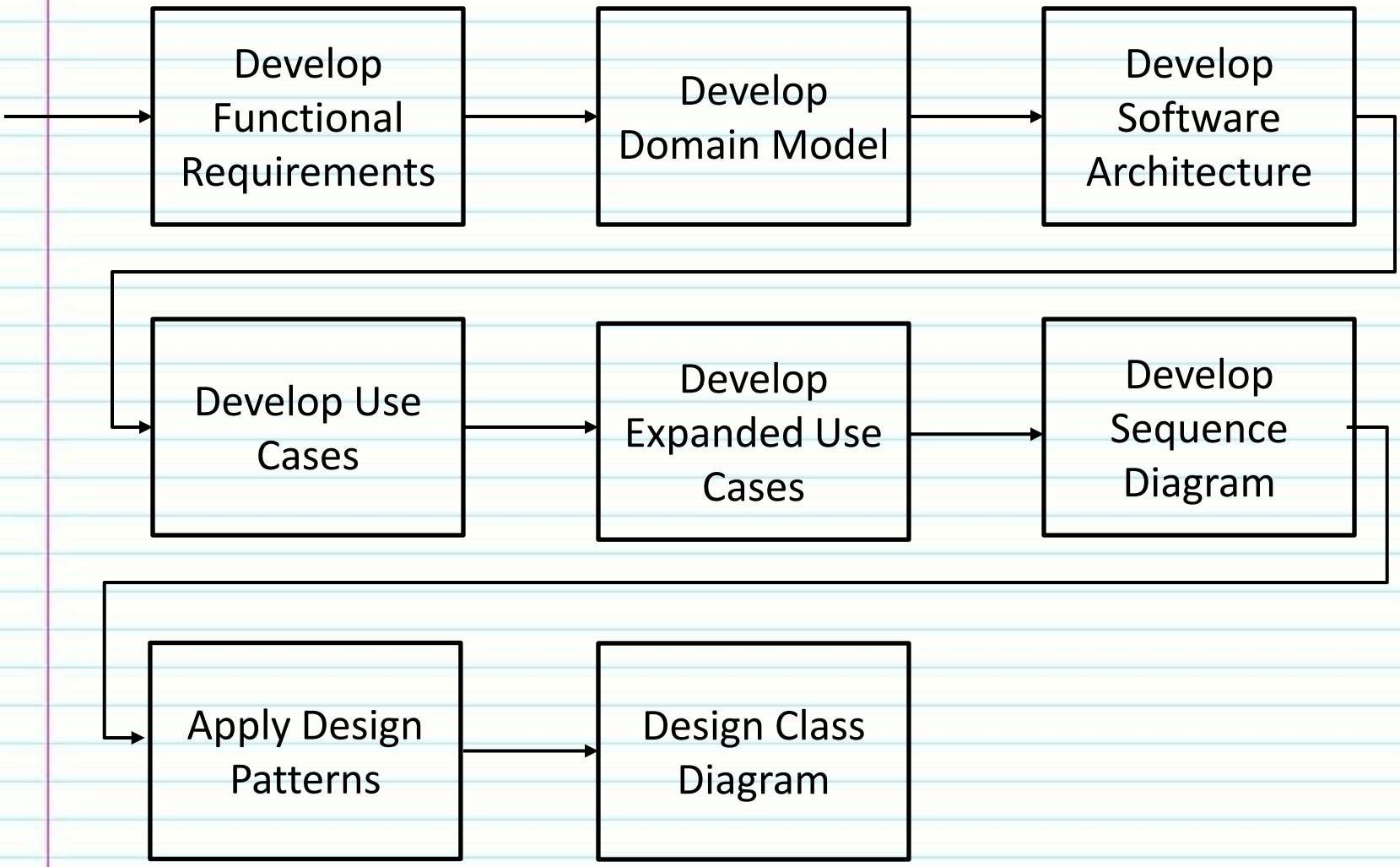
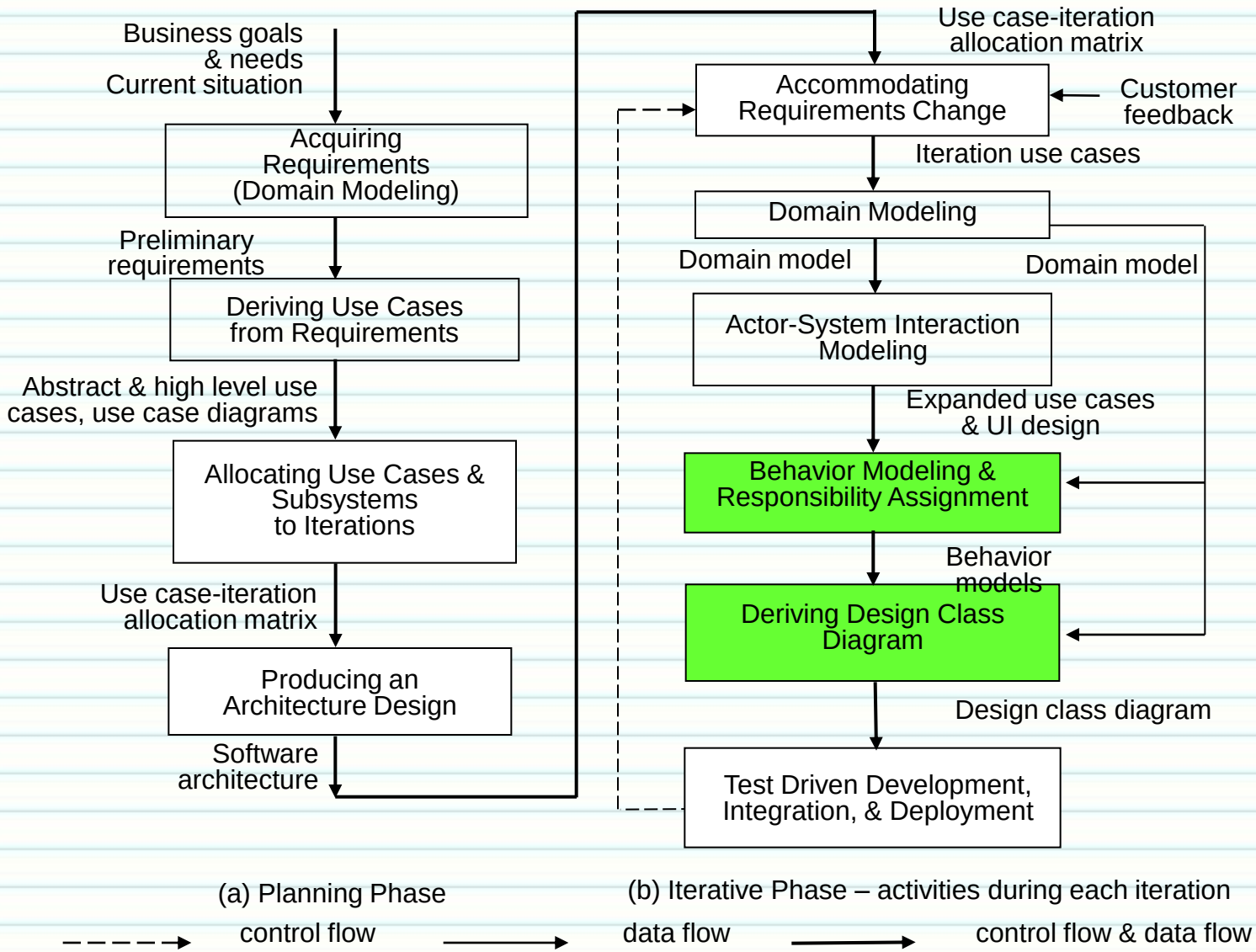
Chapter 10 - Design Patterns

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

Book Approach to OOSE



Applying Patterns in the Methodology Context



What Are Design Patterns?

- Design patterns are proven design solutions to commonly encountered design problems.
- Each pattern solves a class of design problems.
- Design patterns codify software design principles and idiomatic solutions.
- Design patterns improve communication among software developers.
- Design patterns empower less experienced developers to produce high-quality designs.
- Patterns can be combined to solve a large complex design problem.

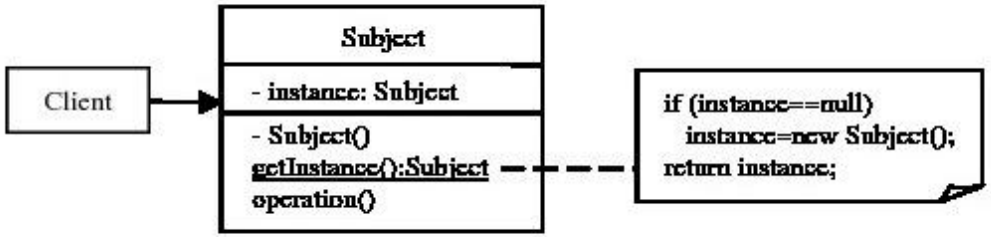
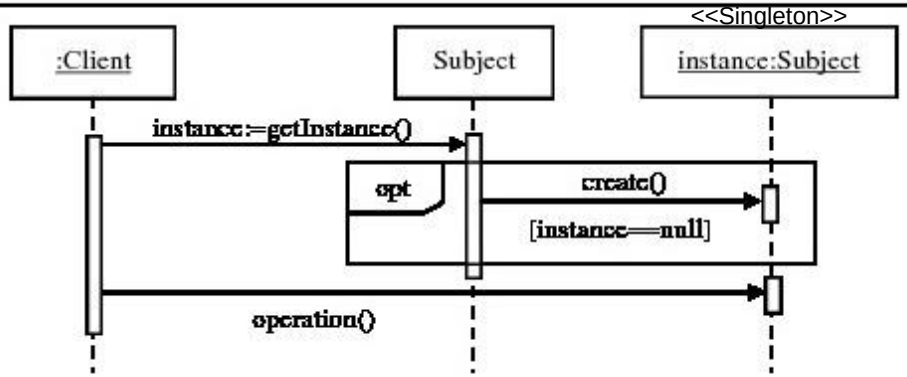
Describing Patterns

- The pattern name conveys the design problem as well as the design solution.
- Example: Singleton
  - How to design a class that has only one globally accessible instance?
  - The *singleton* pattern provides a solution.
- Pattern description also specifies
  - benefits of applying the pattern
  - liabilities associate with the pattern, and
  - possible trade-offs

Example: The Singleton Pattern

- Pattern name: Singleton
- Design Problem: How do we ensure that a class has only one globally accessible instance?
- Example uses:
  - System configuration class
  - System log file

Specification of the Singleton Pattern

Name	Singleton
Type	GoF/Creational
Specification	
Problem	How to design a class that has only a limited number of globally accessible instances?
Solution	Make the constructor of the class private and define a public static method to control the creation of the instances.
Design	
Structural	
Behavioral	
Roles and Responsibilities	• Subject: It provides a public static getInstance() method for the client to retrieve its instance. • Client: It calls the getInstance() method of Subject to retrieve the instance and calls its operation().
Benefits	• It limits the number of instances. • It supports global access to the instance(s).
Liabilities	• Concurrent update to the shared instance may cause unwanted effect.
Guidelines	
Related Patterns	• Singleton limits the number of instances of a class. Flyweight supports numerous occurrences of an object. Prototype reduces the number of classes. • Visitor is often a Singleton.
Uses	Singleton is used in many applications.

The Singleton Pattern

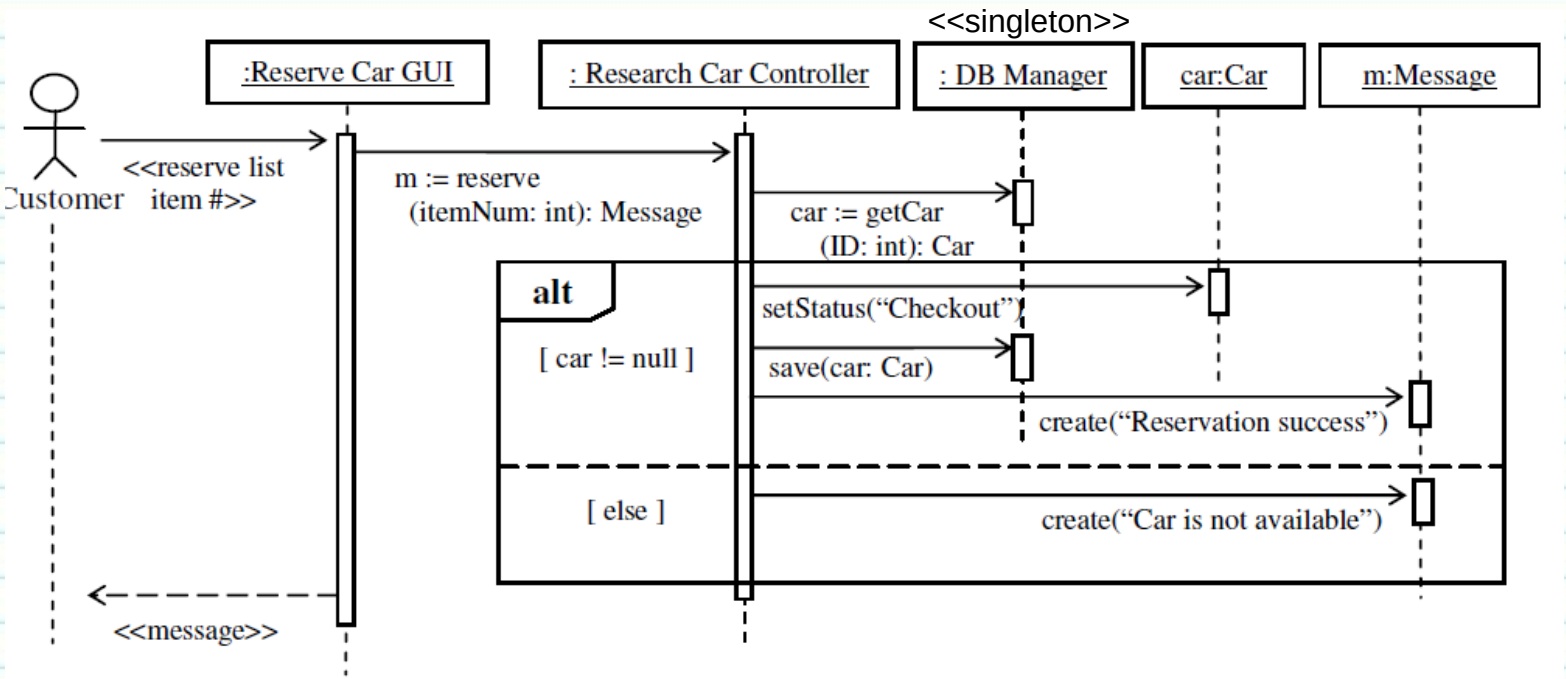
```
public class Catalog {
  private static Catalog instance;
  private Catalog() { ... } // private constructor
  public static Catalog getInstance() {
    if (instance==null) instance=new Catalog();
    return instance;
  }
  // other code
}
```

When to Use a Singleton

- There are a few cases where is makes sense to use a Singleton:
  - Hardware interface access: Singleton can be used in case external hardware resource usage limitation required e.g. Hardware printers where the print spooler is a singleton (to avoid multiple concurrent accesses and creating deadlock).
  - Logger : If there are multiple client applications using the logging utility class they might create multiple instances and cause issues during concurrent access to the same logger file.
  - Configuration File: Here we create a single instance of the configuration file which can be accessed by multiple calls concurrently.
  - Cache: Cache can be used as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the single object.

Correct Depiction of the Singleton

- Correct depiction always shows the stereotype <<singleton>> as below - make sure your student projects capture this please



Commonly Used Design Patterns

- The General Responsibility Assignment Software Patterns (GRASP)
- The Gang of Four Patterns due to the four authors of the book.

GRASP Patterns

- Expert
- Creator
- Controller
- Low coupling
- High cohesion

- Polymorphism
- Pure fabrication
- Indirection
- Do not talk to strangers

Our text book for this semester addresses:

1. Singleton
2. Creator
3. Controller
4. Expert (or Information Expert)

Gang of Four Patterns

- Creational patterns deal with creation of complex, or special purpose objects.
- Structural patterns provide solutions
  for composing or constructing large, complex structures that exhibit desired properties.
- Behavioral patterns are concerned with
  - algorithmic aspect of a design
  - assignment of responsibilities to objects
  - communication between objects

The GoF Patterns

Creational Patterns

- Abstract factory
- Builder
- Factory method
- Prototype
- Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns

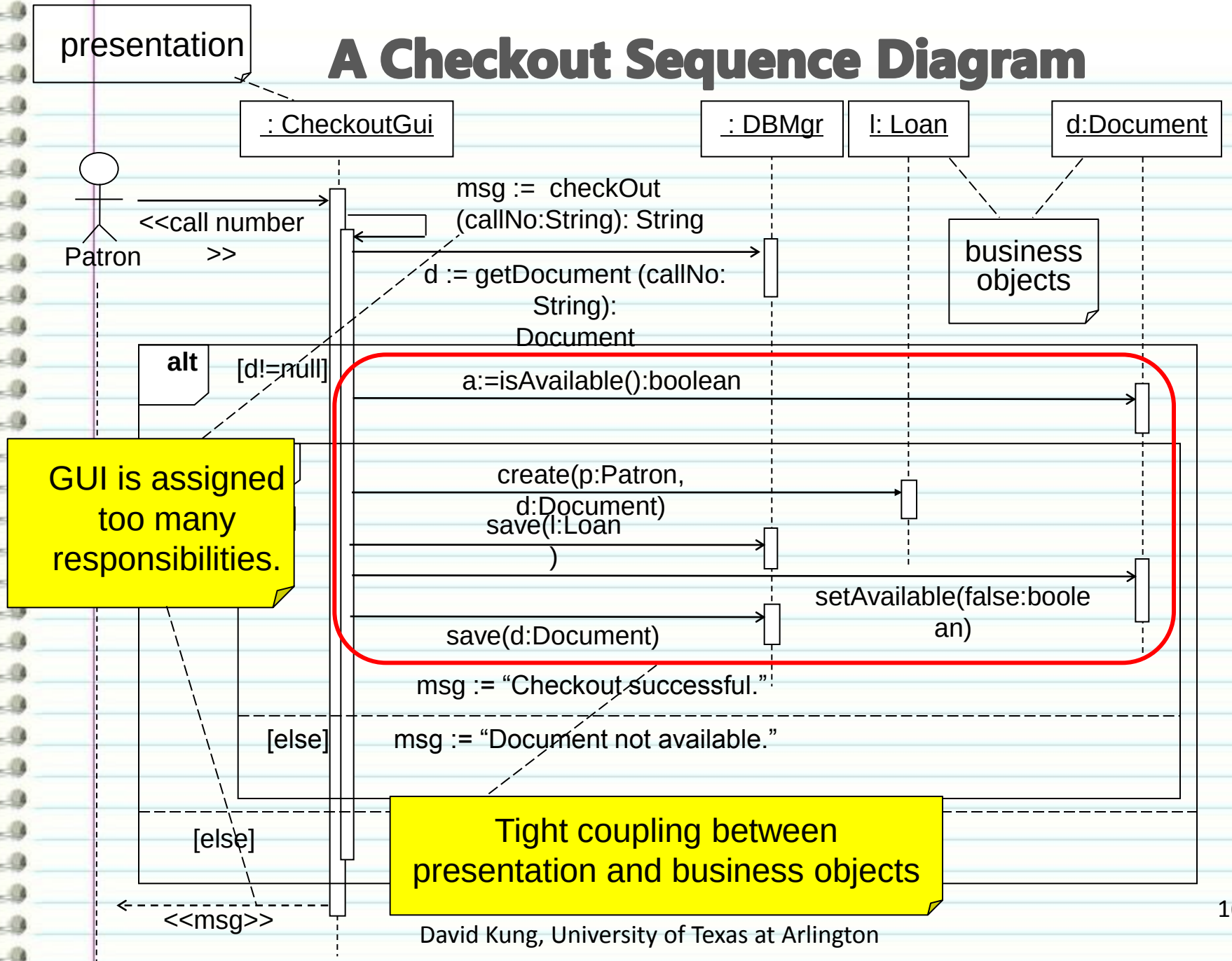
- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

CSE 5322 - Software Design Patterns addresses these and other Design Patterns

Applying GRASP through a Case Study

- Examine a commonly seen design.
- Discuss its pros and cons.
- Apply a GRASP pattern to improve.
- Discuss how the pattern improves the design.
- During this process, software design principles are explained.

A Checkout Sequence Diagram



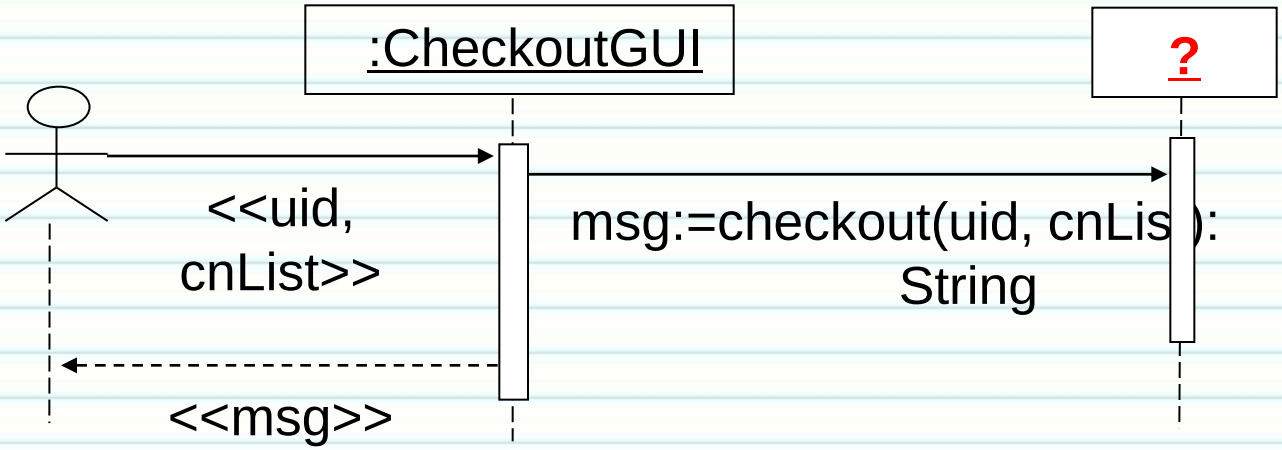
Problems with This Design

- Tight coupling between the presentation and the business objects.
- The presentation has been assigned too many responsibilities.
- The presentation has to handle actor requests (also called system events).
- Implications
  - Not designing “stupid objects.”
  - Changes to one may require changes to the other.
  - Supporting multiple presentations is difficult and costly.

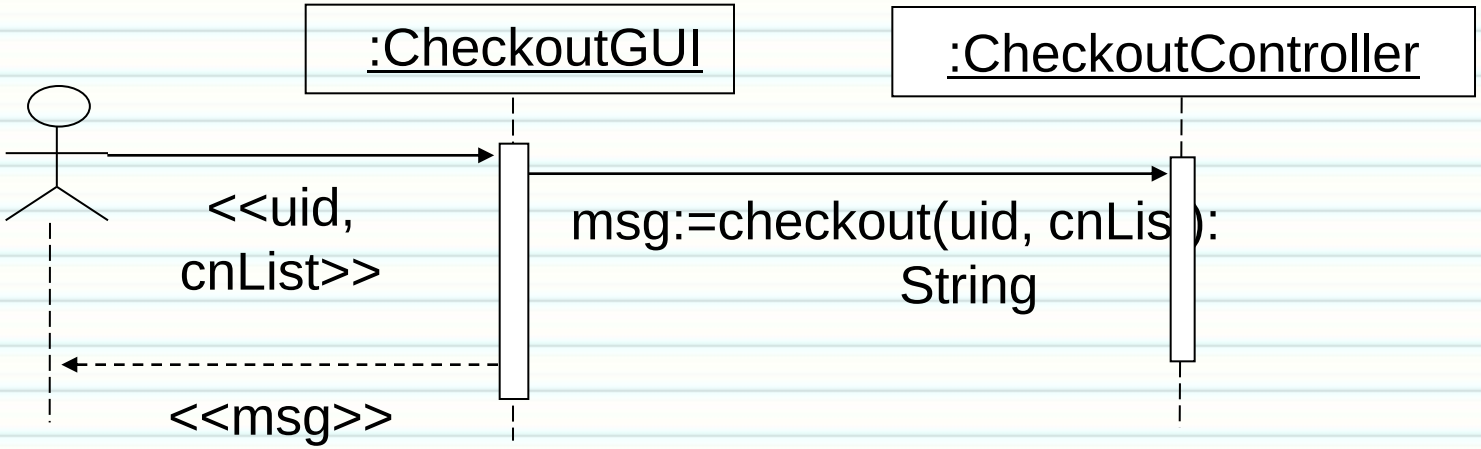
A Better Solution

- Reduce or eliminate the coupling between presentation and business objects.
  - the Low Coupling design principle
- Remove irrelevant responsibilities from the presentation.
  - the separation of concerns principle
  - it achieves high cohesion and
  - designing “stupid objects”
- Have another object (class) to handle actor requests (system events).

Who Should Handle an Actor Request?



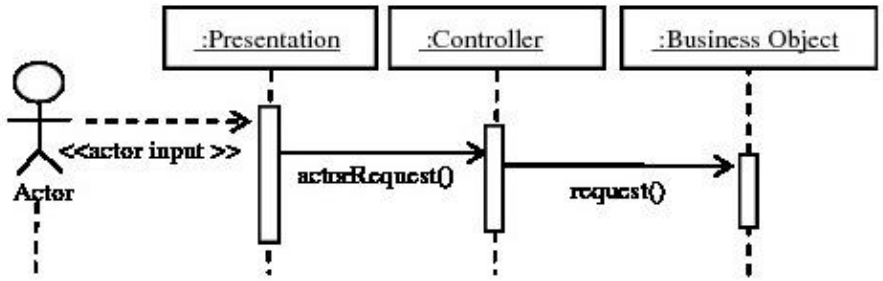
Assign the responsibility for handling an actor request to a controller.



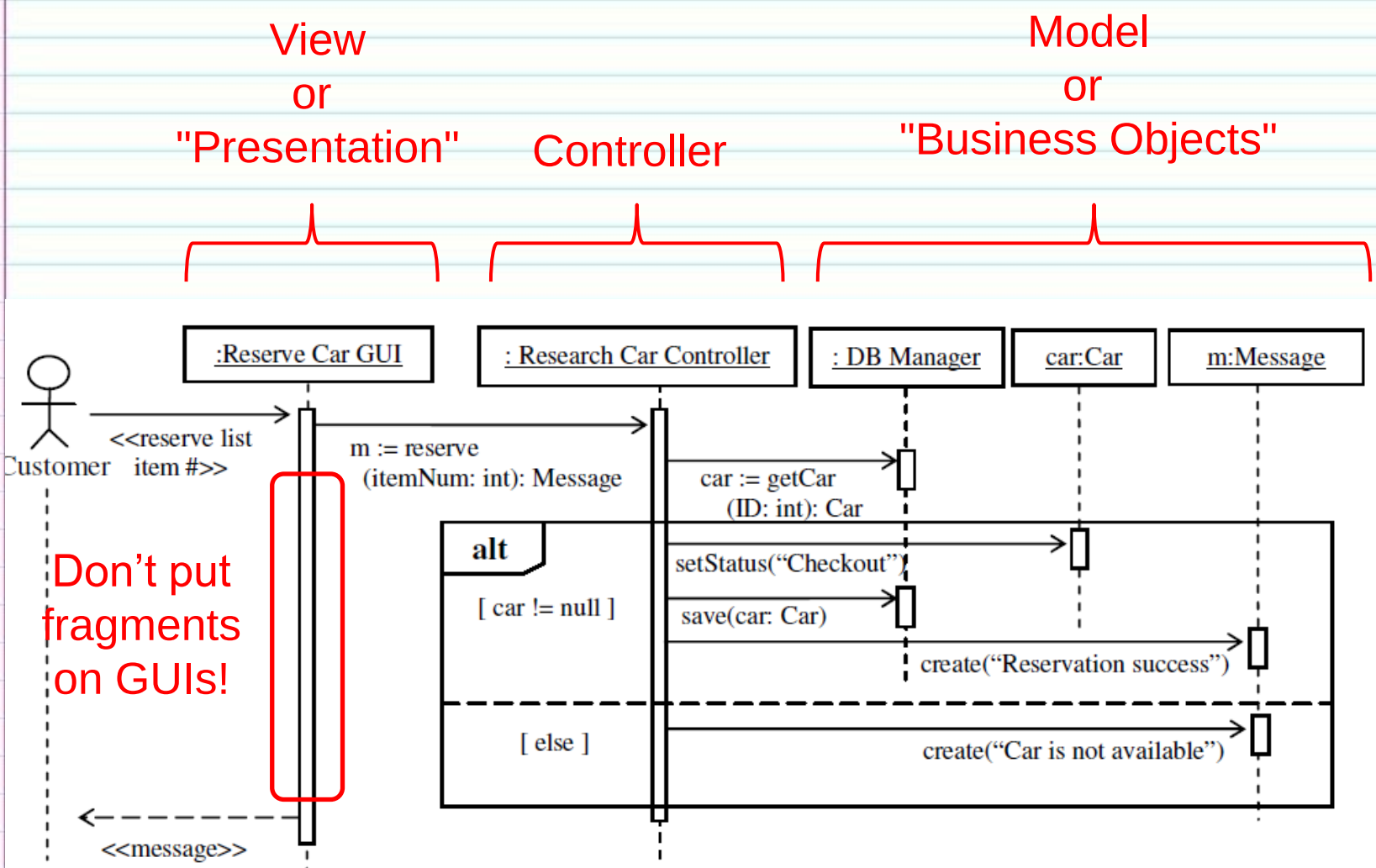
The Controller Pattern

- Actor requests should be handled in the business object layer.
- Assign the responsibility for handling an actor request to a controller.
- The controller may delegate the request to business objects.
- The controller may collaborate with business objects to jointly handle the actor request.
- It supports the following design principles
  1. Design for change. It insulates the changes to business objects/logic from the presentation (GUI).
  2. Separation of concerns. The GUI only deals with the presentation aspects while the controller is responsible for processing.
  3. High Cohesion. The segregation into a presentation and processing class helps to increase cohesion.
  4. Designing "stupid objects". Each object has a specific focus and knows only how to do that task.

Specification of the Controller Pattern

Name	Controller
Type	General responsibility assignment
Specification	
Problem	Who should be responsible for handling an actor request?
Solution	Assign the responsibility to handle the request to a dedicated class called the controller.
Design	
Structural	
Behavioral	
Roles and Responsibilities	• Business Objects: Object classes responsible for the business logic of an application. • Controller: A class dedicated to handle designated actor requests. It takes requests from the Representation and works with the Business Objects to fulfill the request. A use case controller is dedicated to handle all actor requests of a given use case. • Representation: An interface class responsible for interacting with actors of the system. It delegates the actor requests to the Controller and delivers the responses from the Controller to the actor.
Benefits	• It decouples the Representation and the Business Objects. • It reduces the change impact of Representation and Business Objects to one and other. • It supports multiple Representations.
Liabilities	A controller may be assigned too many responsibilities, resulting in a so-called bloated controller. A bloated controller is complex, difficult to understand, implement, test and maintain.
Guidelines	• Adopt use case controllers whenever possible. • Avoid using one controller for more than one use cases.
Related Patterns	Controller is a special case of the Model-View-Controller or MVC pattern.
Uses	In the design of all interactive systems to decouple the representation from business objects.

Sequence Diagrams - Car Rental Example (cont.)



Discussion: DB Manager interacts with Car&Message, interaction w Customer

Types of Controller

- Use case controller
  - It handles all actor requests relating to a use case.
    - A checkout controller handles all actor requests to checkout a document.
    - This is the most common - especially for our class.
- Facade controller
  - It represents the overall system (Library System, Banking System).
  - It represents the organization (Library, Bank).
  - It is used when there are only a few use cases in the entire system.
- Connections with other objects in the sequence diagrams
  - The controller is almost always the object(s) that have the most connections with the rest of the system
  - It represents much of the code and cyclomatic complexity of the software - it will have the most methods

When to Apply the Controller Pattern

1. Apply when writing the use case scenario. The best time to apply this pattern is when the use case is being written - no rework is required to apply this pattern afterward.
2. Apply by modifying an existing scenario. In this case the scenario can be modified to use the controller pattern.
3. Apply when constructing the scenario table. Although not as attractive as applying it earlier, it still offers the advantages of good design principles described earlier.
4. Apply when constructing the design sequence diagram.
5. Apply by modifying a design sequence diagram.

- In all cases when modifying an existing product - care must be taken to make all previous (and subsequent work) consistent.

Applying Use Case Controller

- Applying role controller or facade controller
  - When there are only a few system events and system will not expand in the future.
  - It is not possible to handle the actor request by using a use case controller.
    - example: interlibrary loan
- Applying the Use Case controller
  - Each use case has its own use case controller:
    - Checkout Controller for Checkout Use Case
    - Return Controller for Return Use Case
  - There is only one controller for each use case.
  - There is a one-to-one correspondence:
    - Checkout Use Case, Checkout GUI, Checkout Controller
    - Login Use Case, Login GUI, Login Controller
  - The use case controller maintains the state of the use case, and identifies out-of-sequence system events.

Liabilities of The Controller Pattern

- More classes to design, implement, test and integrate.
- Need to coordinate the developers who design and implement the UI, controllers and business objects.
  - This is not a problem when the methodology is followed.
- If not designed properly, it may result in bloated controllers.

Controller Pattern Guidelines

- Separate the design into GUI, Controller, and Mgr objects
- Never put the UI in the Controller object
- Never put the business interface (e.g., DB) in the Controller object
- Processing sequencing ,logic, procedures, and algorithms should go in the Controller class
- Each Use Case should have its own Controller class
- The Mgr class should perform all the transactions with the business interface
  - Searching attributes, updating attributes, querying attributes, etc
  - A network interface should have its own Mgr object

Bloated Controller

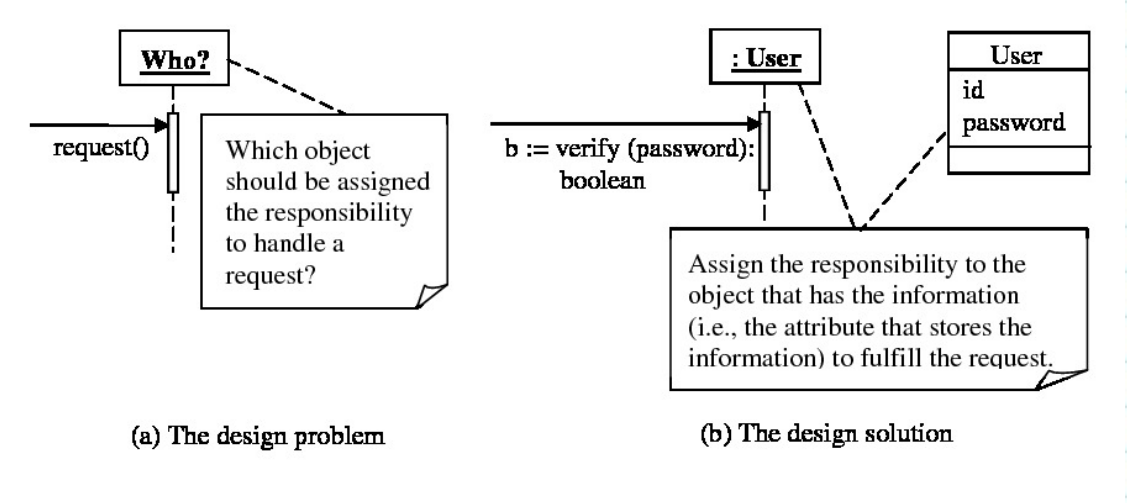
- A bloated controller is one that is assigned too many unrelated responsibilities.
- Symptoms
  - There is only one controller to handle many actor requests.
    - This is often seen with a role controller or a facade controller.
  - The controller does everything to handle the actor requests rather than delegating the responsibilities to other business objects.
  - The controller has many attributes to store system or domain information.

Cures to Bloated Controllers

- Symptoms
  - only one controller to process many system events
  - the controller does all things rather than delegating them to business objects
  - the controller has many attributes to maintain system or domain information
- Cures
  - replace the controller with use case controllers to handle use case related events
  - change the controller to delegate responsibilities to appropriate business objects
  - apply separation of concerns: move the attributes to business objects or other objects

The Information Expert

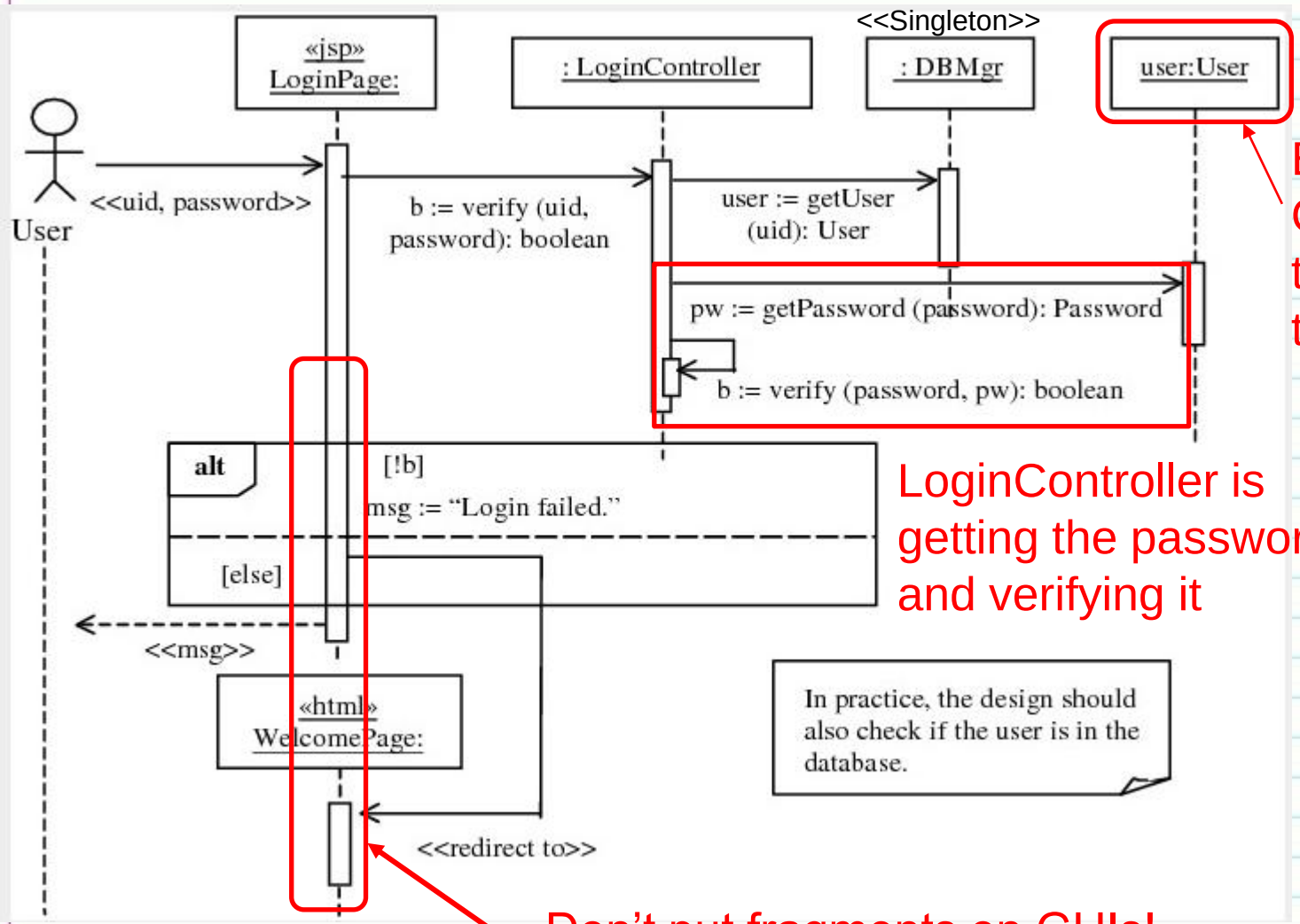
- Clearly, the object that is responsible for handling a request should have the information to fulfill the request.
- As indicated below, the request to verify whether a password is valid should be assigned to a user object because password is one of its attributes



(a) The design problem (b) The design solution

- If this responsibility were assigned elsewhere something else would have to query the user object to get the password and determine its validity or it would have to ask the user object to check the validity – both introduce unwanted coupling
- The (information) expert pattern assigns the responsibility to handle the request to the object that has the information to fulfill the request

A Login Sequence Diagram

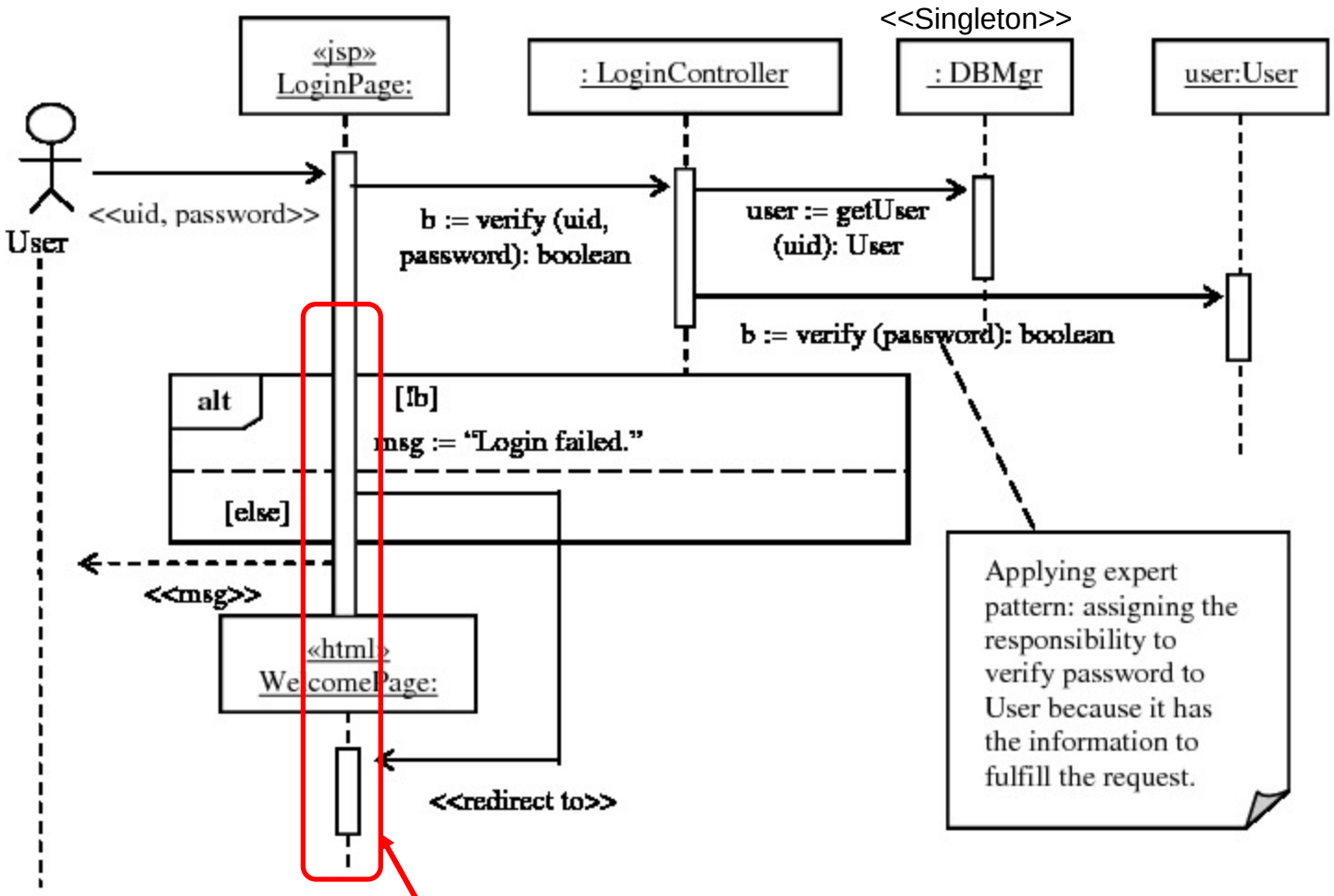


Expert Objects Are the Tables in the DB

LoginController is getting the password and verifying it

Don't put fragments on GUIs!
Put this logic in the Controller

Applying the Expert Pattern (cont.)



Don't put fragments on GUIs!
Put this logic in the Controller

Expert Pattern Trade-off

- For a library system there are three possibilities for which object should process checking out a book
  1. CheckoutController
  2. DB Mgr
  3. Loan
- What if checkout requires a number of checks?
  1. Does the book exist?
  2. Is the book available for checkout?
  3. Is there a limit on the number of books to checkout?
  4. Does the Patron have a hold?
- The ultimate question is – what object should perform this processing?

Expert Pattern Trade-off (cont.)

- What are the trade-offs for implementing these checks in each?

Implement in:	Advantages	Disadvantages	Conclusion
Controller	Maintains the idea that all business logic is in the controller	The controller is now addressing logic that is in the business layer	Not a good idea - don't do this
DBMgr	Most efficient since the DB has all the information about the book being checked out	1) We have a lot of un-related logic in the DBMgr about various objects in the system. 2) DBMgr is prone to change everytime an object usage rule changes	For very simple verifications it might be acceptable to do this in the DBMgr (e.g. Login) if rules get more complicated move this into an object
Loan	1) All of the processing related to the object is contained within that class. It insulates the rest of the system from change. 2) DBMgr simply becomes a series of gets/sets of data in the DB.	Not as efficient as capturing this in the DB Mgr	For non-trivial processing its best to keep the logic internal to that object and use the Expert pattern.

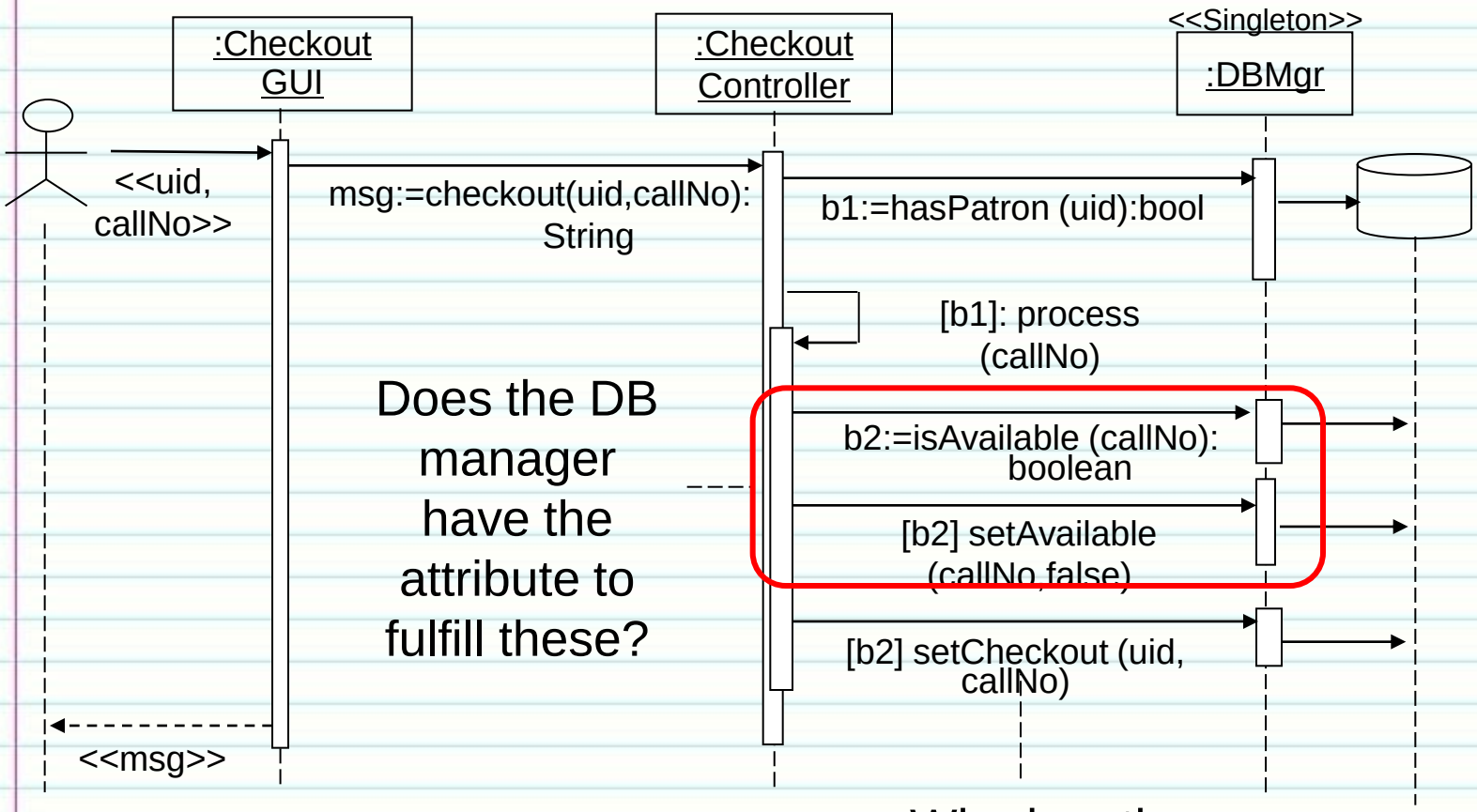
Multiple Objects Use the Expert Pattern

- Often, more than one expert needs to collaborate to fulfill a request.
- In the following sequence diagram this is accomplished within the constructor of the Loan class, that is, create (p: Patron, d: Document) in the UML notation
- However, if the due date depends on:
  - the kind of patron (faculty, staff, or student)
  - The kind of document (book or reserve)

- For this the request to compute the due date involves at least three objects: the patron, the document and the calendar.

- Since the Loan object has access to all these objects that have the needed information it is appropriate to assign this responsibility to the Loan object

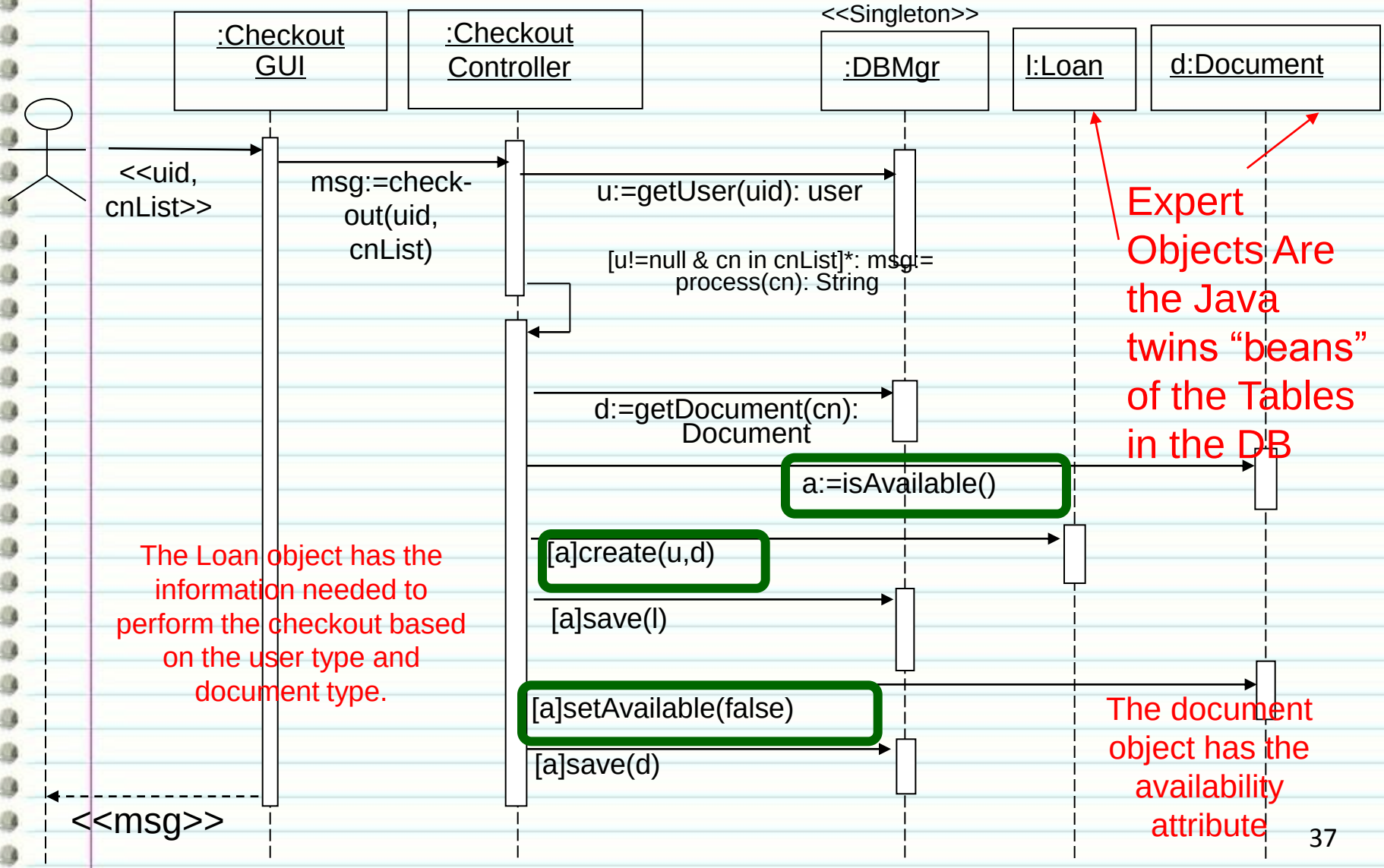
Applying The Expert Pattern



Does the DB manager have the attribute to fulfill these?

Who has the information to fulfill this?

Applying The Expert Pattern



Applying the Expert Pattern (cont.)

- Expert pattern in “reverse” makes more sense. Here is why - when I start a new application these steps help me see the classes that need to be created
  1. the first thing I ask is “what tables will exist in the database and what attributes will be in those tables?”
  2. then for each table in the database we create a Java class that has the same name as the table and has the table attributes in the Java class
     a. Any data validations on those attributes will be in this Java class.
     b. Any rules associated with the table name will be in this class also
     c. Examples here are Loan.java and Document.java
  3. I am going to have a GUI for each **activity** (or Web page) and a Controller for each **form** in my application
     a. For the Checkout UC I will have a CheckoutGUI activity (or Web page) and a CheckoutController (for processing the form)

Applying the Expert Pattern (cont.)

4. Instead of a single DBMgr I create a DAO (data access object) for the same Java “bean”
    a. For this example I will have a DocumentDAO.java and a LoadDAO.java
    b. Creating separate “DBMgrs” let’s team members work on these independently - a single DBMgr doesn’t
5. So I have as an example a Document.java and a DocumentDAO.java - how do I know which one the controller calls?
    a. I call Document.java for
        i. creating/editing any of the specific attributes for a Document (general attribute validation checks in it)
        ii. processing a Document checkout - how many can be checked out at a time, how long, etc
        iii. these are all the general rules that have to do with the entire class and not a specific document

Applying the Expert Pattern (cont.)

b. I call DocumentDAO.java for
   i. determining if it exists
   ii. determining if it is checked out
   iii. determining how many copies are in the library
   iv. I am not querying about the general set of documents but individual documents that exist
   v. DocumentDAO is a proxy (insulates from the DB)

The Expert Pattern

- It is a basic guiding principle of OO design.
- ~70% of responsibility assignments apply the expert pattern.
- It is frequently applied during object interaction design – constructing the sequence diagrams.

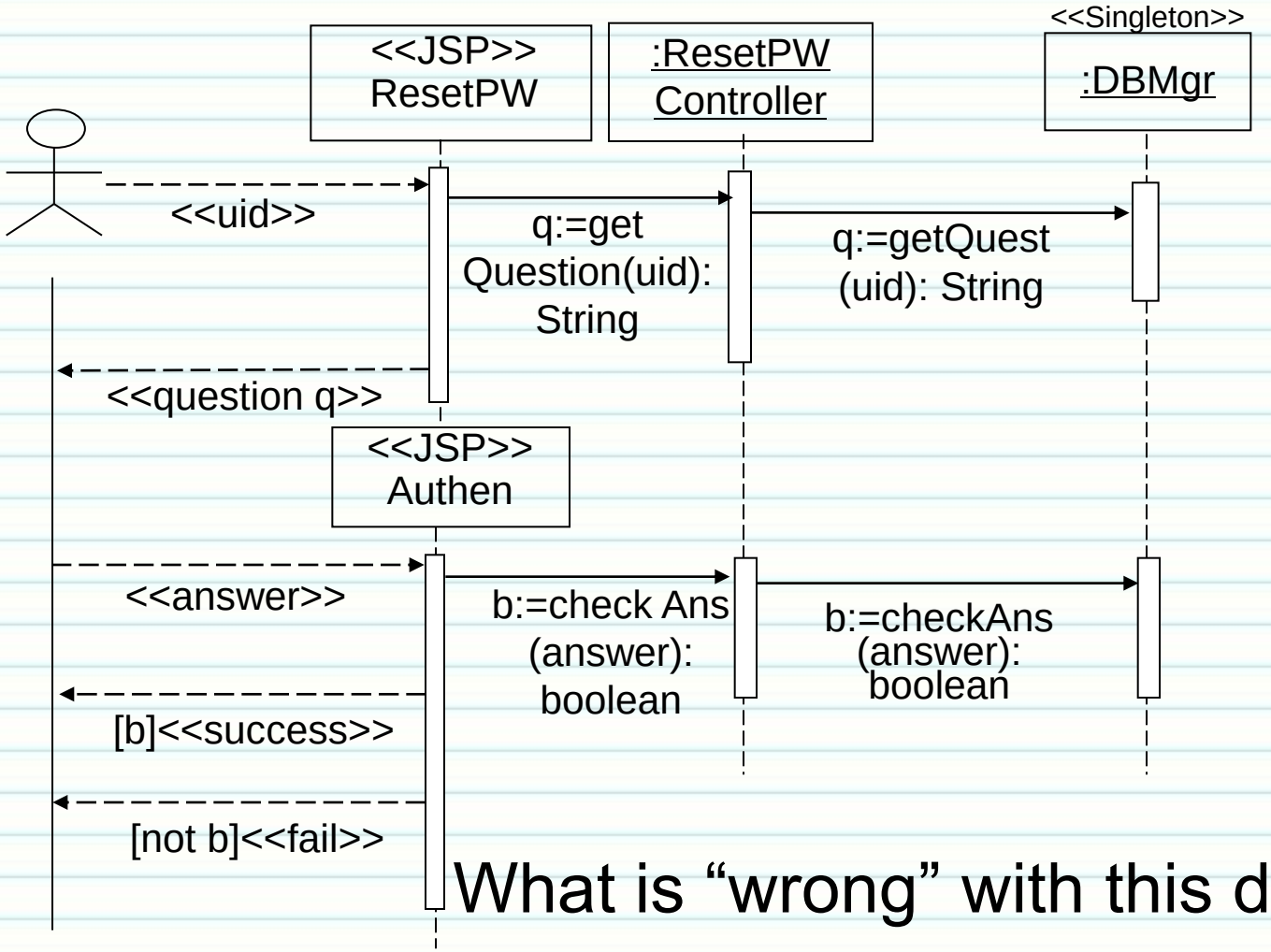
- It has the following benefits
  - Low coupling
  - High cohesion
  - Easy to comprehend and change
  - Tend to result in “stupid objects”

Class Exercise

Identify the non-trivial step in the diagram below

Actor: User	System: Web App
	0) The system displays a page with a “Reset Password” link.
1) TUCBW the user clicks the “Reset Password” link.	2) The system shows a page requesting the user id.
3) The user enters the user id and clicks the Submit button.	4) The system asks an authentication question.
5) The user enters the answer and clicks the OK button.	6) If authenticated, the system shows a confirmation message.
7) TUCEW The user sees the confirmation message.	

A Reset Password Sequence Diagram

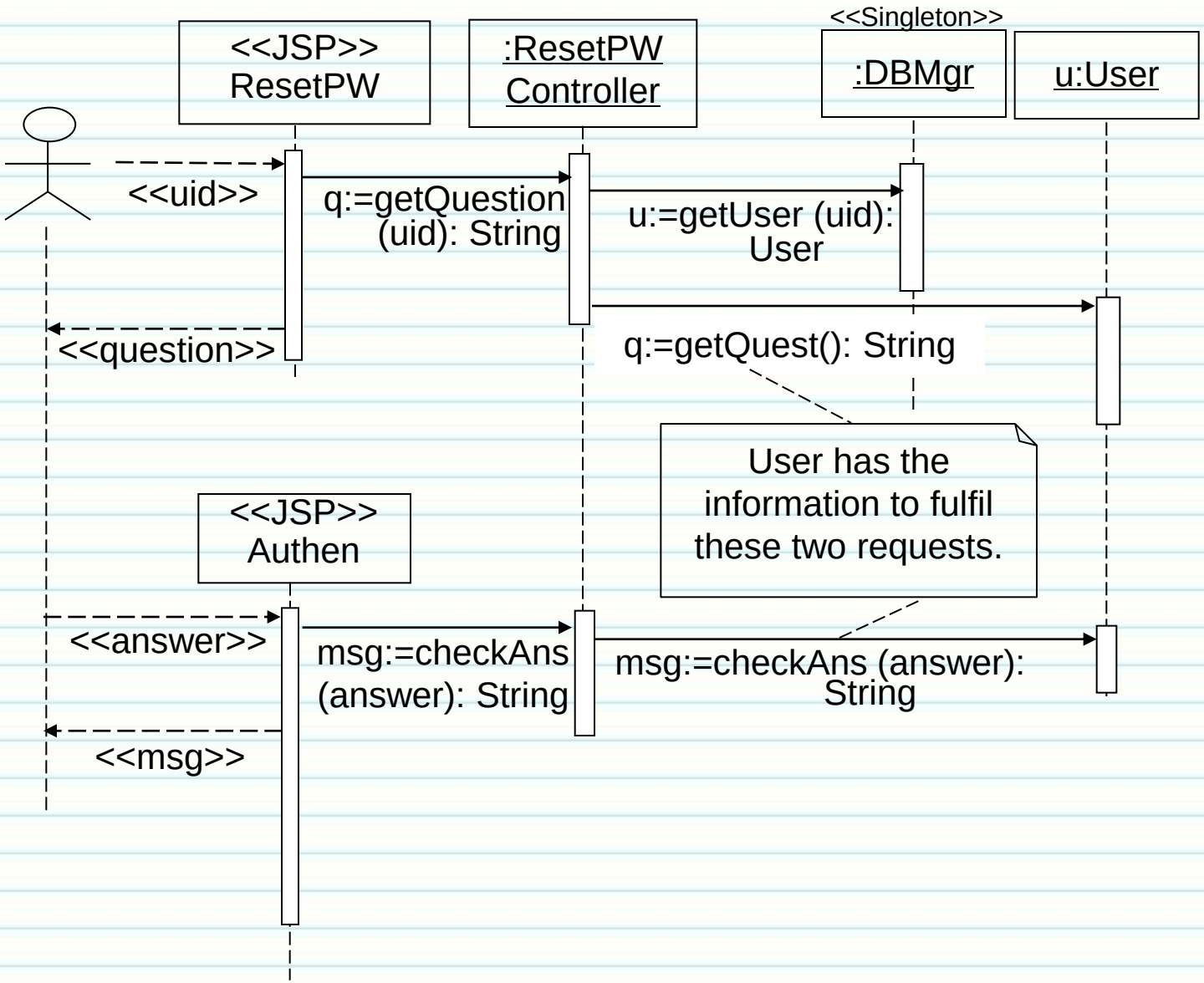


What is “wrong” with this design?

Problems with the Design

- It assigns getQuest() and checkAns() to the wrong object – DBMgr, which does not have the attributes to fulfill the requests.
- It does not design “stupid objects.”
- It violates the expert pattern.
- It is designed with a conventional mindset.

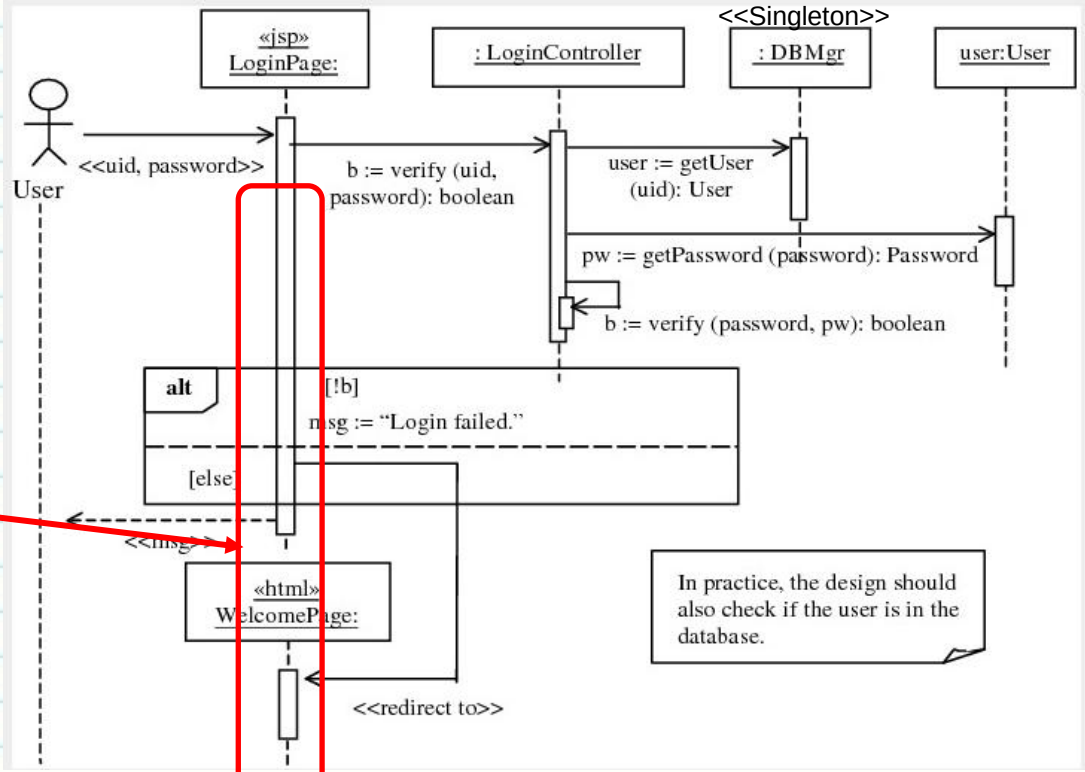
Applying The Expert Pattern



Applying the Expert Pattern to GUIs

- We also have another consideration - in the diagram below the GUI is making logical decisions - is this the best place to put the logic?

Don't put fragments on GUIs! Put this logic in the Controller



- We have two choices on where to put the logic - what are they?
- What are the various trade-offs for putting the logic in the GUI vs. Controller?

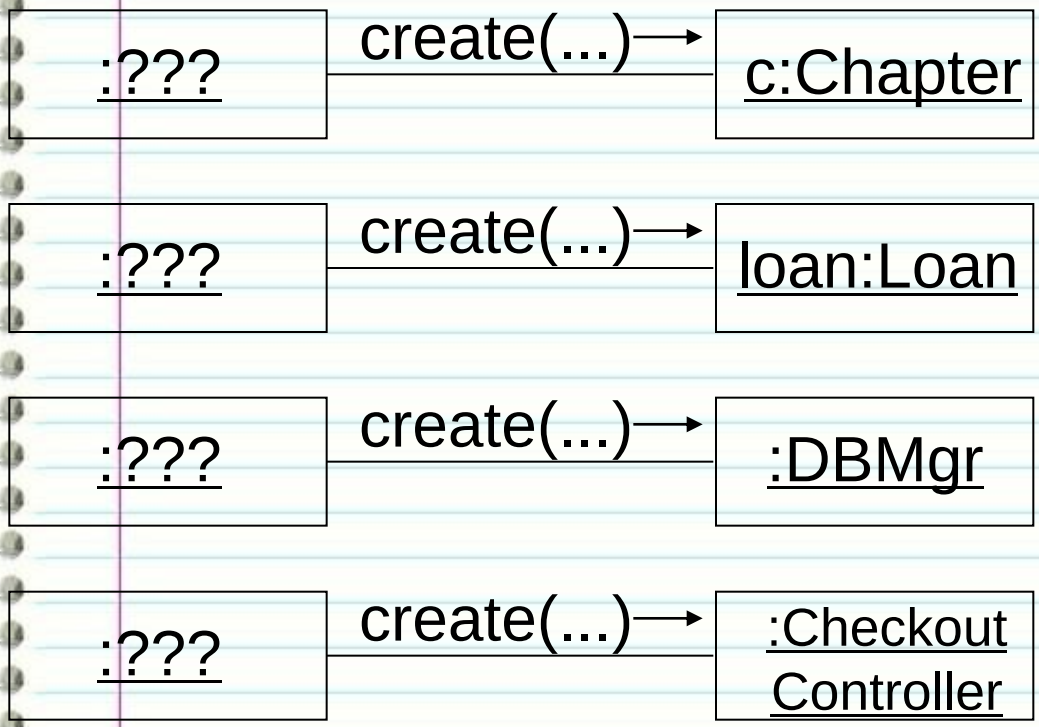
Applying the Expert Pattern to GUIs (cont.)

- What are some of the trade-offs?

Implement in:	Advantages	Disadvantages	Conclusion
Controller	Makes the GUI dumb	May be introducing a coupling between the controller and the GUI	Keep simple checks in the GUI to avoid articifial coupling
GUI	Most efficient since the GUI interfaces directly with the actor	1) We could end up with a lot of un-related logic in the GUI 2) the GUI volatility is increased - we need to change it for both GUI and logic changes	If logic get more complicated might want to move it in the controller

The Creator Pattern

- Who should create a given object?



Who should create a chapter of a book?

Who should create a Loan object in a library system?

Who should create a DB manager?

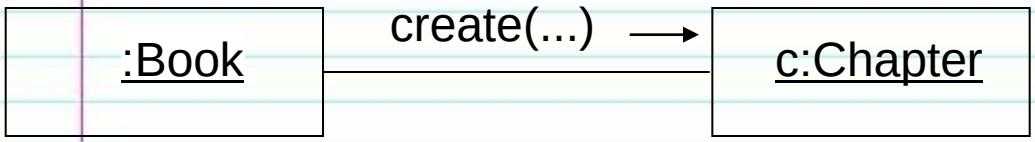
Who should create a checkout controller?

The Creator Pattern

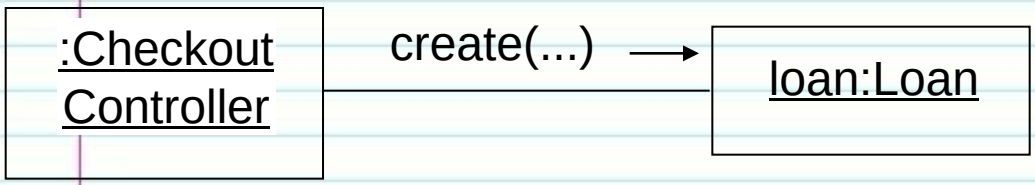
- Object creation is a common activity in OO design – it is useful to have a general principle for assigning the responsibility.
- Assign class B the responsibility to create an object of class A if
  - B is an aggregate of A objects.
  - B contains A objects, for example, the dispenser contains vending items.
  - B records A objects, for example, the dispenser maintains a count for each vending item.
  - B closely uses A objects.
  - B has the information to create an A object.

The Creator Pattern

- Who should create these objects?

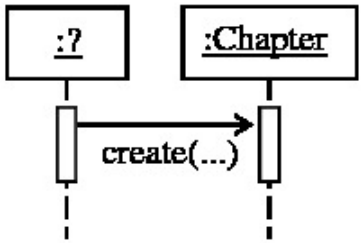


Because a chapter is a part of a book.

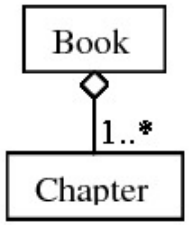


Because Checkout Controller has the information to call the constructor of Loan.

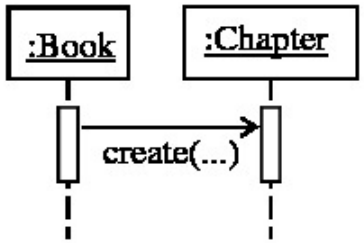
Examples



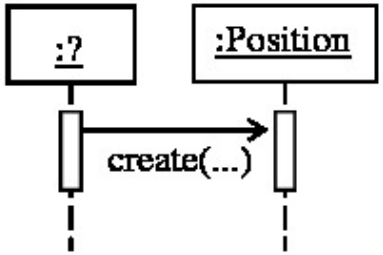
(a) Who should be responsible for creating a Chapter object?



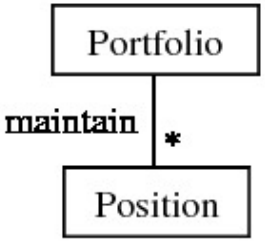
(b) In domain model, book is an aggregate of Chapter.



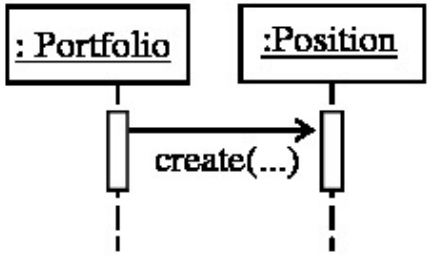
(c) Therefore, :Book should be responsible for creating a Chapter object.



(a) An investment portfolio consists of positions. Who should be responsible for creating a Position object?

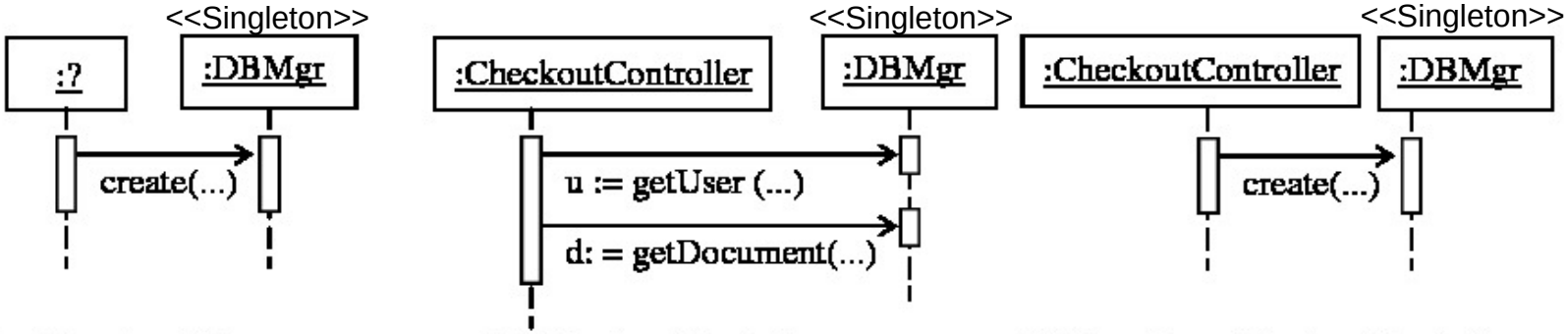


(b) In domain model, Portfolio maintains Positions.



(c) Therefore, :Portfolio should be responsible for creating a Position object.

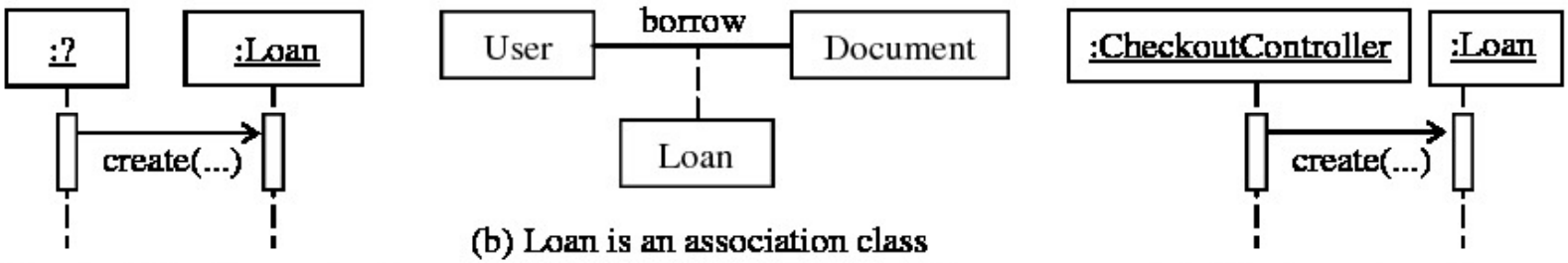
Examples (cont.)



(a) Who should be responsible for creating a DBMgr object?

(b) :CheckoutController uses :DBMgr.

(c) Therefore, :CheckoutController should be the creator.



(a) Who should be the creator?

(b) Loan is an association class between User and Document.

(c) :CheckoutController should be the creator.

Benefits of The Creator Pattern

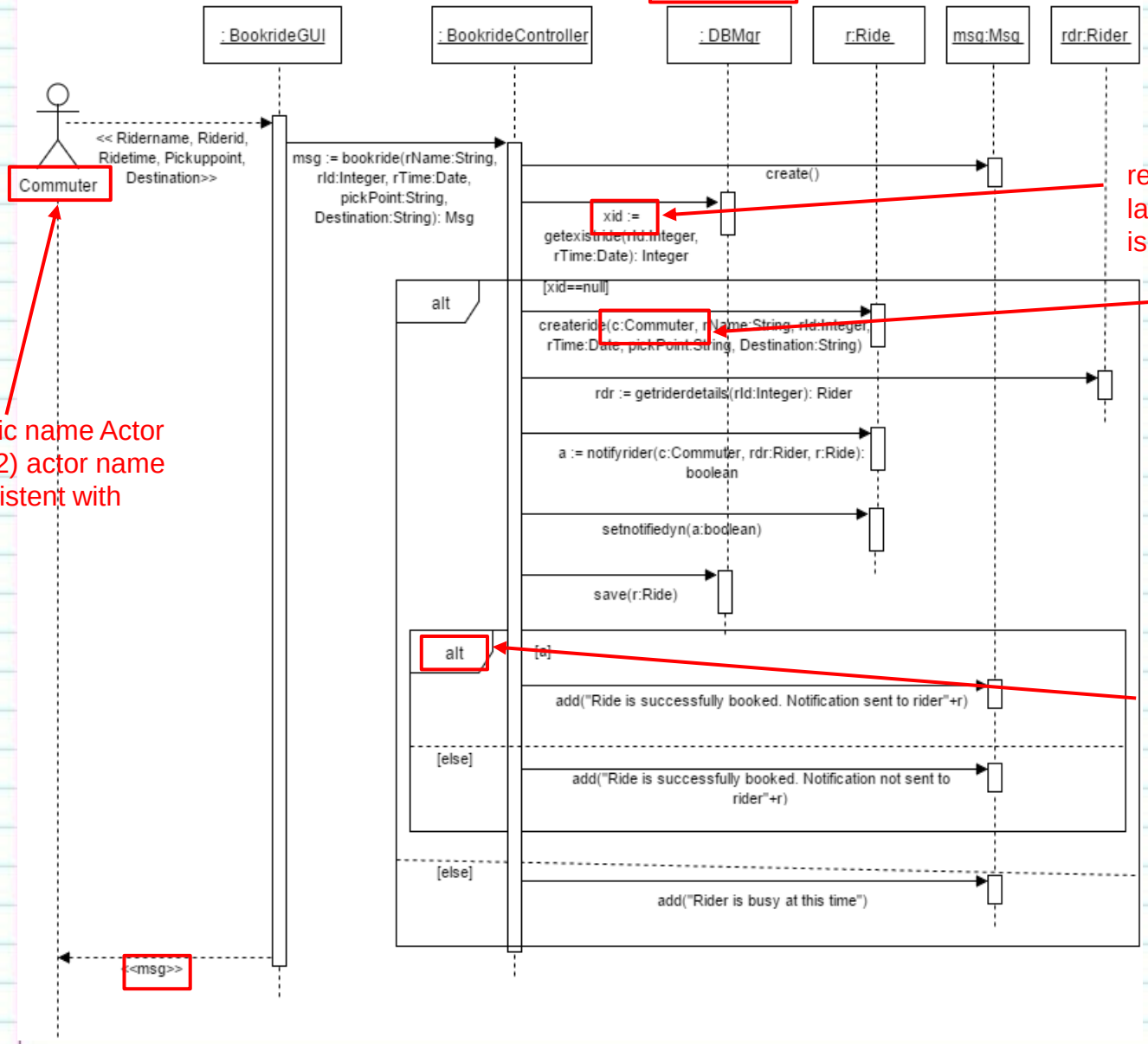
- Low coupling because the coupling already exists.
- Increase reusability.
- Related patterns
  - Low coupling
  - Creational patterns (abstract factory, factory method, builder, prototype, singleton)
  - Composite

Example Student Project

Book Ride

Common mistakes identified

Missing <<singleton>>



return values must be used later - they are here - so this is correct

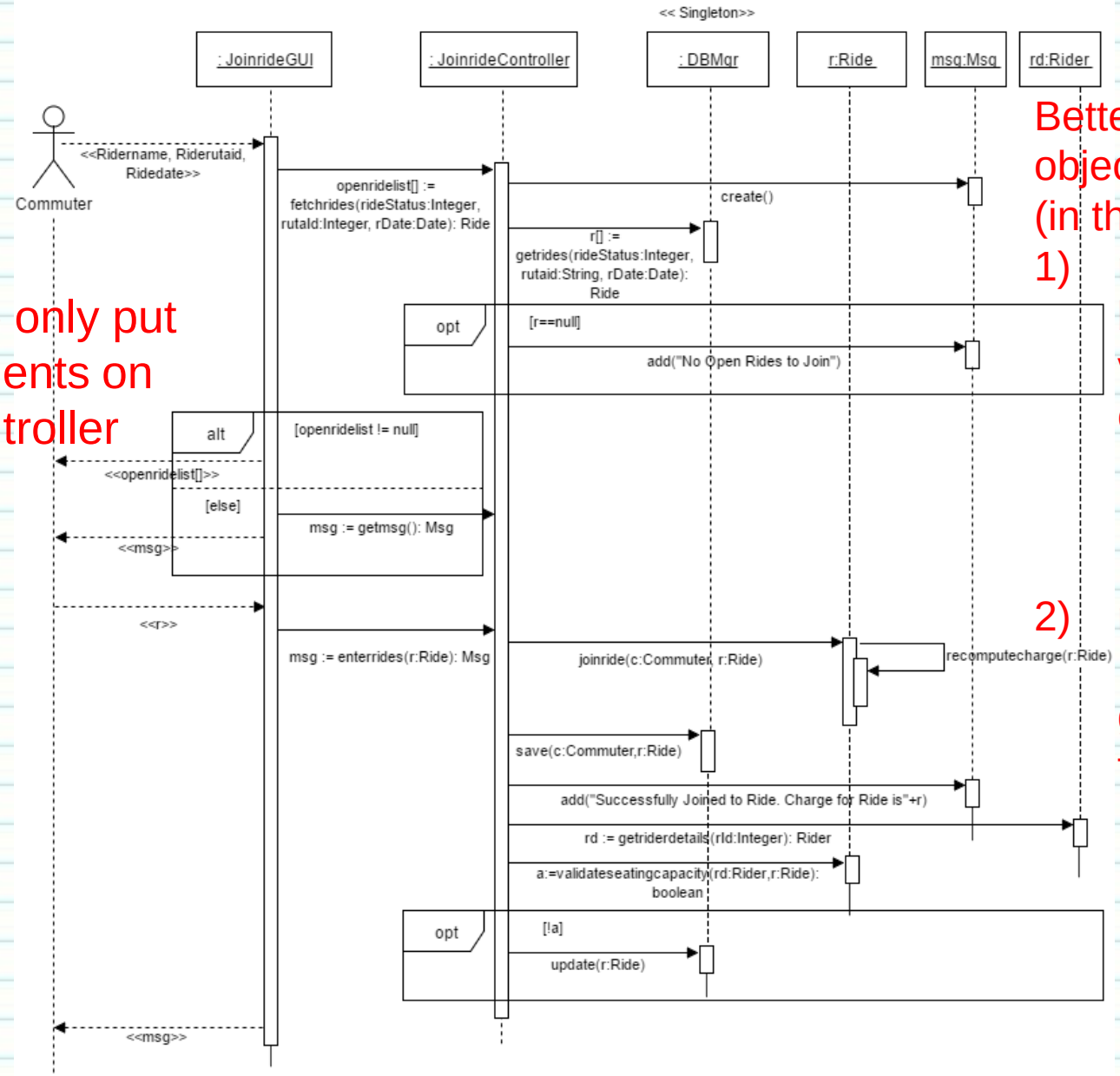
new parameter comes from nowhere! error

1) generic name Actor used or 2) actor name not consistent with EUC

using wrong fragment type - *alt* is an if then else or switch *opt* is if then - usage here correct

this only supports sending a message - if we needed to send a result or an array list we would need to send a return value to the actor

Join Ride



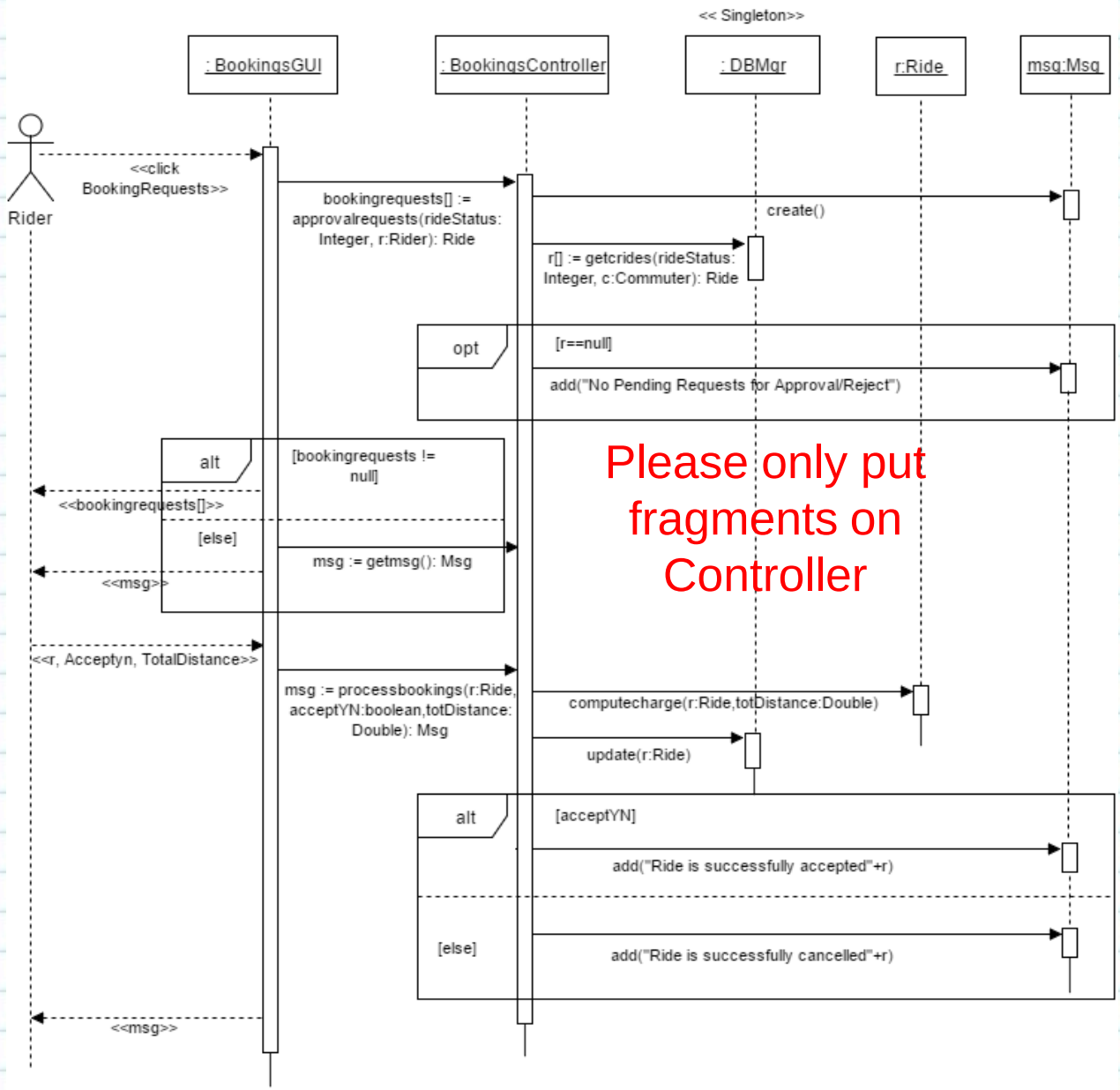
Please only put fragments on Controller

Better flow to Expert objects and DBMgr is (in this order):

1) message with boolean return value to Ride object to validate Ride data from User or to check Ride rules

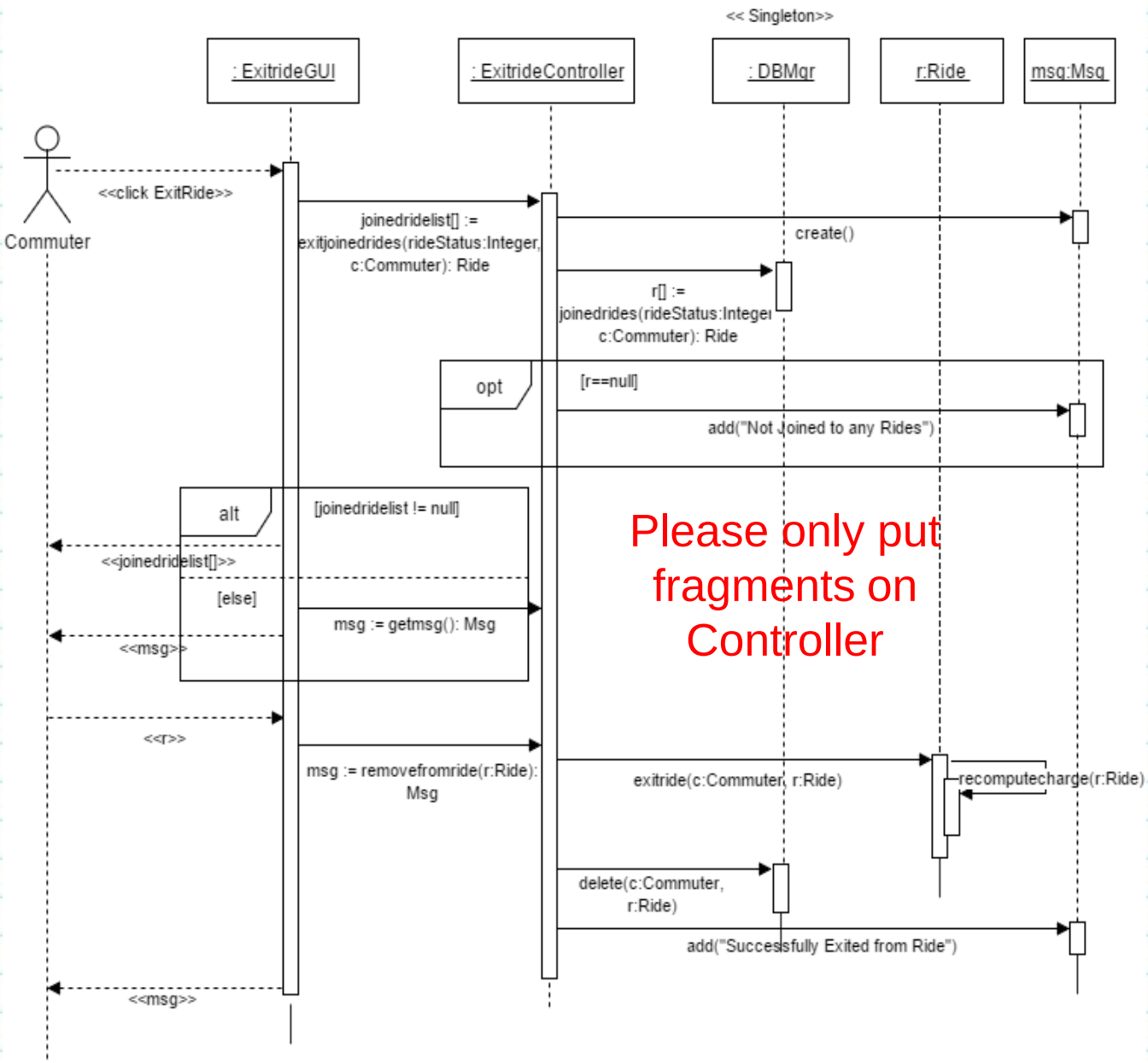
2) message to DBMgr after that to get ride instance from DB

Accept/Reject Booking Request



Please only put fragments on Controller

Exit Ride


<< Singleton>>
: ExitrideGUI
: ExitrideController
: DBMgr
r:Ride
msg:Msg
Commuter
<<click ExitRide>>
joinedridelist[] := exitjoinedrides(rideStatus:Integer, c:Commuter): Ride
create()
r[] := joinedrides(rideStatus:Integer c:Commuter): Ride
opt
[r==null]
add("Not Joined to any Rides")
alt
[joinedridelist != null]
<<joinedridelist[]>>
[else]
msg := getmsg(): Msg
<<msg>>
<<r>>
msg := removefromride(r:Ride): Msg
exitride(c:Commuter, r:Ride)
recomputecharge(r:Ride)
delete(c:Commuter, r:Ride)
add("Successfully Exited from Ride")
<<msg>>
Please only put fragments on Controller

Submit Ratings