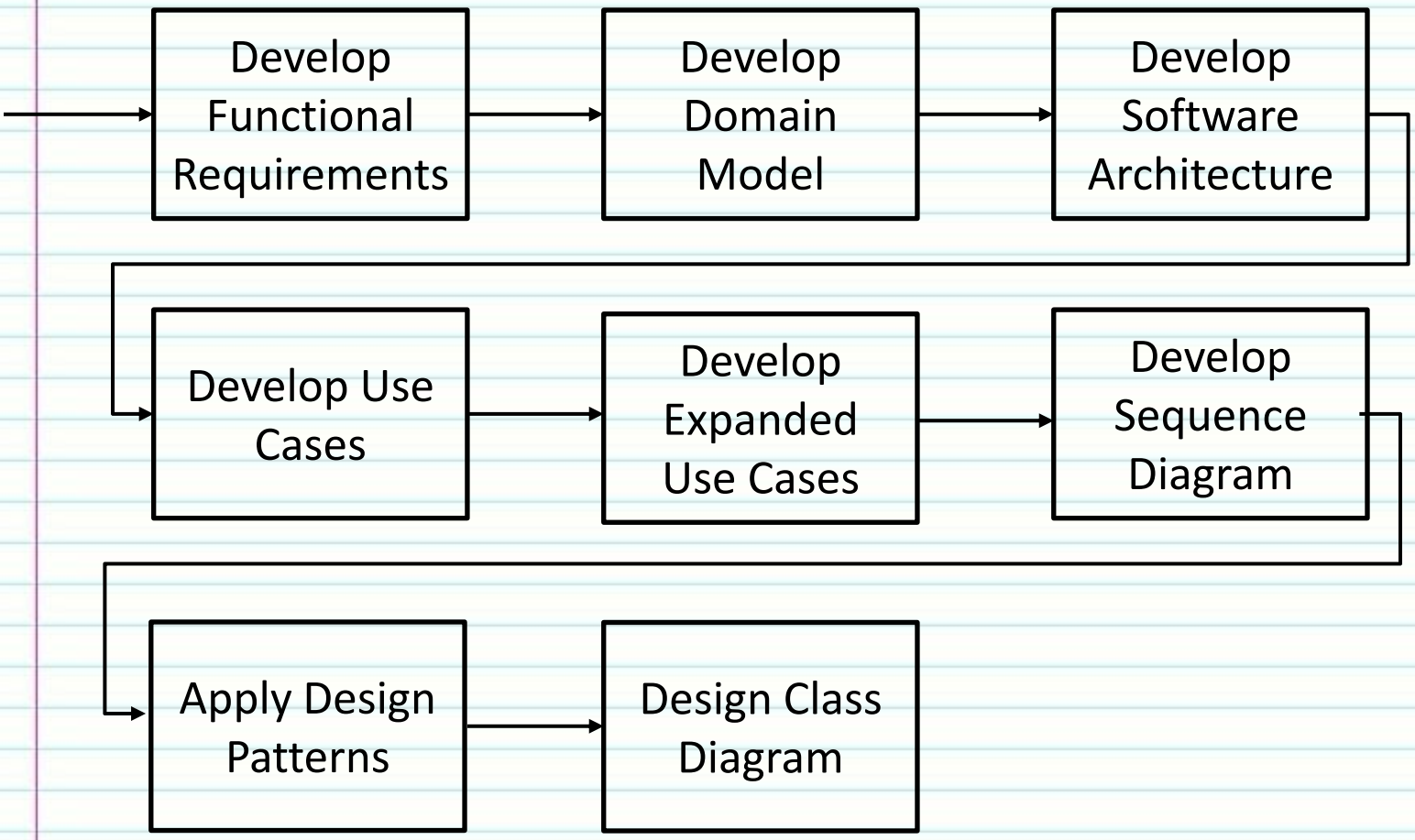
Chapter 11 – Deriving a Design Class Diagram

Dr John H Robb, PMP, SEMC
UT Arlington
Computer Science and Engineering

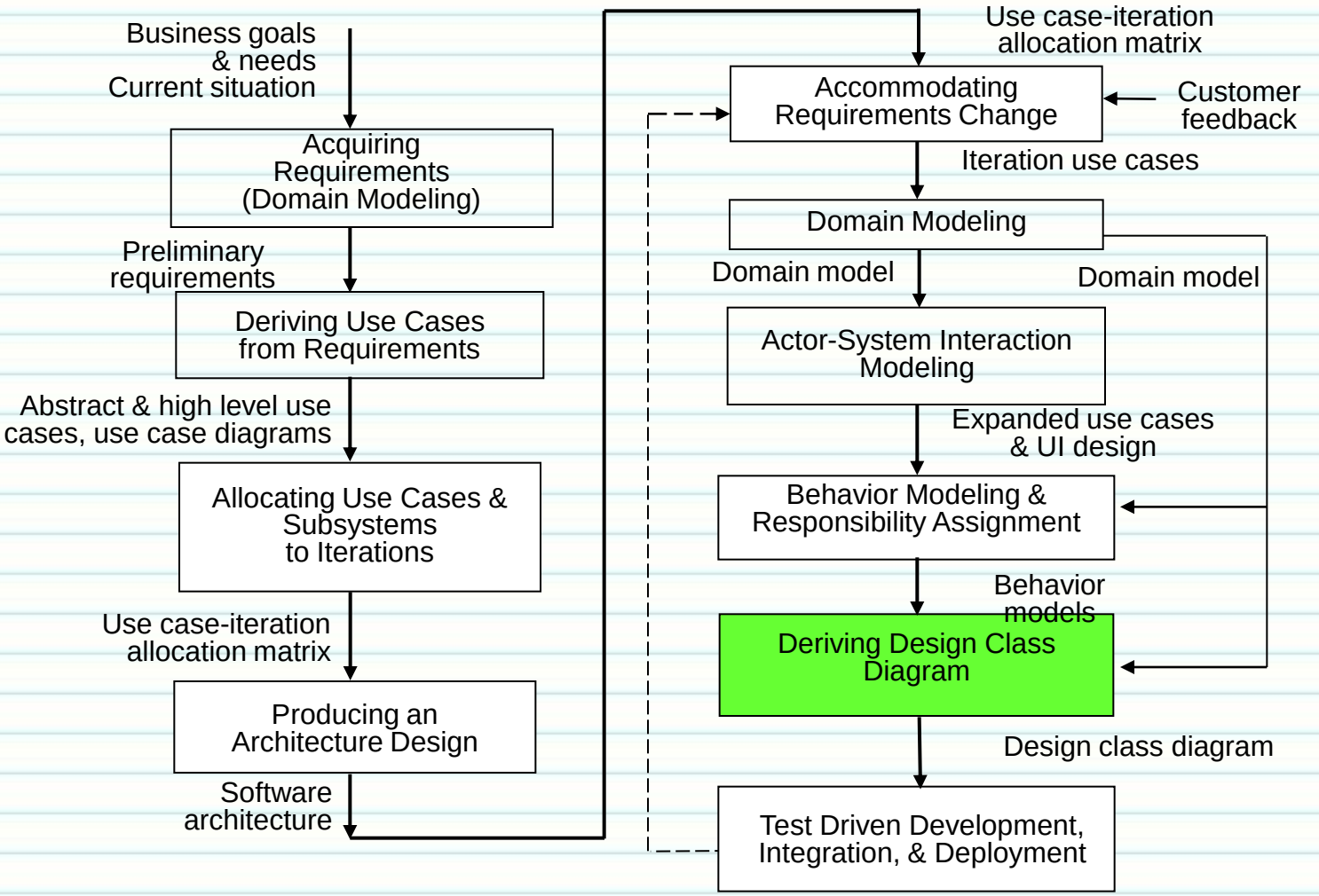
Key Takeaway Points

- A *design class diagram* (DCD) is a UML class diagram, derived from the behavioral models and the domain model.
- It serves as a design blueprint for test-driven development, integration testing, and maintenance.
- Package diagrams are useful for organizing and managing the classes of a large DCD.

Book Approach to OOSE



Deriving DCD in the Methodology Context



Deriving Design Class Diagram

- A design class diagram (DCD) is a structural diagram.
- It shows the classes, their attributes and operations, and relationships between the classes. It may also show the design patterns used.
- It is used as a basis for implementation, testing, and maintenance.
- The collection of the domain model and sequence diagrams does not provide a class structure and road-map to guide subsequent efforts. Information is contained in these diagrams, but is scattered across them.
- The diagram that provides this design specification of the classes and the class structure is the DCD. There is only one DCD for the system, but we add to the DCD based on what is captured in each iteration.

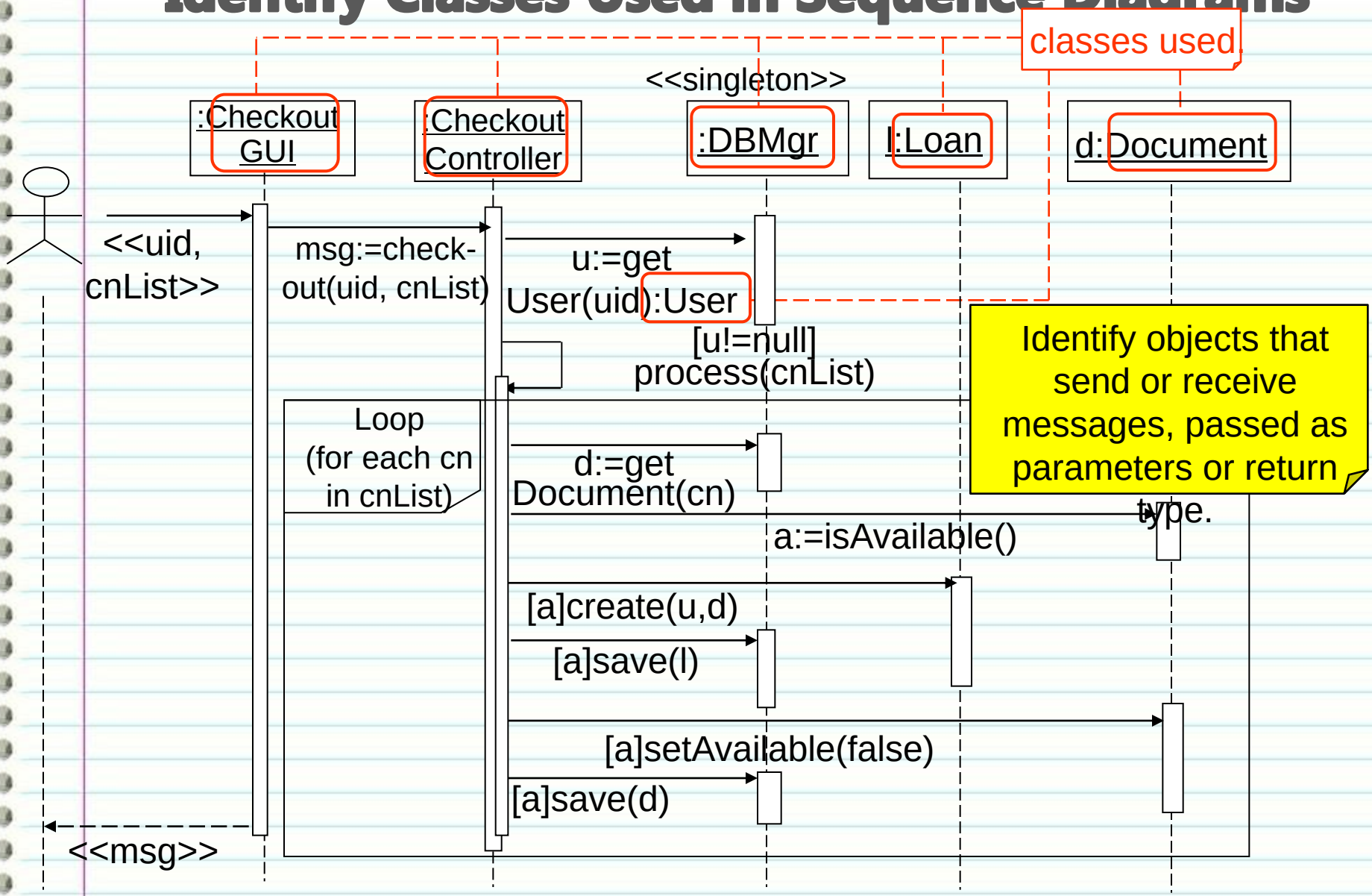
Steps Used to Derive the DCD

1. Identify Classes (from the Design Sequence Diagrams for the current iteration).
2. Identify Methods (from the Design Sequence Diagrams for the current iteration).
3. Identify Attributes (Design Sequence Diagrams and Domain Model).
4. Assess relationships between classes and their implications.

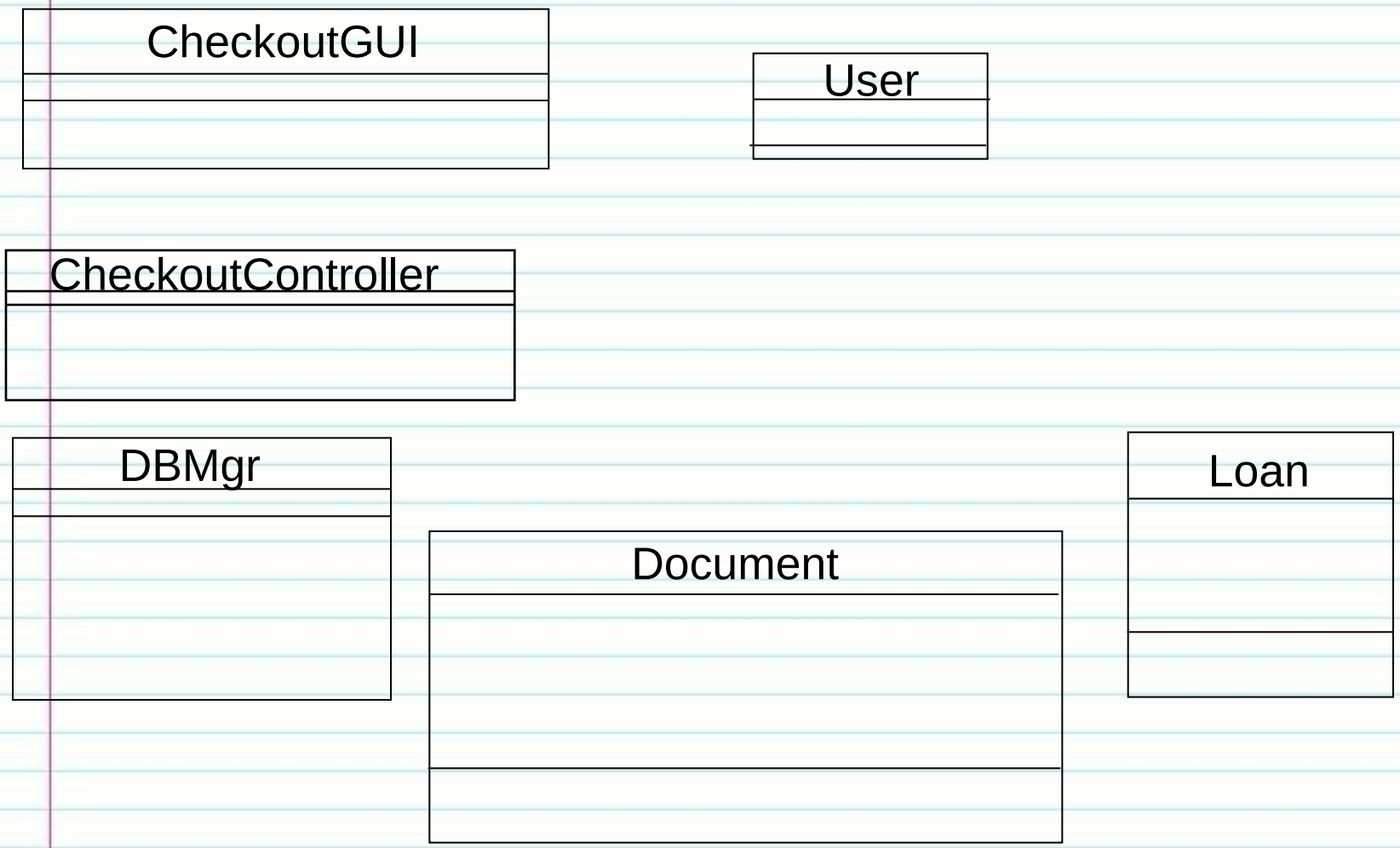
Steps for Deriving DCD - Step 1

- Identify all classes used in each of the sequence diagrams and put them down in the DCD:
  1. Identify classes of objects that send or receive messages (the next slide shows 5 objects)
  2. Identify classes of objects that are passed as parameters or return types/values. Messages between objects represent function calls from one object to another (the next slide shows d:Document, and l:Loan are parameters, so these are identified)
  3. Identify classes that serve as return types. In the next slide Document is the only class used as a return type.
  4. Identify classes from the domain model. Classes from the domain model that are parent classes may be added to the DCD is it promotes understanding.

Identify Classes Used in Sequence Diagrams



Classes Identified



Steps for Deriving DCD - Step 1 (cont.)

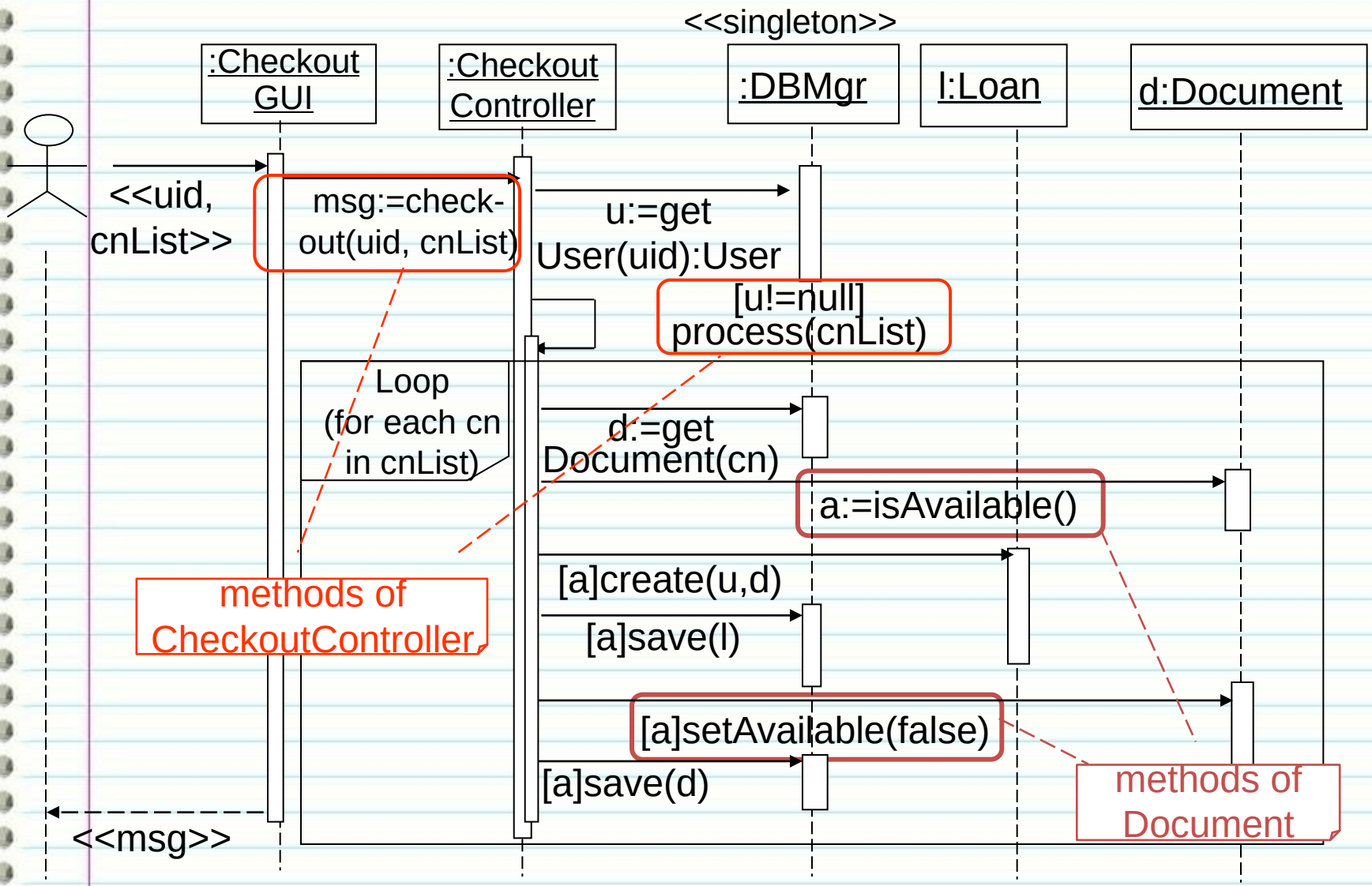
- When identifying these classes keep in mind the following:
  1. These rules should be applied to all of the design sequence diagrams produced in the current iteration, not to just one of the sequence diagrams.
  2. A class may be identified by more than one rule - each class will only be identified once.
  3. Since the DCD serves as the design blueprint for the system, only one DCD will capture all the behavioral models - not one per sequence diagram or each iteration.

Steps for Deriving DCD - Step 2

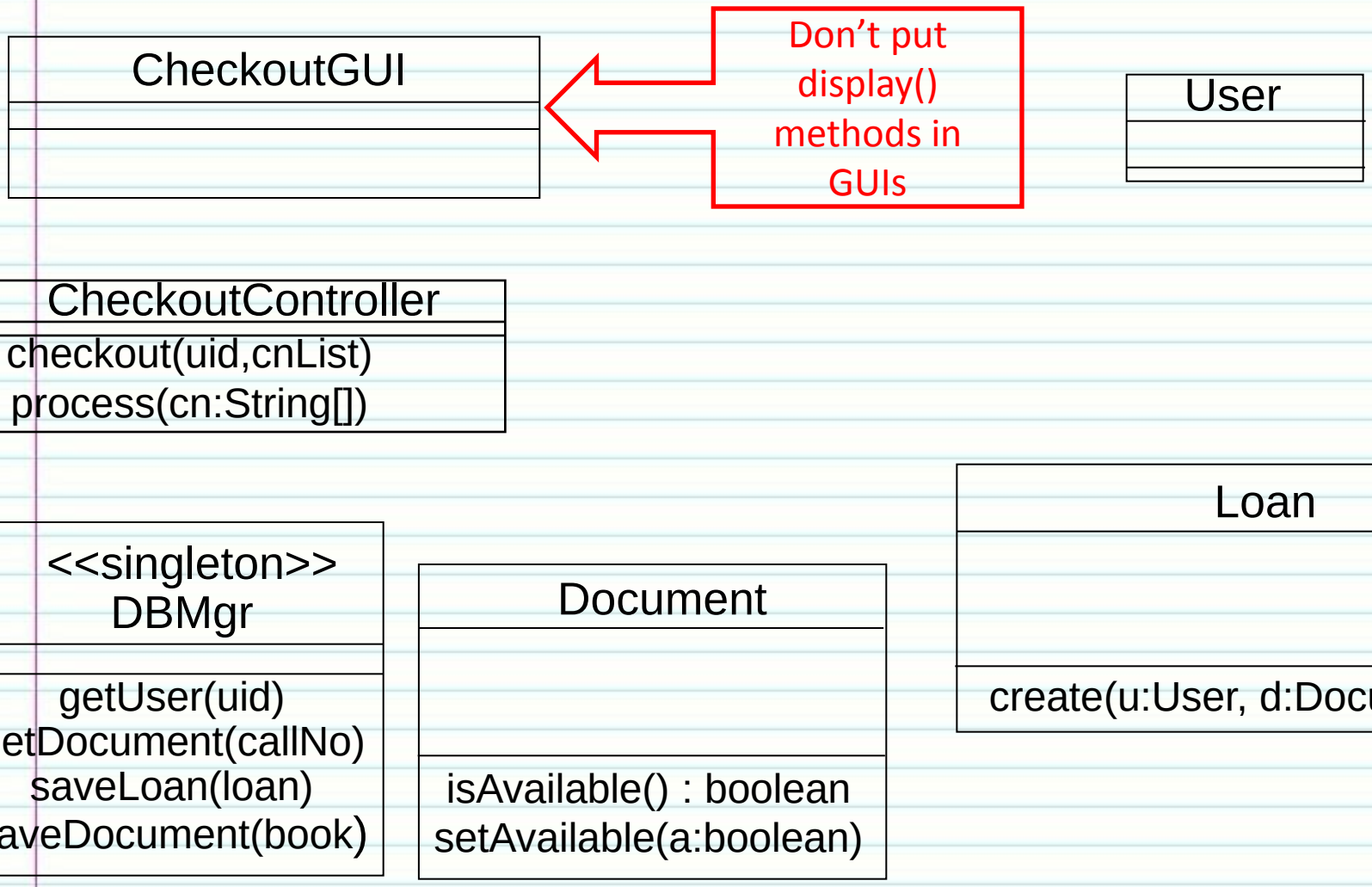
- Identify methods belonging to each class and fill them in the DCD:
  - In a design sequence diagram, a message x:=m(...):X that is sent from object A to object B indicates that A calls the m(...):X function of B and saves the result in variable x of type X. We use this to identify all such methods for each class.
    Methods are identified by looking for messages that label an incoming edge of the object.
  - The sequence diagram may also provide detailed information about the parameters, their types, and return types.

- These are shown on the next slide.

Identify Methods



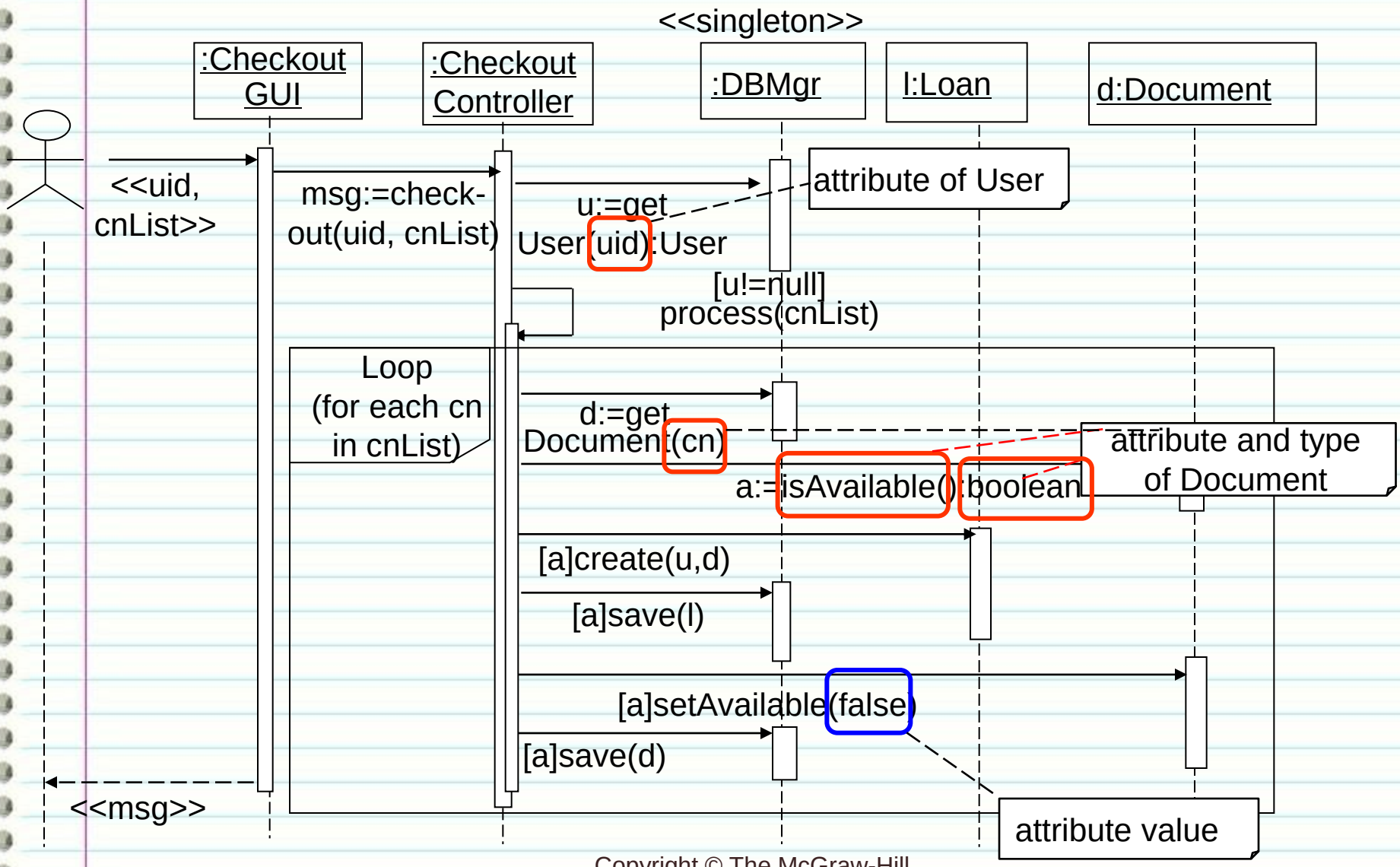
Fill In Identified Methods



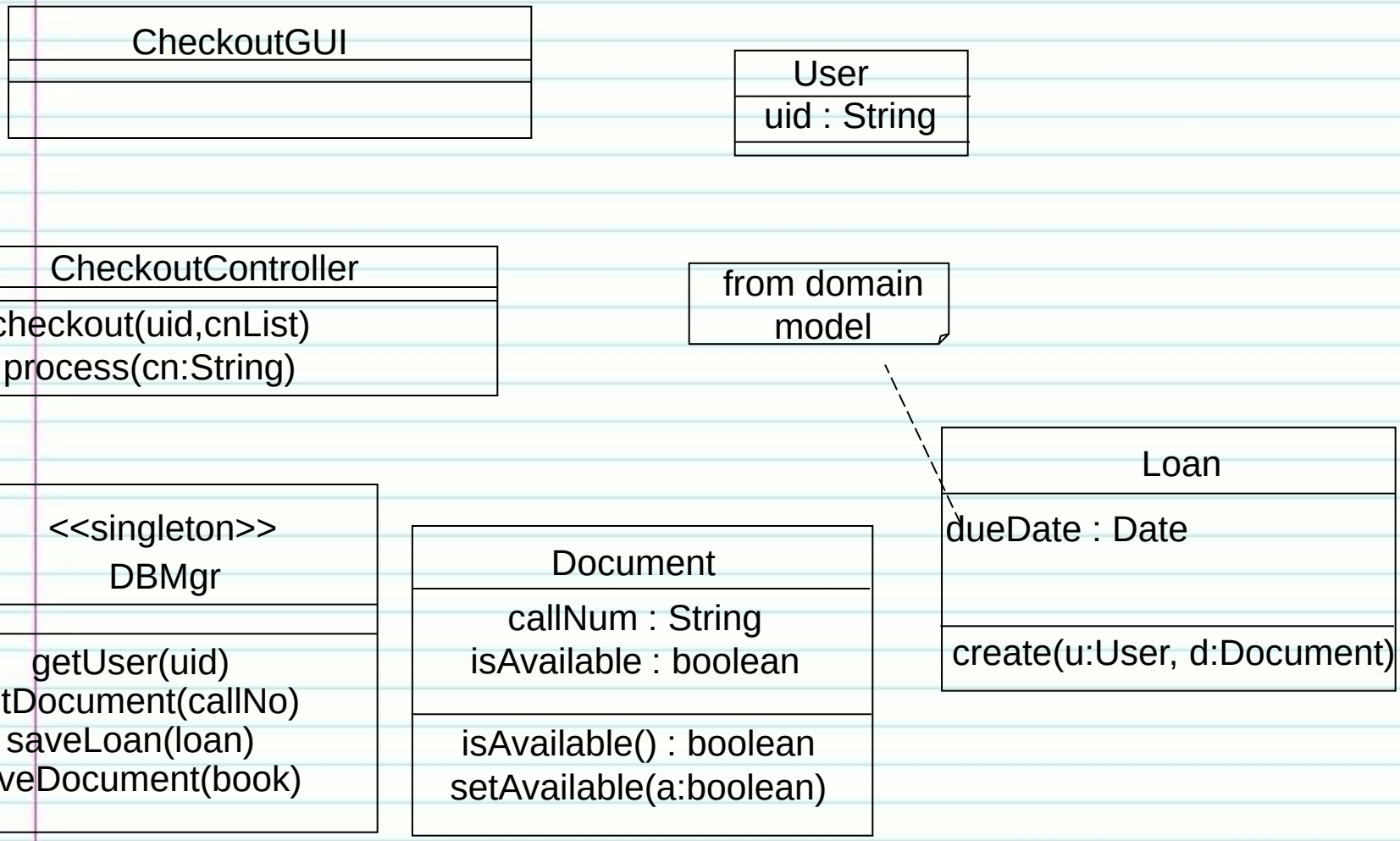
Steps for Deriving DCD - Step 3

- Identify and fill in attributes from sequence diagrams and domain model:
  1. Identify attributes from methods that retrieve objects. A message of the form getX(a:T):X, where X is a class and T is a scalar type, suggests that a is an attribute of X and the type of a is T.
  2. Identify attributes from the normal get and set methods.
  3. Identify attributes from isX():boolean and setX(b:boolean) methods, where X is an adjective. Names such as is and set often imply attributes vs. methods.
  4. Identify attributes from methods that compute a scalar type value. A message of the form computeX(...) or x:=computreX(...):T may suggest that X is an attribute of the object that receives the message. The type of that attribute would be T.
  5. Identify attributes from the parameters to a constructor. A message create(callNo: String, title: String...) the parameters are typically attributes.
  6. Attributes are identified from the domain model. Compare each DCD class with the domain model and identify attributes that are needed by the operations of the DCD

Identify Attributes



Fill In Attributes



Steps for Deriving DCD - Step 4

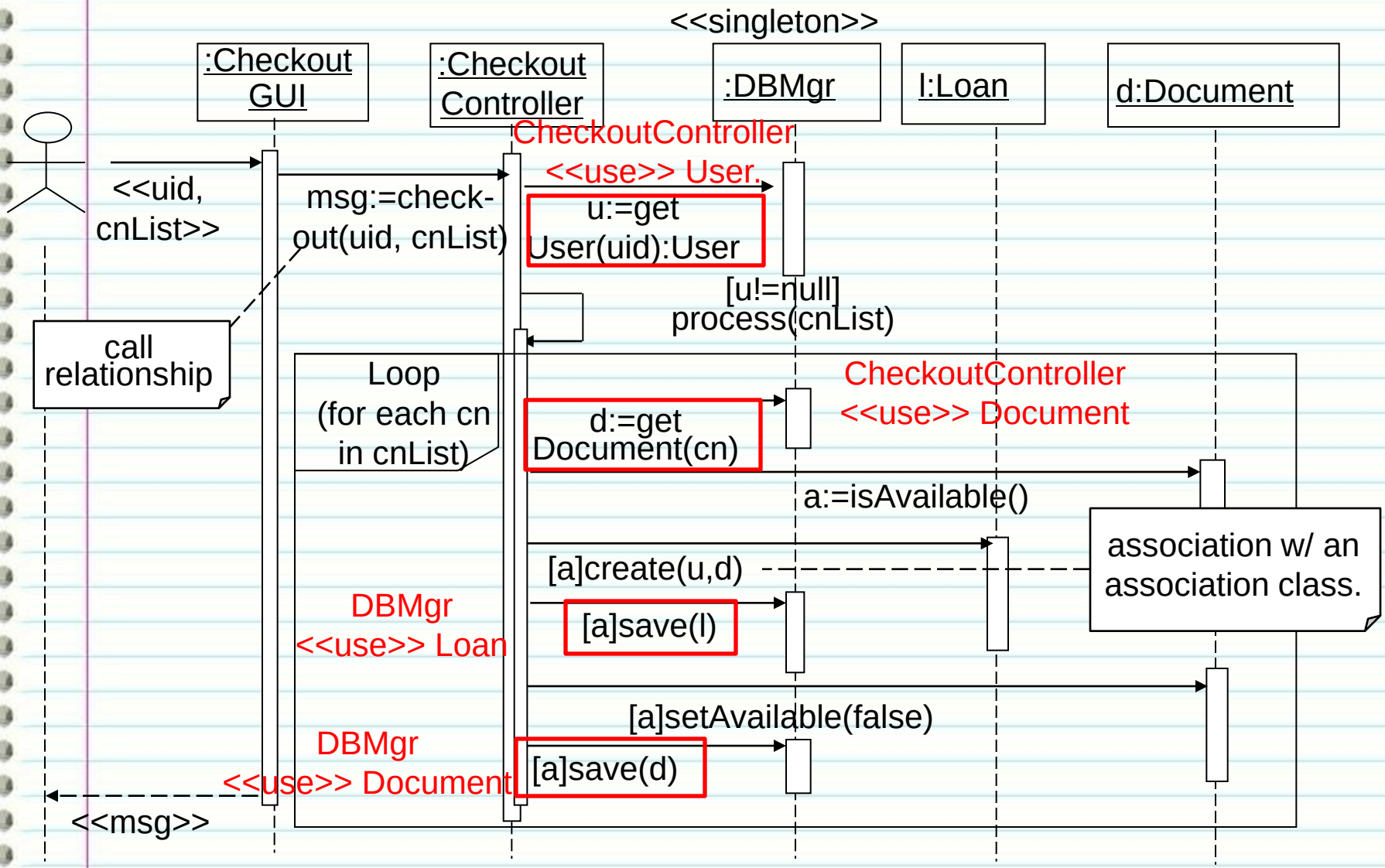
- Identify and fill in relationships from sequence diagram and domain model:
  1. Identify create relationships from calls to constructors. If an object of class A invokes a constructor of Class B, then A creates B.
  2. Identify **use** relationships from classes as parameters or return types. Class A uses class B is one of the following is true:
     a. An object of class A passes an object of class B as a parameter in a function call. On the next slide CheckoutController object calls the save(l:Loan) and save(d:Document) methods of the DBMgr - therefore CheckoutController uses Loan and Document.
     b. An object of class A receives an object of class B as a return value. A receives the B object because it intends to use it - the d:=getDocument(callNo:String): Document call means that CheckoutController uses Document.
     c. Class B is a parameter type of a function of class A - DBMgr uses Loan and Document because save(l:Loan) and save(d:Document) are functions of DBMgr.

Steps for Deriving DCD - Step 4 (cont.)

   d. Class B is the return type of a function of class A. DBMgr uses Document because Document is the return type of the getDocument(...): Document function of DBMgr.

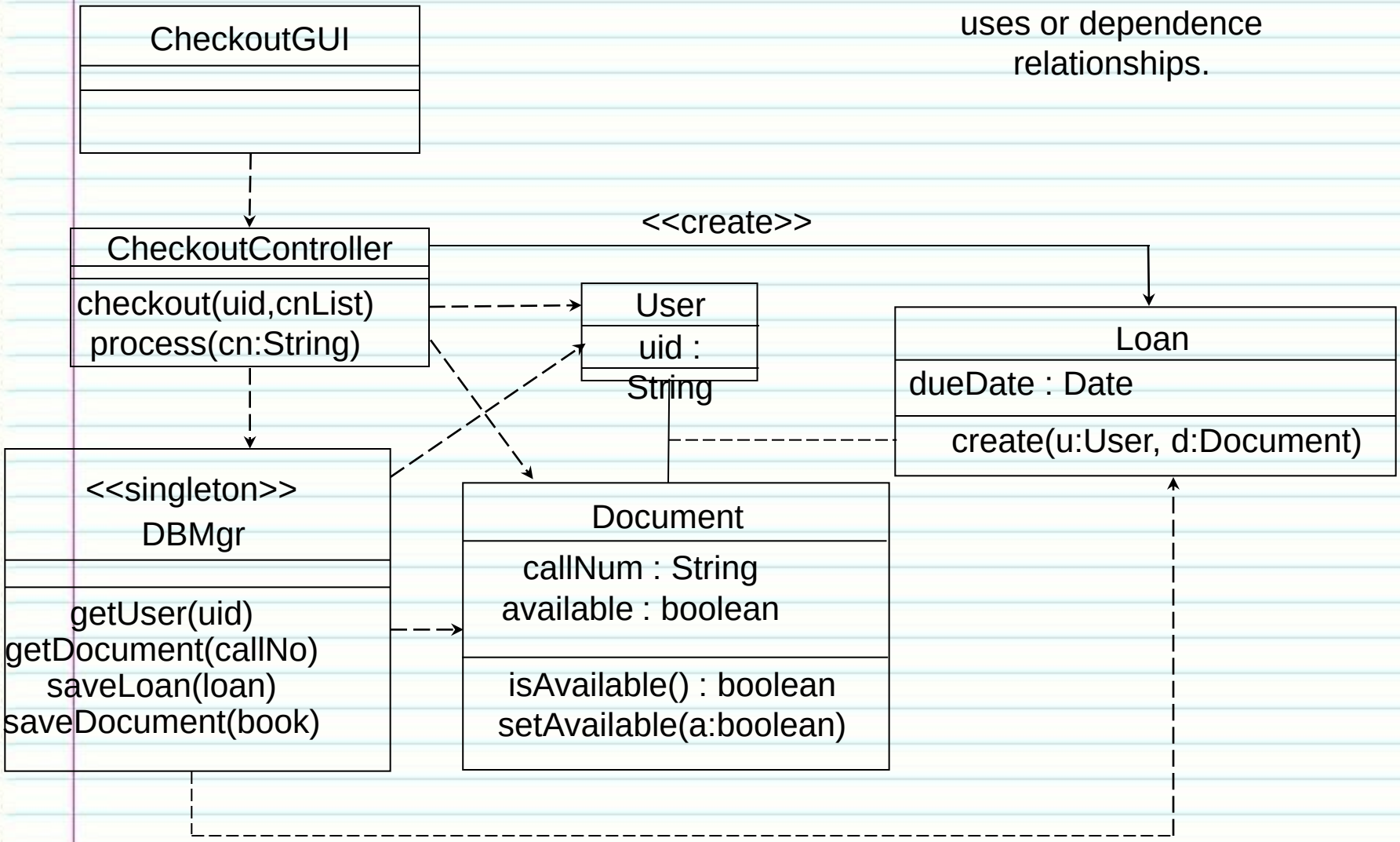
3. Identify association relationships from calls to constructors that pass two objects as parameters. Possible association class. Patron and Document are used to invoke the constructor of a loan.
4. Identify relationships from get object and set object messages.
   a. A is an aggregation of B if an object of B is a part of an object of A.
   b. A uses B, as explained in item 2 above
   c. A associates with B if A and B are neither of the two above.
5. Identify call relationships.
6. Identify containment relationships from messages to add, get, or remove objects. These are messages of the form add(b: B), get(…):B, getB(…):B, etc.
7. Identify additional relationships from the domain model. Inheritance, aggregation, and association relationships.

Identify Relationships

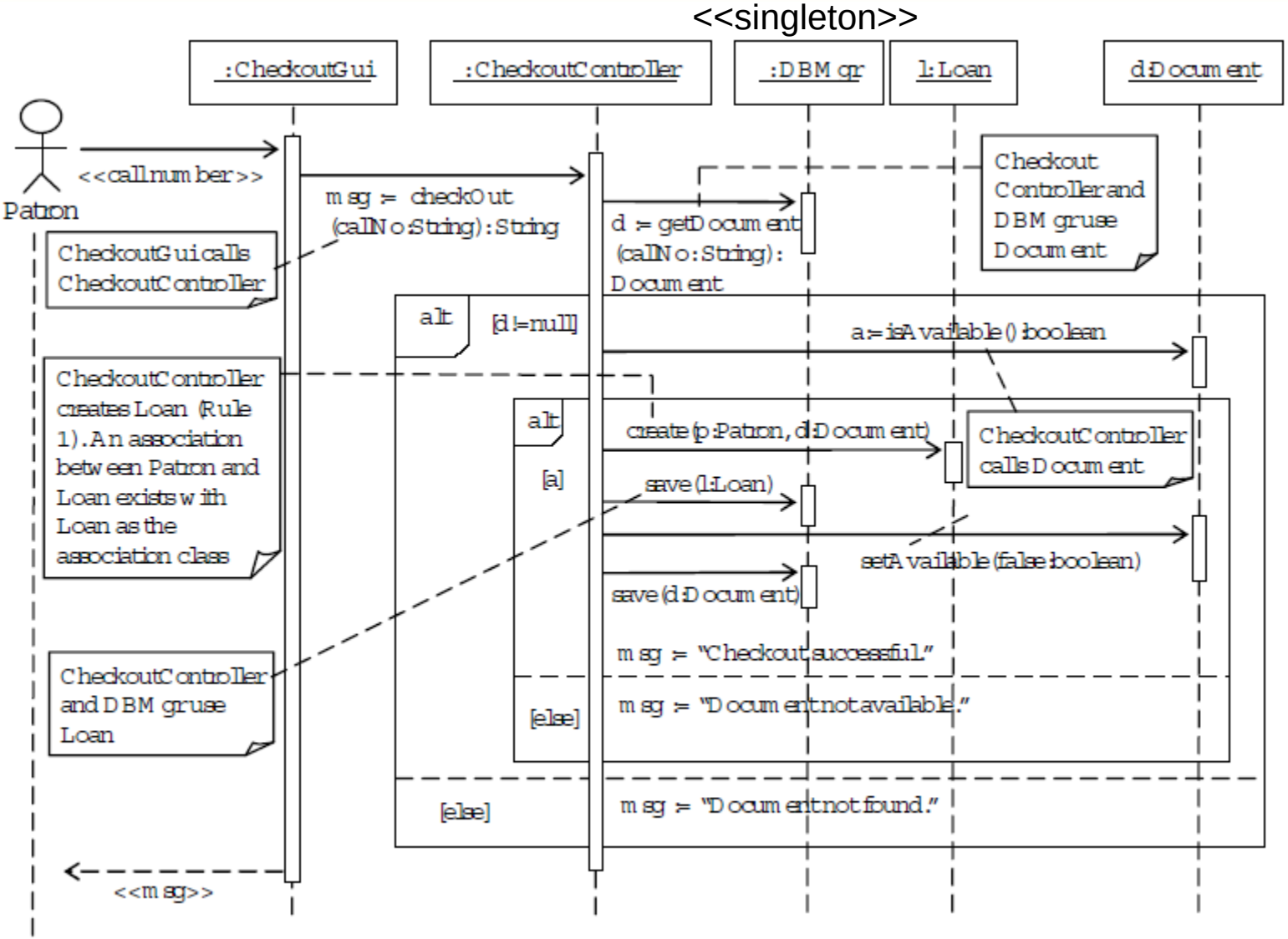


Fill In Relationships

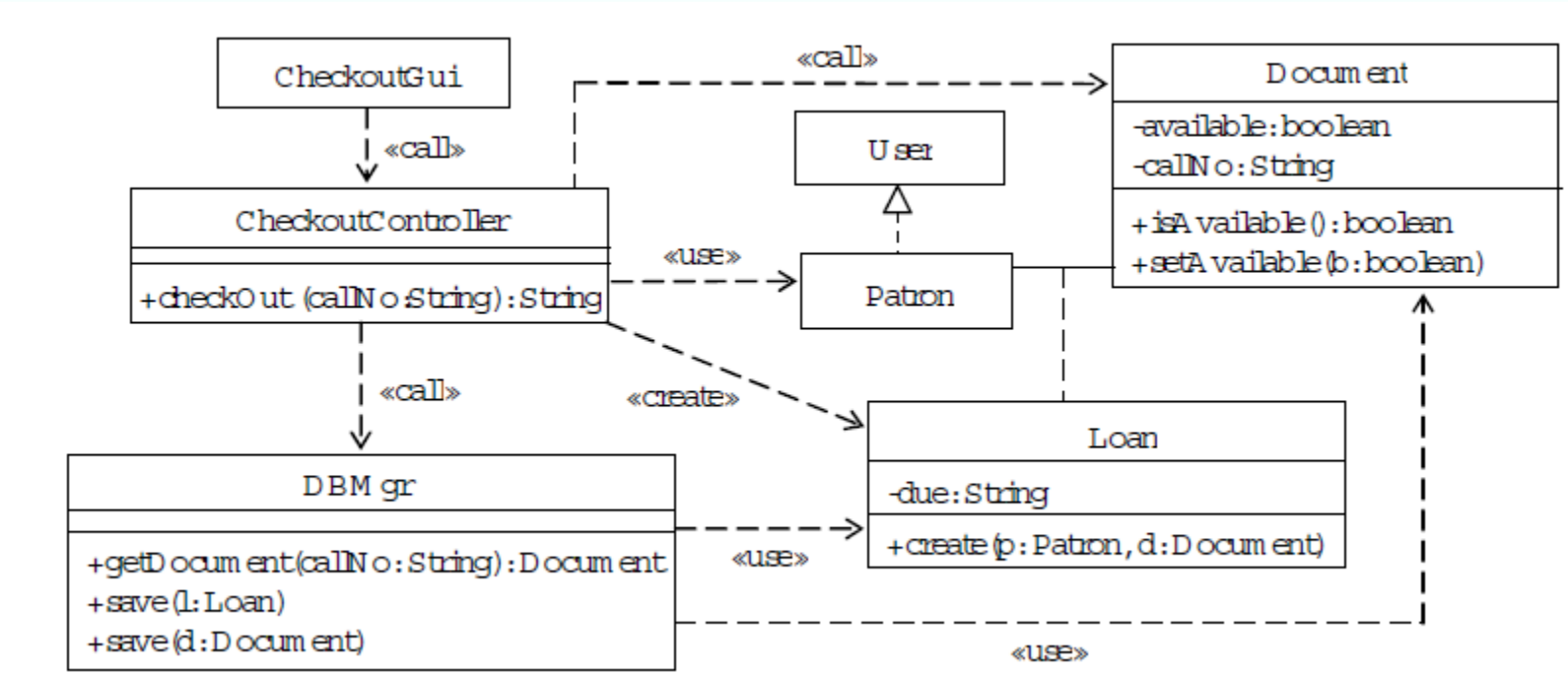
The dashed arrow lines denote uses or dependence relationships.



From Sequence Diagram to Implementation

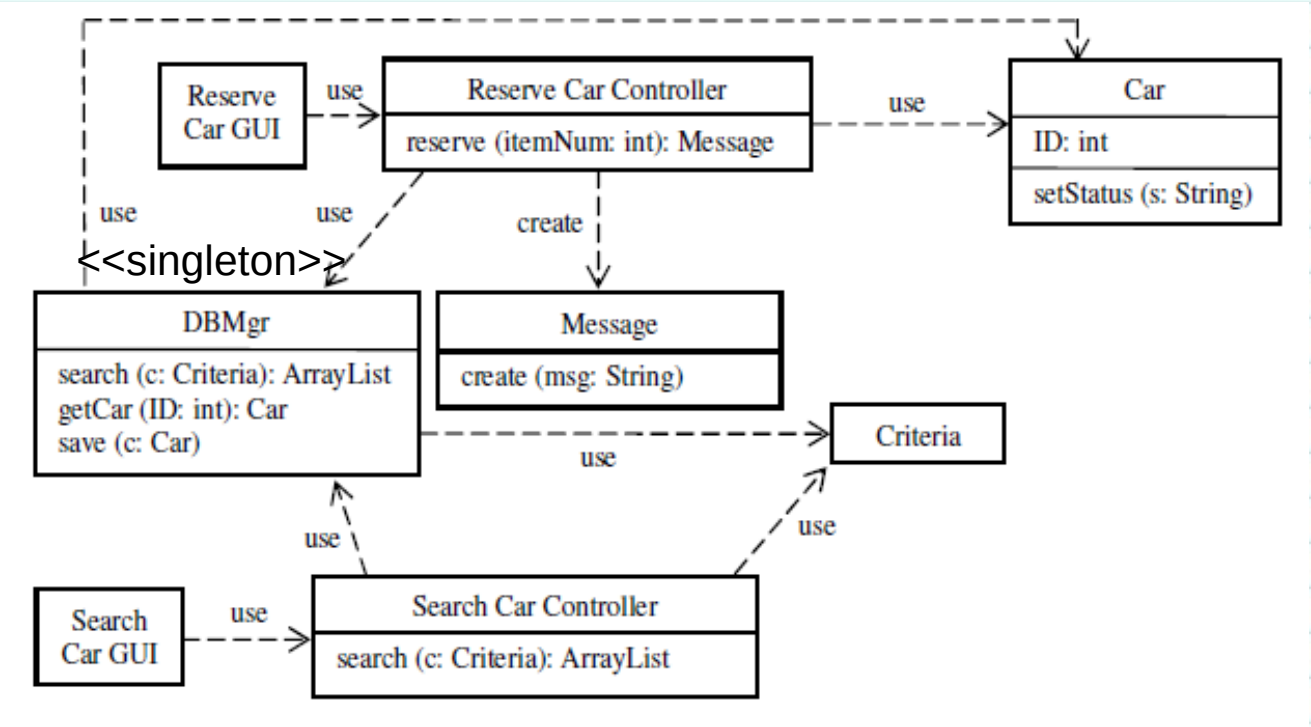


The Complete DCD



Another DCD

- This is for the first two Use Cases of the Car Rental System (App D.1)



DCD Review Checklist

- Ensure that the classes, attributes, operations, parameter types, return types, and relationships in the DCD are derived correctly according to the steps presented in this chapter.
- Does the DCD contain unnecessary classes, operations, or relationships?
- Does the naming of the classes, attributes, operations, and parameters communication concisely the intended functionality and is it easy to understand?
- Does the DCD clearly indicate the design patterns used?
- Compute metrics such as fan-in, fan-out, class-size, depth in inheritance tree, and coupling between classes and identify potential problems. (Chapter 19).

Common Mistakes on DCD

1. Not starting with Domain Diagram when developing the DCD
2. Leaving off <<singleton>> on DBMgr
3. Putting attributes in the wrong place - Controller vs System User
   a. use the DD to tell you where the attribute goes
   b. only add SD attributes that are not already on the DD
4. Forgetting the <<create>> for DD aggregations
5. Forgetting the <<create>> for DD association classes (remember message is from Controller to the association class
6. Classes that have verb names - e.g., Login
7. Putting SD methods in the wrong class - should be in the message recipient class, not sender
8. <<use>> relationship not correct - see slide 19