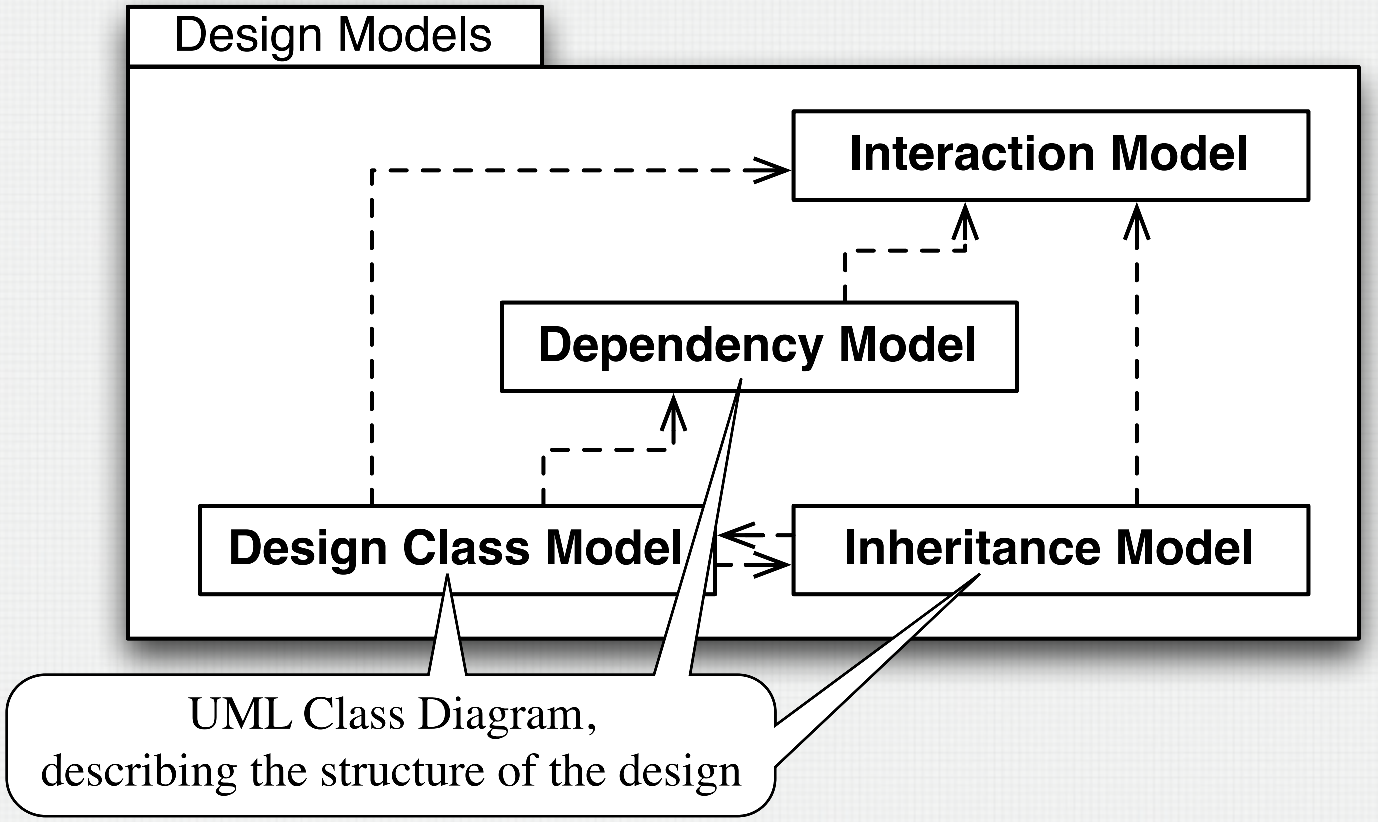
Structural Design

Jörg Kienzle & Alfred Strohmeier

Structural Design Overview

- Purpose and Process of Design
- Design Class Model
  - Deriving the Design Class Model from the Interaction Model
  - Refining the Concept Model
- Dependency Model
  - Usage Dependencies / Transient References
  - Navigable Associations / Permanent References
- Inheritance Model
- Detailed Design Process Summary

Fondue Models: Design



Design Models

- Interaction Model

  Describes the Complete System Behaviour

  - The Interaction Model shows how objects interact at run-time to support the functionality specified in the Operation Model.
- Dependency Model
  - The Dependency Model describes dependencies between classes and communication paths between interacting objects.
- Inheritance Model
  - The Inheritance Model describes the superclass/subclass inheritance design structure.
- Design Class Model

  Describes the Complete System Structure

  - The Design Class Model is composed of the contents of all design classes, i.e. their (value) attributes and methods, all the navigable associations between design classes, and the inheritance structure.

The Design Process in a Nutshell

5. Develop the Interaction Model
    1. Develop communication diagrams for all System Operations
    2. Derive a consistent architecture assigning responsibilities to classes
    3. Revise the communication diagrams, yielding the Interaction Model
6. Develop the Dependency Model based on the Interaction Model
    1. All communication designed in the Interaction Model result in dependencies
7. Develop the Design Class Model
    1. Develop a first version of the Design Class Model based on the Interaction Model
    2. Factor out common properties of classes and build the Inheritance Model
    3. Update and build the final Design Class Model

Design Class Model (1)

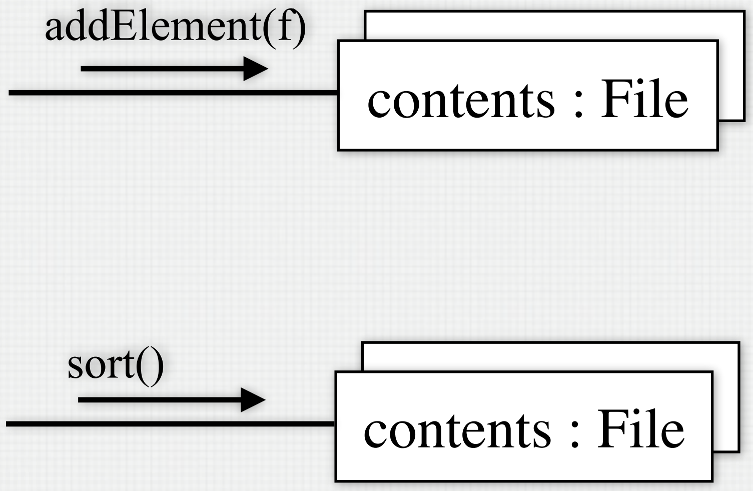
- The Design Class Model shows the static structure of the system
  - Design classes with their attributes and methods
  - Relationships between classes and in which direction they are navigated during execution
- The classes in the design class model must capture all the concepts specified in the concept model
  - Some concept classes are imported into the design class model
  - Some concept classes are split into several design classes
  - Some concept classes disappear, e.g. because they are implemented as enumeration types, or because they are transformed into attributes
  - Some new classes appear
    - To implement relationships, especially ternary associations, if any
    - To provide design-related functionality as described in the Interaction Model

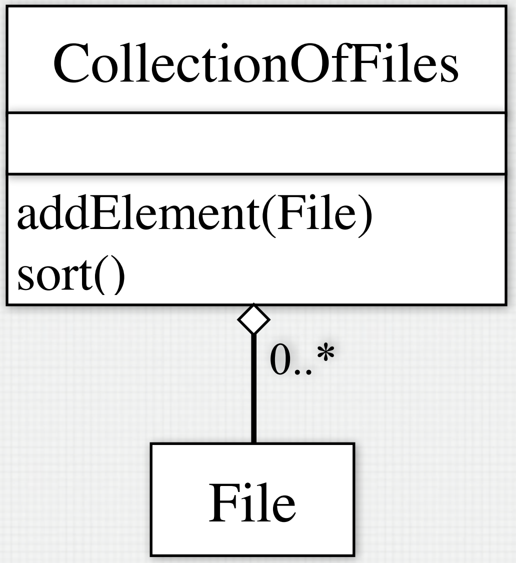
Building the Design Class Model (1)

- Elaborate a first draft of the Design Class Model by adding all the design classes discovered while elaborating the Interaction Model
  - Add the designed operations to the appropriate classes
  - Add value attributes
- Since the Concept Model was used as a basis to inspire the design of the Interaction Model, the Design Class Model and the Concept Model are related
  - The Concept Model is an abstraction of the Design Class Model
  - The Design Class Model is a solution-specific refinement of the Concept Model
  - Attributes of classes of the Concept Model might need to be moved to attributes in the Design Class Model

Building the Design Class Model (2)

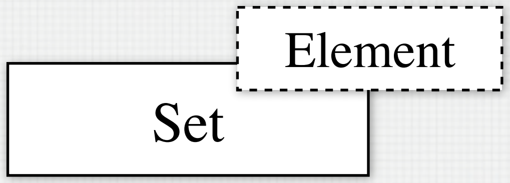
- For each multiobject in the Interaction Model, add an aggregation class to the class model.
  - Add the operations and value attributes required by the interaction model to the classes realizing multiobjects.





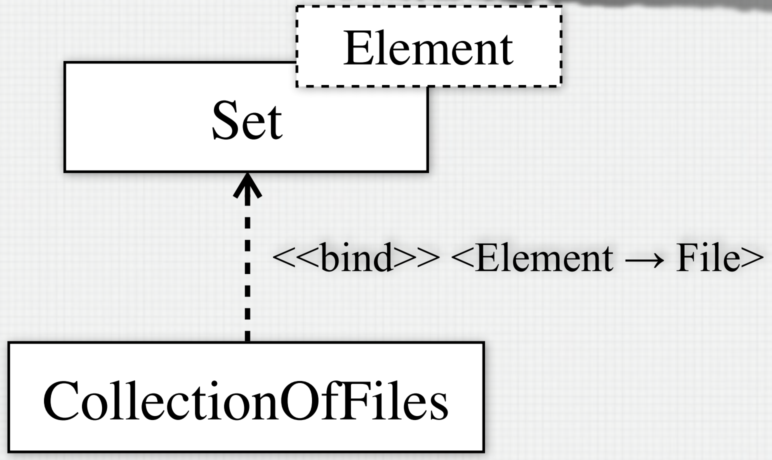
UML Template Classes (1)

- Collection classes are very common:
  - List, Queue, Stack, Set, Sorted... , Blocking... , HashMap
- Modern programming languages often provide generic implementations of collections
  - Java generics
  - C++ templates
- UML provides templates
  - Allows model elements to be parameterized
  - Formal parameters are shown in a dashed rectangle, usually in the top right corner of the model element



UML Template Classes (2)

- A parameterized UML model element is a reusable model element
- It can be instantiated by specifying a binding, which assigns actual parameters to the formal parameters
  - Element → File
  - Or visually:



Dependency Model (1)

- When developing the Interaction Model, objects interact with each other by invoking methods
- Decisions are made on how the communication paths in the system are realized
  - Who calls whom?
  - How does one object know about the other?
  - Who has permanent references to other objects?
  - When are object references passed as parameters?
- The Dependency Model focusses on the reference structure of the classes in the system
  - A Dependency Diagram is a class diagram, but only a subset of the class diagram notations is used

Dependency Model (2)

- An object must have a reference to another object when it wants to communicate with it.
  - The server object must be visible to the client object when the client sends a message to the server.
- The reference might be transient
  - The result is a Usage Dependency between the classes.
- The reference might be permanent
  - The result will be a Navigable Association.
- The decision, transient or permanent, is recorded in the Dependency Model

Usage Dependency (1)

- A usage dependency is a situation in which a class A, called the client class, depends on a class B, called the server class, for its correct implementation or functioning.
  - Usage corresponds to a transient link, that is, a connection between instances of classes that is not meaningful or present all the time, but only in some context, such as the execution of a method.
- A usage dependency is depicted by a dashed arrow from the client to the server, with the keyword <<use>>.
  - The arrowhead is on the independent class, and the tail on the dependent class.

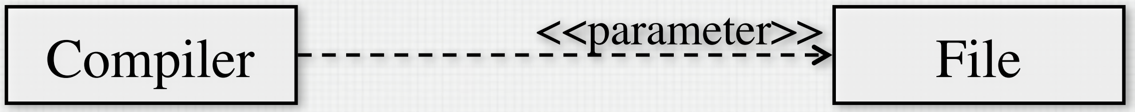
Usage Dependency Types

- <<call>>
  - The client class calls a method of the server class.
- <<instantiate>>
  - The client class creates an instance (object) of the server class.
- <<parameter>>
  - An operation of the client class has a parameter belonging to the server class.
- <<use>> is the superset of all transient dependencies, including call, instantiate and parameter

Transient Reference

- The client only needs to message a server in the context of a single method invocation
- Access is given through a parameter, or the object is created by the invoking method
- The result is a usage dependency
- Example
  - A compiler needs to know the source file to process:

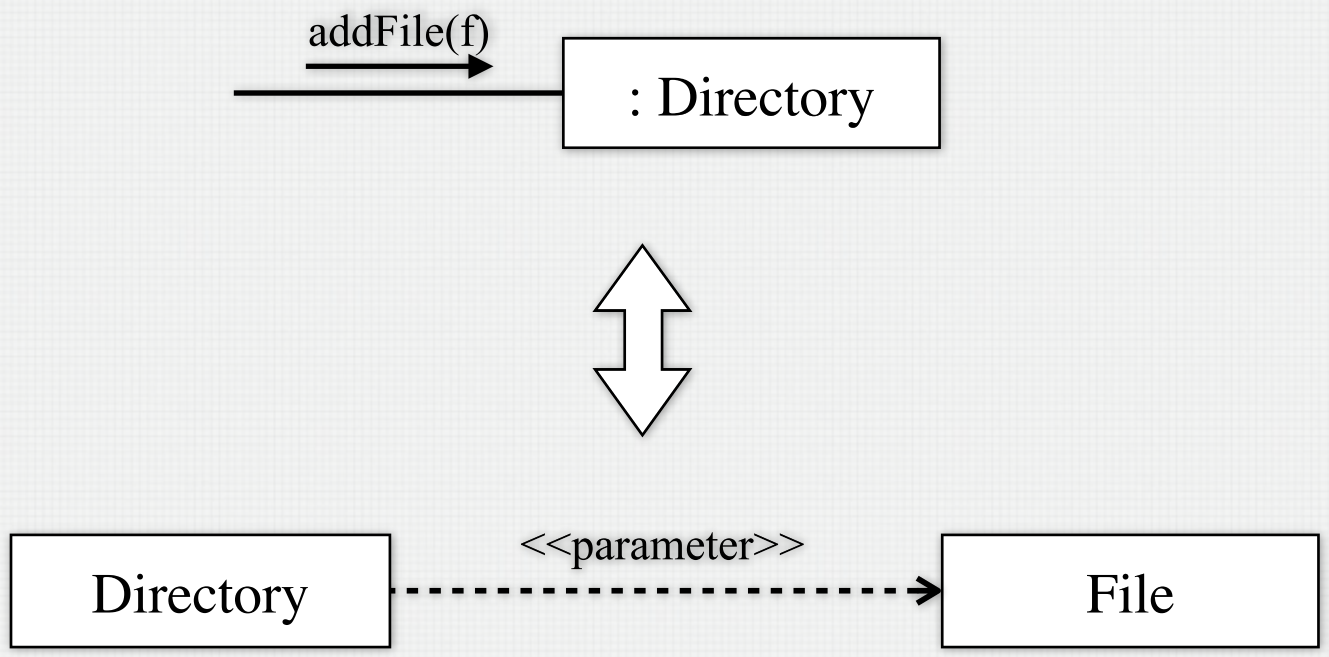
operation Compiler::compile(f: File)



From Interactions to Dependencies (1)

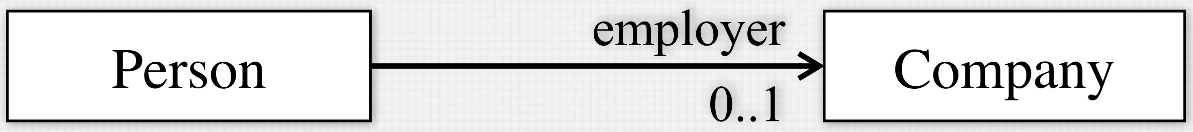
Interaction Model:

Dependency Model:



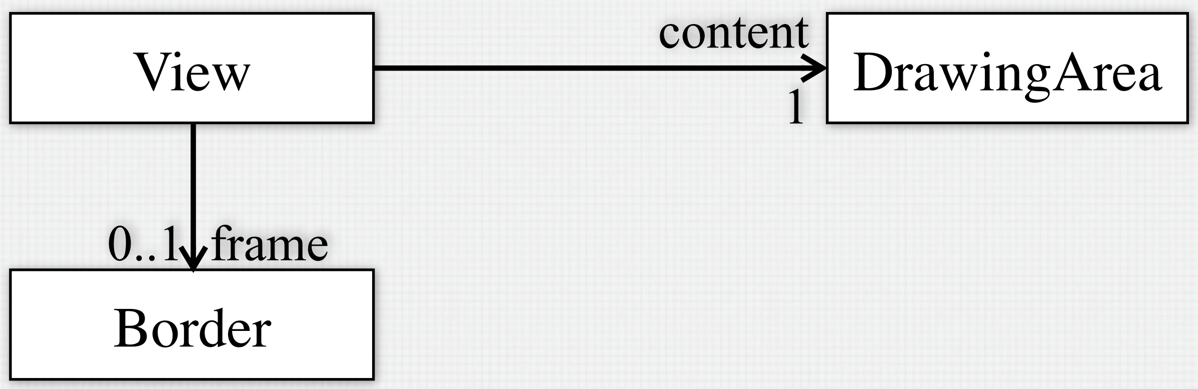
Navigable Association (1)

- Navigability indicates whether it is possible to traverse a binary association to obtain the object (multiplicity 0 or 1) or the collection of objects associated with an instance of the class.
- A Navigable Association is shown by adding an arrowhead to the association end.
- Arrowheads may be attached to zero, one or both ends of an association.
  - No arrowhead means not navigable
  - Non-navigable associations are never used, i.e., they are typically not needed in a design, and therefore do not have to be implemented



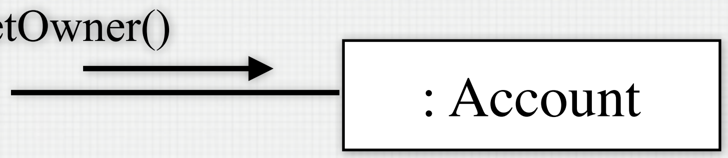
Navigable Association Example

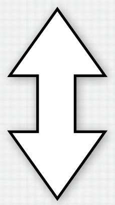
- From a View object it is possible to navigate to its contents, an object of the class DrawingArea. It is also possible to navigate to its frame, a Border object.
- Neither the drawing area nor the border “know” about their view.
- Perhaps it is possible to find the view of a drawing area, but it must be searched for by means not part of the DrawingArea class.



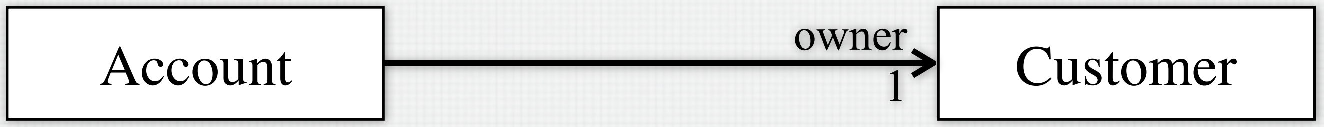
From Interactions to Dependencies (2)

Interaction Model:





Dependency Model:



Permanent Reference

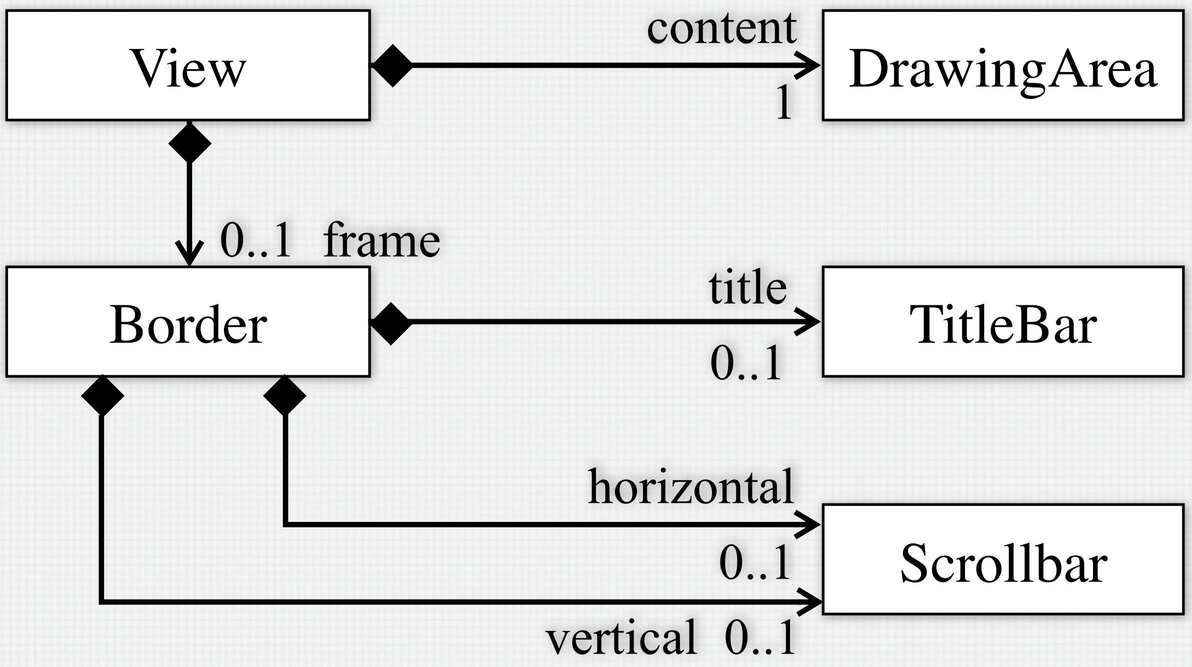
- A permanent reference is needed when an object needs the same reference in many contexts, e.g. when the reference to the server object must persist between method calls
- A permanent reference results in a navigable association
  - In the implementation, a permanent reference will become an object attribute of the class, sometimes called a pseudo-attribute
- If a class is able to return a reference to an object or a collection of objects (without messaging other objects), then each instance has to store a permanent reference

Bound Life Time

- If the lifetime of the server is enclosed within that of the client, i.e. the lifetime of the server cannot start before that of the client, and the server cannot outlive the client, then the permanent reference is then said to be bound
  - A composition association in the concept model often leads to a reference with a bound lifetime
  - A collection implementing a navigable association end of multiplicity “many” often has a bound lifetime
    - Beware, it does not mean that the members of the collection have a bound lifetime

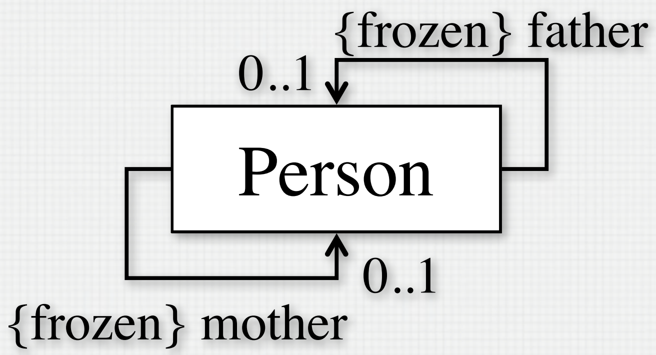
Bound Life Time Example

- A view knows about its border. The border is created when the view is created, and can not outlive the view.
  - The border knows about its title bar and scrollbars. They are created when the border is created, and can not outlive the border.



Changeability

- If a reference cannot be changed after initialization, then the corresponding association link is said to be frozen.
  - Default is changeable
- It is the link that is frozen, the object itself is usually changeable, i.e. its attribute values can change.
- Example
  - The mother / father of a person cannot be changed.

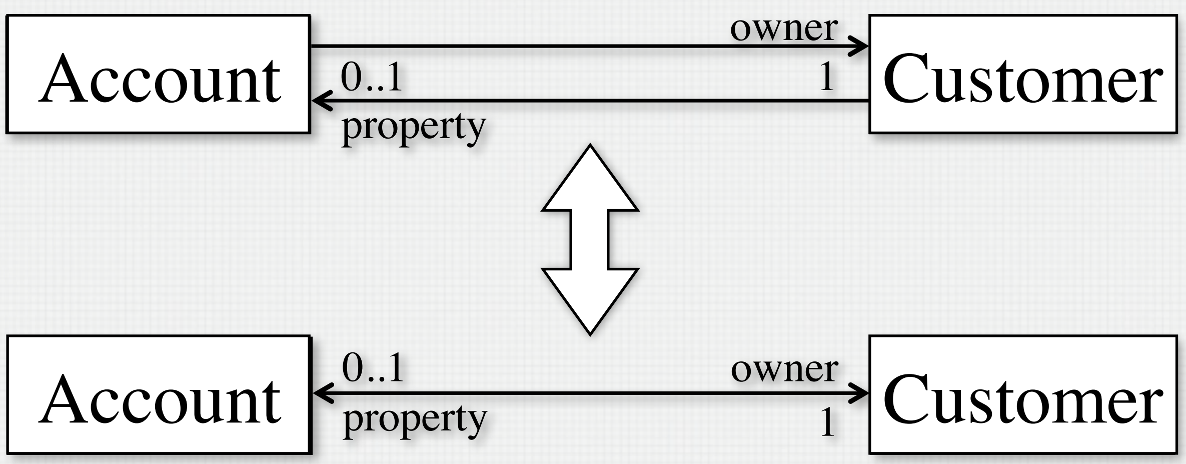


Categories of Reference Relationships

- Reference lifetime
  - Transient reference,
    - shown by a dependency relationship
  - Permanent reference
    - shown by a navigable association
    - Server binding
      - Unbound lifetime (default)
      - Bound lifetime, composition association
    - Changeability (mutability)
      - Frozen (constant), property (tag) {frozen}
      - changeable (mutable) (default)

Mutual References

- There are often pairs of links in the Dependency Model which are the inverse of each other
  - Such links can stem from the same association in the Concept Model
  - To ensure consistency of the underlying association, referential integrity of the links must be enforced by the implementation.
- Replace the two navigable associations by a single two-way navigable association

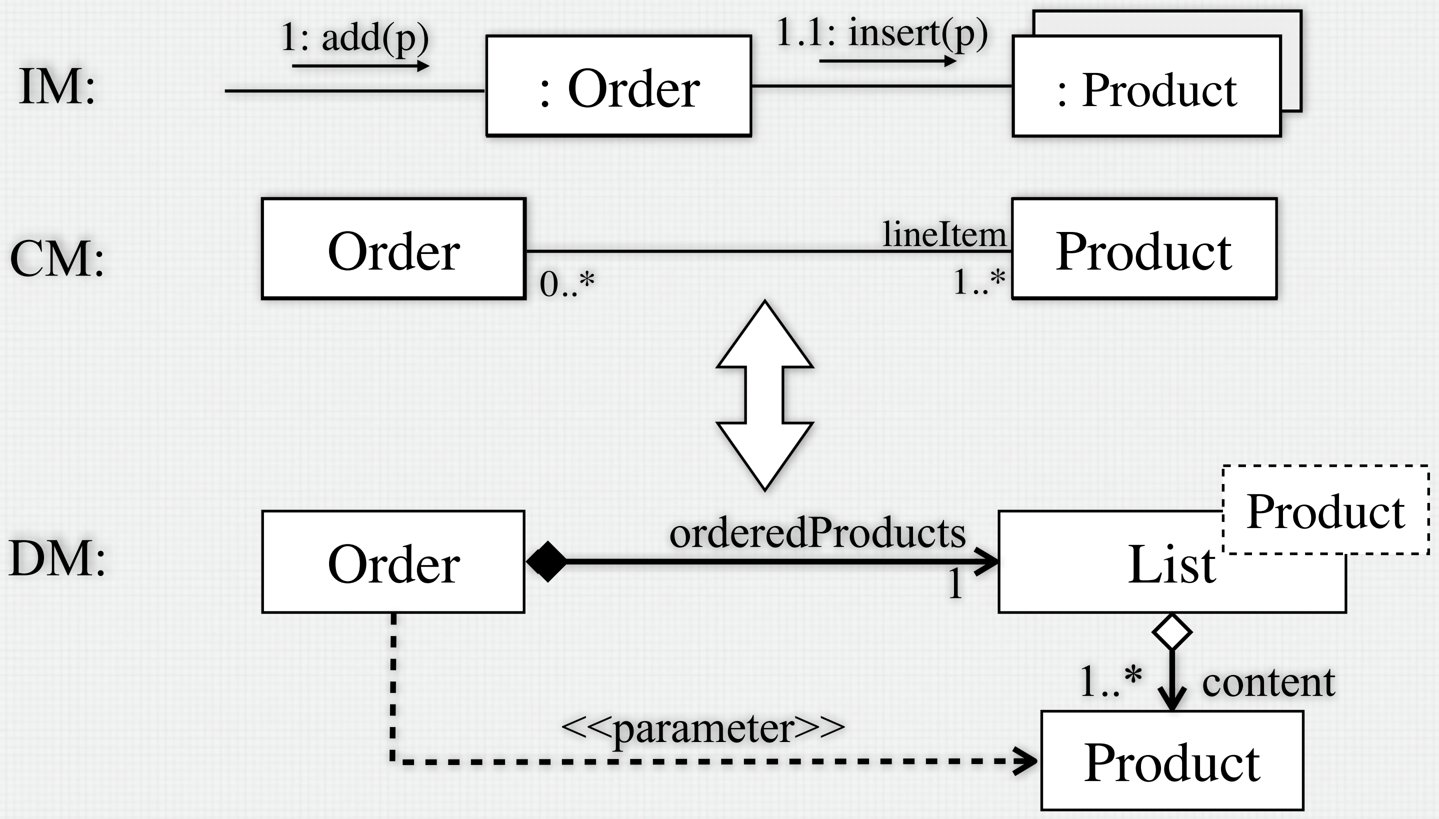


Multiobjects and Associations

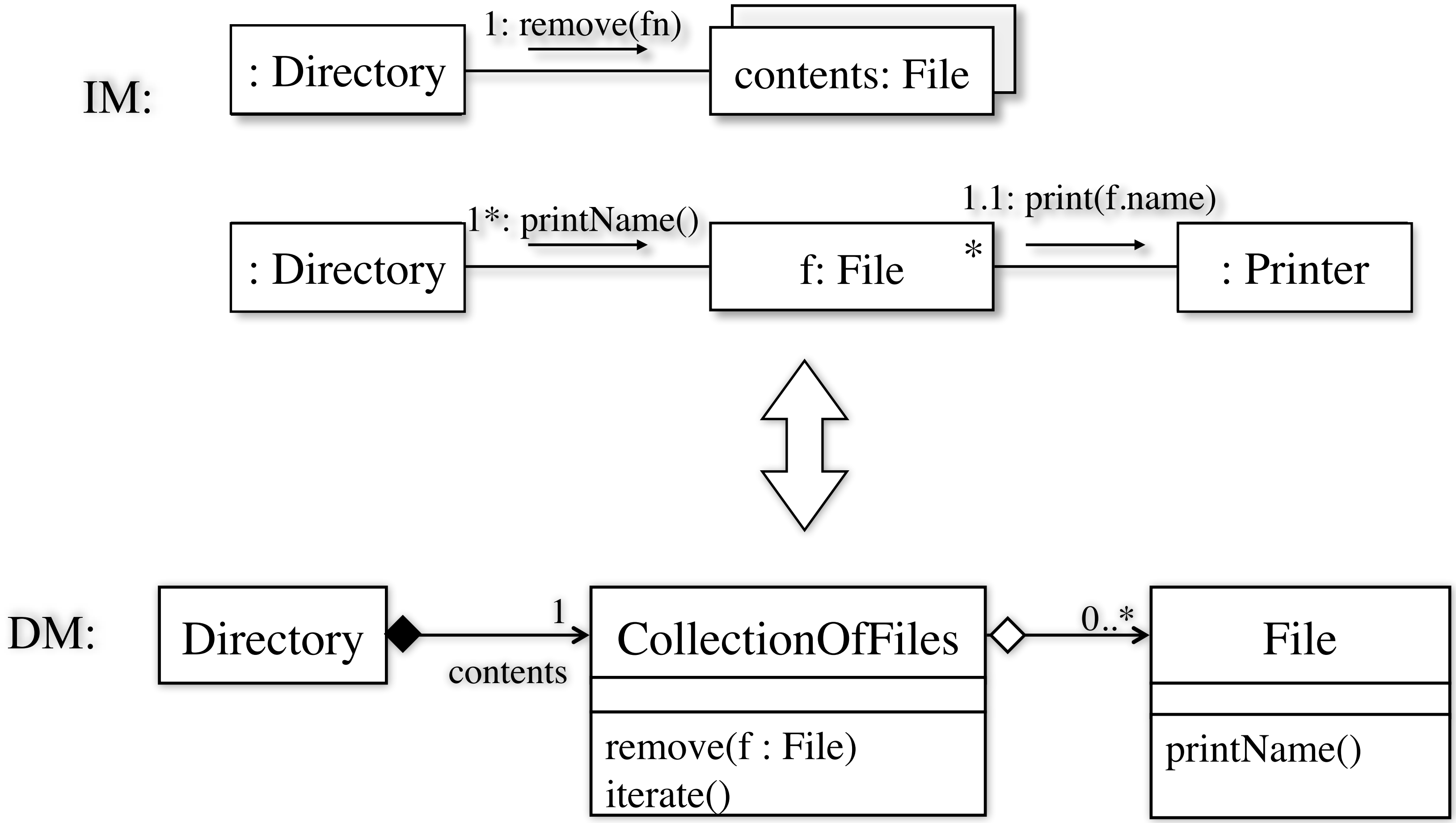
- Multiobjects are usually the realization of association ends with multiplicity “many”
- As already said, for every multiobject, an aggregation class must be added to the class model
  - This class is often an instantiation of a common template collection class
- The “many” association end is then replaced by a navigable association end of multiplicity 1 or 0..1 that connects to the collection.

Multiobject Example

- Interaction Model: “insert a product into an order”
- Concept Model: Orders are composed of line items, each being related to a product.

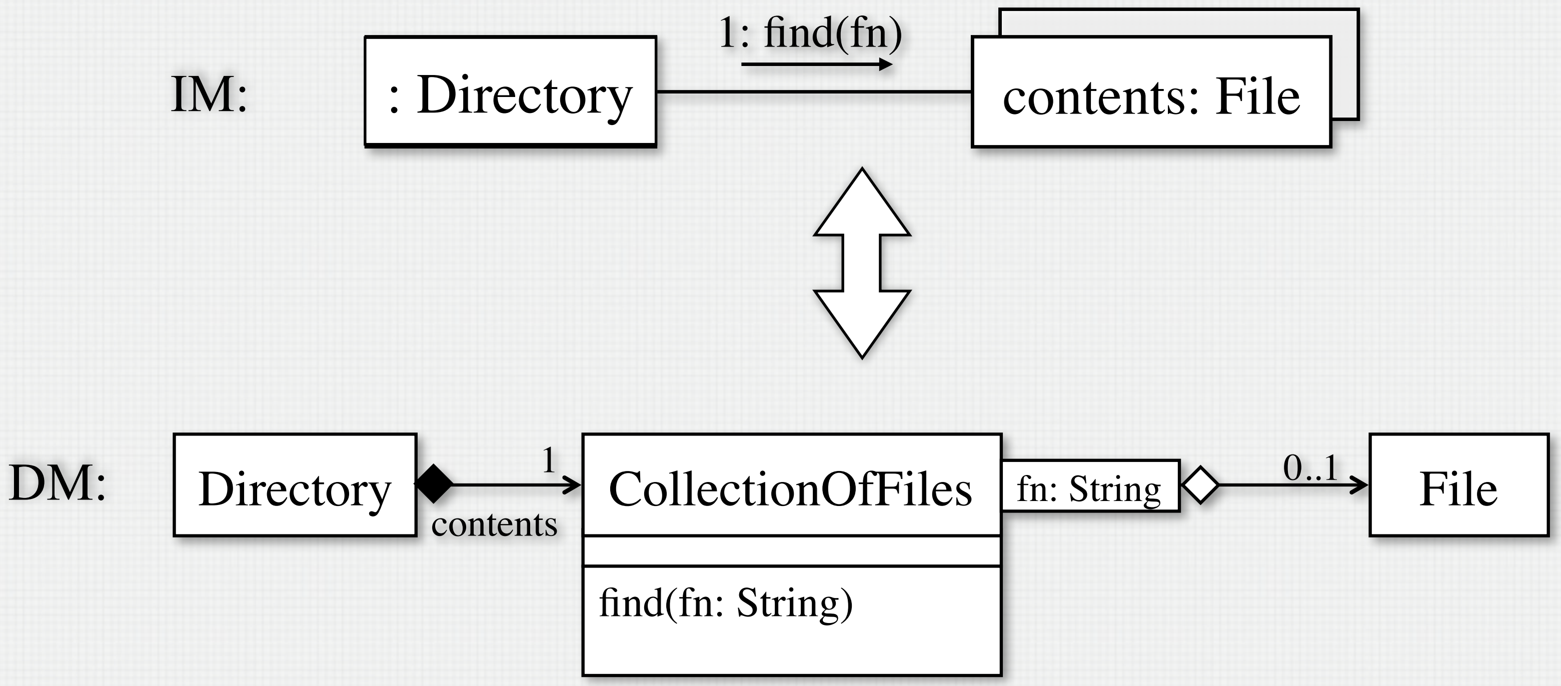


"Many" Roles Example



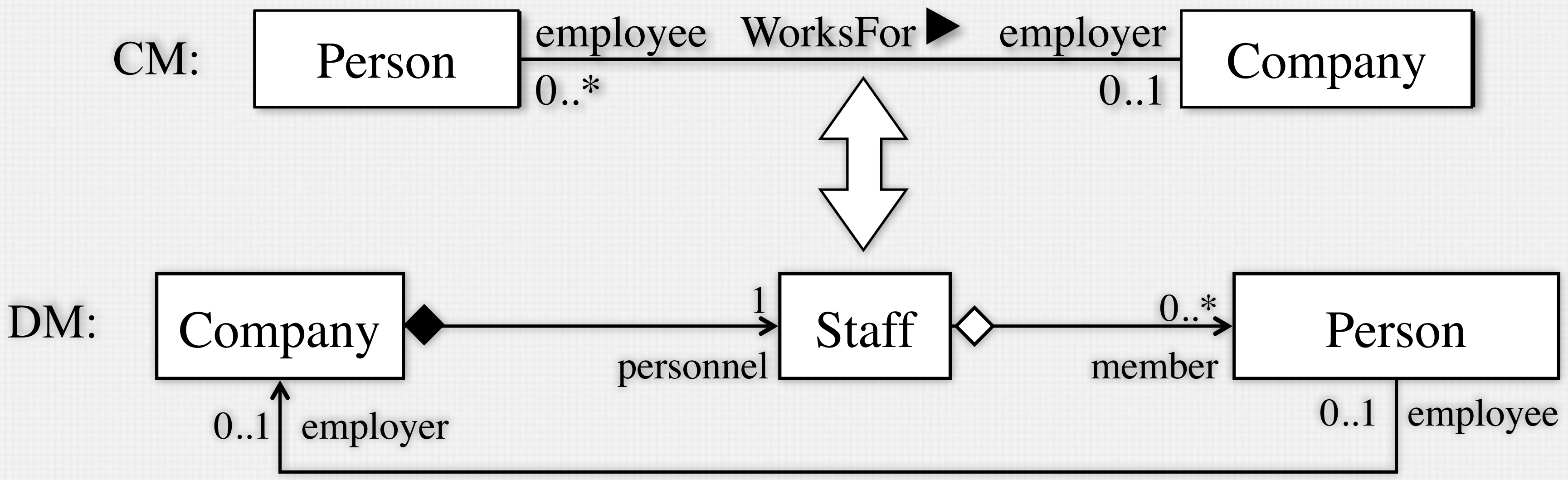
Multiobjects and Qualified Associations

- Multiobjects are often used for retrieving an object based on some search criteria. It is useful to show this kind of navigation by a qualified navigable association.
  - The implementation can then use a map or dictionary for realizing the navigable association end.



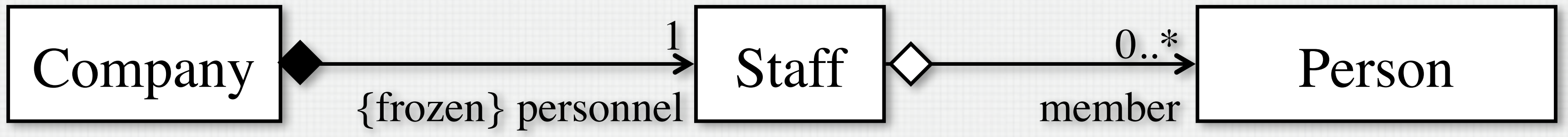
Multiobjects and Bi-directional Associations

- Associations with multiplicity “many” from the Concept Model that are navigable in both directions create cyclic associations in the Design Class Model.
  - Use OCL to clarify the relation between the different associations.



Reference Properties of Collections

- For a collection, the reference properties apply to the collection, and not to its members.
- For a collection, the case: frozen and bound is the most frequent.
- Example
  - The personnel of a company remains the same during the whole lifetime of a company, but its content, i.e. the employed persons don't remain the same. The personnel's lifetime is bound to that of the company.



Dependency Model Process (1)

- For each message in every interaction diagram, a communication path is needed from the client class to the server object
    - If the server object is a parameter or is instantiated, then we report that the client class has a usage dependency on the server class
    - If a message yields an object (or collection of objects), the server object must hold a permanent reference to the object (or collection), either directly, or through a permanent reference to an intermediate object
        - This shows up as a navigable association in the Dependency Model
    - If the server as part of the operation calls objects that are not passed into the operation as a parameter, then the server object must hold a permanent reference to the object

Dependency Model Process (2)

- For all navigable associations, determine if the association is already in the Dependency Model
  - If no, add it
- When there are two or more associations between the same classes navigable in the opposite directions, determine if they are inverse associations of each other
- The same analysis should be performed on navigable associations leading to cycles
- Annotate navigable associations with the aggregation or composition diamond and the reference property {frozen}, if appropriate.

Simplifying the Dependency Model

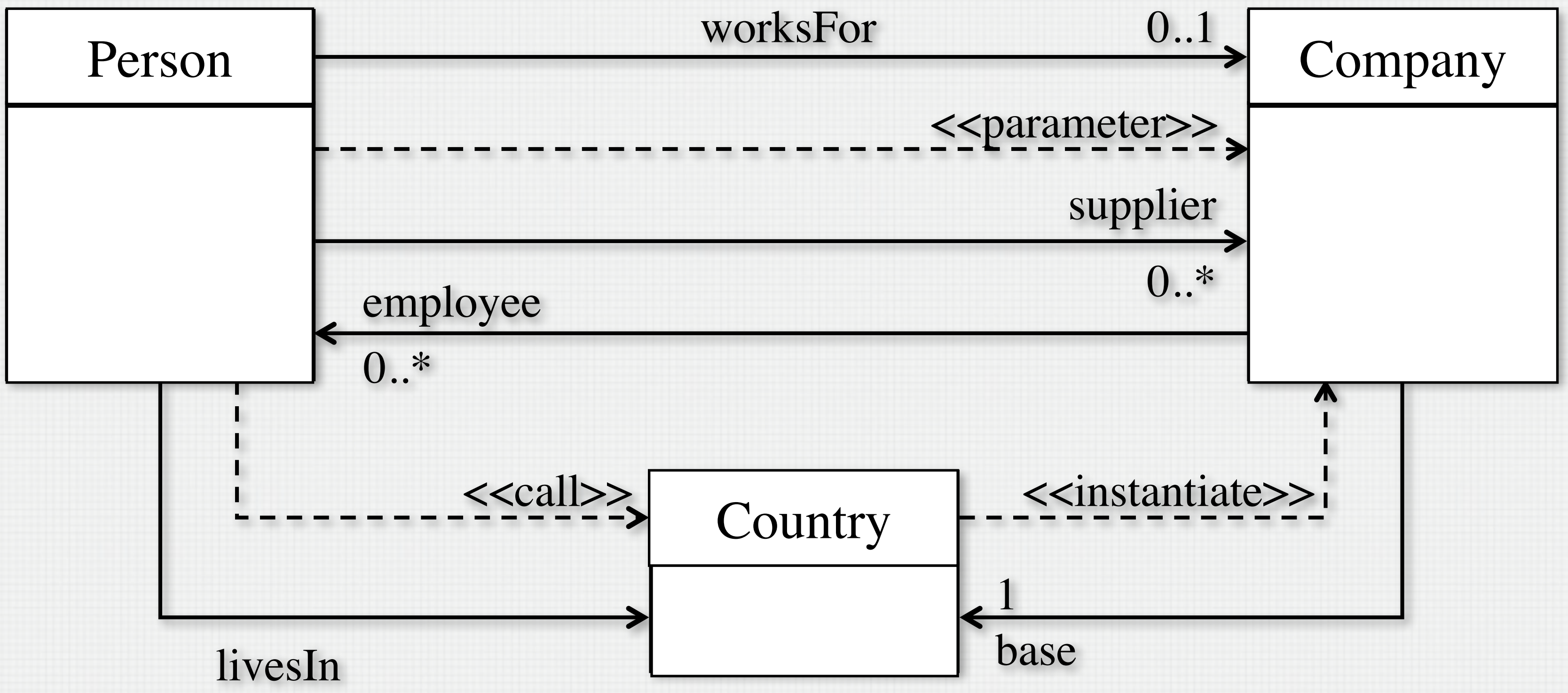
- Permanent references are “stronger” than temporary ones
  - If there is a navigable association between two classes, then the source class is always dependent on the target class, and usage dependency does not have to be shown explicitly
- Some entities are used all over the system. For readability reasons, we propose to drop them from the Dependency Model
  - Look out for unique system-wide objects, a.k.a. singletons, which are used, and therefore visible, from all classes.
  - Look out for system-wide classes, which are used, and therefore visible, from all classes.
- Document this decision, e.g. by annotating / stereotyping the classes using the <<system-wide>> stereotype in the Design Class Model
    - System-wide objects show up in the Design Class Model as classes stereotyped <<system-wide>> with multiplicity 1

Dependency Model Questions

A, B and C are three classes; OA is an object (role name) of class A, and OB is an object (role name) of class B. For each statement below, decide if it is correct or not. Then justify your answer.

1. If OA messages OB, and OB is a system-wide object, then no navigational association from A to B has to be shown on the Dependency Diagram.
2. If OB is a system-wide object, and OA messages OB, then there is a usage dependency of A on B.
3. A navigable association from A to B in the Dependency Diagram includes any usage dependency of B on A (UML uses a dashed arrow from B to A to show it).
4. If there is a navigable association from A to B, and a navigable association from B to C, then any navigable association from A to C can be dropped.
5. If there is a navigable association from A to B, and a navigable association from B to C, then any usage dependency of A on C can be dropped from the Dependency Diagram.

Simplify Question



Use of Dependency Model

- The Dependency Model is mainly there to provide insight into your design during development
  - Can be used to assess the coherence of the design
    - Are creational responsibilities coherently modularized?
    - Is the design of each class in line with the responsibilities assigned to it?
      - Encapsulation of state
      - Provision of behaviour
  - Exposes unnecessary coupling / circular dependencies
    - Is it really necessary that everyone depends on certain classes?
    - Can restructuring of a specific interaction design break circular dependencies?
    - Can refactoring be applied to decrease coupling?
  - Can help plan the software development
    - Strongly related classes should be designed / implemented by the same team / individual
- The Design Class Model is the blueprint of the implementation

From Dep. Model to Design Class Model

- To build the Design Class Model
  - Transfer all classes and all navigable associations from the Dependency Model to the Design Class Model
  - Transfer bound and frozen constraints
  - Supply a role name for each navigable association end
  - Remove the usage dependencies
  - Identify attributes that store the state needed for the behaviour described in the Interaction Model
- The Design Class Model is the blueprint of the implementation
  - Can easily be used to generate code skeletons

From Concept Model to Design Class Model

- The Design Class Model is a refinement of the Concept Model
- The developer needs to:
  - Remove associations that are never used (i.e. navigated over) in the Interaction Model
  - Add navigable associations from the dependency model
  - Add role names to all navigable association ends
  - Remove actors together with the associations they are connected to
  - Remove the system object (you can make it a package)
    - Or transform it into a SystemProtocolEnforcer class that takes care of enforcing the constraints defined by the Protocol Model

Inheritance vs. Gen./Specialization

- Specialization/generalization defines a semantic relationship between two classes. This semantic relationship is inherent in the application domain.
- Specialization/generalization are properties of the domain model and not of the system design or implementation.
- At design, an inheritance relationship is a property of the design classes, and not necessarily of the domain
    - Inheritance can be used as a design construct for specialization
    - Inheritance relationships at design are prescriptive, whereas specialization/generalization relationships at requirements specification are descriptive
    - Inheritance may be used for efficiency reasons or code reuse

Inheritance Model Process

- The Inheritance Model shows the inheritance relations of the design
- Look out for common functionality and common structure between classes. The commonalties can be extracted to build new ancestor classes.
  - Note: Common functionality and common structure can also be factored out by genericity (template classes)
- Integrate the changes into the Design Class Model
  - The notation for inheritance is the same as for generalization/ specialization

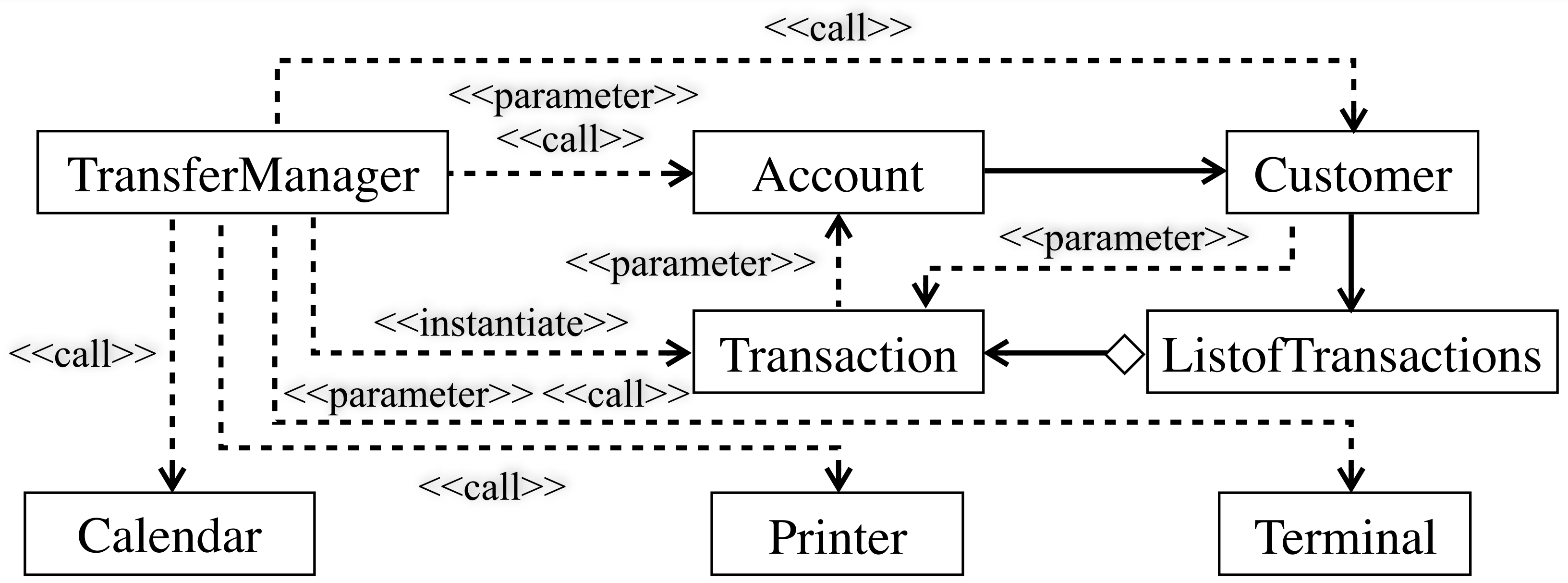
Design Class Model Summary

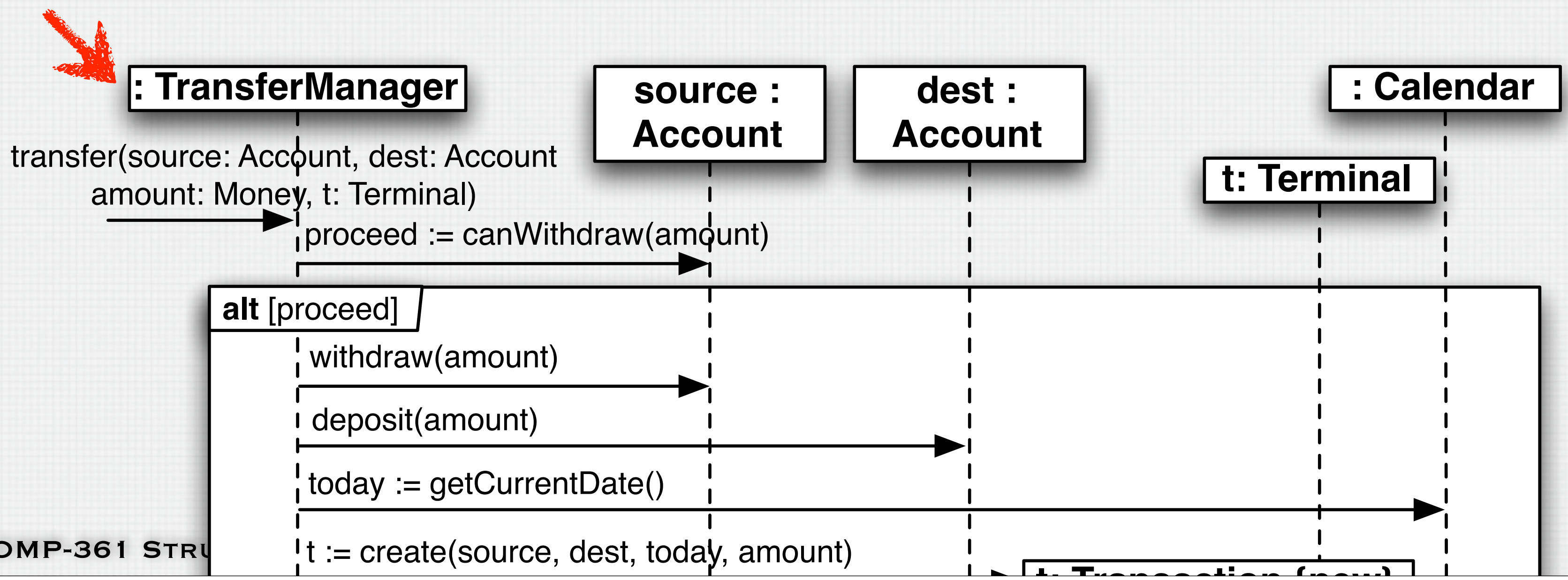
- The Design Class Model contains all classes which are controllers and (direct or indirect) collaborators of system operations.
- The design classes provide methods.
  - The design classes do not have object attributes or navigation methods.
  - We want to show associations graphically!
- The Design Class Model contains all navigable associations needed for object communication at run-time.
- The Design Class Model integrates the Inheritance Model, i.e., it shows inheritance relationships among classes

Bank Design Example

- Establish a Dependency Model for the Bank System based on the Transfer and GenerateMonthly designs
- Establish a Design Class Model for the Bank System

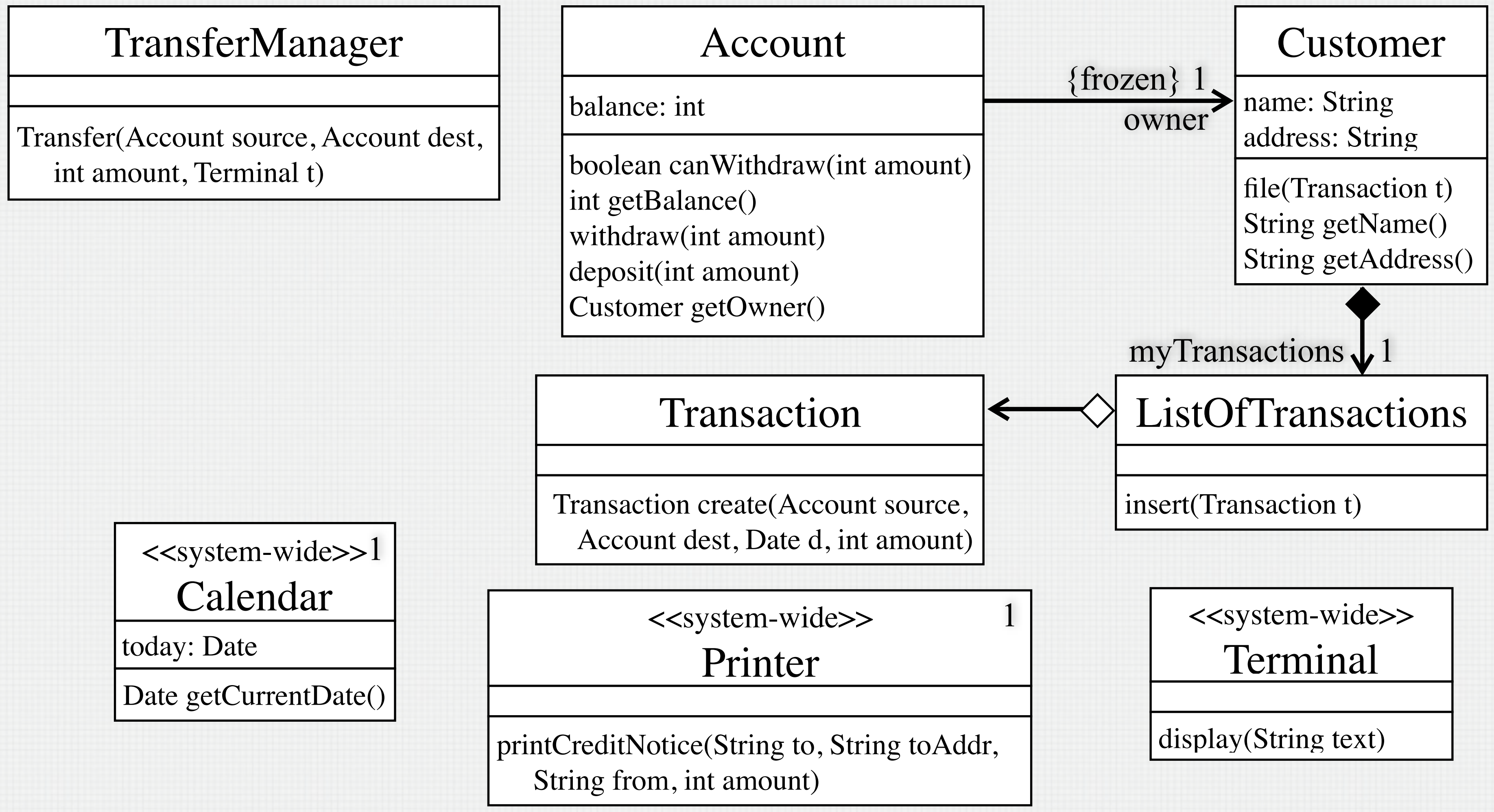
Transfer Dependency Model



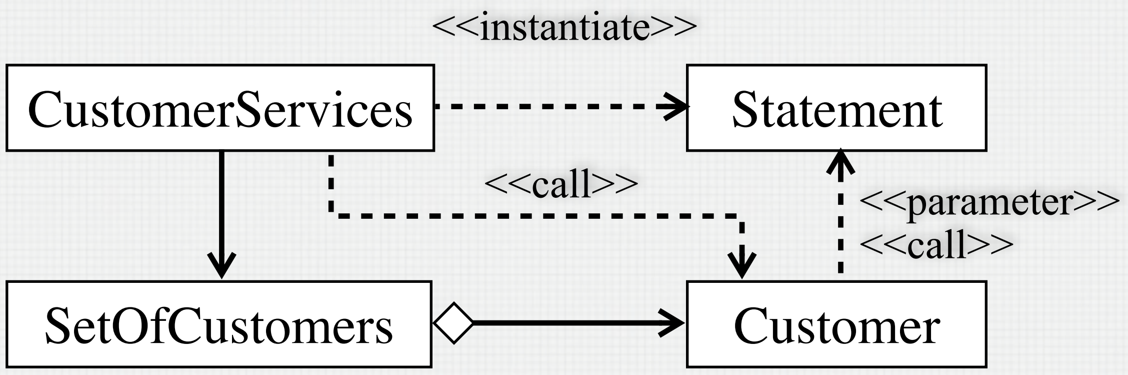


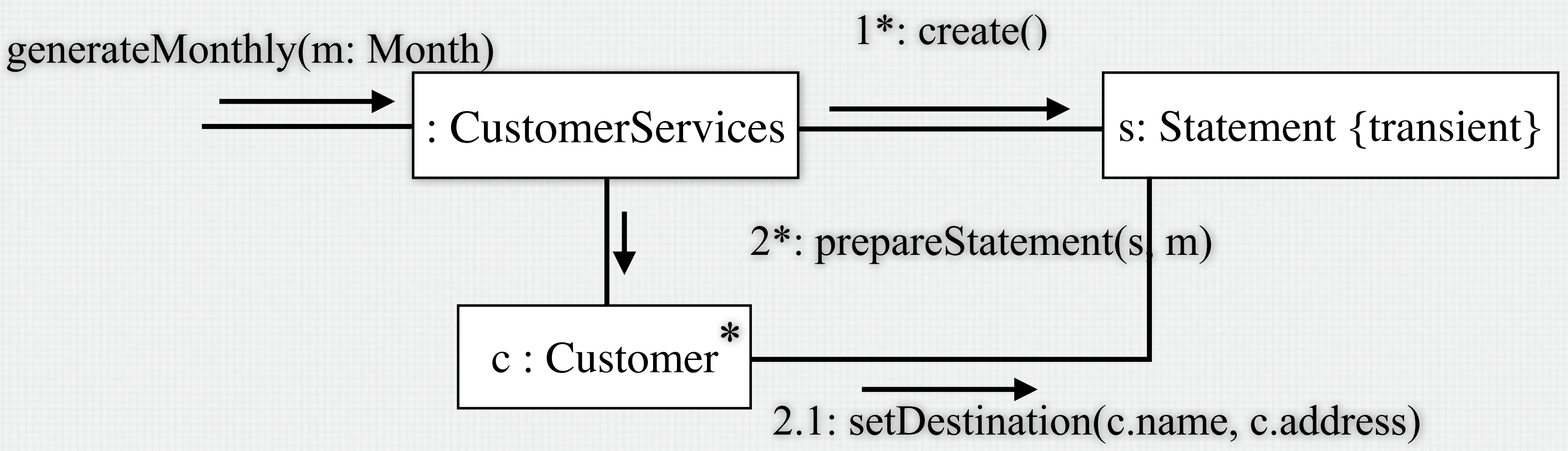
: TransferManager
source : Account
dest : Account
: Calendar
t: Terminal
transfer(source: Account, dest: Account amount: Money, t: Terminal)
proceed := canWithdraw(amount)
alt [proceed]
withdraw(amount)
deposit(amount)
today := getCurrentDate()
t := create(source, dest, today, amount)

Transfer Design Class Model



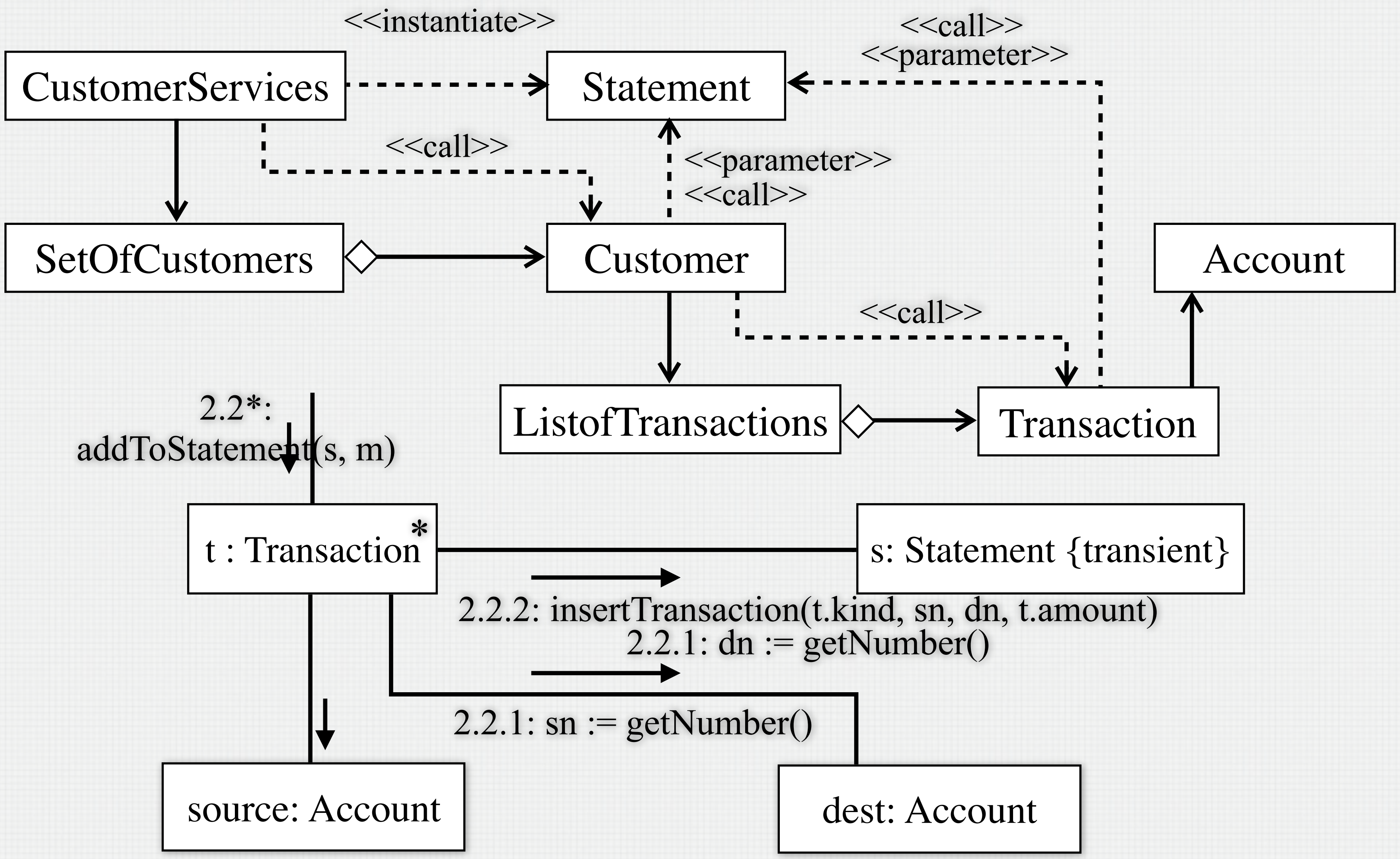
GenerateMonthly Dependency Model



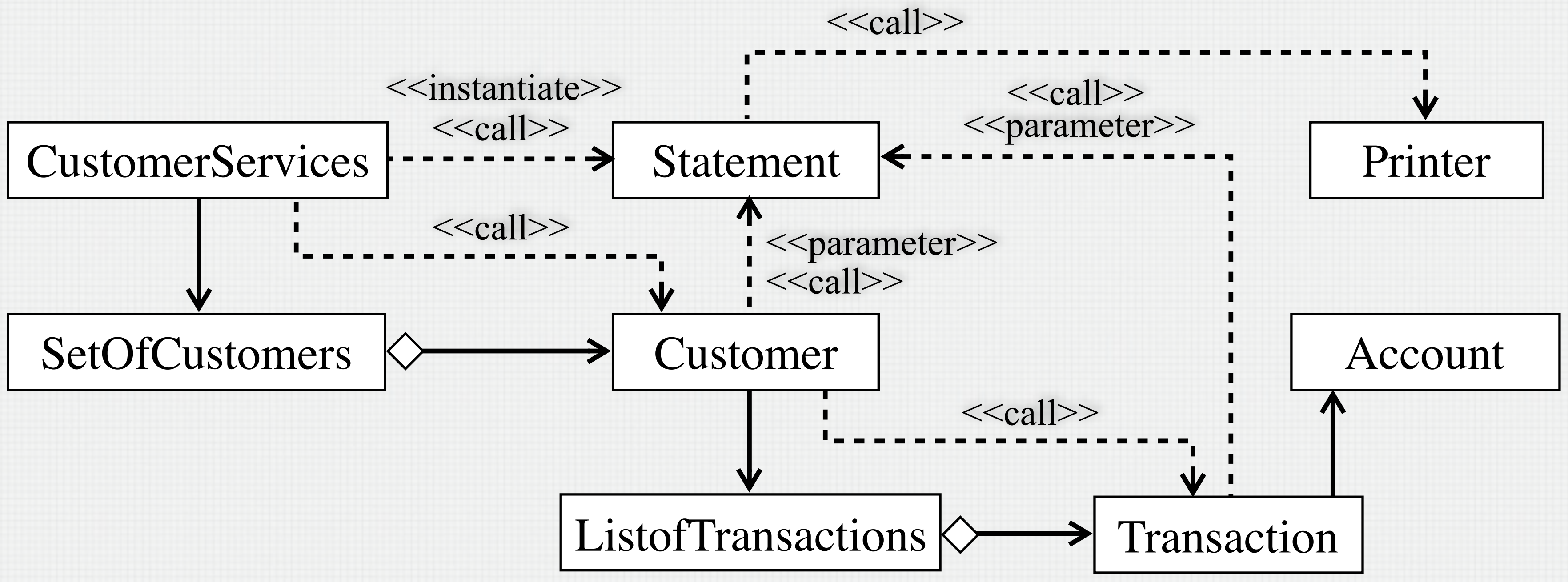


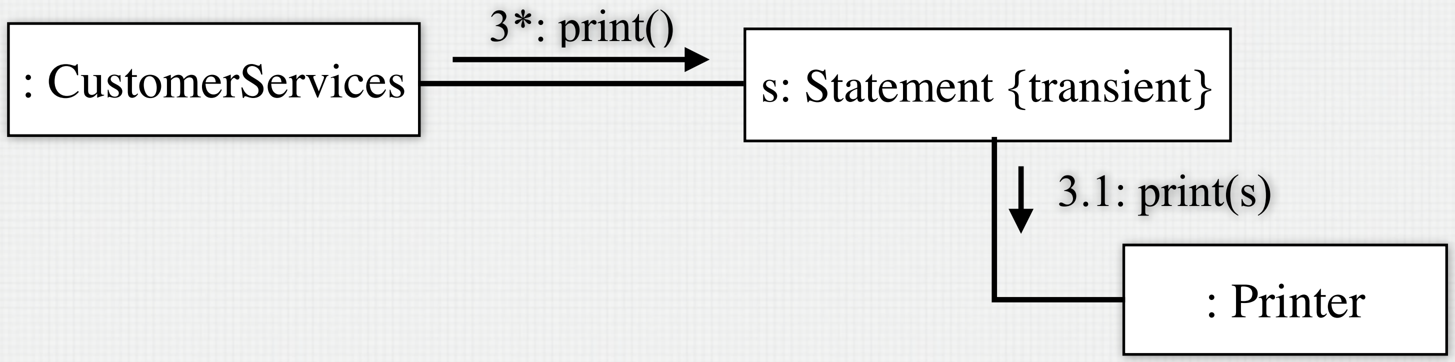
generateMonthly(m: Month)
1*: create()
: CustomerServices
s: Statement {transient}
2*: prepareStatement(s, m)
c : Customer
*
2.1: setDestination(c.name, c.address)

GenerateMonthly Dependency Model

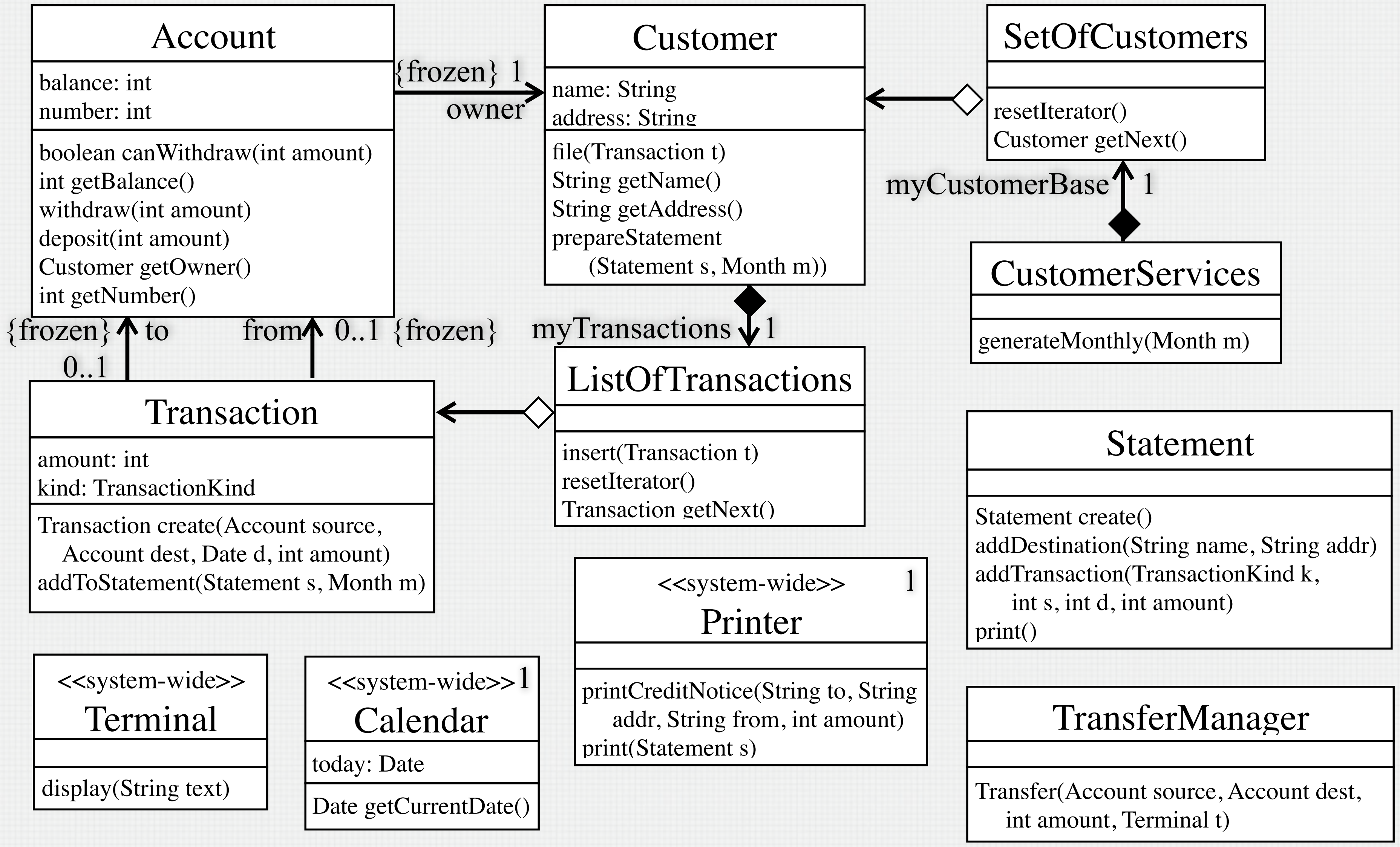


GenerateMonthly Dependency Model





Transfer+Generate Design Class Model



Principles of Good Design (1)

- Design with the aim to support future change
  - It is clear though that we must bias certain futures in that we cannot cover all futures, thus good designers are often good predictors.
- Develop modular systems
- Maximize module coherence
- Minimize module coupling, e.g. object interaction
- Minimize data and functional dependencies
- Cleanly separate functionality

Principles of Good Design (2)

- Distribute responsibilities evenly between classes
  - Avoid the extremes, i.e. “dumb” and “god” objects
- Have one class, one abstraction, with the right name
- Strive for class cohesion
  - A class should engage in just one general type of responsibility
  - Concentrate on the responsibilities of a class rather than the data it must encapsulate, because it will often lead you to a more extensible design
- Strive for method cohesion
  - A method should carry out a single specific function. This is analogous to functional cohesion in structured design.
- Be sure an abstraction is a class and not simply a role played by an object

Principles of Good Design (3)

- Encapsulate representation
  - Remember that the state of an object can only be changed through its public interface.
- Delegate rather than enquire
  - A client object tells a service object what it wants that object to do, rather than asking for information from that object so that it can do it
  - Remember data and related behaviour should be encapsulated together.
- Use inheritance wisely!
  - Obey the Liskov Substitution Principle (LSP):
    - Derived classes must be usable through the base class interface without the need for the user to know the difference.
  - Develop shallow inheritance hierarchies
  - Most root classes should be abstract

Additional Design Process Activities

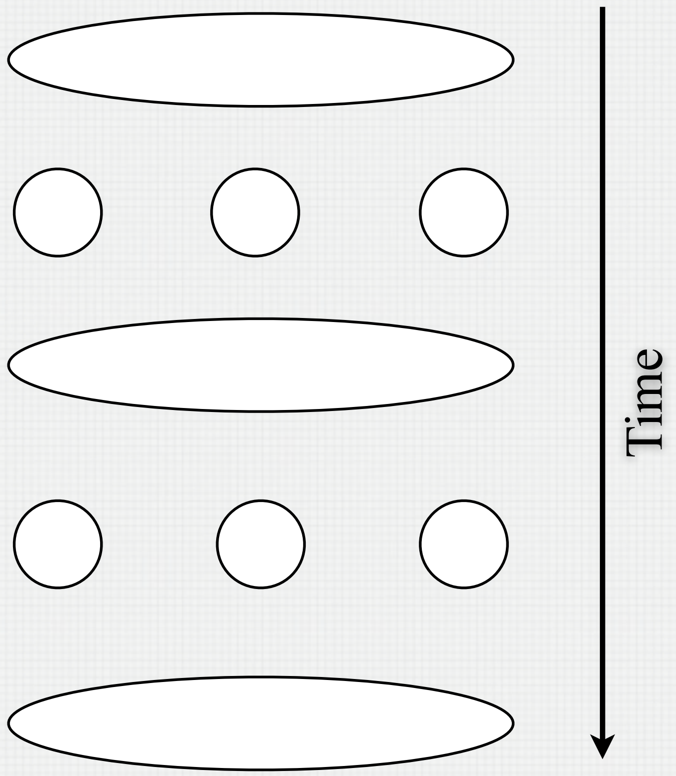
- System Architecture Design for Distributed Systems
  - Start drafting an architecture of the system before entering the design process.
    - Divide system-wide responsibilities between the client(s) and the server.
    - Divide responsibility for the system operations between the client(s) and the server.
- User Interface Design
  - Determine the mechanisms used by the user interface to call system operations, to provide values and objects for the system operations, and to report error messages to the user.
- How should Exceptional Situations be Handled?
  - If not specified in the operation model, determine how the system should react to exceptional situations
  - Determine the responsibilities of the finite state machine implementing the Protocol Model

Splitting the Work

- After having agreed on an architecture, it is possible to divide work among team members
- Determine the main classes by designing the communication diagrams for some important (orthogonal) system operations
    - Each team member may perform this task for some system operations
    - It might be worthy to have several designs for the same (important) operation.
    - The designs are then compared and consolidated.
- Allocate classes and system operations to team members.
    - A team member is responsible for all design models for his/her classes and system operations.

Work Breakdown

- Alone
  - Design communication diagrams / sequence diagrams
  - Prepare questions and suggestions,
  - Rework communication diagrams (yours or those of others).
- Group work
  - Consolidate individual work
  - Discuss variants
  - Walk-throughs



Detailed Design Process (1)

- Preliminary decisions:
  - Decide on system architecture, e.g. client/server
  - Decide on communication means between system and environment for each message, if not already done
    - Decide how to identify object parameters (i.e. instances of classes of the Concept Model) in messages
      - During design, a class will need to be developed that is responsible of mapping from the identification means to the design class implementing the concept and vice versa.

Detailed Design Process (2)

1. Design the Interaction Model

- Develop a communication diagram or sequence diagram for each system operation in the Operation Model
  - Identify relevant objects involved in the computation
  - Establish role of each object in the computation:
    - Identify controller
    - Identify collaborators
  - Decide on messages (and parameters) and control flow between objects
- At the end, make the following checks:
  - Consistency with the Requirement Model
    - Check that the design-equivalent of each class in the Concept Model is used in at least one interaction diagram.
  - Verification of functional effect
    - Check that the functional effect of each interaction diagram satisfies the specification as defined by the operation schema in the Operation Model.

Detailed Design Process (3)

2. Dependency Model

- Record class dependencies resulting from the design in the Dependency Model
  - For each message in the Interaction Model, a communication path is needed from the client class to the server object
    - Decide on the kind of communication path required, taking into account: its lifetime, the server binding, and reference changeability.
  - Make the following checks:
    - Completeness: All message passing in the Interaction Model must be realized in the Dependency Model.
    - Consistency with the Concept Model: Trace back navigable design associations to requirements associations. Look out for requirements associations not realized in the Dependency Model.
    - Referential Integrity: Look out for multiple associations between the same classes. Are they synonyms? Also: mutual inverse associations must be unified.
- Use the Dependency Model to evaluate your design / explore other options

Detailed Design Process (4)

3. Design Class Model

- The Design Class Model is derived from the Interaction and Dependency Models.
  - Classes from the Dependency Model
  - Methods and parameters from the Interaction Model
  - Value attributes from the Interaction Model (look also at the Concept Model)
  - Navigable associations from the Dependency Model
  - Inheritance relationships from the Inheritance Model (see next slide)
- Perform the following checks:
  - All classes needed by the Interaction Model appear in the Design Class Model.
  - All methods of the Interaction Model are realized in the Design Class Model.
  - All associations shown in the Dependency Model appear in the Design Class Model.
  - There are no non-navigable associations in the Design Class Model.
  - Each navigational association end has a role name.
  - The Design Class Model is consistent with the Inheritance Model.

Detailed Design Process (5)

4. Inheritance Model

- Build the Inheritance Model
  - The inheritance structure is built by identifying commonalities between classes and discovering abstractions. Identify superclasses and subclasses and construct the inheritance diagrams.
    - Examine generalizations and specializations in the Concept Model
    - Look out for methods common to several classes
    - Look out for navigable association ends common to several classes
- Use the Inheritance Model to update the Design Class Model

Questions?