SCM

***FIRST LAW OF SYSTEM ENGINEERING* STATES:**

“NO MATTER WHERE YOU ARE IN THE SYSTEM LIFE CYCLE, THE SYSTEM WILL CHANGE, AND THE DESIRE TO CHANGE IT WILL PERSIST THROUGHOUT THE LIFE CYCLE.”

SCM

- Software Configuration Management is a set of tracking and control Activities that are initiated **when a software engineering project begins and terminates only when the software is taken out of operation**

SCM

- Software configuration management (SCM) is an umbrella activity that is applied throughout the software process.

- SCM activities are developed to

1) **Identify change**
2) **Control change**
3) **Ensure that change is properly implemented**
4) **Report changes to others** who may have an interest(stakeholders)

SCM

- The output of the software process is divided into 3 categories:
  (1) computer programs (both source level and executable forms),
  (2) work products that describe the computer programs (targeted at various stakeholders),
  (3) data or content (contained within the program or external to it).

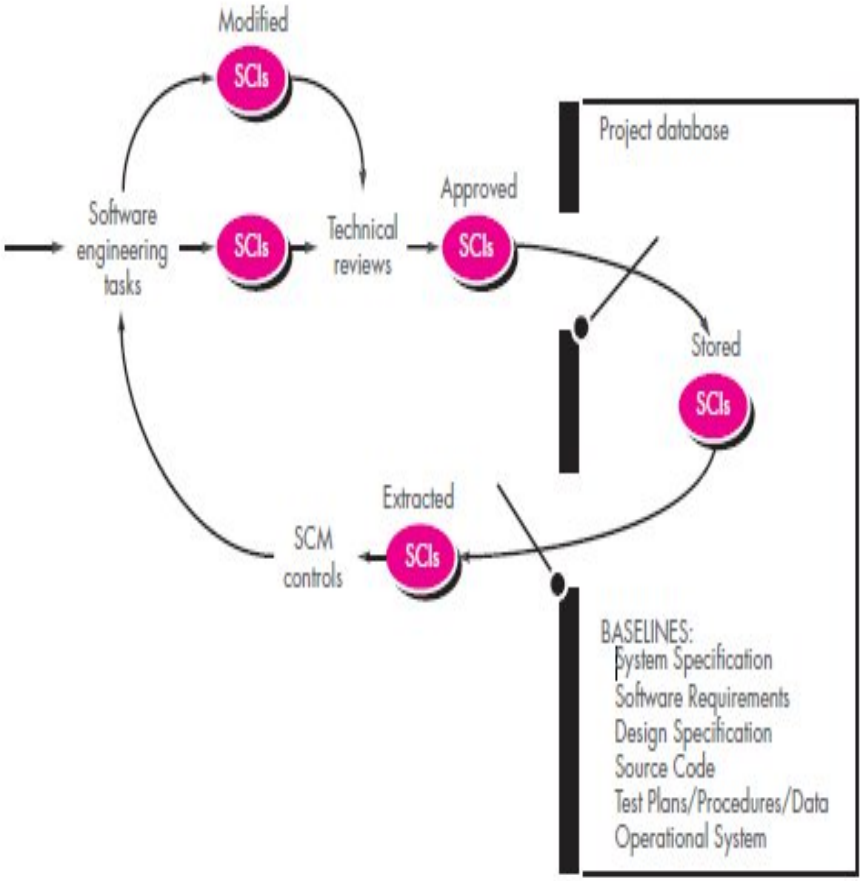
- The **items that comprise all information produced as part of the software process are collectively called a *software configuration*.**

SCM

- As software engineering work progresses,
  - a hierarchy of *software configuration items* (SCIs)
    - a named element of information
    - that can be as small as a single UML diagram or as large as the complete design document
  - is created.

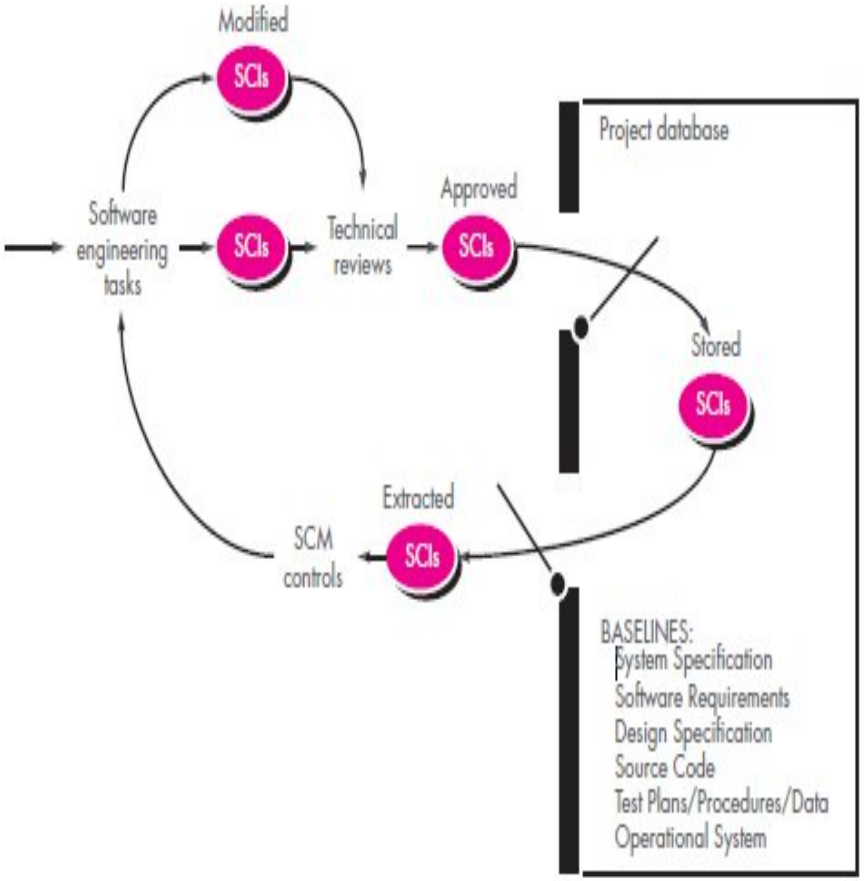
Baselines

- The progression of events that lead to a baseline is illustrated in Figure.
- Software engineering tasks produce one or more SCIs.
- After SCIs are reviewed and approved,
  - they are placed in a *project database* (also called a *project library* or *software repository*).



Baselines

- When a member of a software engineering team wants to make a modification to a baselined SCI,
- It is copied from the project database into the engineer's private workspace.
- However, this extracted SCI can be modified only if SCM controls are followed.
- The arrows illustrate the modification path for a baselined SCI.



An SCM Scenario

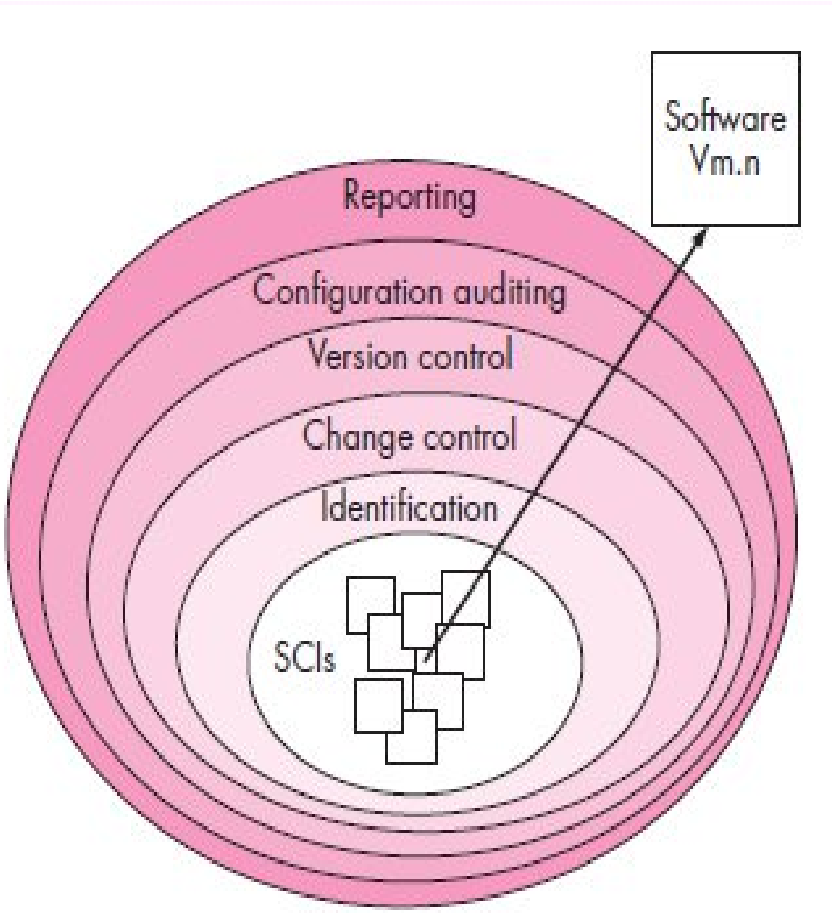
- A typical CM operational scenario involves a
  - **project manager** who is in charge of a software group,
  - **a configuration manager** who is in charge of the CM procedures and policies,
  - **the software engineers** who are responsible for developing and maintaining the software product,
  - **the customer** who uses the product.

SCM Features

- **Versioning**
- **Dependency tracking and change management**
- **Requirements tracing**
- **Configuration management**
- **Audit trails**

THE SCM PROCESS

1) Identification of Objects in the Software Configuration
2) Version Control
3) Change Control
4) Configuration Audit
5) Status Reporting



Identification of Objects in the Software Configuration

- To control and manage software configuration items,
  - each should be separately named
  - then organized using an object-oriented approach.

- Two types of objects can be identified :
  - basic objects
  - aggregate objects.

Identification of Objects in the Software Configuration

- *Basic object* –
  - **a unit of information that you create during analysis, design, code, or test.**

  For example,
  - section of a requirements specification,
  - part of a design model,
  - source code for a component, or
  - a suite of test cases that are used to exercise the code.

- An *aggregate object*
  - **a collection of basic objects and other aggregate objects.**

  For example,
  - a **DesignSpecification** is an aggregate object.

What is Version Control?

Version control combines

- procedures and tools
- **to manage different versions of configuration objects that are created during the software process.**

Version Control System

Implements or is directly integrated with these major capabilities:
(1) a project database that stores all relevant configuration objects,
(2) a *version management* capability that stores all versions of a configuration object
(3) a *make facility* that enables you to collect all relevant configuration objects and construct a specific version of the software.

Version Control

- Version control and change control systems often implement
  - an *issues tracking*
  - also called *bug tracking*
  - capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

Version Control

- A number of version control systems establish a ***change set—***
  - a collection of all changes (to some baseline configuration)
  - that are required to create a specific version of the software.
- A change set captures
  - **all changes to all files** in the configuration along
  - **with the reason for changes** and
  - details of **who made the changes and when**.

Version Control

- A number of named change sets can be identified for an application or system.
- This enables you to
  - **construct a version of the software**
  - **by specifying the change sets** (by name)
  - **that must be applied to the baseline configuration.**

- To accomplish this, a *system modeling* approach is applied.

Version Control

The system model contains:
(1) a *template* that includes a component hierarchy and a "build order" for the components that describes how the system must be constructed,
(2) construction rules, and
(3) verification rules.

Change Control

WHY?

- For a large software project,
  - uncontrolled change rapidly leads to chaos.

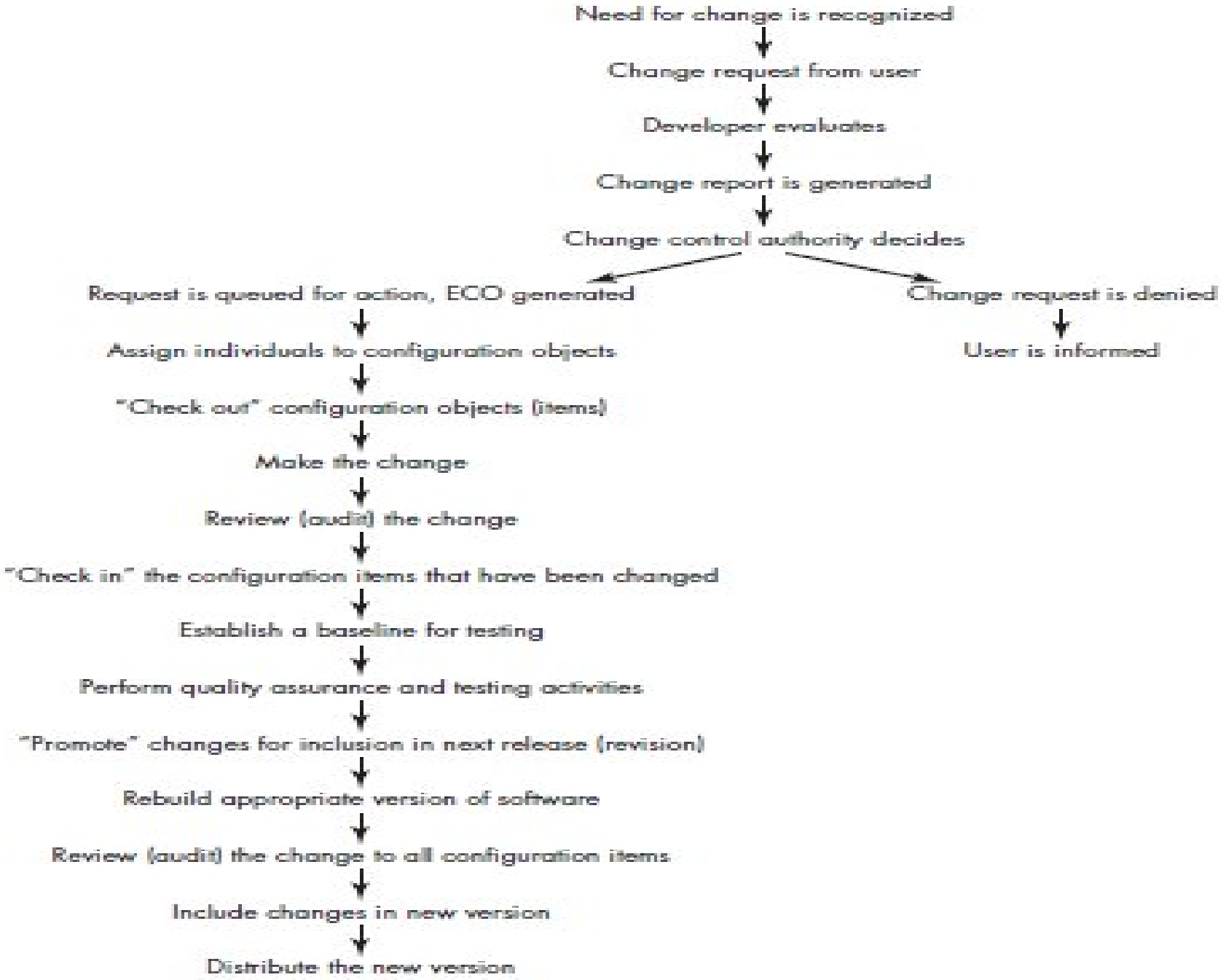
Change Control

HOW?

- For such projects, change control combines
  - human procedures and automated tools
  - to provide a mechanism for the control of change.

Change Control

The change control process is illustrated schematically in Figure.

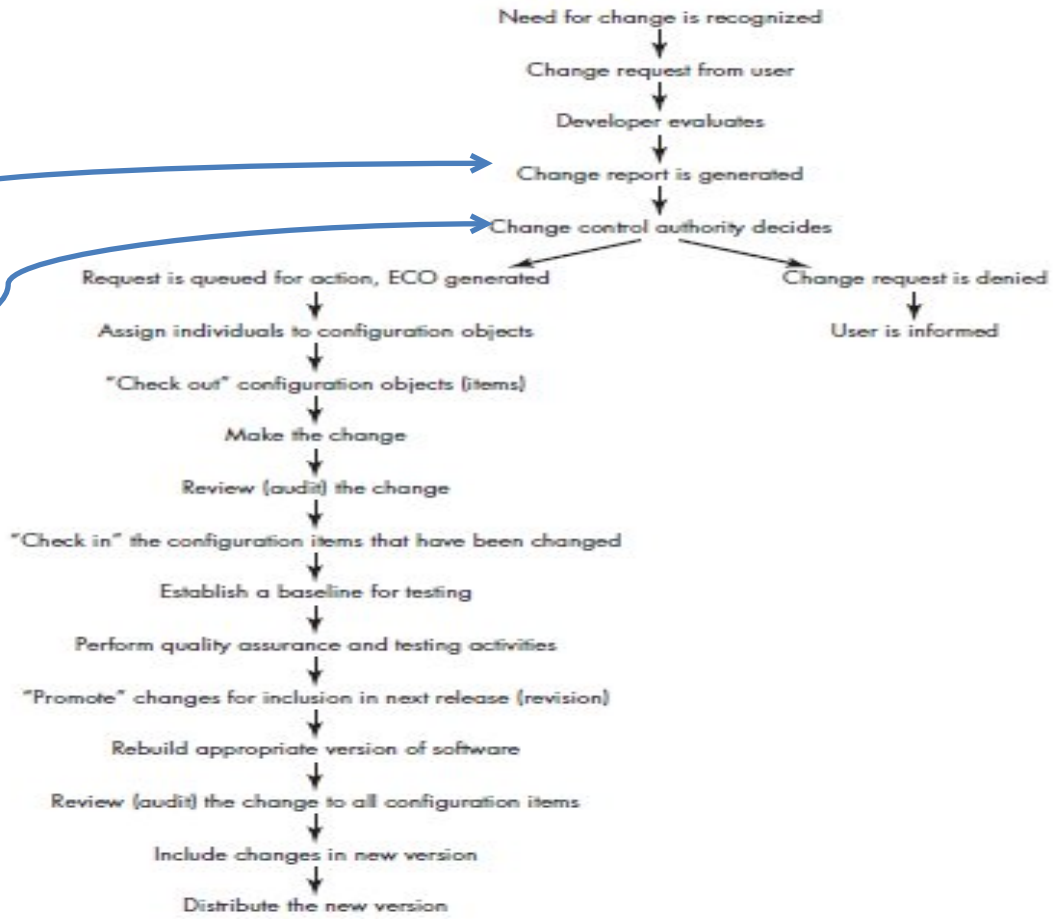

Need for change is recognized
Change request from user
Developer evaluates
Change report is generated
Change control authority decides
Request is queued for action, ECO generated
Change request is denied
Assign individuals to configuration objects
User is informed
"Check out" configuration objects (items)
Make the change
Review (audit) the change
"Check in" the configuration items that have been changed
Establish a baseline for testing
Perform quality assurance and testing activities
"Promote" changes for inclusion in next release (revision)
Rebuild appropriate version of software
Review (audit) the change to all configuration items
Include changes in new version
Distribute the new version

Change Control

- A *change request*
  - is submitted and
  - evaluated to
    - assess technical merit,
    - potential side effects,
    - overall impact on other configuration objects
    - system functions,
    - projected cost of the change.

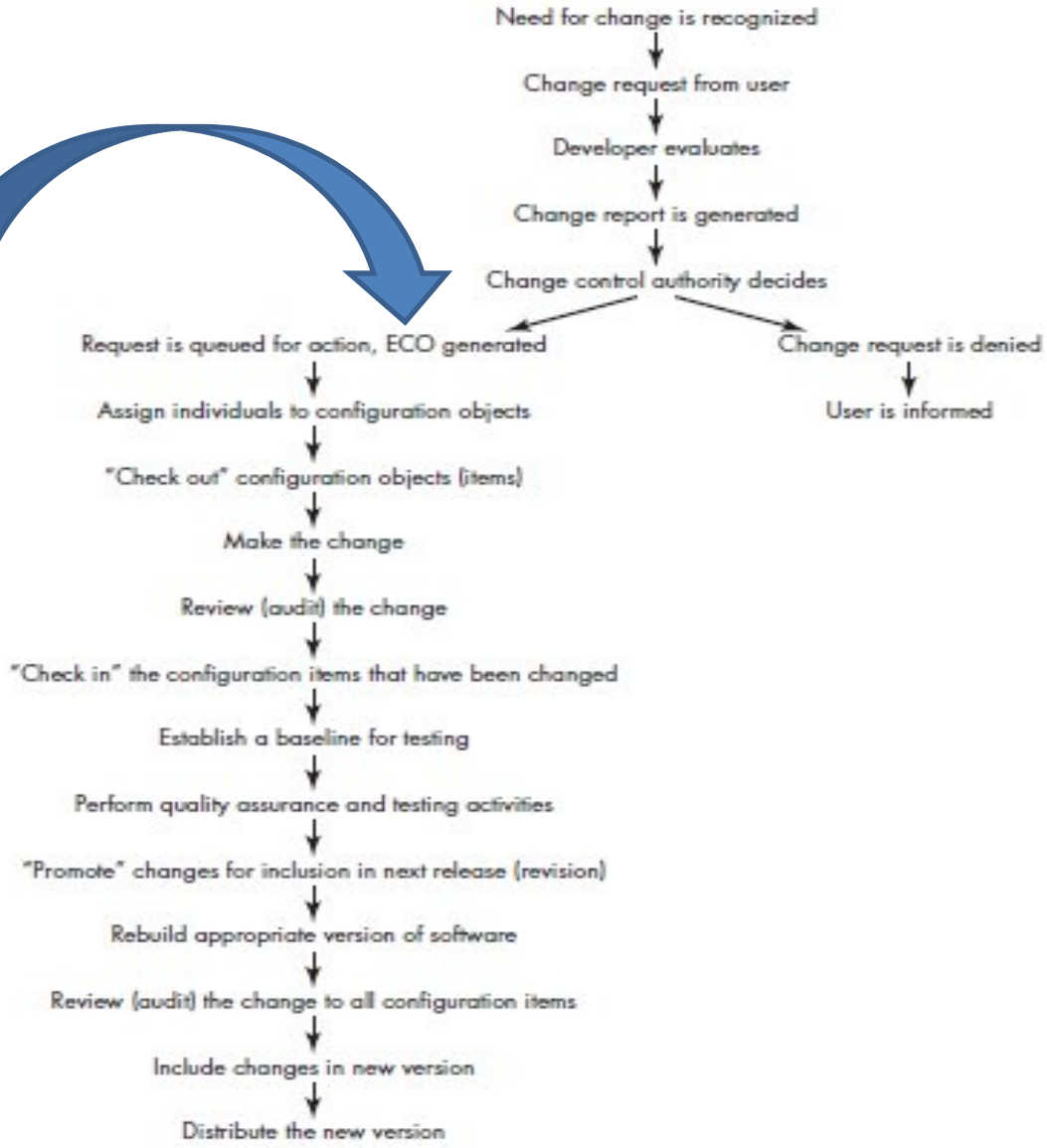
Change Control

- The results of the evaluation are presented as a *change report*,
  - which is used by a *change control authority* (CCA)
  - a person or group that makes a final decision on the status and priority of the change.



Change Control

- An *engineering change order* (ECO) is generated for each approved change.
- The ECO describes
  - the change to be made,
  - the constraints that must be respected, and
  - the criteria for review and audit.

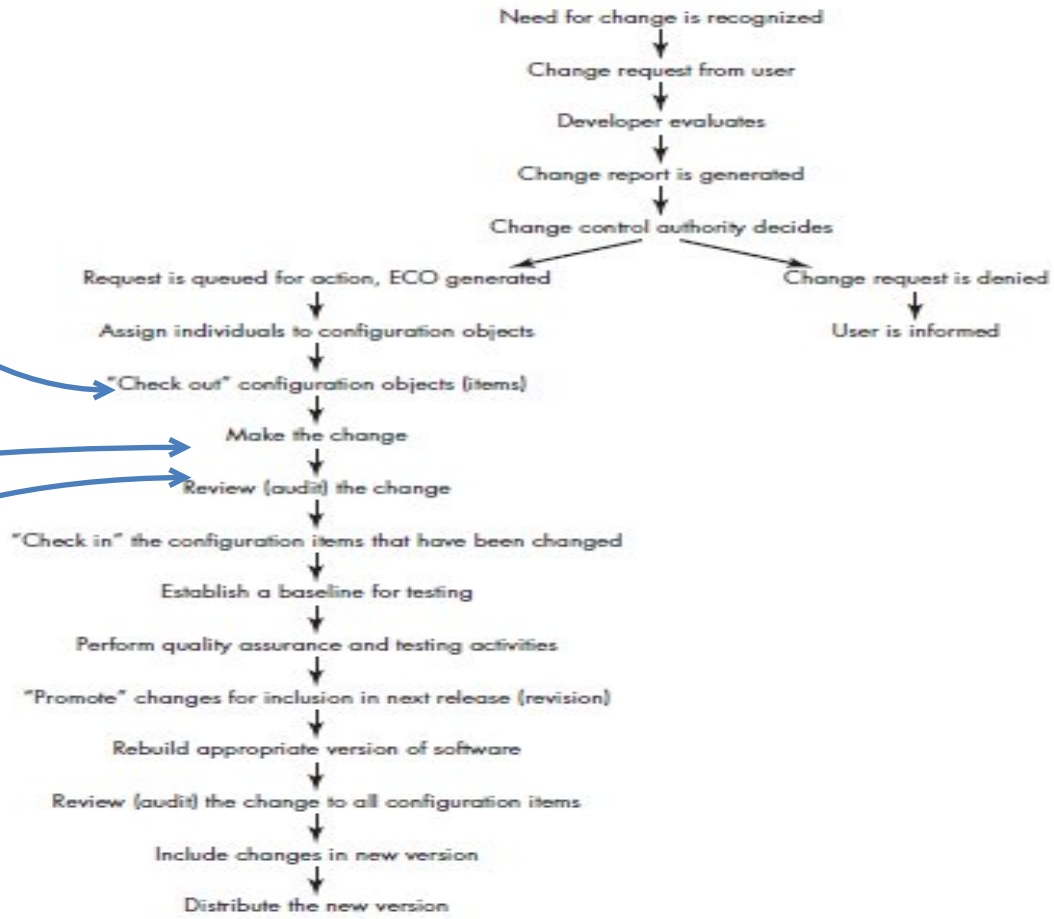


Change Control

- The object(s) to be changed
  - can be placed in a directory that is controlled solely by
  - the software engineer making the change.
- **A version control system updates the original file once the change has been made.**

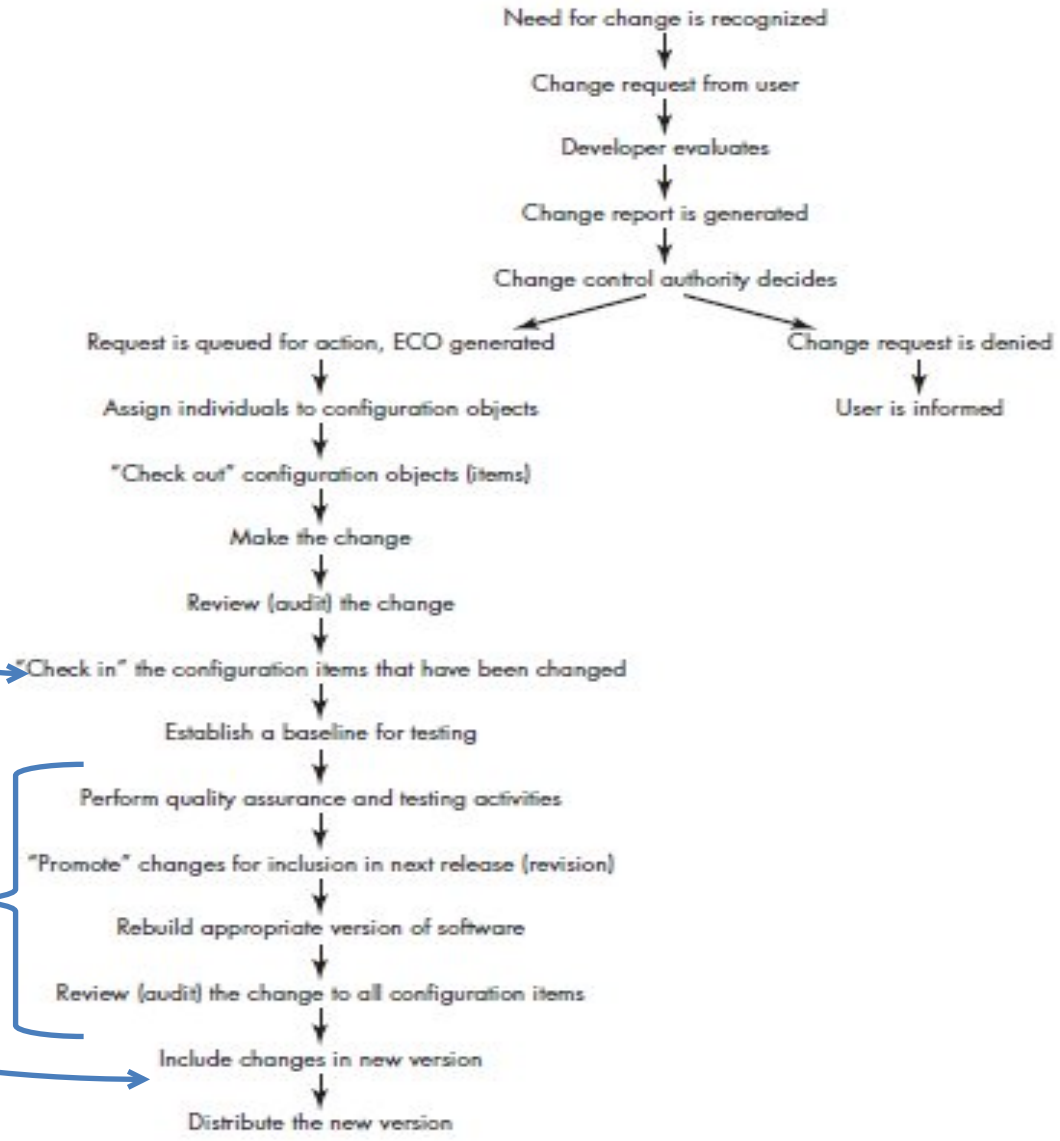
Change Control

- The object(s) to be changed can be "checked out" of the project database (repository),
- The change is made, and
- Review the Change



Change Control

- The object(s) is (are) then "checked in" to the database and
- Appropriate SQA activities are applied
- Appropriate version control mechanisms are used to
- Create the next version of the software.



How can a software team ensure that the change has been properly implemented?

The answer is two fold:
(1) technical reviews and
(2) the software configuration audit.

Configuration Audit

- A *software configuration audit*
  - complements the technical review
  - by assessing a configuration object for characteristics
    - that are generally not considered during review.

Configuration Audit

The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?

2. Has a technical review been conducted to assess technical correctness?

3. Has the software process been followed and have software engineering standards been properly applied?

5. Have SCM procedures for noting the change, recording it, and reporting it been followed?

Configuration status reporting

- *Also called status accounting*
- Is an SCM task that answers the following questions:
  - (1) What happened?
  - (2) Who did it?
  - (3) When did it happen?
  - (4) What else will be affected?

1) Each time an SCI is assigned new or updated identification, a CSR entry is made.
2) Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made.
3) Each time a configuration audit is conducted, the results are reported as part of the CSR task.

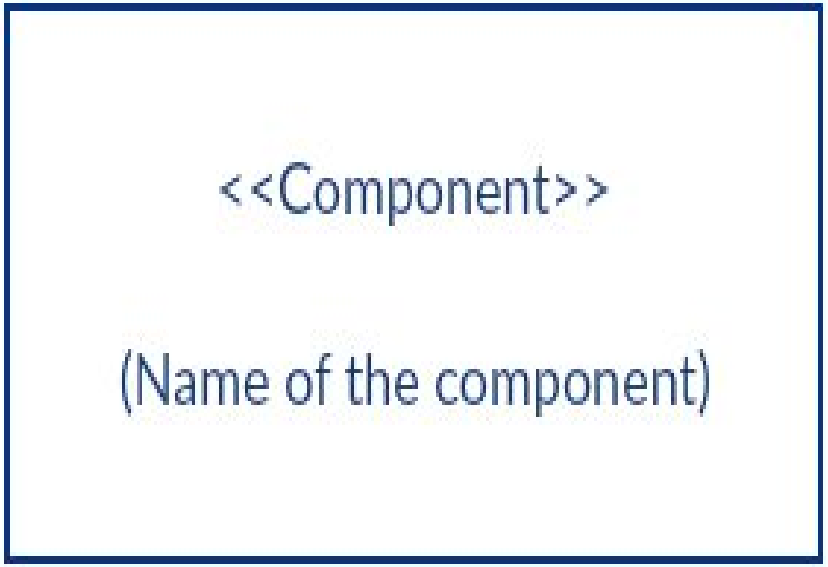
Component Diagrams

Component Diagram Notations

Component

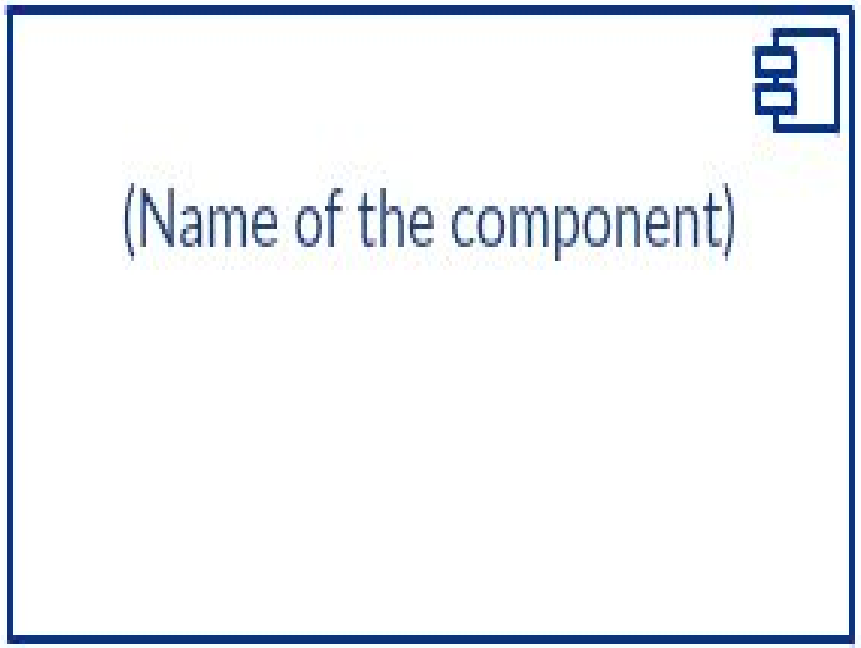
There are three ways the component symbol can be used.

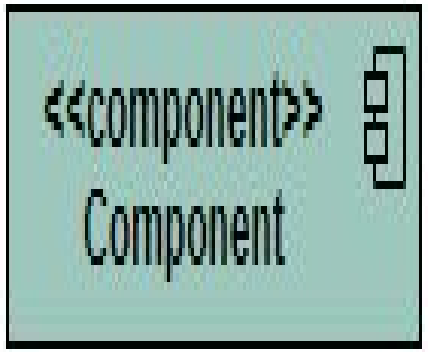
1) Rectangle with the component stereotype (the text <<component>>).
The component stereotype is usually used above the component name to avoid confusing the shape with a class icon.



Component

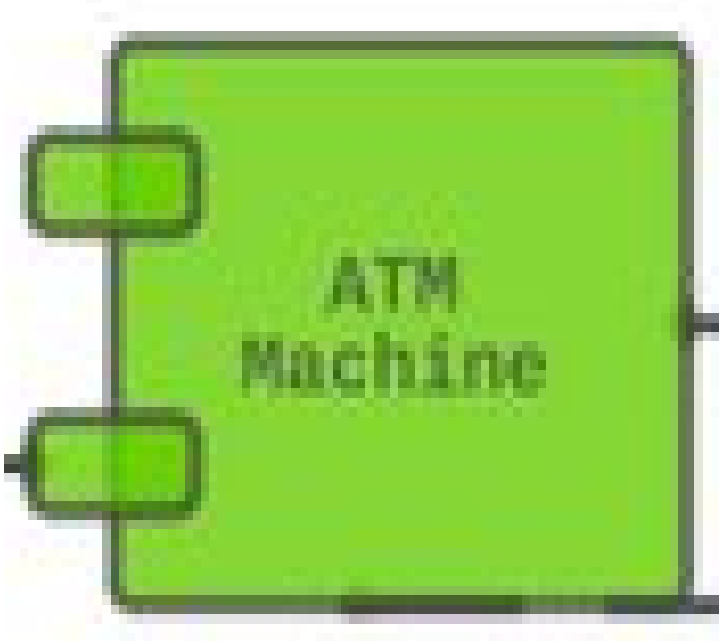
2) Rectangle with the component icon in the top right corner and the name of the component.





Component

3) The component stereotype icon is a rectangle with two smaller rectangles protruding on its left side.



Component Interfaces

- **Provide Interface**

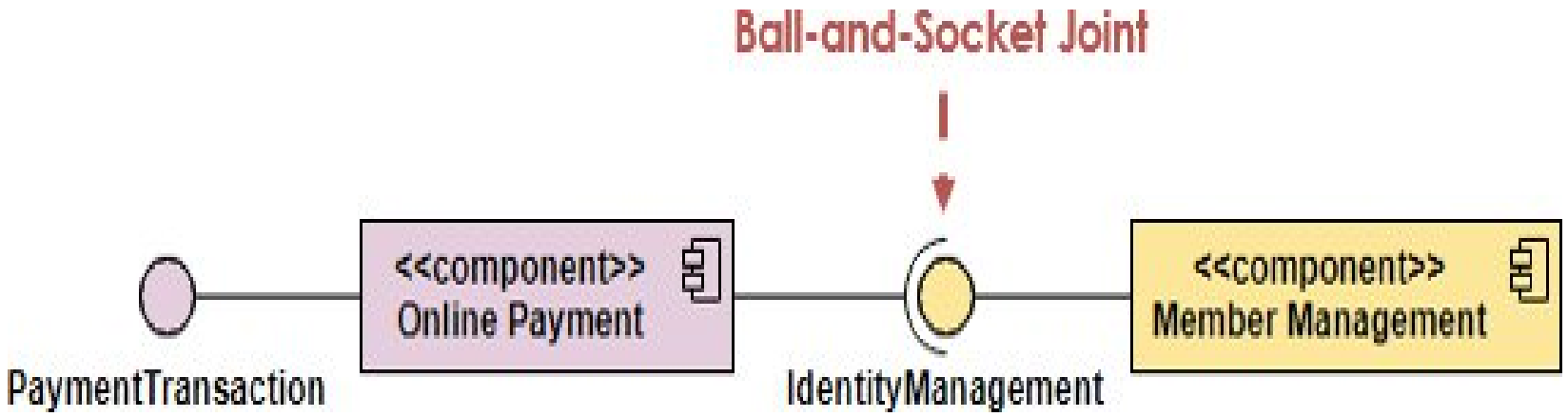
Provided interfaces define "a set of public attributes and operations that must be provided by the classes that implement a given interface".

- **Required Interface**

Required interfaces define "a set of public attributes and operations that are required by the classes that depend upon a given interface".

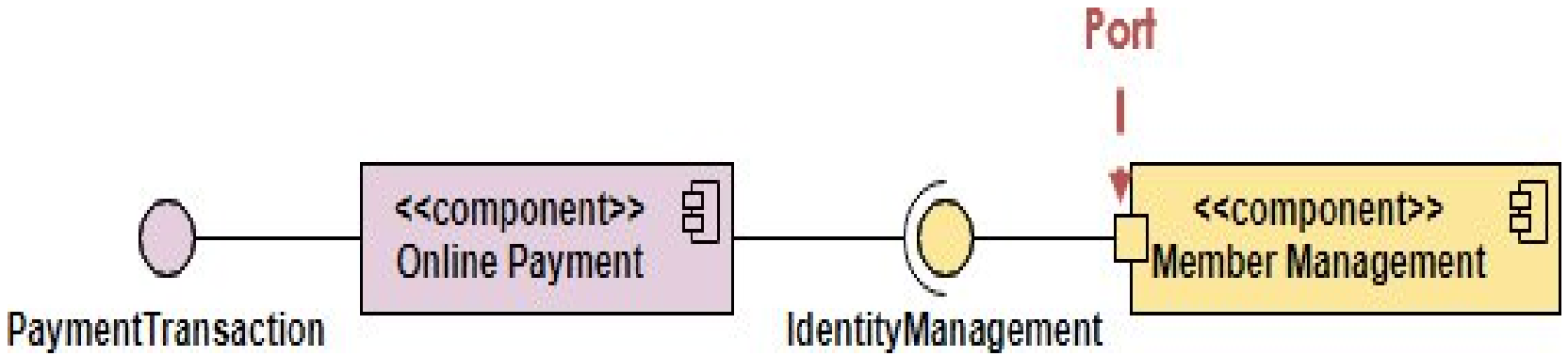
Component Assemblies

- Components can be "wired" together using to form subsystems, with the use of a ball-and-socket joint.
- Component Diagram ball and socket joint



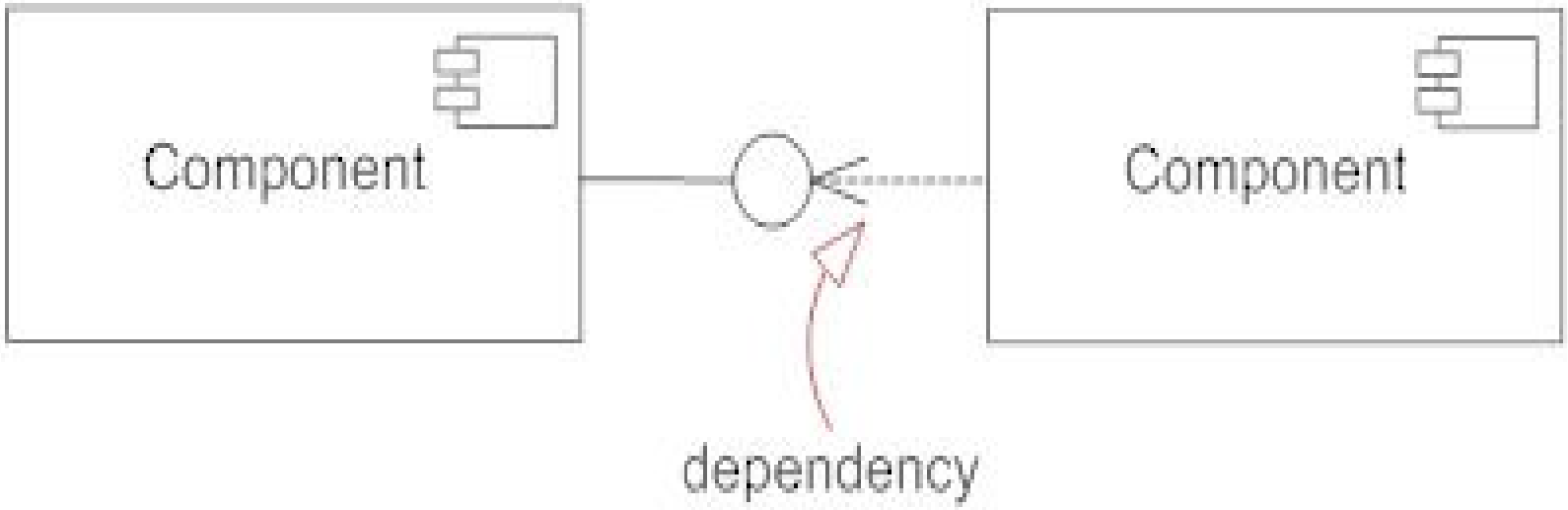
Port

- A port (definition) indicates that the component itself does not provide the required interfaces (e.g., required or provided).
- Instead, the component delegates the interface(s) to an internal class.

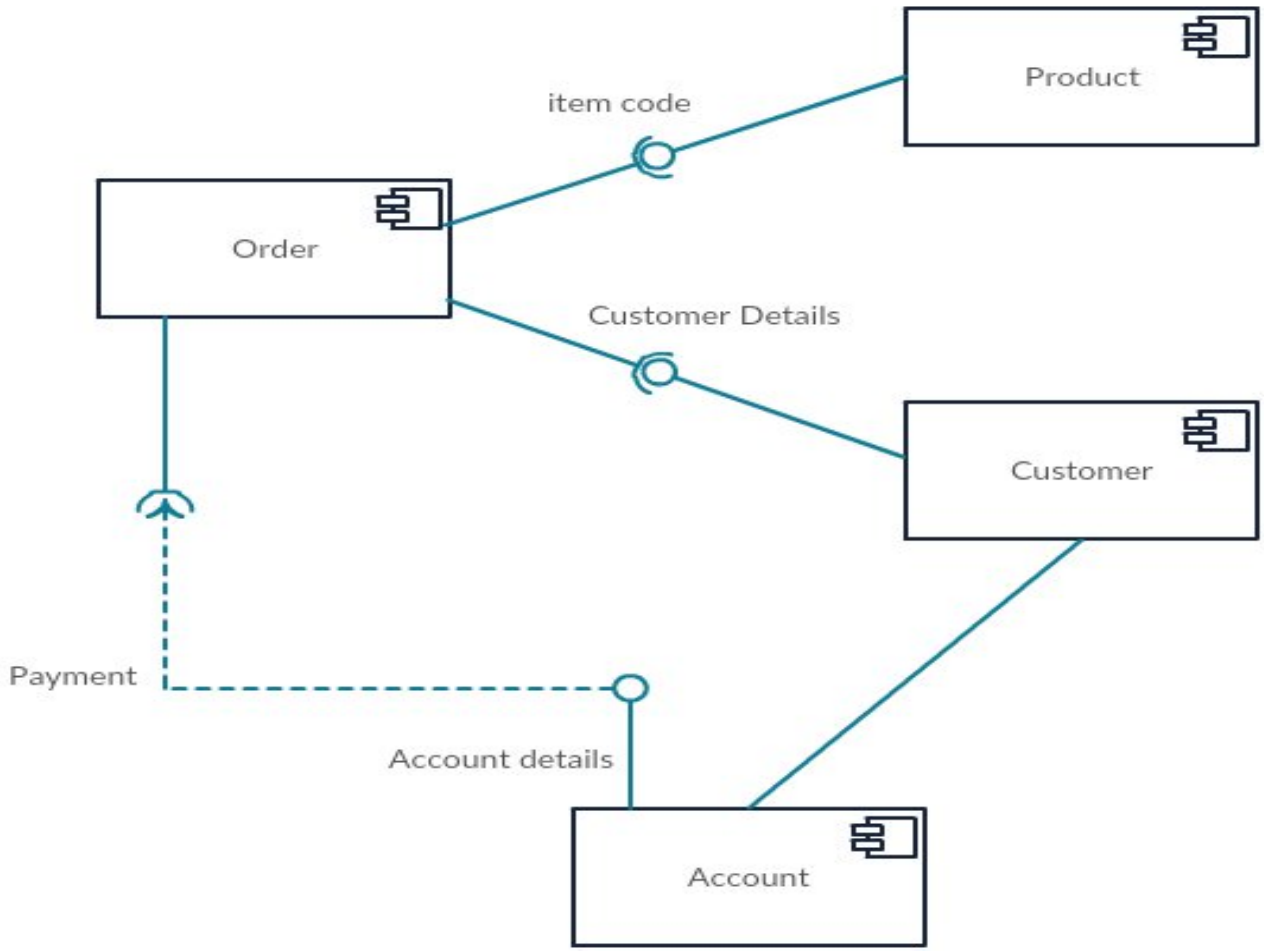


Dependencies

- Draw dependencies among components using dashed arrows.



COMPONENT DIAGRAM FOR ONLINE SHOPPING SYSTEM



When to Draw Component Diagram?

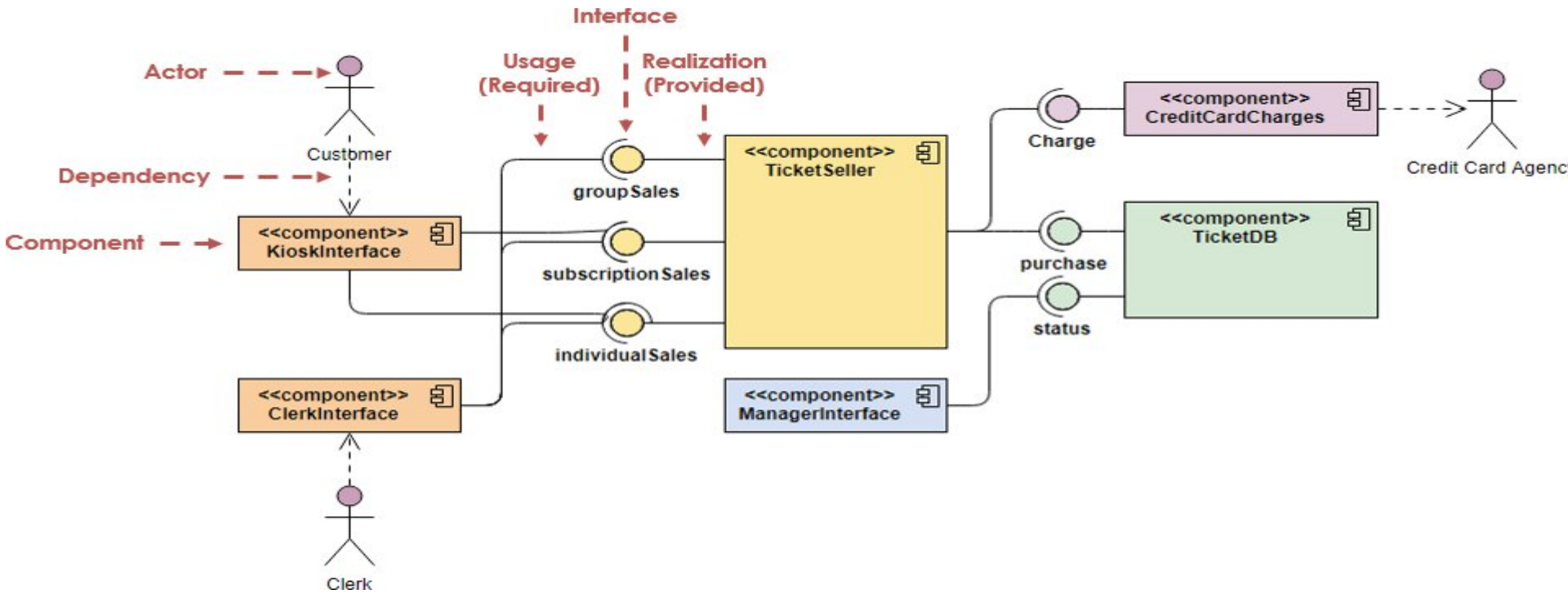
- Use component diagrams when you are dividing your system into components and want to show their interrelationships through interfaces.
- The breakdown of components into a lower-level structure.

How to Draw a Component Diagram?

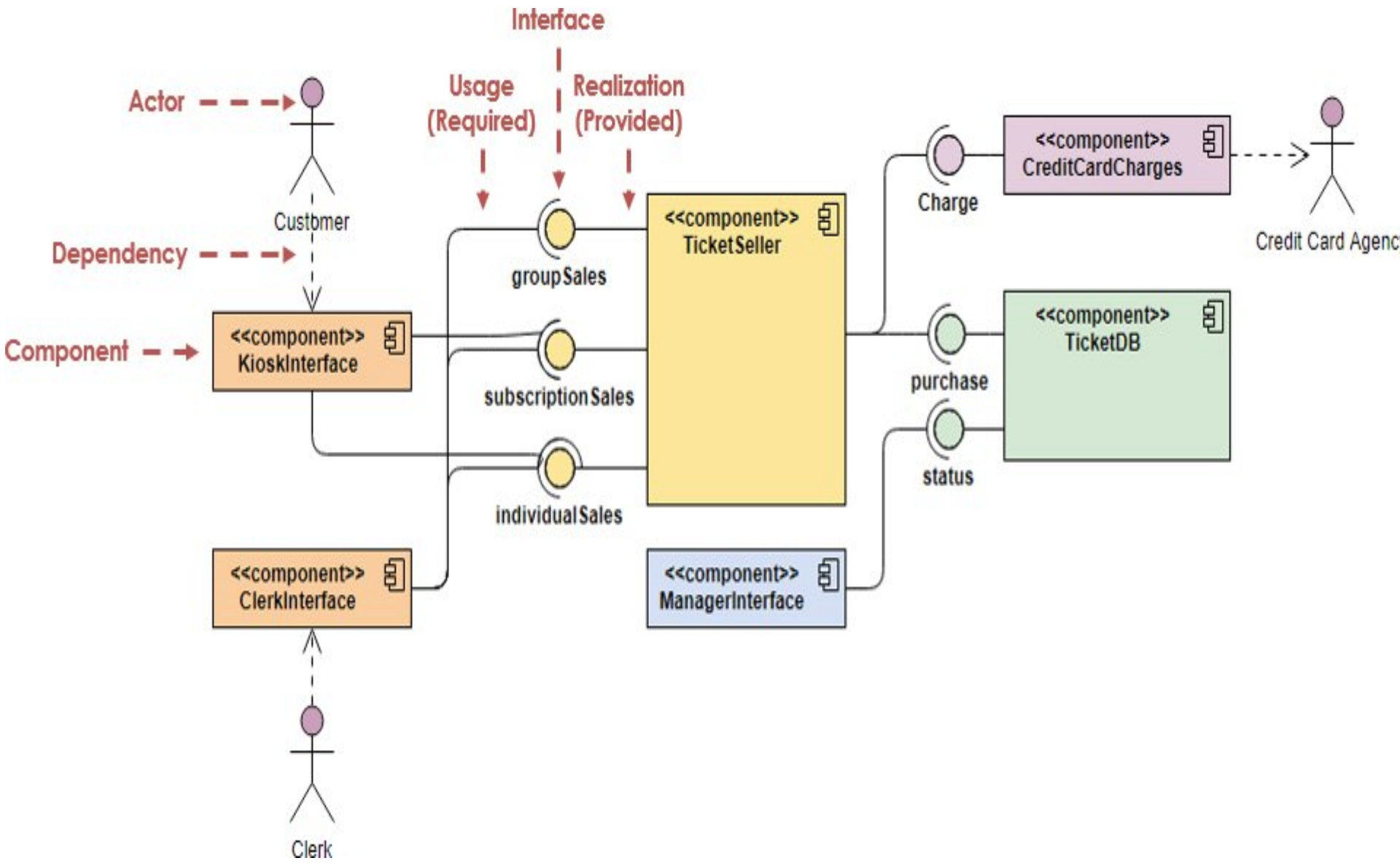
- Decide on the purpose of the diagram
- Add components to the diagram, grouping them within other components if appropriate
- Add other elements to the diagram, such as classes, objects and interface
- Add the dependencies between the elements of the diagram

Ticket Selling System Component Diagram

- Kiosk component
- Clerk component
- Ticket seller component
  - that sequentializes requests from both kiosk and clerks.
- A component that processes credit card charges
- Ticket Database component

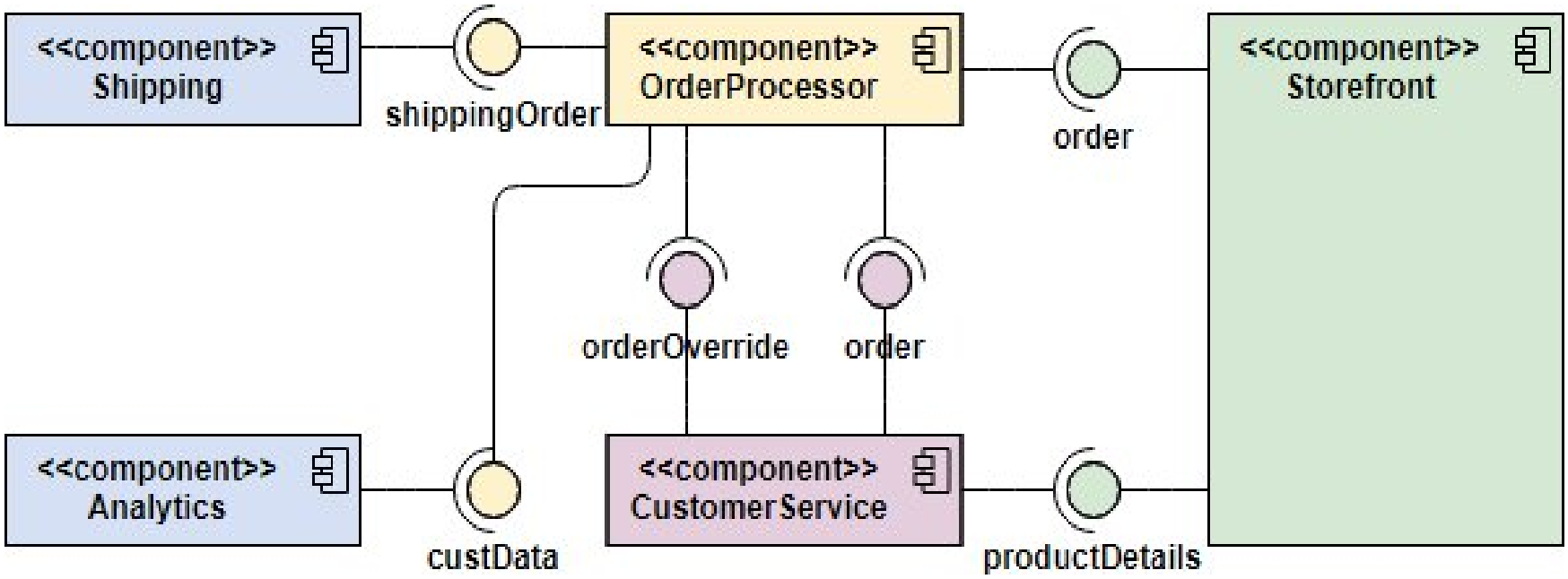


Ticket Selling System Component Diagram



Order Processing System Component diagram example

- The Figure below is a much larger view of what is involved in a online store.
- By using a component diagram we see the system as a group of nearly independent component or subsystems that interact with each other in a specifically defined way.

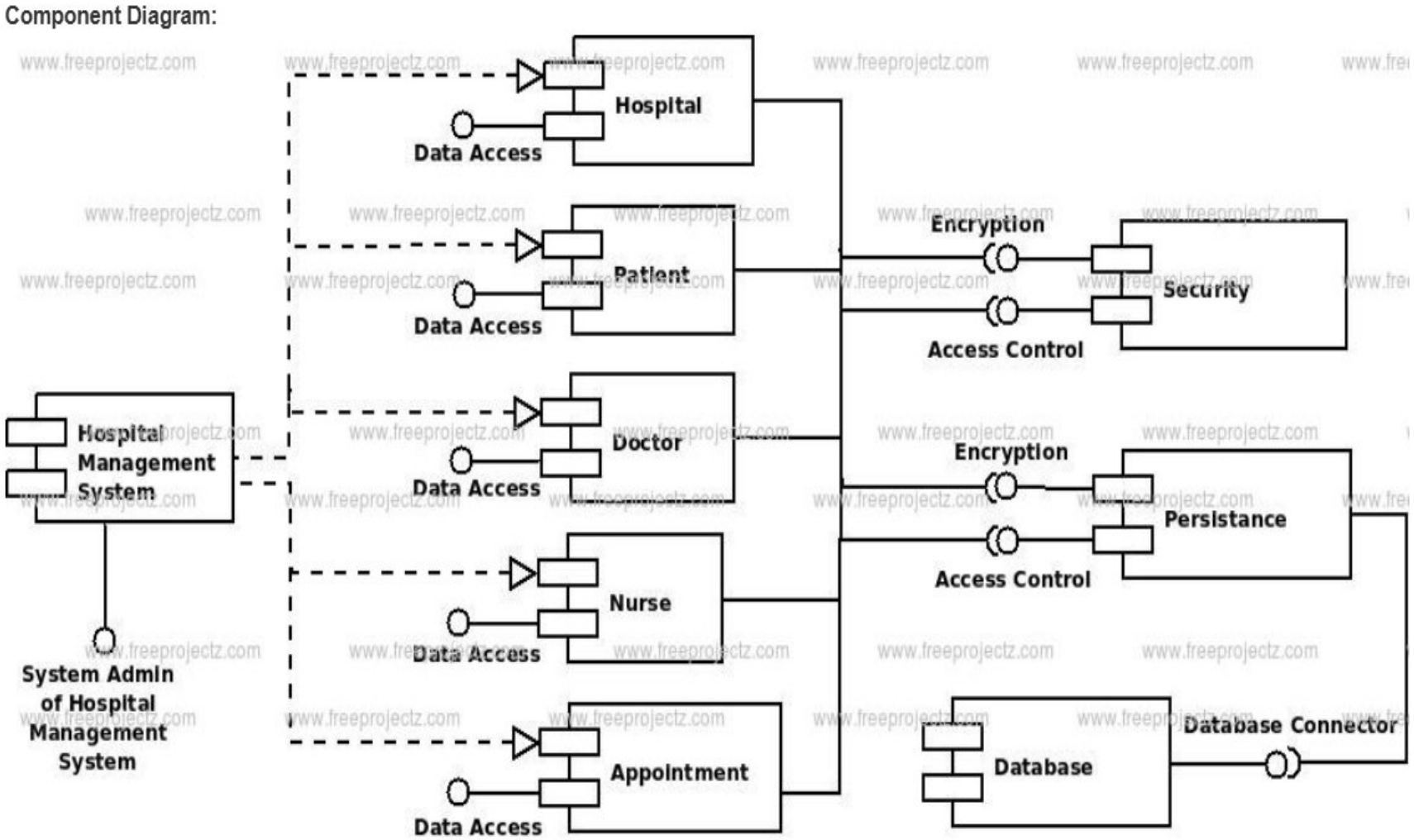


Component Diagram for Hospital Management System

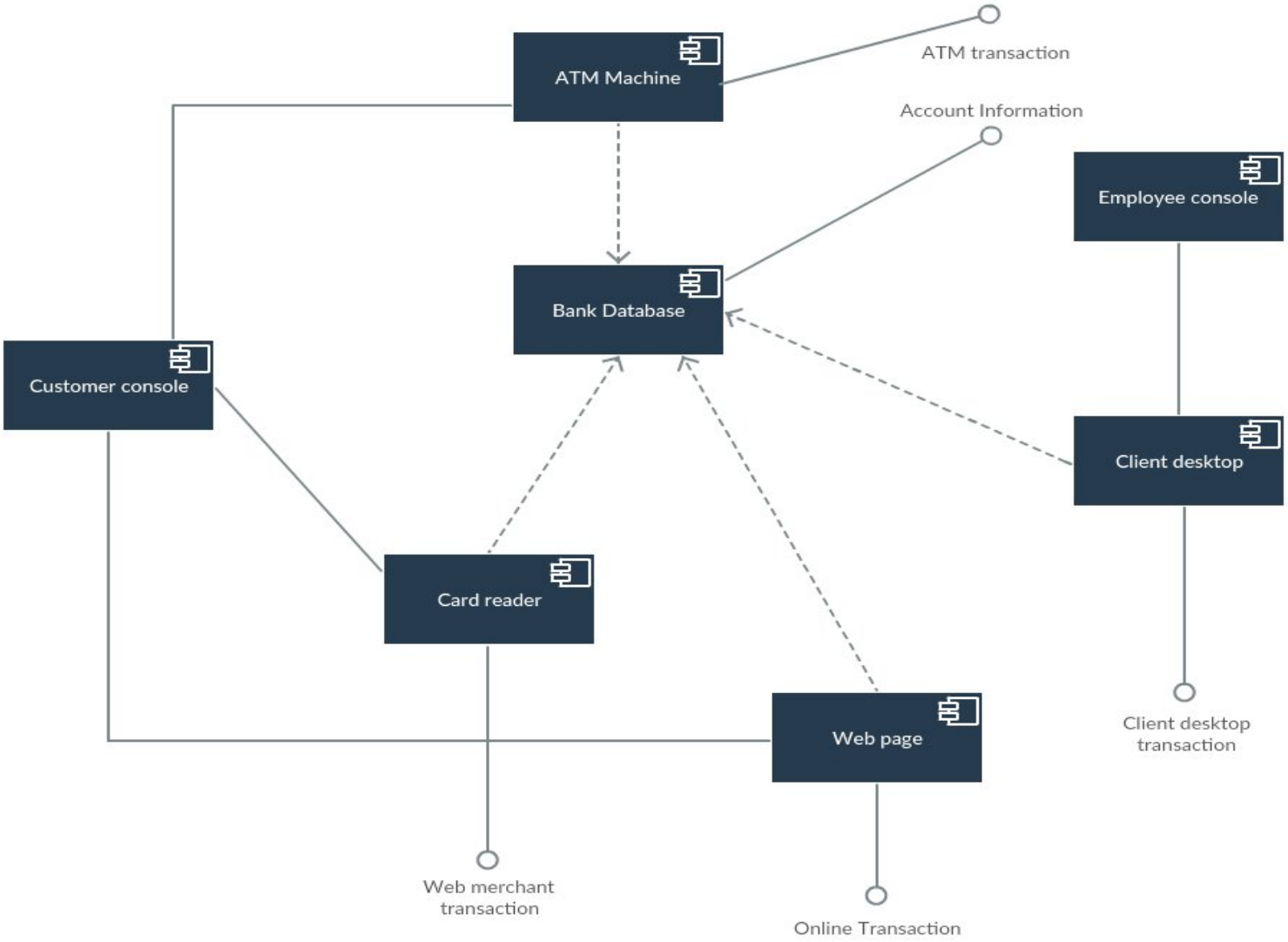
Components -

- Hospital
- Patient
- Doctor
- Nurse
- Appointment
- Database
- Security

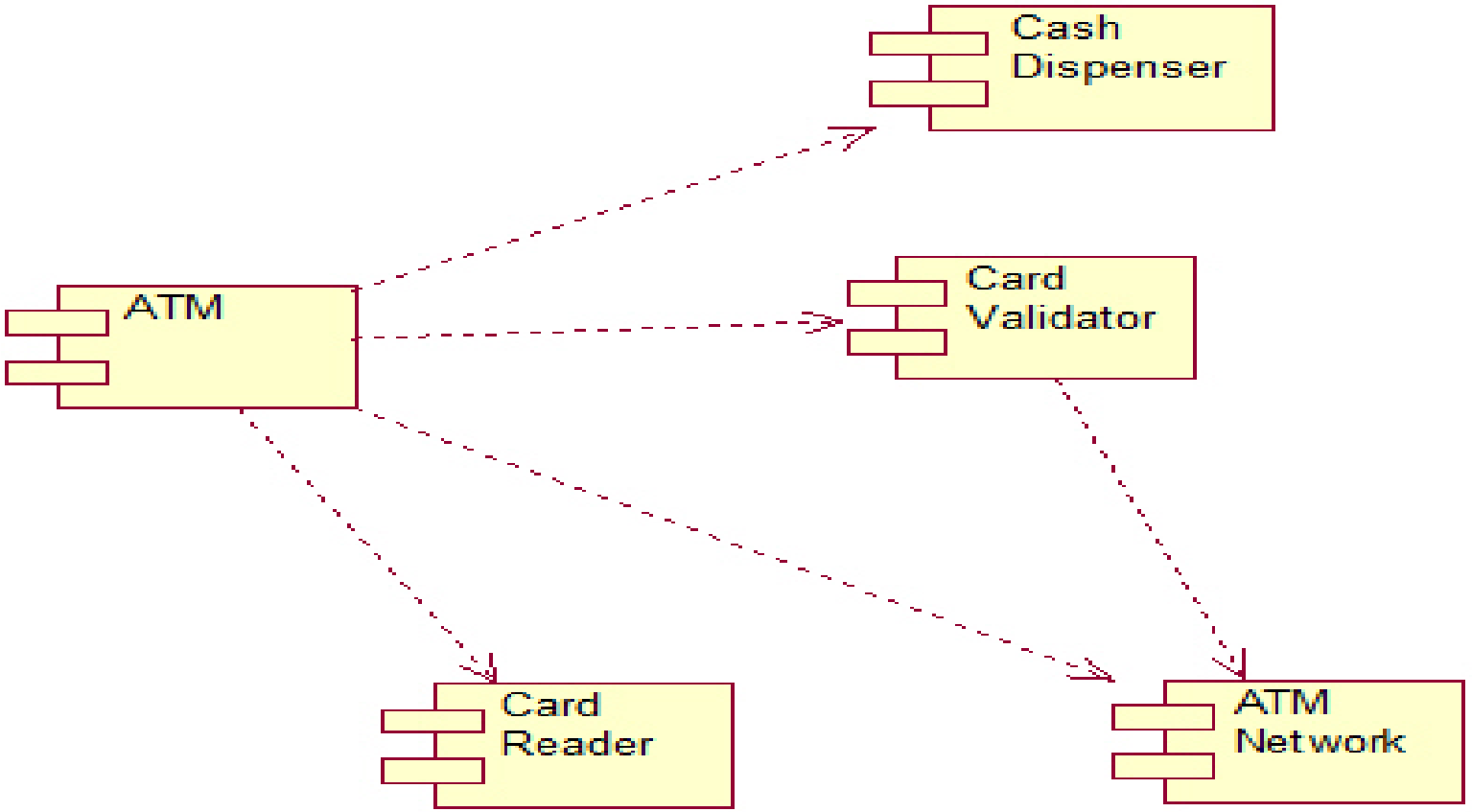
Component Diagram for Hospital Management System



COMPONENT DIAGRAM for ATM

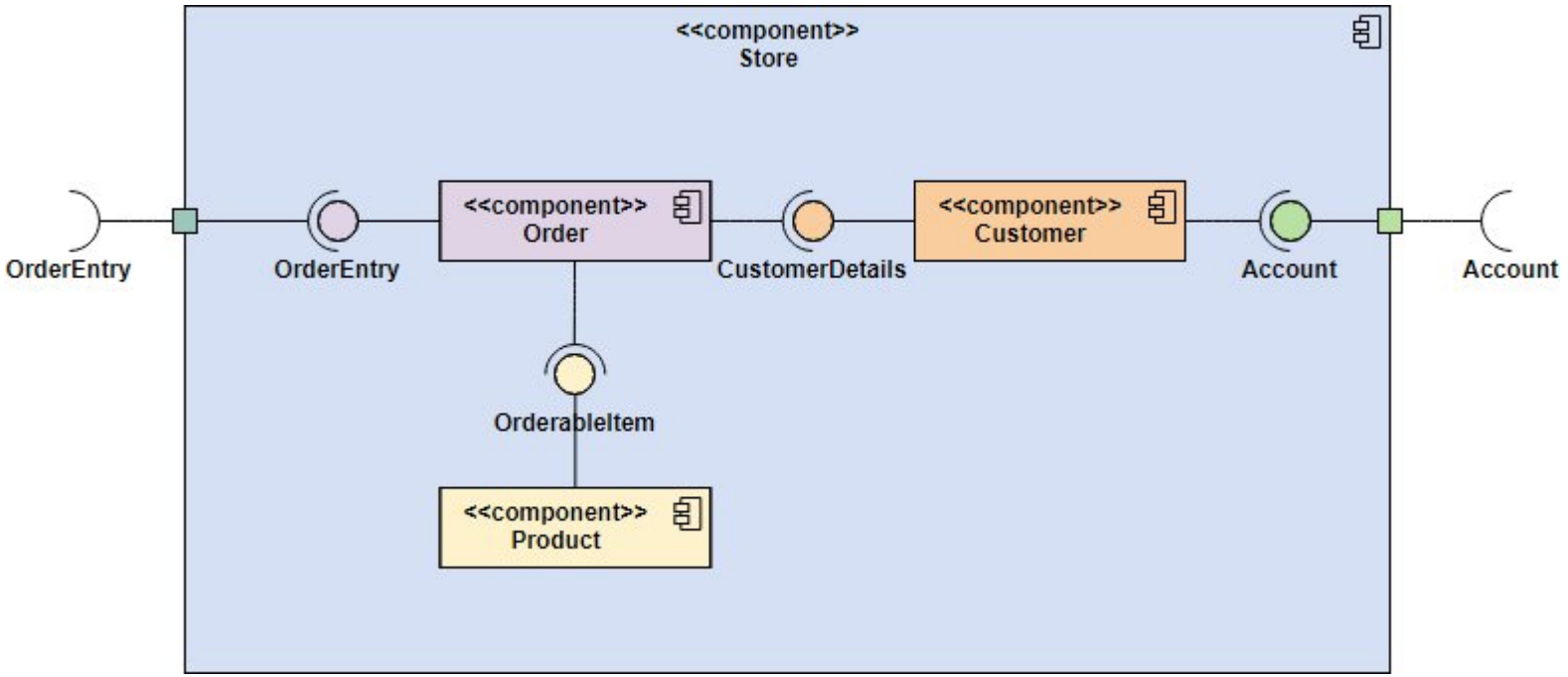


Component diagram for ATM Application



Component Example - Store Component - nested component structure

- To show a nested component structure, you merely draw the component larger than normal and place the inner parts inside the name compartment of the encompassing component.
- The Figure below show's the Store's component nested structure.



Deployment Diagram

When to Draw Deployment Diagram?

When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.

- To model embedded systems
- To model client/server systems
- To model fully distributed systems

How to Draw a Deployment Diagram?

- Firstly, identify the nodes that represent your system's client and server processors and then highlight those devices that are relevant to the behavior of your system.
  - For example, you'll want to model
    - special devices, such as credit card readers, badge readers, and
    - display devices other than monitors,
    - because their placement in the system's hardware topology are likely to be architecturally significant.

How to Draw a Deployment Diagram?

- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram.
- Specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

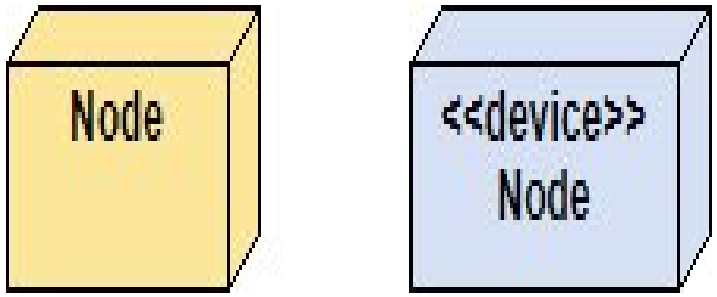
Deployment Diagram Notations

Component

- A component is a grouping of classes that work together closely.
- Components can be classified by their type. Some components exist only at compile time, some exist only at link time, some exist only at run time; and some exist at more than one time.

Node

- A node is a run-time physical object that represents a computational resource, generally having memory and processing capability.
- **Node:** A hardware or software object, shown by a three-dimensional box.

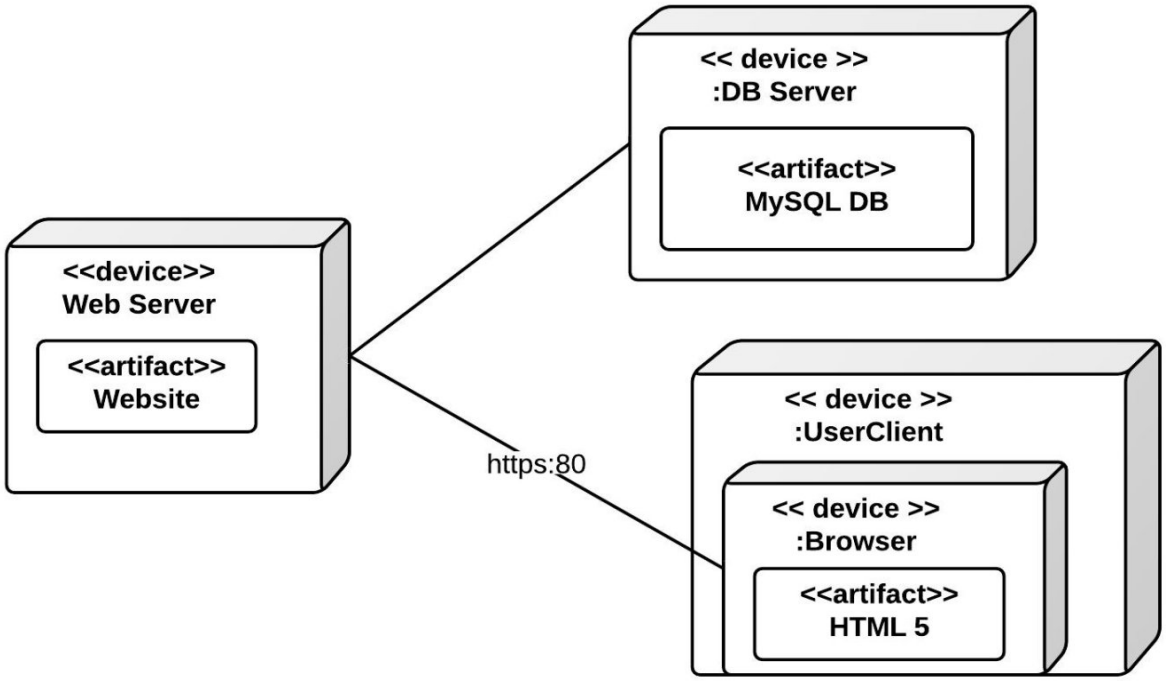


Node as container

- You may model the component instances that run or live on a node by drawing them within the node.
- A node that contains another node inside of it.

Connection Line

- You may model which nodes communicate with one another using the Connection relationship line.

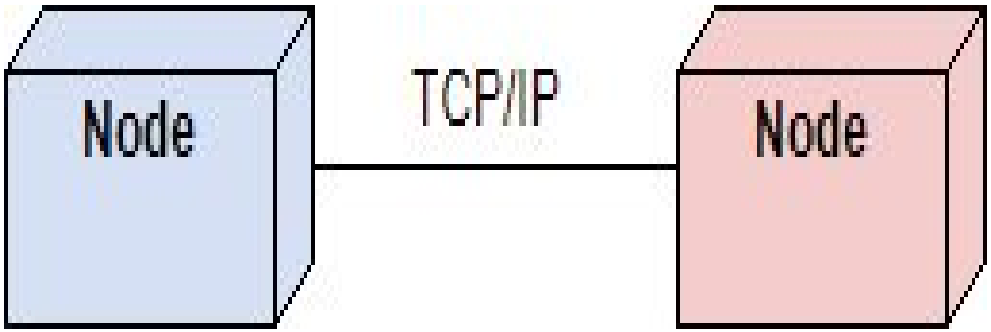


Dependency

- A dependency indicates that one model element (source) depends on another model element (target), such that a change to the target element may require a change to the source element in the dependency.
- In a deployment diagram, you can use the dependency relationship to show the capability of a node type to support a component type.

Connection

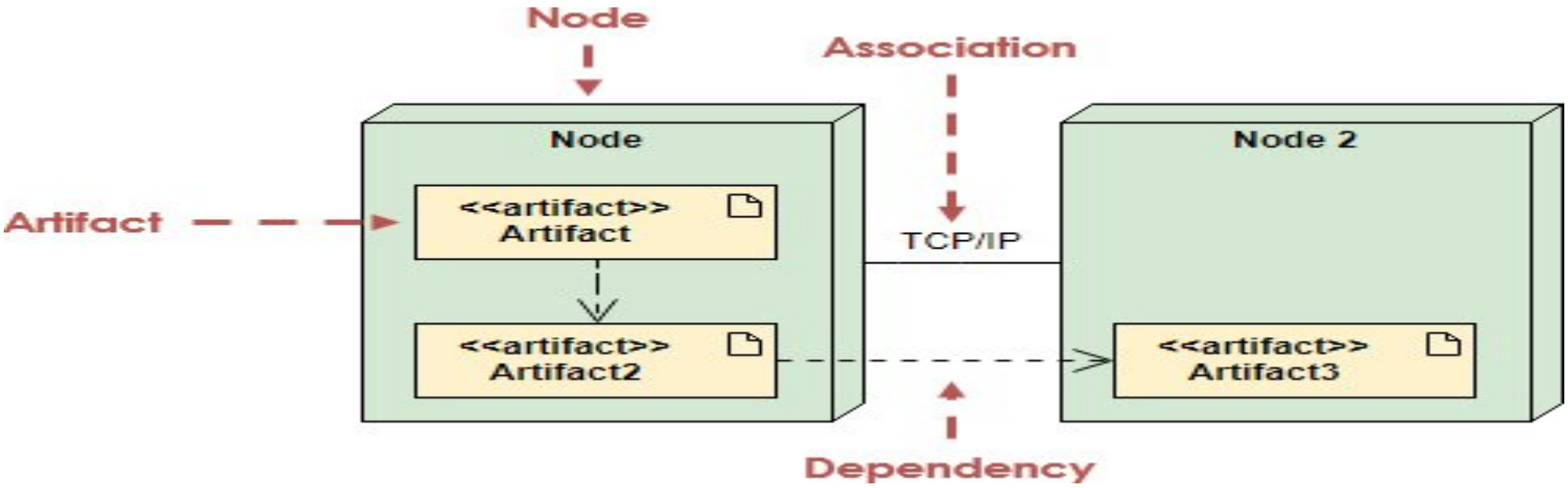
- A connection depicts the communication path used by the hardware to communicate usually indicates the method i.e. TCP/IP.



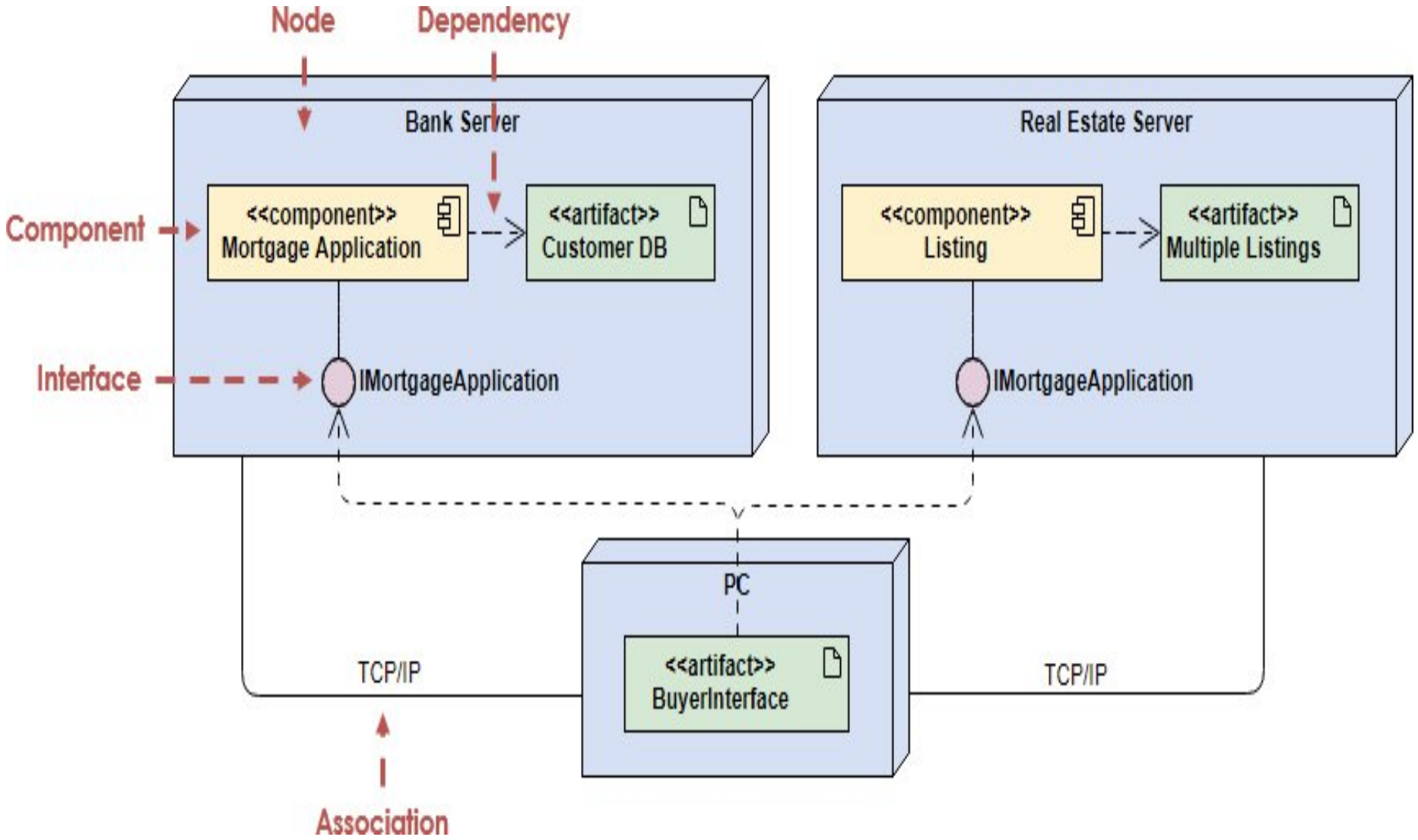
Artifact

- Artifacts represent concrete elements in the physical world that are the result of a development process.

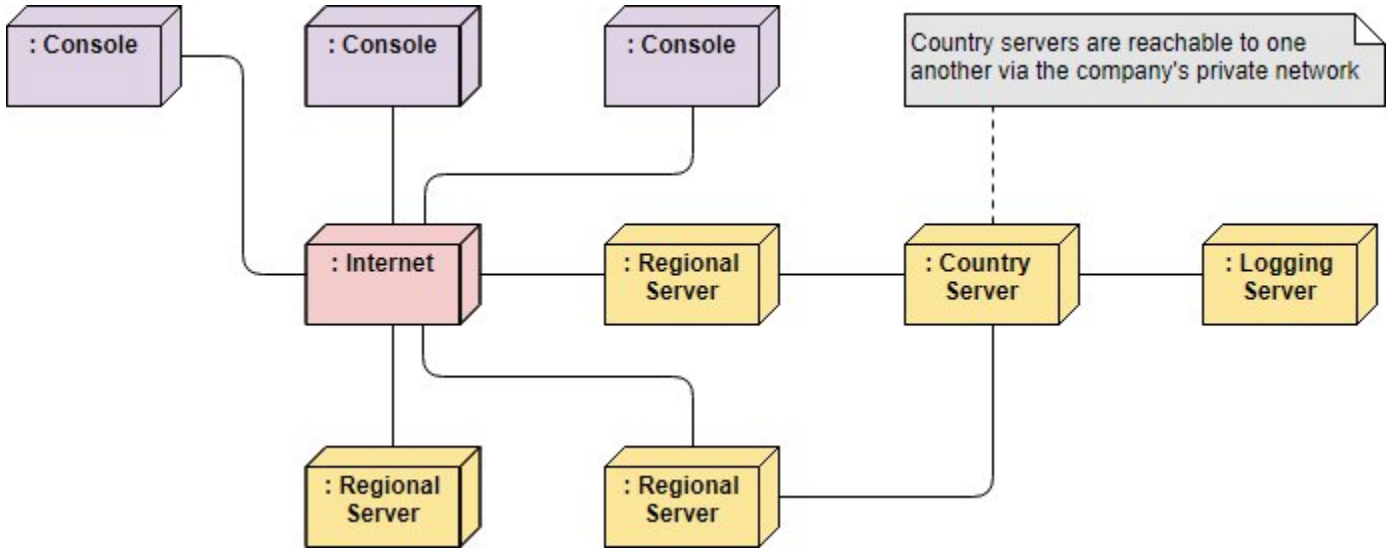
- Examples of artifacts are executable files, libraries, archives, database schemas, configuration files, etc.



- Deployment diagram shows the relationships among software and hardware components involved in real estate transactions.



Deployment Diagram Example - Modeling a Distributed System



Mapping Models to Code

Mapping Models to Code

- A **transformation** aims at improving one aspect of the model (e.g., its modularity) while preserving all of its other properties (e.g., its functionality).

- Transformation Activities:

1. *Optimization*

- This activity addresses the performance requirements of the system model.

Mapping Models to Code

- **Examples**
- Reducing multiplicities
- adding redundant associations for efficiency,
- adding derived attributes to improve the access time to objects.

2. Realizing associations

- map associations to source code constructs,
- **Example**
- references and collections of references.

Mapping Models to Code

3. Mapping contracts to exceptions

- describe the behavior of operations when contracts are broken.

- This includes raising exceptions when violations are detected.

4. Mapping class models to a storage schema

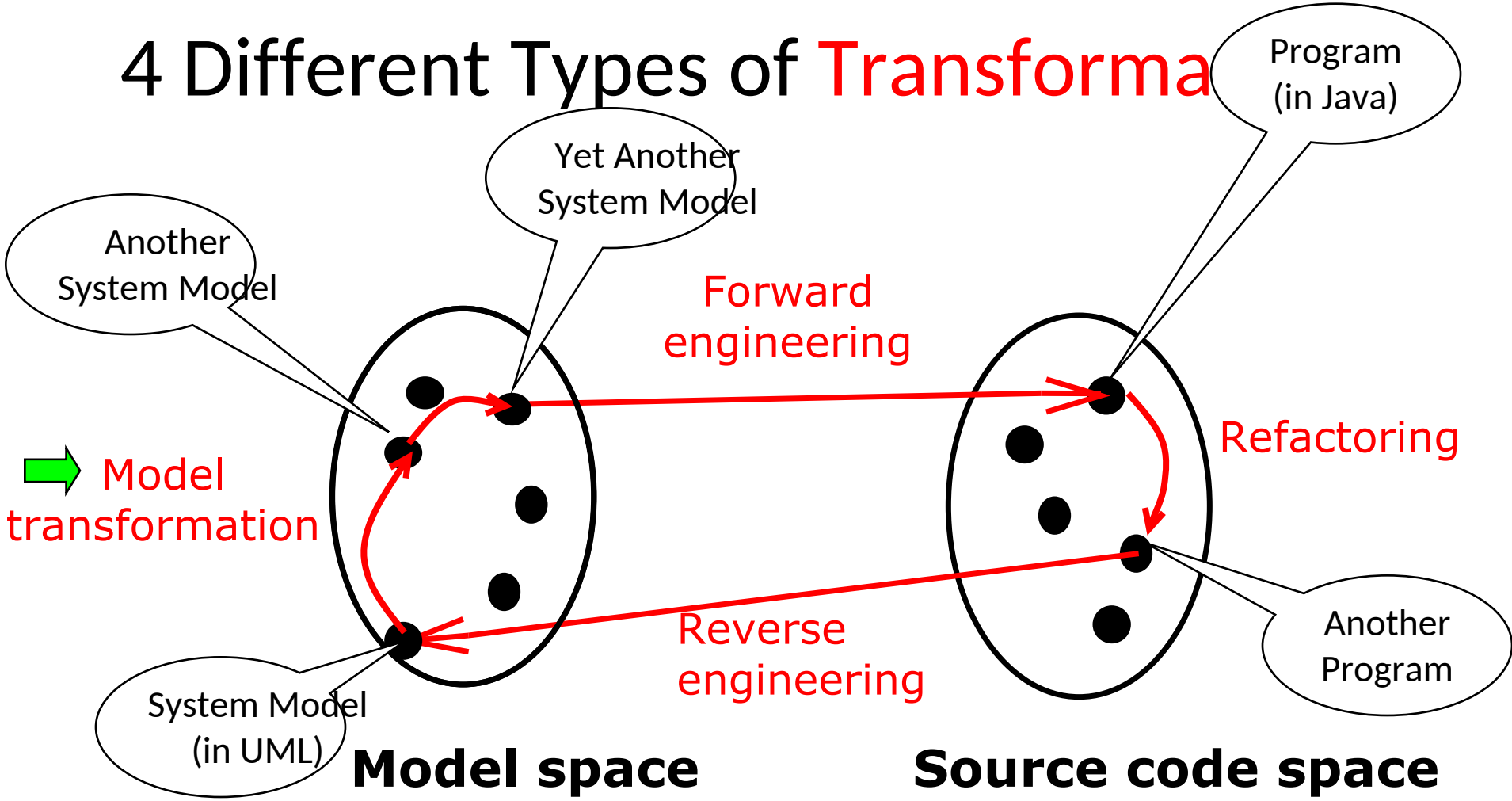
- selected a persistent storage strategy, such as a database management system, a set of flat files, or a combination of both.

- map the class model to a storage schema, such as a relational database schema.

Transformations

- Let us get a handle on these problems
- To do this we distinguish two kinds of spaces
  - the model space and the source code space
- and 4 different types of transformations
  - Model transformation
  - Forward engineering
  - Reverse engineering
  - Refactoring.

4 Different Types of Transforma



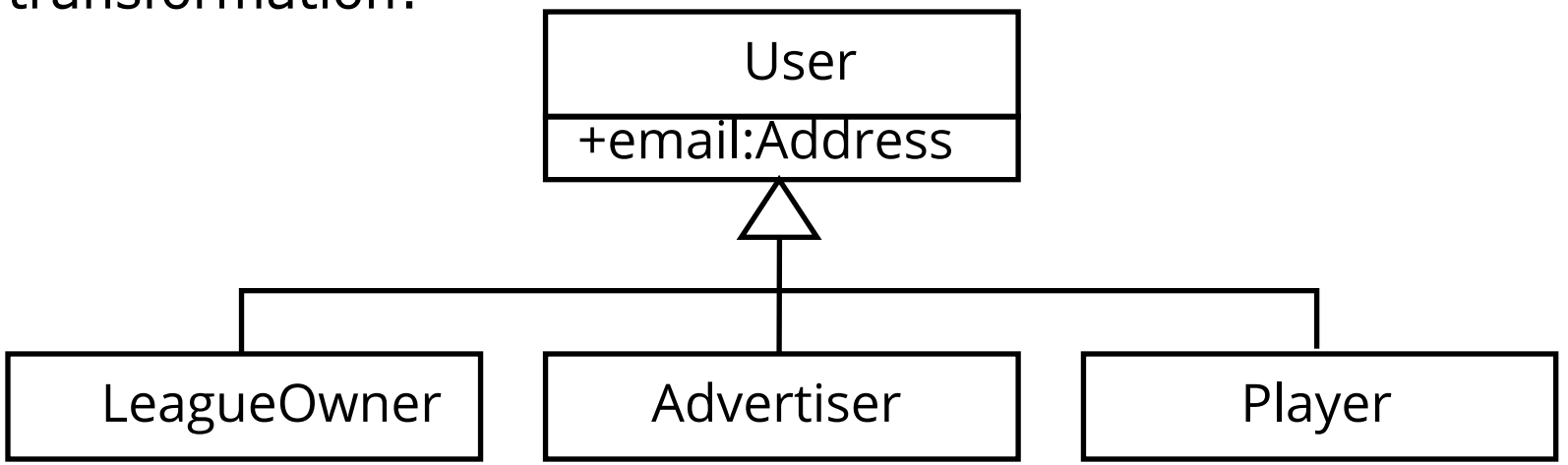
Model Transformation

- A **model transformation** is applied to an object model and results in another object model.
- purpose of object model transformation is to bringing it into closer compliance with all requirements.
- A transformation may add, remove, or rename classes, operations, associations, or attributes.

Model Transformation Example

Object design model before transformation:

Object design model after transformation:



Forward Engineering

- **Forward engineering** is applied to a set of model elements and results in a set of corresponding source code statements, such as a class declaration, a Java expression, or a database schema.

- Purpose of forward engineering is **to maintain a strong correspondence between the object design model and the code**, and to **reduce implementation errors.**

Forward Engineering Example

Object design model before transformation:

Source code after transformation:

```
public class User {
        private String email;
        public String getEmail() {
                return email;
        }
        public void setEmail(String value){
                email = value;
        }
        public void notify(String msg) {
                // ....
        }
}
```

```
public class LeagueOwner extends User {
        private int maxNumLeagues;
        public int getMaxNumLeagues() {
                return maxNumLeagues;
        }
        public void setMaxNumLeagues
                        (int value) {
                maxNumLeagues = value;
        }
}
```

Refactoring

- A **refactoring** is a transformation of the source code that improves its readability or modifiability without changing the behavior of the system.
- Refactoring aims at improving the design of a working system.
- the refactoring is done in small incremental steps that are interleaved with tests.

Refactoring

There are Three Methods for Refactoring:

1) Pull Up Field
2) Pull Up Constructor Body
3) Pull Up Method

Refactoring Example: Pull Up Field

```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

```
public class User {
    private String email;
}
public class Player extends User {
    //...
}

public class LeagueOwner extends User {
    //...
}
public class Advertiser extends User {
    //...
}
```

Mapping Models to Code

- The first one, Pull Up Field, moves the email field from the subclasses to the superclass User.

Pull Up Field relocates the `email` field using the following steps (Figure 10-3):

1. Inspect `Player`, `LeagueOwner`, and `Advertiser` to ensure that the `email` field is equivalent. Rename equivalent fields to `email` if necessary.
2. Create public class `User`.
3. Set parent of `Player`, `LeagueOwner`, and `Advertiser` to `User`.
4. Add a protected field `email` to class `User`.
5. Remove fields `email` from `Player`, `LeagueOwner`, and `Advertiser`.
6. Compile and test.

Refactoring Example: Pull Up Constructor Body

```
public class User {
    private String email;
}


public class Player extends User {
    public Player(String email) {
        this.email = email;
    }
}
public class LeagueOwner extends User{
    public LeagueOwner(String email) {
        this.email = email;
    }
}
public class Advertiser extendsUser{
    public Advertiser(String email) {
        this.email = email;
    }
}
```

```
public class User {
    public User(String email) {
        this.email = email;
    }
}


public class Player extends User {
    public Player(String email) {
        super(email);
    }
}
public class LeagueOwner extends User {
    public LeagueOwner(String email) {
        super(email);
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        super(email);
    }
}
```

- **Using ‘this’ keyword to refer current class instance variables**
- **super keyword can also be used to access/invoke the parent class constructor.**

Mapping Models to Code

- The second one, Pull Up Constructor Body, moves the initialization code from the subclasses to the superclass.

Then, we apply the *Pull Up Constructor Body* refactoring to move the initialization code for `email` using the following steps (Figure 10-4):

1. Add the constructor `User(Address email)` to class `User`.
2. Assign the field `email` in the constructor with the value passed in the parameter.
3. Add the call `super(email)` to the `Player` class constructor.
4. Compile and test.
5. Repeat steps 1–4 for the classes `LeagueOwner` and `Advertiser`.

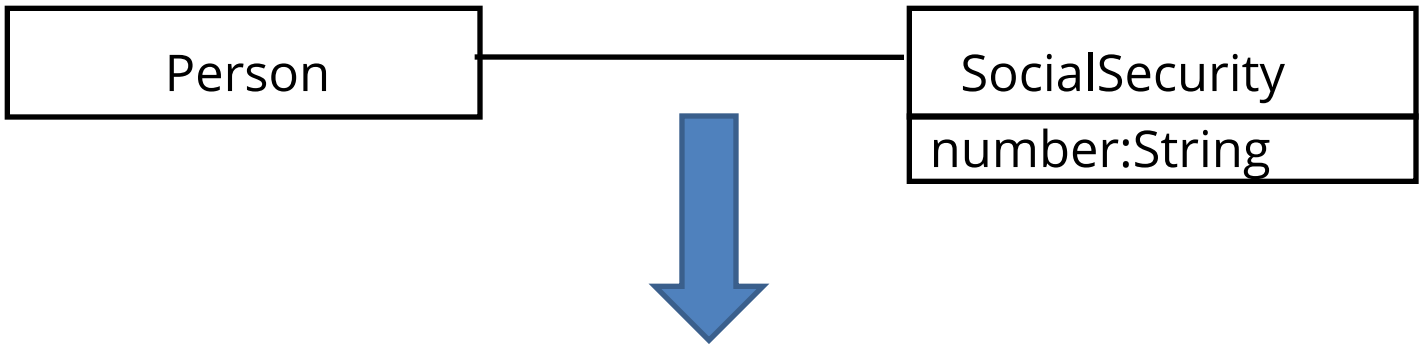
Mapping Models to Code

- The third and final one, Pull Up Method, moves the methods manipulating the email field from the subclasses to the superclass.

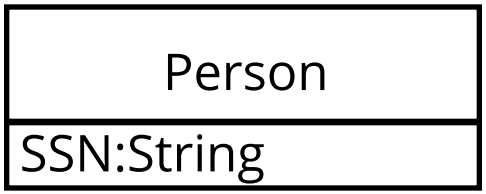
1. Examine the methods of `Player` that use the `email` field. Note that `Player.notify()` uses `email` and that it does not use any fields or operations that are specific to `Player`.
2. Copy the `Player.notify()` method to the `User` class and recompile.
3. Remove the `Player.notify()` method.
4. Compile and test.
5. Repeat for `LeagueOwner` and `Advertiser`.

Collapsing Objects

Object design model before transformation:



Object design model after transformation:



Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

Reverse Engineering

Reverse engineering is applied to a set of source code elements and results in a set of model elements.

- **purpose** - to recreate the model for an existing system, either because the model was lost or never created, or because it became out of sync with the source code.

- Reverse engineering is essentially an inverse transformation of forward engineering.

Mapping Models to Code

- **Transformation Principles**

A transformation aims at improving the design of the system with respect to some criterion.

- To avoid introducing new errors, all transformations should follow certain principles:

Mapping Models to Code

1. ***Each transformation must address a single criteria***.

2. ***Each transformation must be local***. A transformation should change only a few methods or a few classes at once.

3. ***Each transformation must be applied in isolation to other changes.***

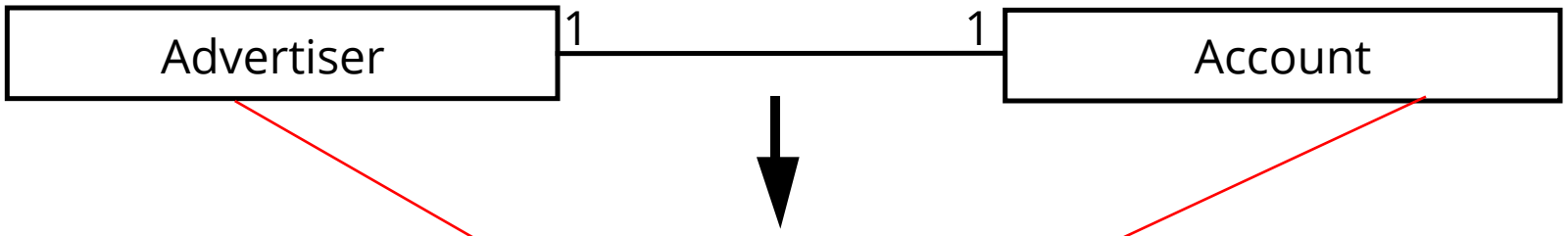
4. Each transformation must be followed by a validation step.

Mapping Associations to Collections

1. Unidirectional one-to-one association
2. Bidirectional one-to-one association
3. Bidirectional one-to-many association
4. Bidirectional many-to-many association

Unidirectional one-to-one association

Object design model before transformation:

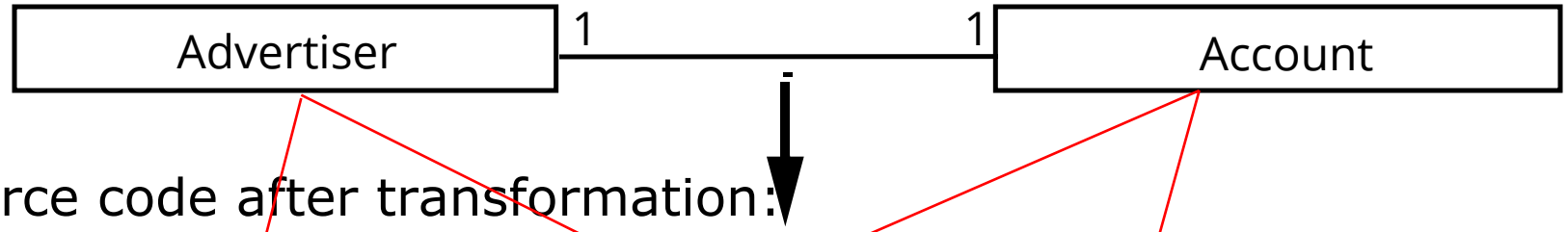


Source code after transformation:

```
public class Advertiser {
        private Account account;
        public Advertiser() {
                account = new Account();
        }
        public Account getAccount() {
                return account;
        }
}
```

Bidirectional one-to-one association

Object design model before transformation:



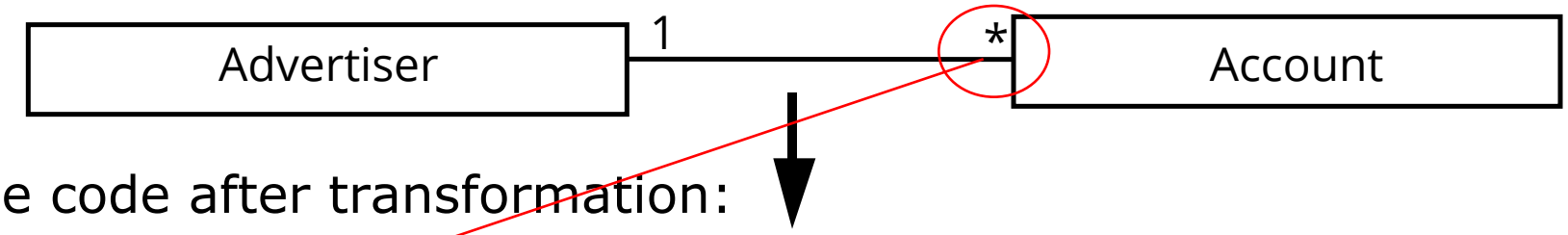
Source code after transformation:

```
public class Advertiser {
/* account is initialized
 * in the constructor and never
 * modified. */
      private Account account;
      public Advertiser() {
            account = new Account(this);
      }
      public Account getAccount() {
            return account;
      }
}
```

```
public class Account {
      /* owner is initialized
       * in the constructor and
       * never modified. */
      private Advertiser owner;
      publicAccount(owner:Advertiser) {
            this.owner = owner;
      }
      public Advertiser getOwner() {
            return owner;
      }
}
```

Bidirectional one-to-many association

Object design model before transformation:



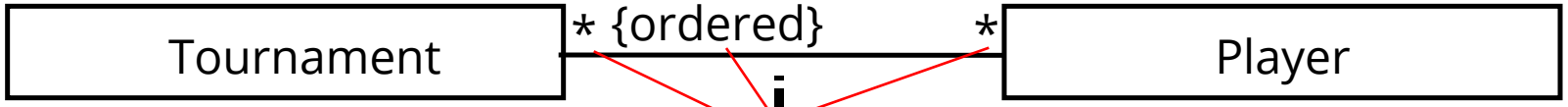
Source code after transformation:

```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount(Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}
```

```
public class Account {
    private Advertiser owner;
    public void setOwner(Advertiser newOwner)
    {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
                newOwner.addAccount(this);
            if (oldOwner != null)
                old.removeAccount(this);
        }
    }
}
```

Bidirectional many-to-many association

Object design model before transformation



Source code after transformation

```
public class Tournament {
    private List players;
    public Tournament() {
        players = new ArrayList();
    }
    public void addPlayer(Player p) {
        if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
        }
    }
}
```

```
public class Player {
    private List tournaments;
    public Player() {
        tournaments = new ArrayList();
    }
    public void addTournament(Tournament
    t) {
        if (!tournaments.contains(t)) {
          tournaments.add(t);
          t.addPlayer(this);
        }
    }
}
```

Mapping Contracts to Exceptions

1. **Checking preconditions.**

- Preconditions should be checked at the beginning of the method, before any processing is done.

- There should be a test that checks if the precondition is true and raises an exception otherwise.

- Each precondition corresponds to a different exception

Mapping Contracts to Exceptions

2. Checking post conditions.

- Post conditions should be checked at the end of the method, after all the work has been accomplished and the state changes are finalized.

- Post condition corresponds to a Boolean expression in an if statement that raises an exception if the contract is violated.

Mapping Contracts to Exceptions

3. Checking invariants.

- When treating each operation contract individually, invariants are checked at the same time as post conditions.

4. Dealing with inheritance.

- The checking code for preconditions and post conditions should be encapsulated into separate methods that can be called from subclasses.

Mapping Contracts to Exceptions

5. Coding effort.

- In many cases, the code required for checking preconditions and post conditions is longer and more complex than the code accomplishing the real work.

6. Increased opportunities for defects.

- Checking code can also include errors, increasing testing effort.

Mapping Contracts to Exceptions

7. Performances drawback.

Checking systematically all contracts can significantly slow down the code, sometimes by an order of magnitude. Although correctness is always a design goal, response time and throughput design goals would not be met.

```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
            throw new KnownPlayerException(p);
        }
        //... Normal addPlayer behavior
    }
}
public class TournamentForm {
    private TournamentControl control;
    private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and
                // proceed to the next player.
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```

Figure 10-14 Example of exception handling in Java. `TournamentForm` catches exceptions raised by `Tournament` and `TournamentControl` and logs them into an error console for display to the user.

```
public class Tournament {
//...
    private List players;

    public void addPlayer(Player p)
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,
            IllegalNumPlayers, IllegalMaxNumPlayers
    {
        // check precondition!isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();

        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // check post condition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() + 1
        if (getNumPlayers() != pre_getNumPlayers + 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
    }
    //
```