe-and-filter style



- Due to the successive implementation steps, it is difficult to implement interactive applications
- Low productivity
 - Each filter must analyze the data
 - Difficult to share global data
 - Filters must agree on the format of the data

Software Architecture Document

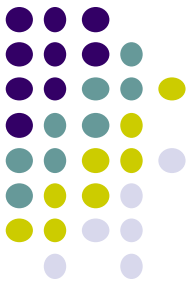


**Software
Architecture
Document**

The Software Architecture Document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different system aspects.

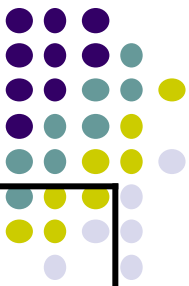
Software Architecture

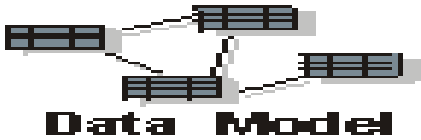
Document may include:



1. Objectives
2. Scope - what it applies to
3. References
4. Architectural Representation
5. Architectural Goals and Constraints
6. Use-Case View
7. Logical View
8. Process View
9. Deployment View
10. Implementation View
11. ***Data View (optional)***
12. Size and Performance
13. Quality: extendibility, reliability, portability...

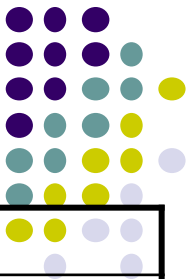
The Data Model



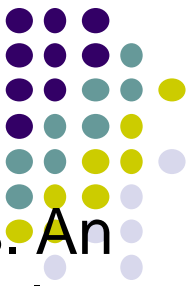
 <p>Data Model</p>	<p>The data model is a subset of the implementation model which describes the <i>logical and physical representation of persistent data</i> in the system. It also includes any behavior defined in the database, such as stored procedures, triggers, constraints, etc.</p>
<p>UML representation</p>	<p>A top-level Package stereotyped as «data model», containing a set of Components which represent the physical storage of persistent data in the system.</p>

A **Data Model** is a description of the persistent data storage perspective of the system. This section is *optional* if there is little or no persistent data, or trivial translation between the Design and Data Model.

Data Model Properties



Name	Brief Description	UML Representation
Packages	The packages used for organizational grouping purposes.	Owned via the association "represents", or recursively via the aggregation "owns".
Tables	The tables in the data model, owned by the packages.	Components, stereotyped as <<table>>.
Relationships	The relationships between tables in the model.	Associations, stereotyped as <<foreign key>>.
Columns	The data values of the tables.	Attributes, stereotyped as <<column>>.
Diagrams	The diagrams in the model, owned by the packages.	- " -
Indexes	Event-activated behavior associated with tables.	Components, stereotyped as <<index>>.
Triggers	Event-activated behavior associated with tables.	Operation, stereotyped as <<trigger>>.
Procedures	Explicitly invoked behavior, associated with tables or with the model as a whole.	Component, stereotyped as <<procedure>>.

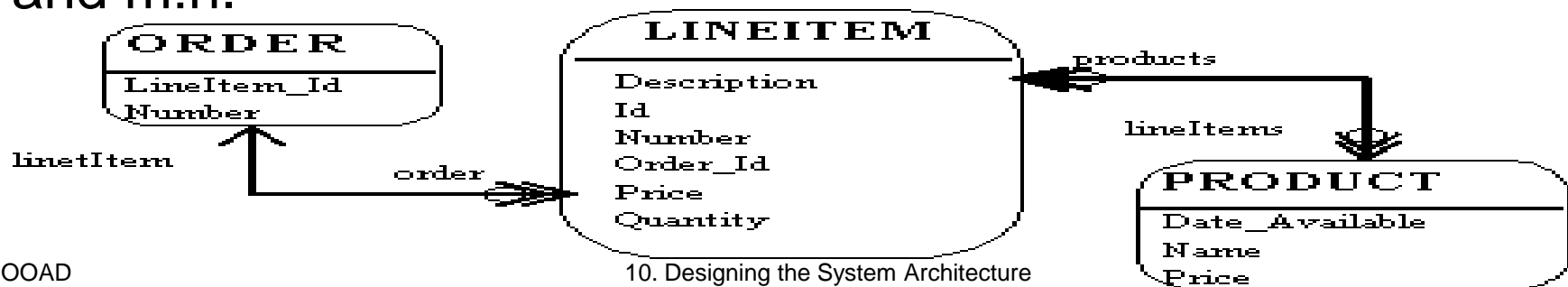


The Relational Model

The **relational model** is composed of **entities** and **relations**. An entity may be a physical table or a logical projection of several tables also known as a view.

An entity has **columns** and **records** or rows. Each entity has one or more **primary keys**. The primary keys uniquely identifies each record. **Foreign key** columns contain data which can relate specific records in the entity to the related entity.

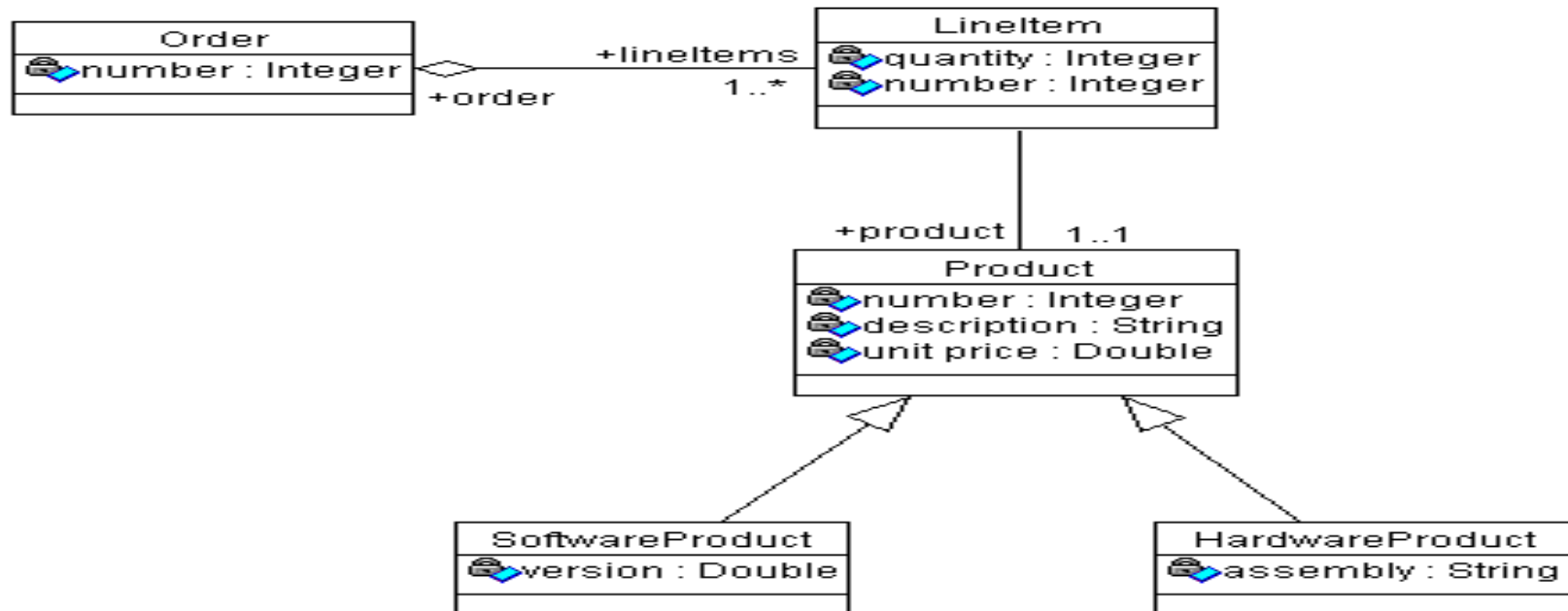
In the physical model relations are typically implemented using **foreign key/primary key references**. Relations have **multiplicity** (also known as cardinality). Common cardinalities are 1:1, 1:m, m:1, and m:n.

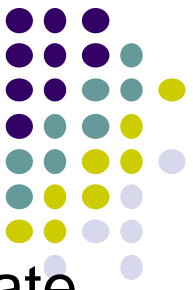




The Object Model

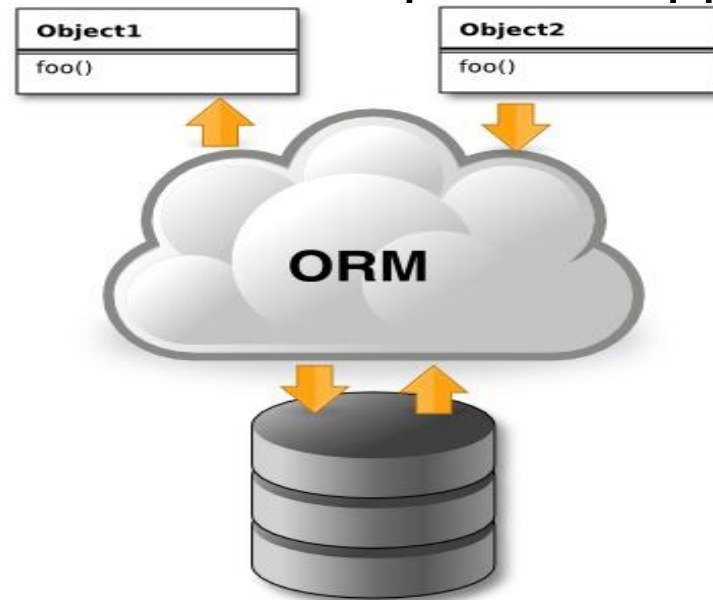
The object model contains **classes** defining the structure and behavior of a set of **objects**; sometimes called objects **instances**. The structure is represented as **attributes** (data values) and **associations** (relationships between classes). Supports **inheritance**. The following figure illustrates a simple class diagram model, showing only attributes (data) of the classes.



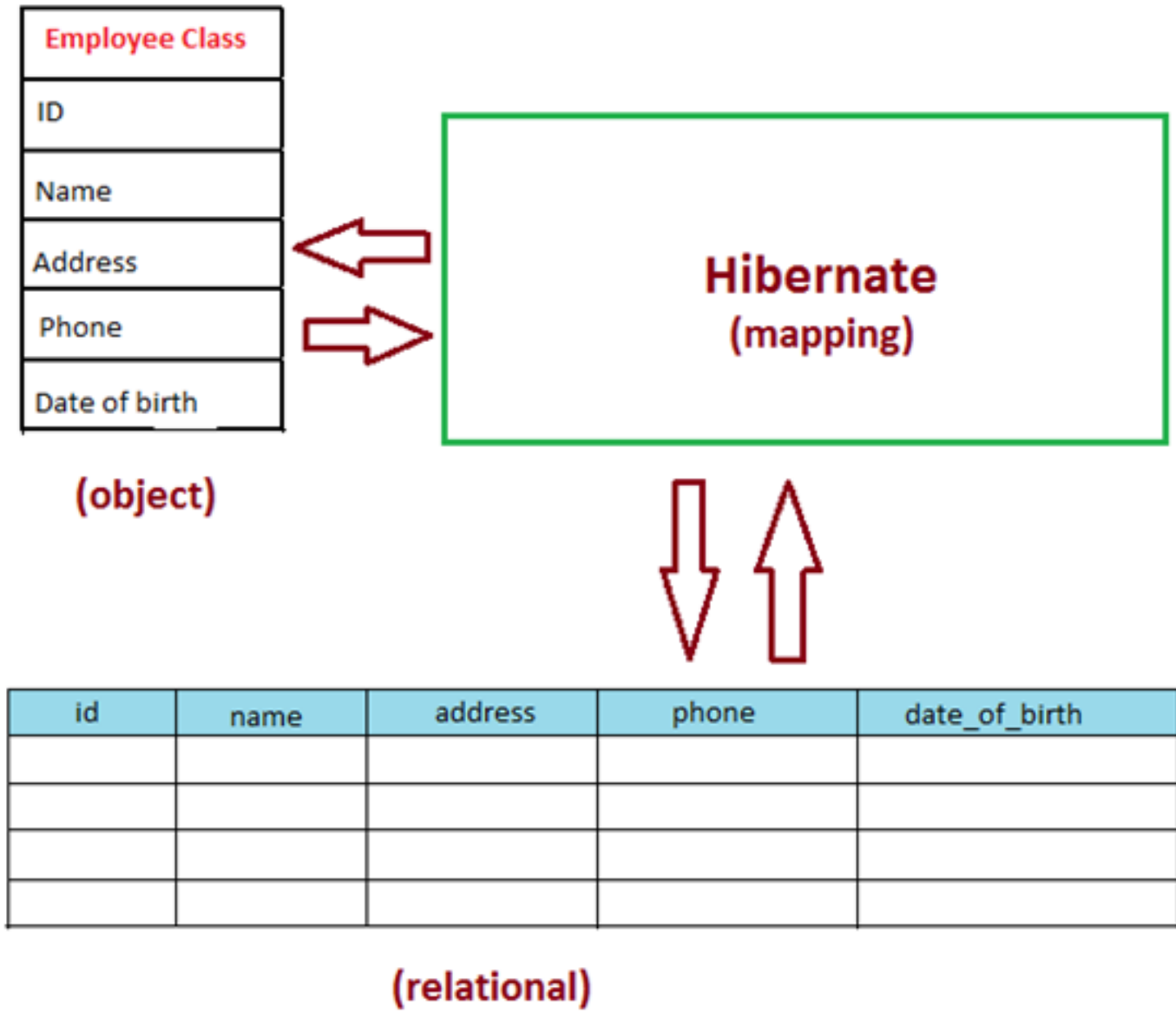
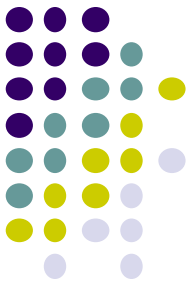


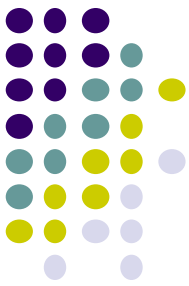
Persistence Frameworks (ORM Tools)

The role of the object-relational framework is to generically encapsulate the physical data store and to provide appropriate object translation services.



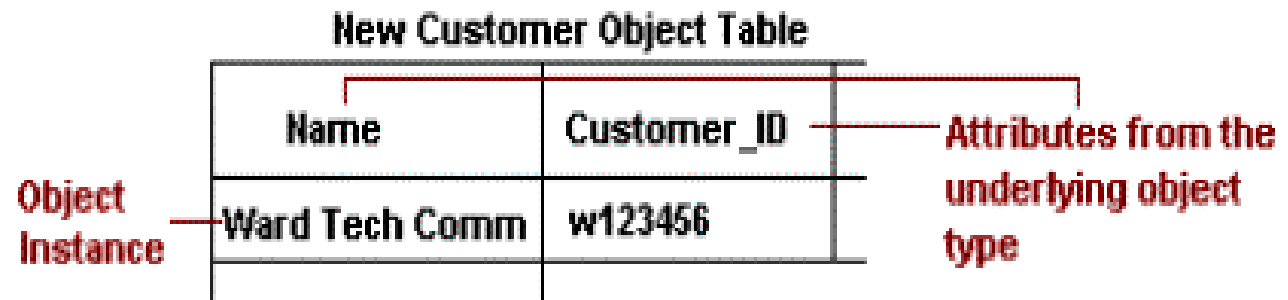
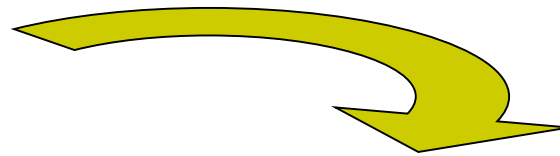
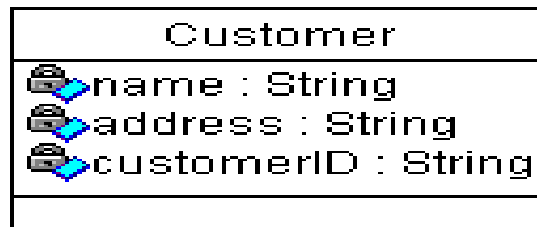
Application developers spend over 30% of their time implementing relational DB access in OO applications. Implementing an object-relational framework captures this investment. The object-relational mapping (ORM) tools can be reused in subsequent applications reducing the object-relational implementation cost to less than 10% of the total implementation costs.



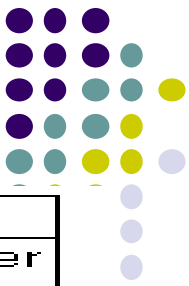


Mapping Persistent Classes to Tables

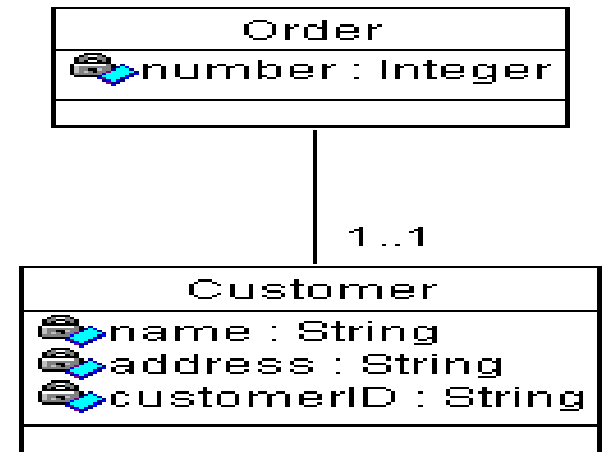
In a relational database written in third normal form, every row in the tables – every "tuple" – is regarded as an object. A column in a table is equivalent to a persistent attribute of a class. So, in the simple case where we have no associations to other classes, the mapping between the two worlds is simple. The data type of the attribute corresponds to one of the allowable data types for columns.



Mapping Associations between Persistent Objects



Associations between two persistent objects are realized as **foreign** keys to the associated objects. A **foreign key** is a column in one table which contains the **primary key** value of associated object.



When we map this into relational tables, we get an Order table and a Customer table. The Order table will have columns for attributes listed, plus an additional column Customer_ID which contains foreign-key references to associated rows in the Customer table. For a given Order, the Customer_ID column will contain the identifier of the Customer to whom the Order is associated. Foreign keys allow the RDBMS to **join** related information together.

Mapping of 1:N association

[Bruegge & Dutoit'2004]

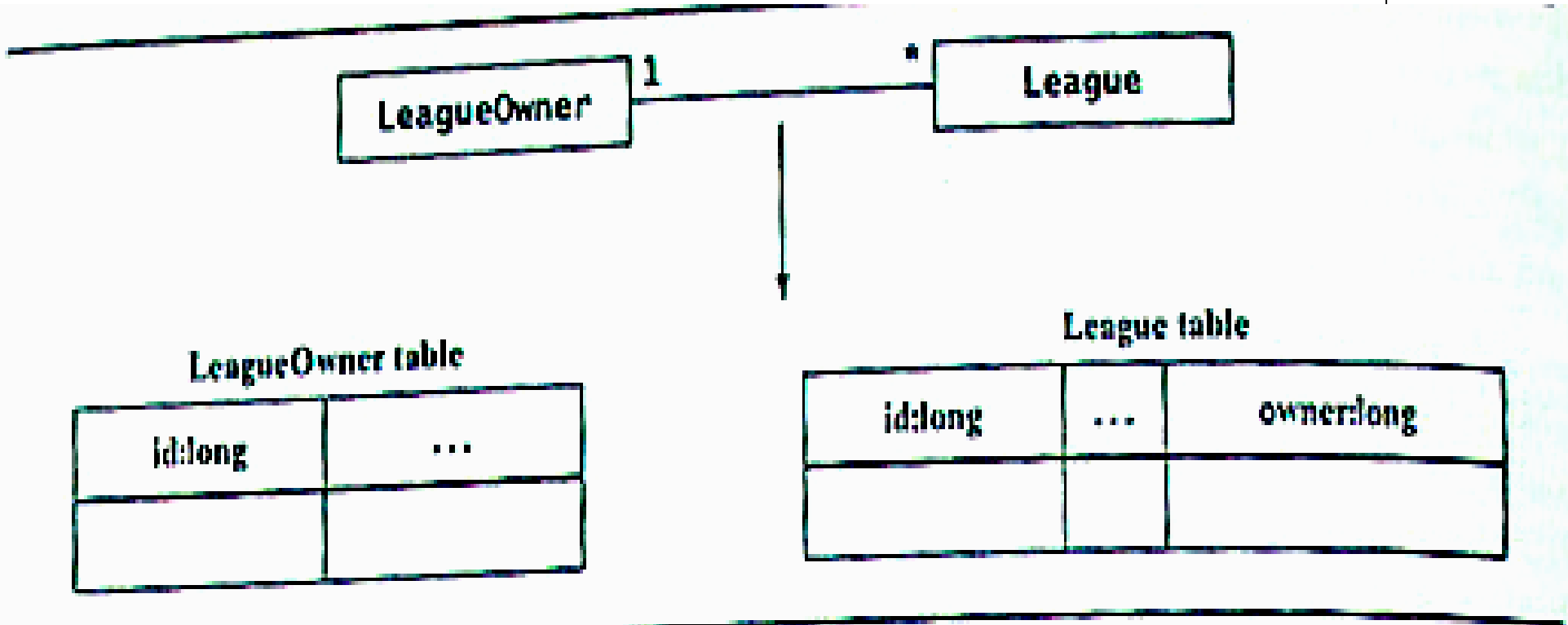
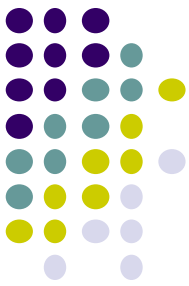


Figure 10-19 Mapping of the LeagueOwner/League association as a buried association.

Mapping of M:N association

[Bruegge & Dutoit'2004]

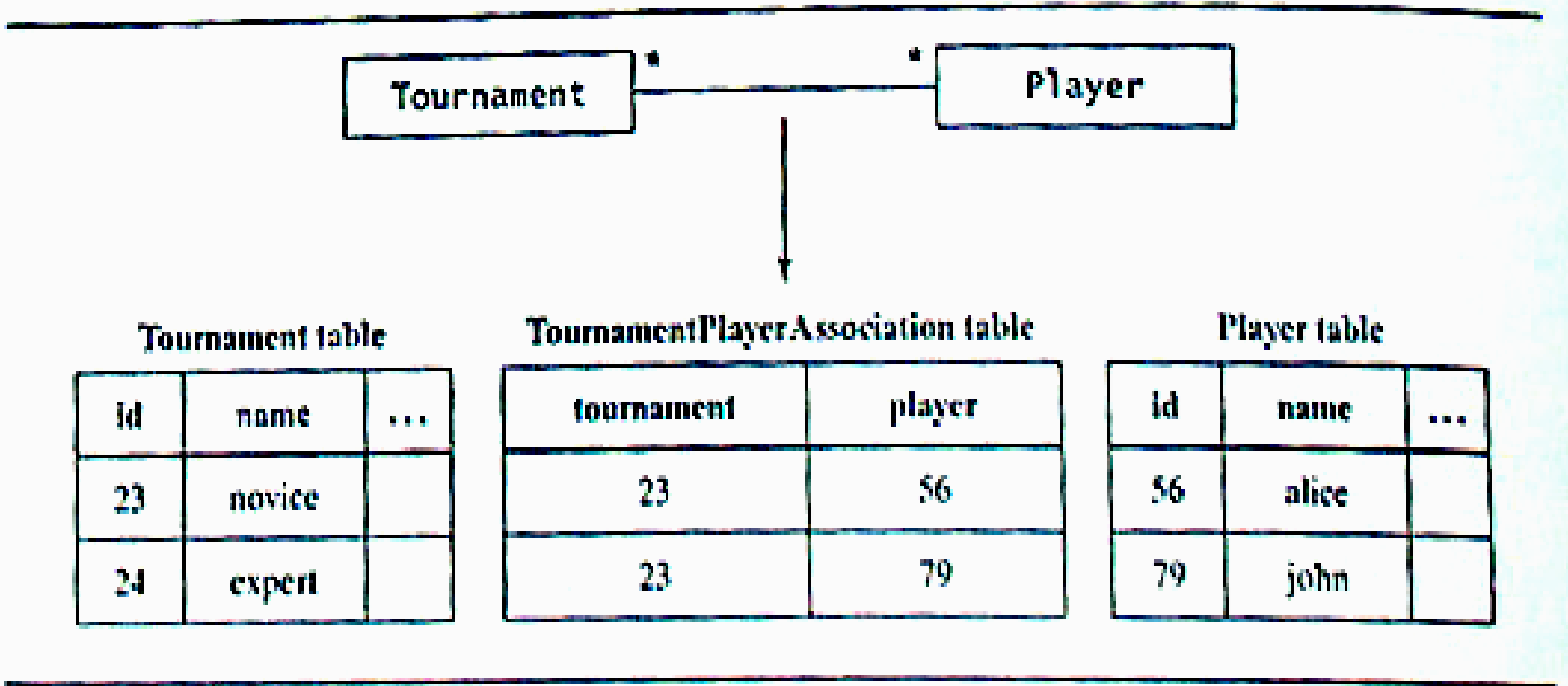
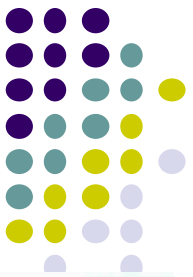


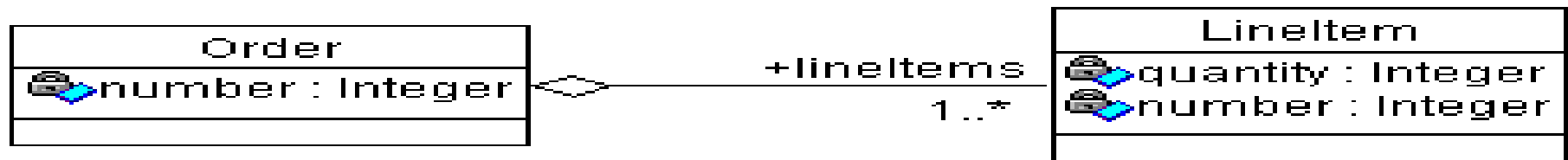
Figure 10-20 Mapping of the **Tournament/Player** association as a separate table.



Mapping Aggregation Associations to the Data Model

Aggregation is also modeled using foreign key relationships.

When we map this into relational tables, we get an Order table and a Line_Item table. The Line_Item table will have columns for attributes listed, plus an additional column Order_ID which contains foreign-key references to associated rows in the Order table. *For a given Line Item, the Order_ID column will contain the Order_ID of the Order that the Line Item is associated with.* Foreign keys allow the RDBMS to **join** related information together.



Modeling Inheritance and Many-to-Many Associations

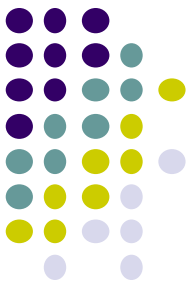


The standard relational data model does not support modeling inheritance associations in a **direct** way but:

1. Use **separate tables** to represent the super-class and sub-class. Have, in the sub-class table, a foreign key references to the super-class table.
2. **Duplicate all inherited attributes** and associations as separate columns in the sub-class table. This is similar to **de-normalization** in the standard RDBS.

A standard technique in relational modeling is to use an **intersection entity** to represent many-to-many associations. The same approach should be used here: an intersection table should be used to represent the association.

Example: If Suppliers can supply many Products, and a Product can be supplied by many Suppliers, the solution is to create a Supplier/Product table.



Homework: VP ORM Mapping

