Chapter 9 – Sequence Diagrams

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

Key Takeaway Points

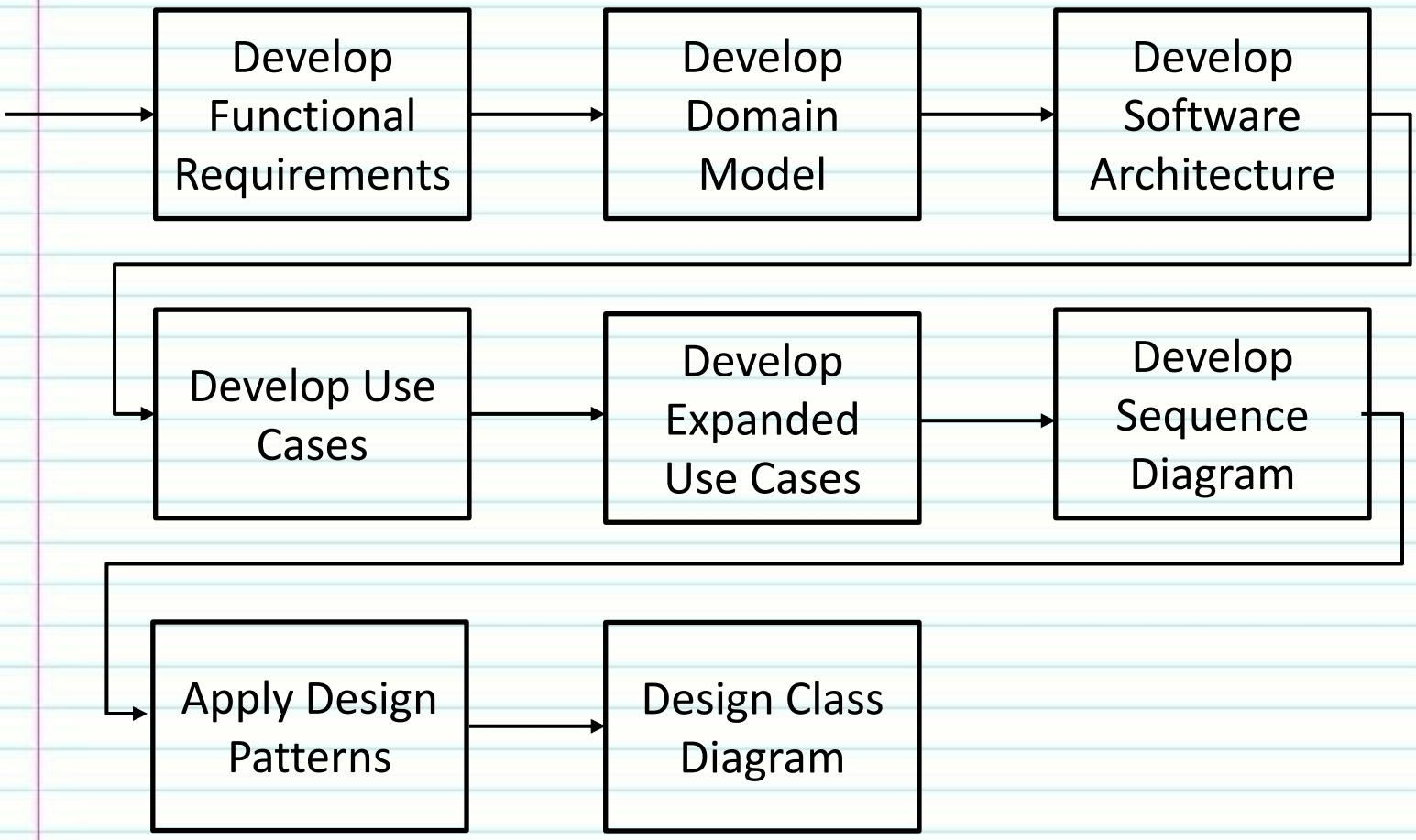
- Sequence Diagram Analysis model helps the development team
  - understand the existing business processes,
  - and design object interaction behaviors to improve the business.

- The Expanded Use Case deals with foreground processing of a use case
  - It's concentrating on the choreograph between the Actor and System
  - System processing steps are abbreviated with emphasis on foreground processing

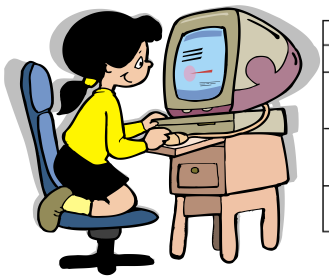
- The Sequence Diagram deals with background processing of a use case
  - We're going to look at the Non-trivial System level steps that require more detail

- The goal of the sequence diagram is to identify the methods needed in each object

Book Approach to OOSE



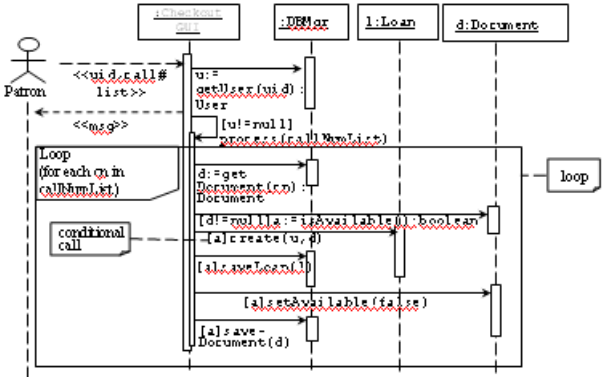
Actor-System Interaction & Object Interaction



UC1: Checkout Document

Actor: Patron	System: LIS
	0) System displays the main GUI.
1) TUCBW patron clicks the "Checkout Document" button on the main GUI.	2) The system displays the Checkout GUI.
3) The patron enters the call numbers of documents to be checked out and clicks the "Submit" button.	4) The system displays a checkout message showing the details.
5) TUCEW the patron sees the checkout message.	

Actor-system interaction

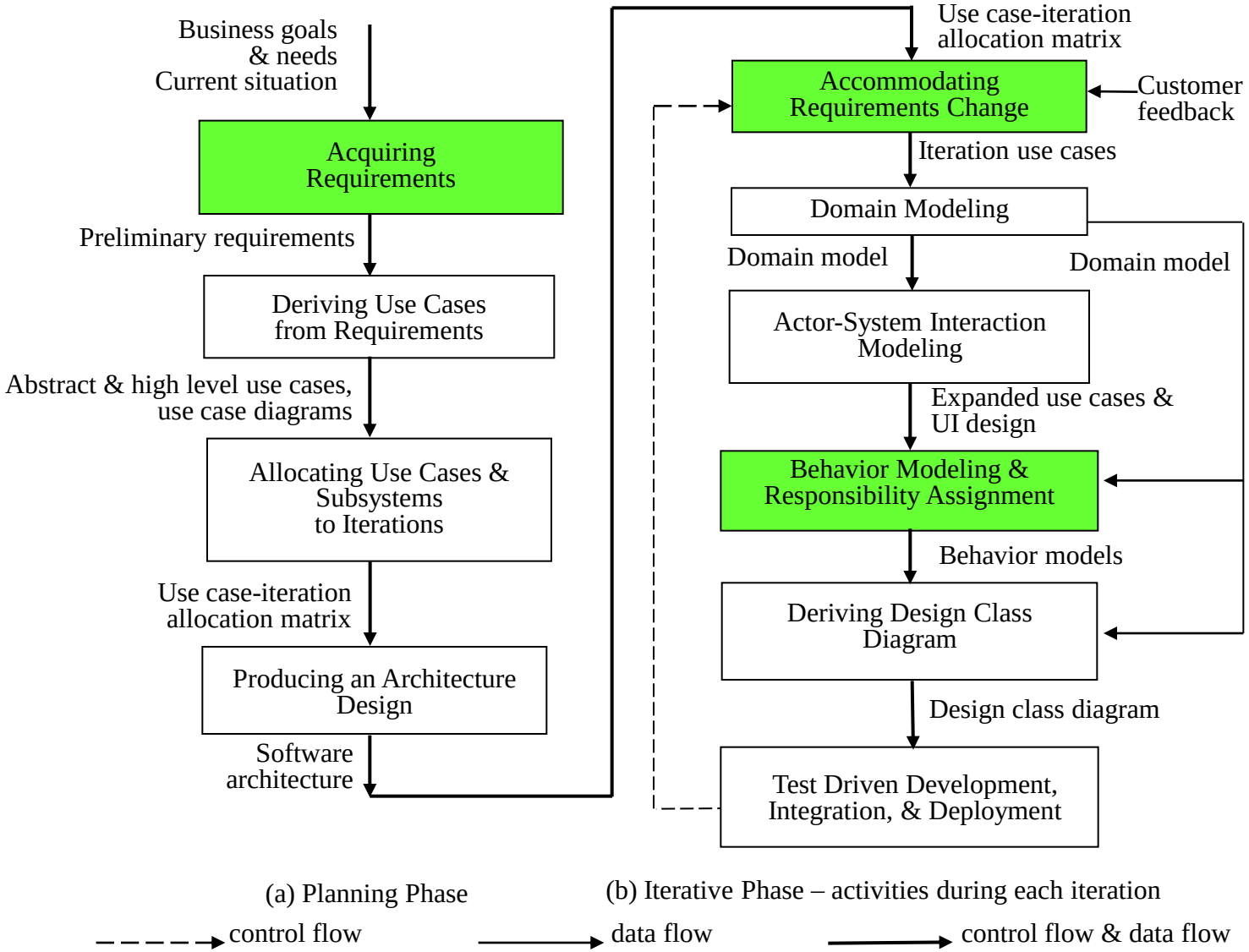


Object interaction

- Foreground processing of use case.
- Acquiring actor input and actor action.
- Displaying system responses.

- Background processing of use case by objects.
- Designing high-level algorithms to process actor requests.
- Producing system responses.

OIM in the Methodology Context



Two Perspectives

- For sequence diagrams there are two perspectives to consider
  1. The analysis perspective – how do the objects interact with each other to accomplish the business tasks in the existing, possibly manual business processes?
  2. The design perspective – how should the objects interact in the proposed system to improve the business process?

- OIM is important from an analysis perspective because, the development team
  - may not be familiar with the existing business process
  - may need to collect information and construct models to help understand them
  - may need object interaction models to identify problems or weaknesses in the existing business process

Two Perspectives (cont.)

- OIM is important from a design perspective because, the existing business processes may have been designed years ago
  - may have expanded considerably
  - may have changed dramatically (technological improvements)
  - existing processes may need to be redesigned to better address technology or business needs

Sequence Modeling

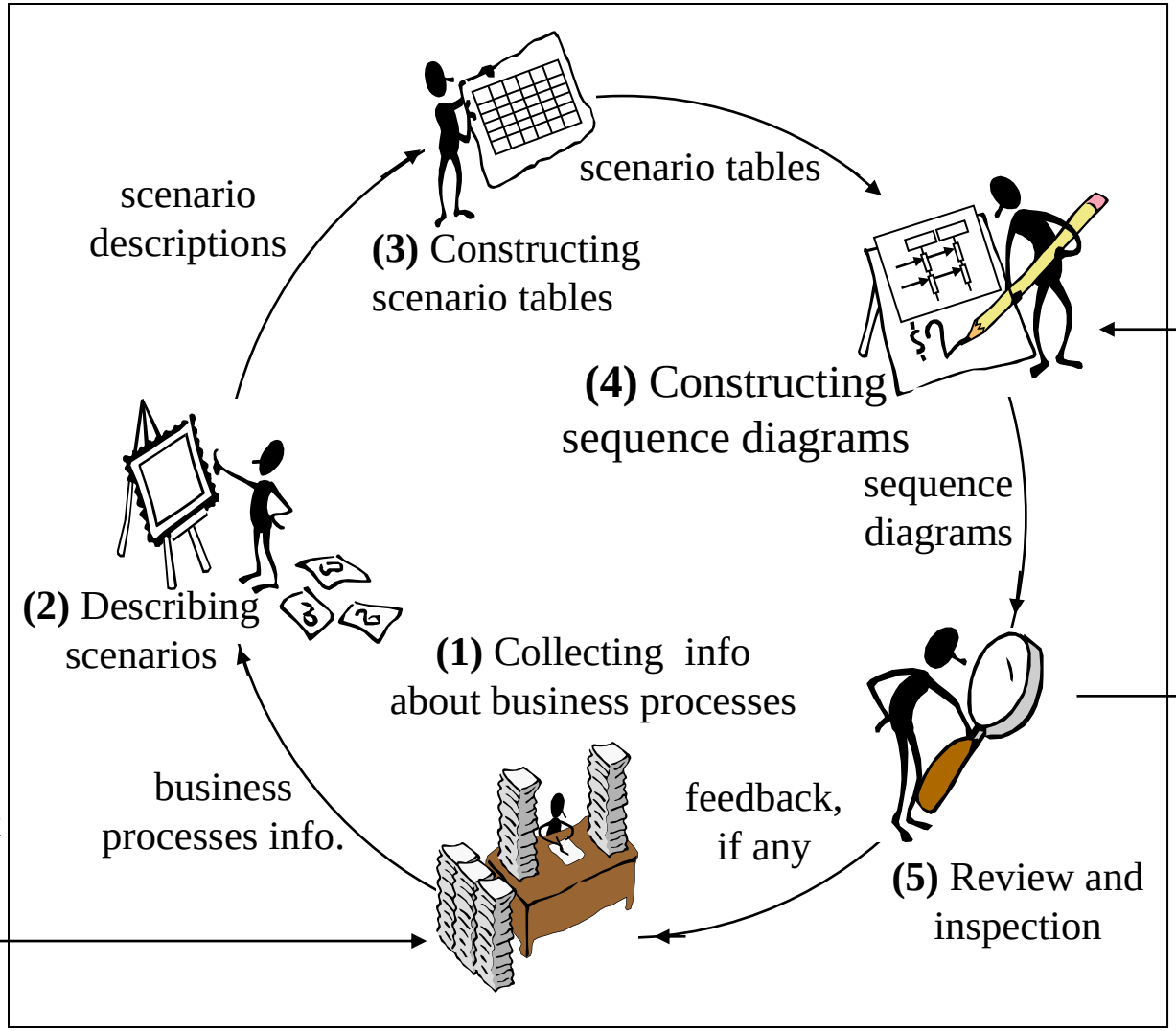
- Sequence modeling specifies how objects interact with each other to carry out the background processing of a business process.

- Sequence modeling is aided by the specification of
  - scenarios
  - scenario tables.

- A scenario is an informal, step-by-step description of object interaction.

- A scenario table organizes the interaction into a five column table - it facilitates translation to a sequence diagram.

Object Interaction Modeling Steps



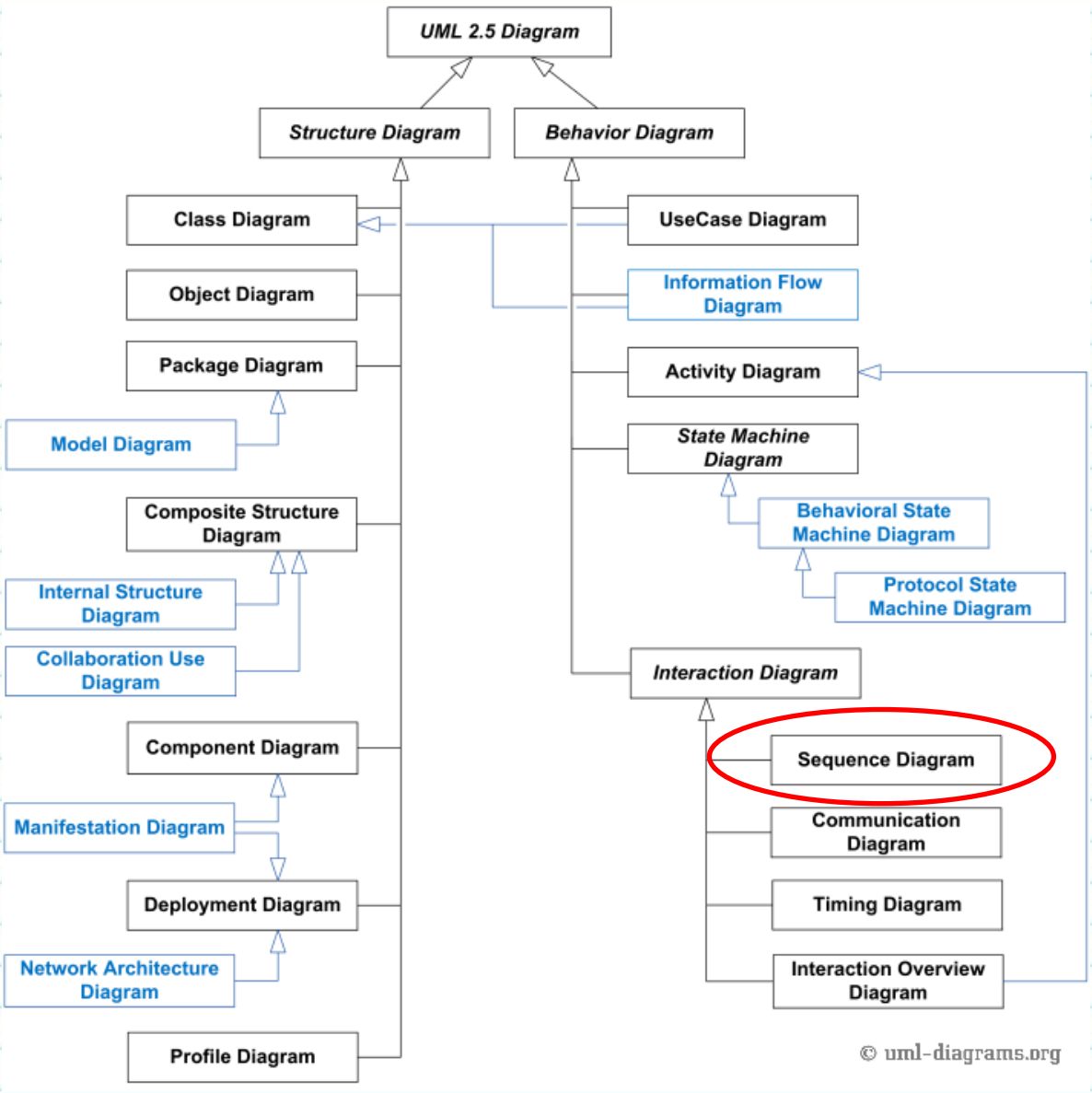
domain model, design class diagram from last iteration

design sequence diagrams

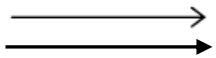
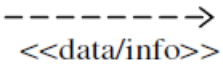
to deriving a design class diagram (chapter 11)

expanded use cases for current iteration

What is a Sequence Diagram?



Sequence Diagram Notions and Notations

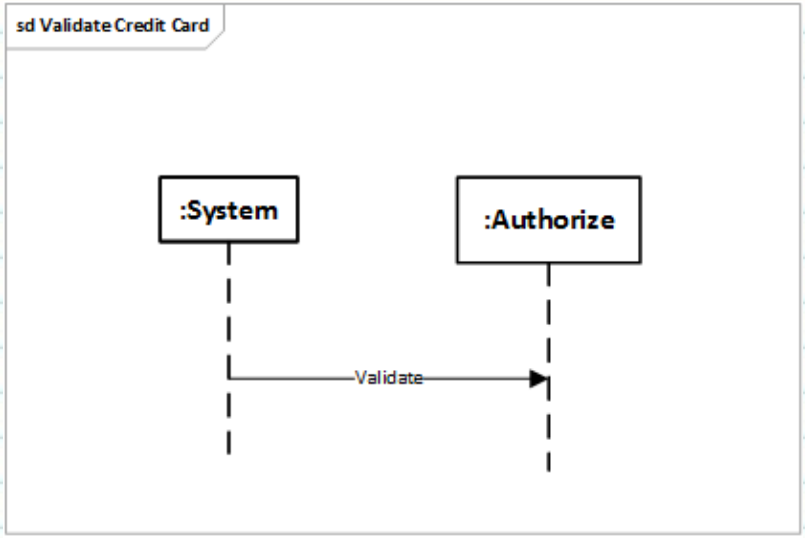
Notion	Notation	Semantics
Actor		A role played by a set of entities or stakeholders that are outside of the system and interact with the system.
Object	object:Class	An object of a certain Class. (It is placed on top of the lifeline. The underline indicates an object. The object name is placed before the colon and class name after the colon.)
Lifeline or "activation bar"		Represents the lifetime of the specified object in the system
Method execution		Indicating that the object exists in the system and is executing a method.
Object destruction		The cross at the tip of the lifeline indicates that the object is destroyed or ceases to exist.
Message, or message passing	m 	A message m is sent from one object to another object, i.e., a function call from one object to another object. Synchronous vs. asynchronous
Stereotyped message	 <<data/info>>	A stereotyped, or user-defined, message is communicated between two objects, or an object and and an actor.
Combined fragment	operator	A combined fragment defines an expression on a portion of a sequence diagram using an interaction operator like loop or alt.

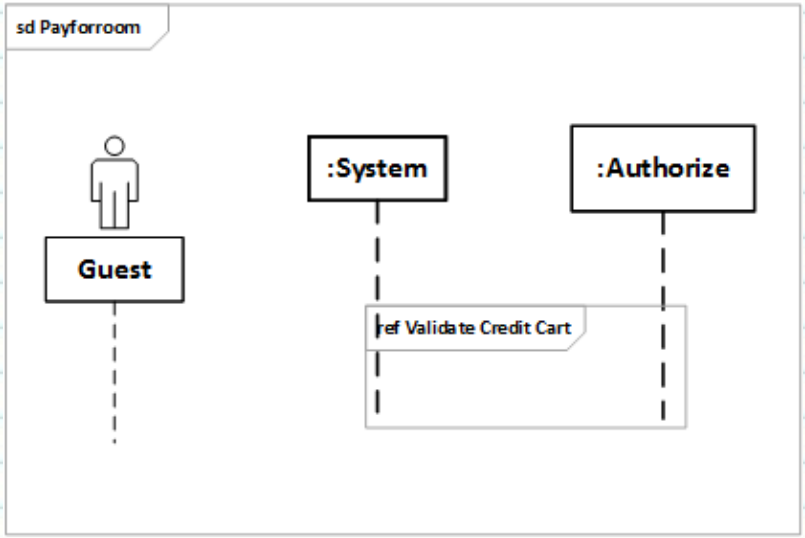
Operators in a Combined Fragment

A combined fragment is an interaction fragment which defines a combination (expression) of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of combined fragments the user is able to describe a number of traces in a compact and concise manner.

Operator	Keywords	Description
alt	[guard1] [guard2] ... [else]	Selects one interaction to execute from a set of interactions. The interaction selected is from the True guard condition or the [else] if none are true. Corresponds to a switch in Java.
assert		The selected interaction must occur in exactly the way indicated - an assertion.
break		If the selected action occurs the enclosing interaction is abandoned (usually used in a loop). Similar to a break statement in a switch.
loop	minint, maxint, [guard]	The interaction is executed a minimum of minint times and up to maxint times as long as the [guard] condition is true.
opt	[guard]	this interaction only occurs if the guard is true. Corresponds to an If in Java.
ref		reference to an interaction defined elsewhere. Corresponds to a method call in Java or a <<include>> in a use case

The Reference Operator in a Combined Fragment

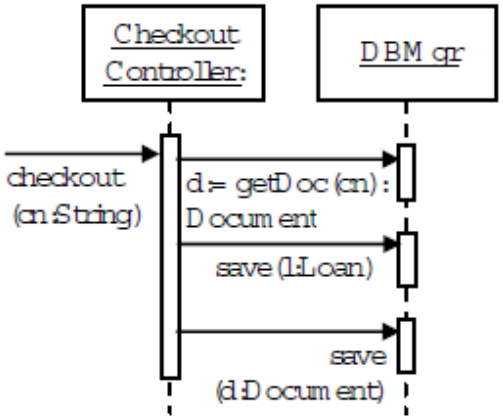




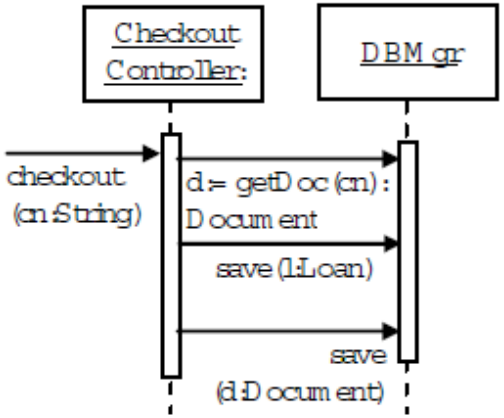
Representing Various Object Instances

Notation	Notion and Meaning
:Car	An unnamed instance of the Car class. The name is not important, or not used elsewhere in the sequence diagram
car:	A named instance of an unnamed class. A class without a type, the type is not important, unknown, or to be determined at run time
car: Car	A named instance of the Car class. A named instance with a type, this is commonly used.
:Car	A collection of instances of the Car class. Or a collection of objects of the Car class
<<jsp>> LoginPage:	A stereotyped object. A stereotyped class might have some not directly supported in UML.

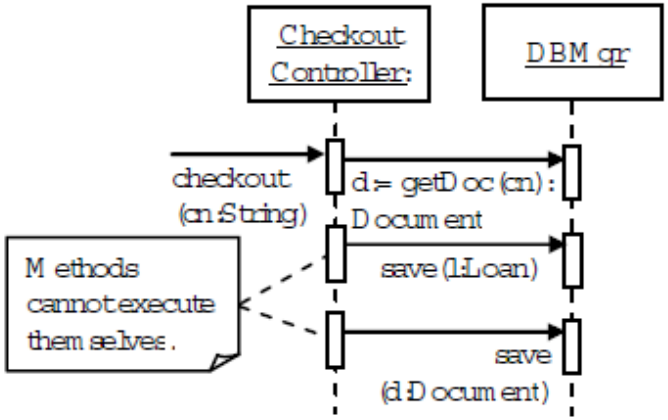
Using the Notations Correctly



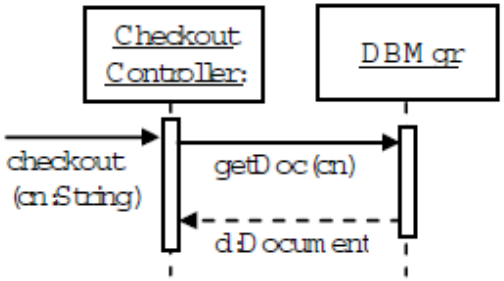
(a) Correct: during the execution of checkout(...), three separate calls to DBMgr are made.



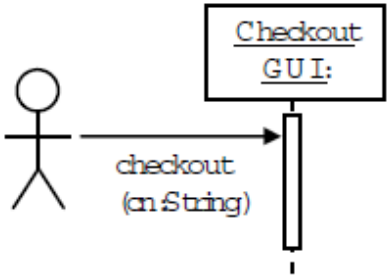
(b) Incorrect: the long rectangle beneath DBMgr should split into three as in (a)



(c) Incorrect: methods must be called to execute.



(d) Not preferred: the back dashed arrow line can be interpreted differently. Do as (a) is preferred.



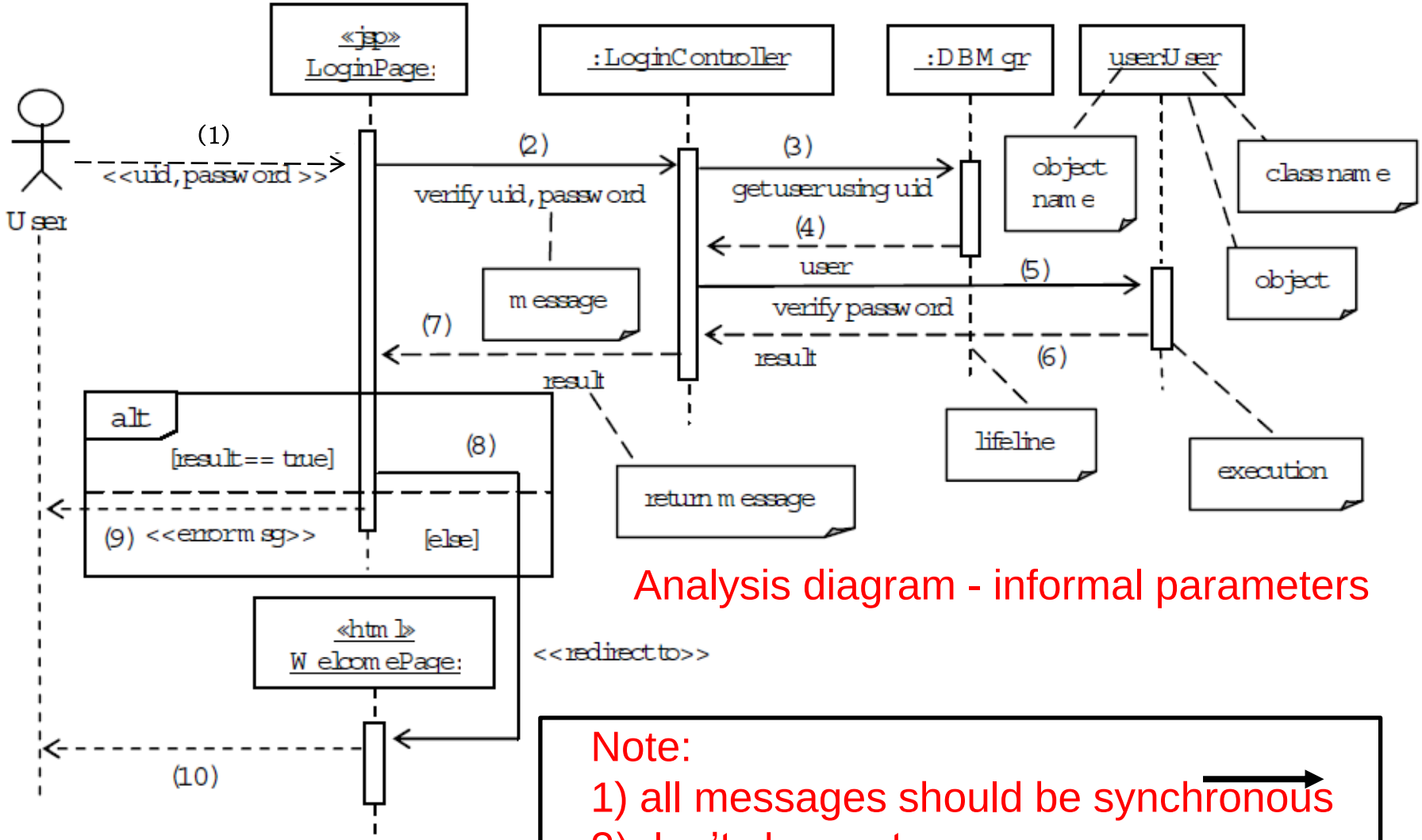
(e) Incorrect: an actor cannot call a function of an object; should use a dashed line and stereotype message.

Sequence Diagrams Illustrated

Simplified Login use case

1. User submits UID and password to LoginPage
2. LoginPage verifies with LoginController using UID and password
3. LoginController gets user (object) from the database manager (DBMgr) using UID
4. DBMgr returns user (object) to LoginController
5. LoginController verifies with user (object) using password
6. User (object) returns result to LoginController
7. LoginController returns result to LoginPage
8. If result is true, LoginPage redirects to WelcomePage
9. If result is false, LoginPage shows an error message to user
10. User is shown the WelcomePage (or the error message)

Sequence Diagram of a Login Scenario



Analysis diagram - informal parameters

Note:
1) all messages should be synchronous
2) don't show return messages
3) Don't put fragments on GUI

Asynchronous vs. Synchronous Messages

- There are two types of messages that can be sent in the Sequence diagram
  - Asynchronous - represented by an open ended arrow
    - Represents a tasking hand-off or call to middle-ware - something where the activity is started but does not complete before control is returned back
  - Synchronous - represented by a closed arrow
    - Represents a method invocation - control is returned back only after the invoked method completes
    - This is the normal use in the Sequence diagram and one we will always use regardless of how they appear on some of the class charts (which I can't change)
    - I believe that asynchronous should not be depicted - we don't want to predict implementation details that could change
- **<u>For all class project work - use synchronous messaging</u>**

Scenario for a Checkout Document Use Case

Patron enters the call number to the CheckOutGUI

CheckOutGUI calls checkout(callNo:String) of CheckoutController

CheckoutController calls getDocument(callNo:String) of DBMgr

DBMrg returns the document d to CheckoutController

If d is not null, then

    CheckoutController creates the Loan object 1

    CheckoutController calls save(1:Loan) of DBMgr

    CheckoutController calls setAvailable(false:Boolean) of document d

    CheckoutController calls save(d:Document) of DBMgr

    CheckoutController sets msg to “Checkout successful”

Else

    CheckoutController sets msg to “Document not found”

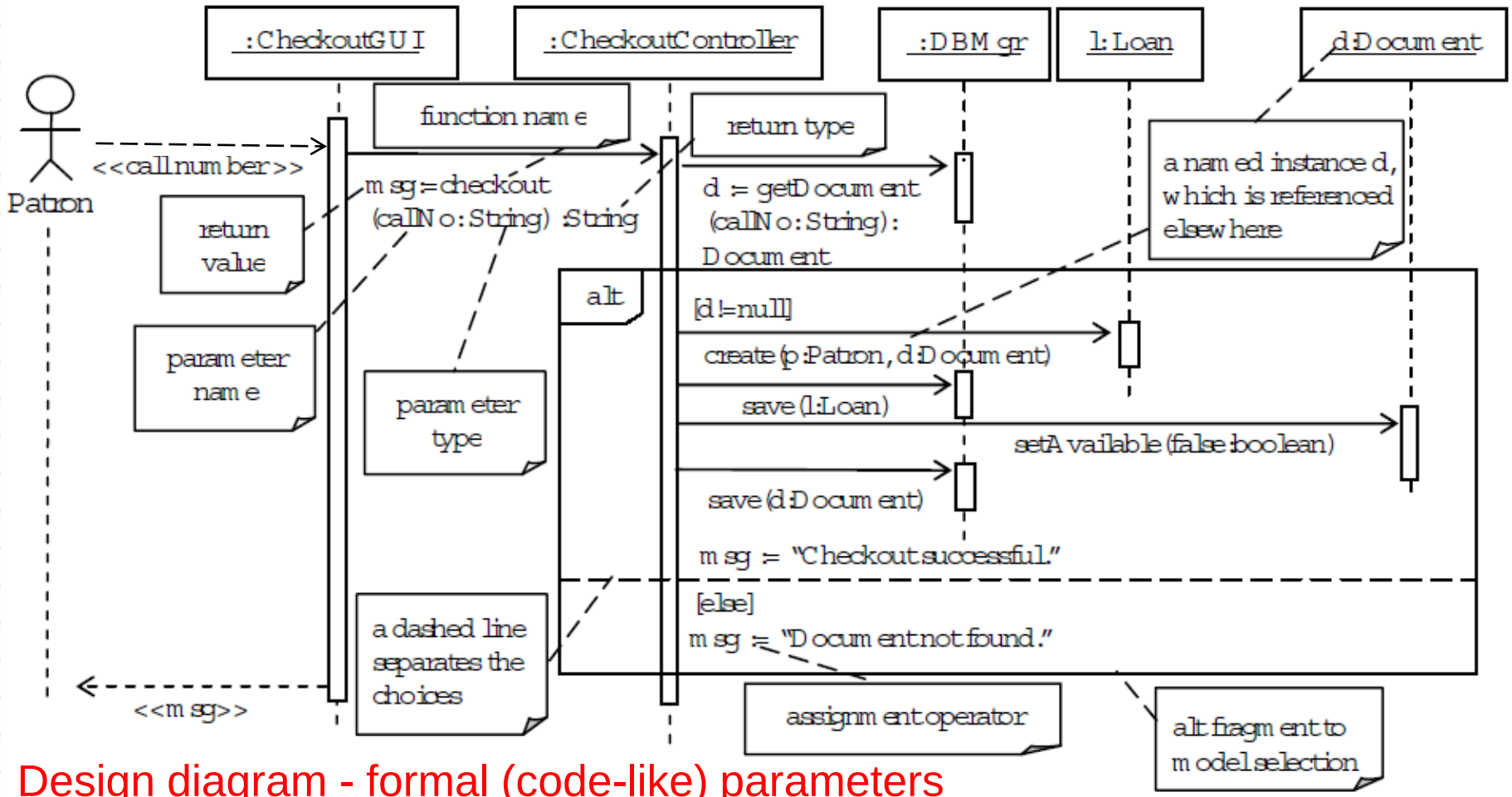
CheckoutController rturns msg to CheckoutGUI

CheckoutGUI shows message to Patron

Patron sees the msg

Its corresponding Design Sequence Diagram is on the next slide

Sequence Diagram for a Checkout Document



Design diagram - formal (code-like) parameters

Note: all messages should be synchronous

When Do We Need a Sequence Diagram?

- A trivial step is
  - If the step does not require background processing
  - If the system response simply displays a menu, or input dialog
  - If the step displays the same system response for all actors – Step 2 from the previous expanded use case – the Checkout GUI is the same for all actors
- A **non**trivial step is
  - The system response requires background processing
  - The system response is different for different actors (not just a standard GUI)
  - Key question: does it require other objects to interact and collaborate with each other to fulfill the request?
  - Checkout message of step 4 has the following considerations
    1. Does the document exist?
    2. Is it available?
    3. Has to create a loan record
    4. Error messages as appropriate above

Example

UC1: Checkout Document

Actor: Patron	System: LIS
	0) System displays the main GUI.
1) TUCBW patron clicks the "Checkout Document" button on the main GUI.	2) The system displays the Checkout GUI.
3) The patron enters the call numbers of documents to be checked out and clicks the "Submit" button.	*4) The system displays a checkout message showing the details.
5) TUCEW the patron sees the checkout message.	

nontrivial step

Modeling the Non-trivial Step

- If we look at step 4 in the expanded use case – the step is
  - "The checkout GUI displays a checkout message showing the details."
  - This task can be broken down into a number of sub-tasks
  - Checkout documents (use objects in task steps)
    - Get the document objects from the database
    - Create loan objects
    - Set document objects to unavailable
    - Save loan objects to database
    - Save document objects to database
    - Return a checkout message

Guidelines for Scenario Construction

- Specify the normal scenario first (i.e., assume everything will go as expected).
- If needed, augment the normal scenario with alternative flows.

- Scenario writing steps:
  1. What must be done to fulfill the non-trivial actor request?
  2. What order is needed to carry out these tasks?
  3. For each of these tasks determine the object that is acted upon.
  4. For each of these tasks determine the object to issue the request to perform them.

Checkout Document Scenario Description

3) The patron enters the call numbers of documents to be checked out and clicks the Submit button.

4.1) Checkout GUI checks out the documents with the checkout controller using the document call numbers.

4.2) Checkout controller creates a blank msg.

4.3) For each document call number,

4.3.1) The checkout controller gets the document from the database manager (DBMgr) using the document call number.

4.3.2) DBMgr returns the document d to the checkout controller.

4.3.3) If the document exists (i.e., d!=null)

4.3.3.1) the checkout controller checks if the document is available (for check out).

4.3.3.2) If the document is available for check out,

4.3.3.2.1) the checkout controller creates a Loan object using patron p and document d,

4.3.3.2.2) the checkout controller sets document d to not available,

4.3.3.2.3) the checkout controller saves the Loan object with DBMgr,

4.3.3.2.4) the checkout controller saves document d with the DBMgr,

4.3.3.4.5) the checkout controller writes “checkout successful” to msg.

4.3.3.3) else,

4.3.3.3.1) the checkout controller writes “document not available” to msg.

4.3.4) else

4.3.4.1) the checkout controller writes “document not found” to msg.

4.4) The checkout controller returns msg to Checkout GUI.

4.5) Checkout GUI displays msg to patron.

Constructing the Scenario Table

- Each sentence of the scenario is a declarative sentence consisting of
  1. a subject,
  2. an action of the subject,
  3. an object that is acted upon, and
  4. possibly other objects required by the action.

- The sentences are arranged in a scenario table to
  - facilitate scenario description and
  - facilitate translation into a sequence diagram
  - Key note: for most of the rows the subject is either the subject or object acted upon from the previous row

- The approach is to
  1. highlight the subject, subject action, data or objects required by the subject action, and the object acted upon.
  2. enter these into the scenario table row by row.

Scenario Table

	Subject	Action of Subject	Other Data/Objects	Object Acted Upon
4)	Checkout GUI	checks out	call numbers	checkout controller
4.1)	checkout controller	creates		msg
4.2)	For each document call number			
4.2.1)	checkout controller	gets document	call number	DBMgr
4.2.2)	DBMgr	returns	document d	checkout controller
4.2.3)	If document exists (d!=null)			
4.2.3.1)	checkout controller	checks is available		document
4.2.3.2)	If document is available			
4.2.3.2.1)	checkout controller	creates	patron, document	Loan object
4.2.3.2.2)	checkout controller	set available to	false	document
4.2.3.2.3)	checkout controller	saves	loan	DBMgr
4.2.3.2.4)	checkout controller	saves	document	DBMgr
4.2.3.2.5)	checkout controller	appends	"checkout successful"	msg
4.2.3.3)	else			
4.2.3.3.1)	checkout controller	appends	"document not available"	msg
4.2.4)	else			
4.2.4.1)	checkout controller	appends	"document not found"	msg
4.3)	checkout controller	returns	msg	Checkout GUI
4.4)	Checkout GUI	displays	msg	Patron

Assigning Tasks to Objects

Don't fill the Subject column in yet

#	Subject	Subject Action	Other Data/Objects	Object Acted Upon
		Checkout documents	call numbers	checkout controller
		Get the document objects from the database	call numbers	DBMgr
		Create loan objects	patron, documents	loan
		Set document objects to unavailable		document
		Save loan objects to database	loan objects	DBMgr
		Save document objects to database	documents	DBMgr
		Return a checkout message		checkout GUI
		Display the checkout message		

Assigning Tasks to Objects

#	Subject	Subject Action	Other Data/Objects	Object Acted Upon
		Checkout documents	call numbers	checkout controller
			call numbers	DBMgr
			patron, documents	loan
				document
		Save loan objects to database	loan objects	DBMgr
		Save document objects to database	documents	DBMgr
		Return a checkout message		checkout GUI
		Display the checkout message		

To determine which objects perform the task (subject):

1. Look up the objects in the last column
2. Determine if the classes already exist in the domain model – don't introduce new classes

Assigning Tasks to Objects

#	Subject	Subject Action	Other Data/Objects	Object Acted Upon
		Checkout documents	call numbers	checkout controller
		Get the document objects from the database	call numbers	DBMgr
				loan
				document
				DBMgr
				DBMgr
				checkout GUI
		Display the checkout message		

To determine which object should issue the request to perform each of the tasks

1. If the task is a sub-task of a previous row then the requesting object is on the previous row
2. If the task is **<u>not</u>** a sub-task of a previous row then in most cases the requesting object is the subject of the previous row
3. For the first row, the requesting object is the GUI that receives the request and usually named after the use case

Assigning Requestors - Simple Approach

- Scenario Subject Rules - Simplified
  1. We don't depict actor steps in the Scenario
  2. Since the Exp UC starts with an actor step, it always starts with the UC GUI in the Scenario
  3. The Subject in the first step is the UC GUI
  4. The Object in the first step is the subject in the next - it has the message (information from the GUI)
  5. The Object of the second step is the Subject of the next step UNLESS the step is a subtask of the previous step. If a subtask the Subject STAYS as Subject until all subtasks are completed
  6. Since the UC finishes with an Actor confirmation it means that the Scenario ends with the UC GUI as the Subject (sending information to the actor)

Results from the First Part of Assigning Requests

#	Subject	Subject Action	Other Data/Objects	Object Acted Upon
	Checkout GUI	checkout documents	call numbers	checkout controller
	checkout controller	get documents (from database)	call numbers	DBMgr
	checkout controller	create Loan objects	patron, documents	Loan
	checkout controller	set documents to unavailable		document
	checkout controller	save Loan objects (to database)	Loan objects	DBMgr
	checkout controller	save documents (to database)	documents	DBMgr
	checkout controller	return checkout message		Checkout GUI
	Checkout GUI	display checkout message		

Subtask

Assigning Requestors (cont.)

- The are two other activities that need to be performed to complete the scenario
  1. **Returning data** - for each task that returns a result, insert a row to return the results from the object acted upon to the requesting object. There are two such tasks – checkout documents and get documents.
     a. The get documents subtask should return the documents that are requested (as the name implies). So a row indicating that the DBMgr returns the document to the checkout controller is added after the get documents row.
     b. The checkout documents subtask should return a checkout message because the checkout GUI displays this message to the patron. But, this activity is already there, so no row is added.
  2. **Conditional and loop statements** are inserted as appropriate and statement numbers are entered in the first column.

- The results of these two steps are shown on the next slide.

Completing the Scenario Description

#	Subject	Subject Action	Other Data/Objects	Object Acted Upon
3	checkout GUI	Checkout documents	call numbers	checkout controller
4.1	checkout controller	gets document(s)	call numbers	DBMgr
4.2	DBMgr	returns document(s)		checkout controller
4.3	for each d in documents			
4.3.1	if d is available			
4.3.1.1	checkout controller	Create loan object	patron, d	loan
4.3.1.2	checkout controller	set available	false	DBMgr
4.3.1.3	checkout controller	save	loan object	DBMgr
4.3.1.4	checkout controller	save	d	DBMgr
4.3.1.5	checkout controller	writes	d is checked out successfully	checkout message
4.3.2	else			
4.3.2.1	checkout controller	writes	d is not available	checkout message
4.4	checkout controller	return checkout message		checkout GUI
4.5	checkout GUI	display checkout message		

Converting Scenario Tables to Diagrams

- Converting the scenario tables to diagrams involves the following three steps
  1. Converting scenario tables to sequence diagrams. In this step, an informal sequence table is derived from each scenario table

Subject	Object	Case
Actor	Object	1
Object	Actor	2
Object	Object	3
Object	Itself	4

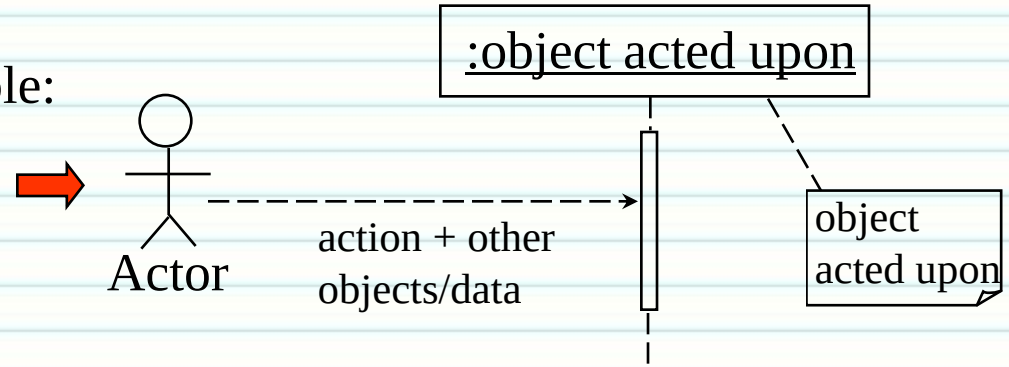
See next diagrams for the specific rules related to each case

  2. Deciding on instance names and types. In this step the names and types of the object instances that send and receive messages are defined
  3. In this step the function names, parameters and return types are determined

Case 1

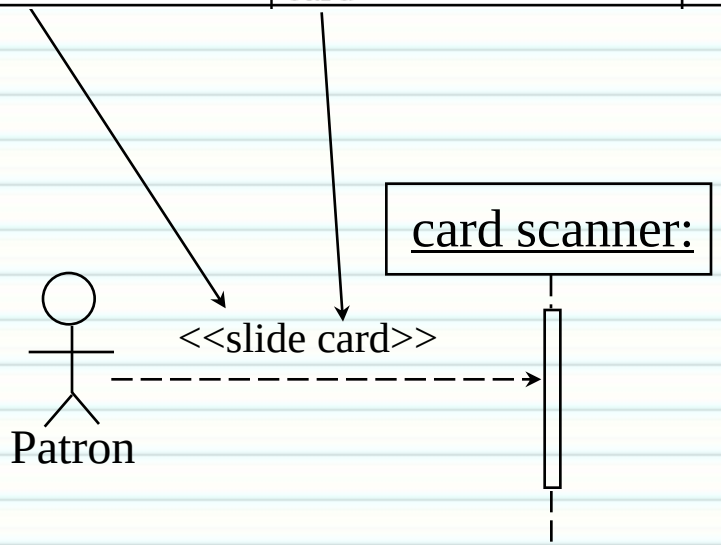
for each line of the scenario table:

Case 1: subject is an Actor.



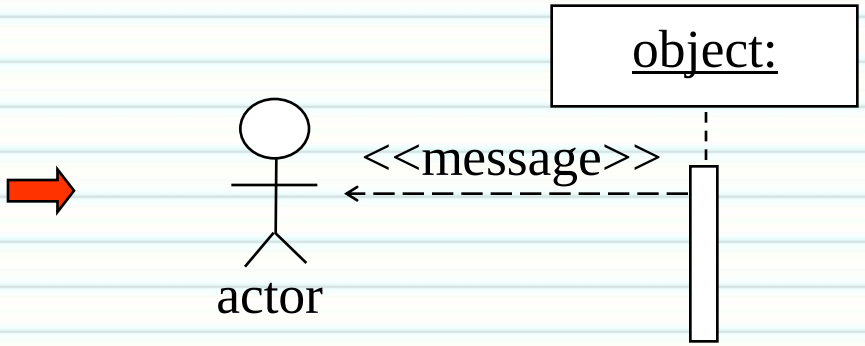
Example:

	subject	action/message	other objects/data	object acted
1	patron	slide	card	through card scanner

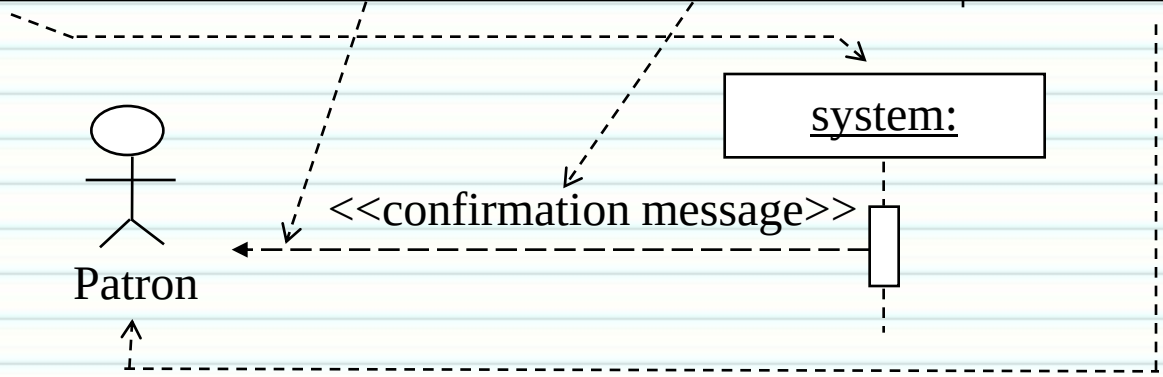


Case 2

Case 2: subject is an object and object acted upon is an actor.

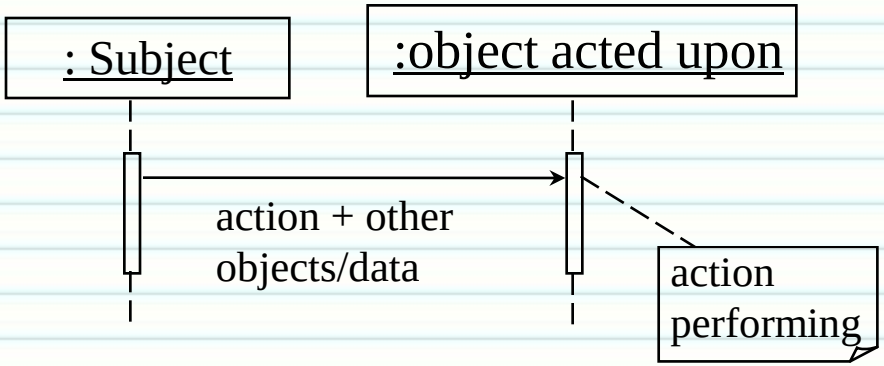


Subject	Subject Action	Other Data / Objects	Object Acted Upon
system	displays	confirmation message	Patron



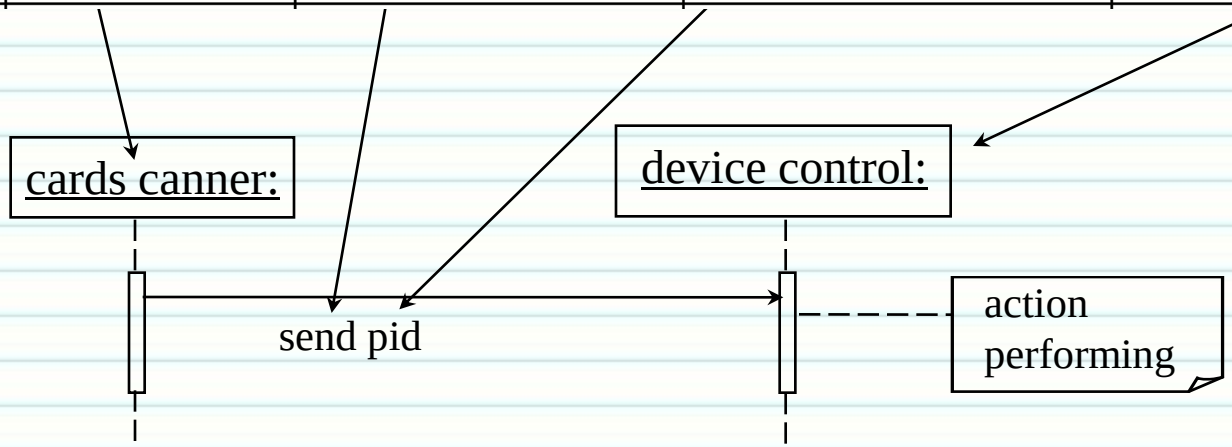
Case 3

Case 3: both subject and object acted upon are objects.



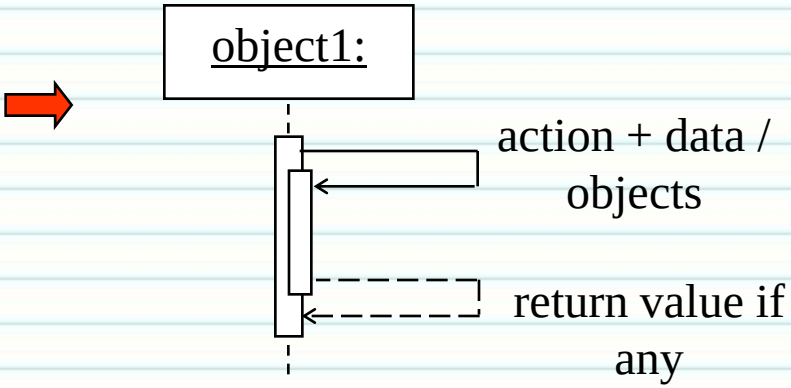
Example

	subject	action/message	other objects/data	object acted
1	patron	slide	card	through card scanner
2	card scanner	read	patron id (pid)	from card
3	card scanner	send	pid	to device control

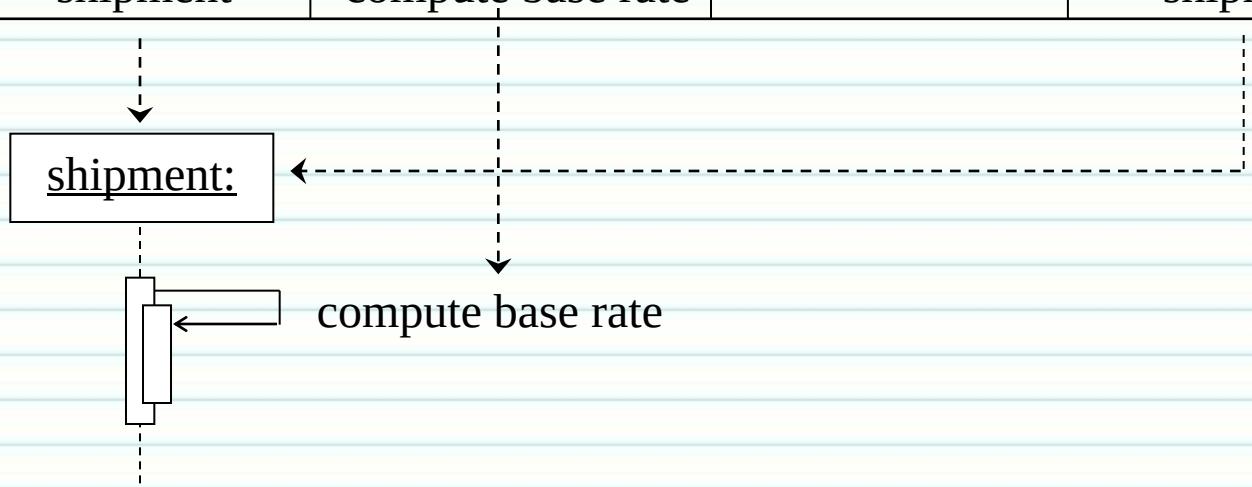


Case 4

Case 4: both subject and object acted upon are the same (a special case of case 3).



Subject	Subject Action	Other Data/ Objects	Object Acted Upon
shipment	compute base rate		shipment



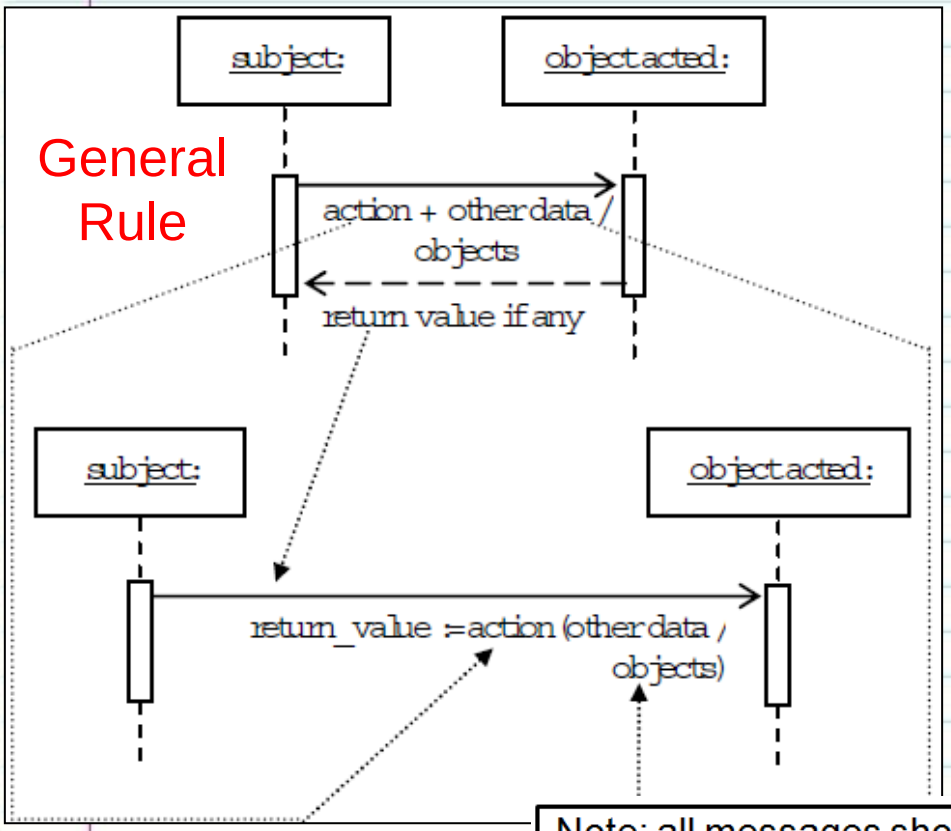
Deciding on Instance Names and Instance Types

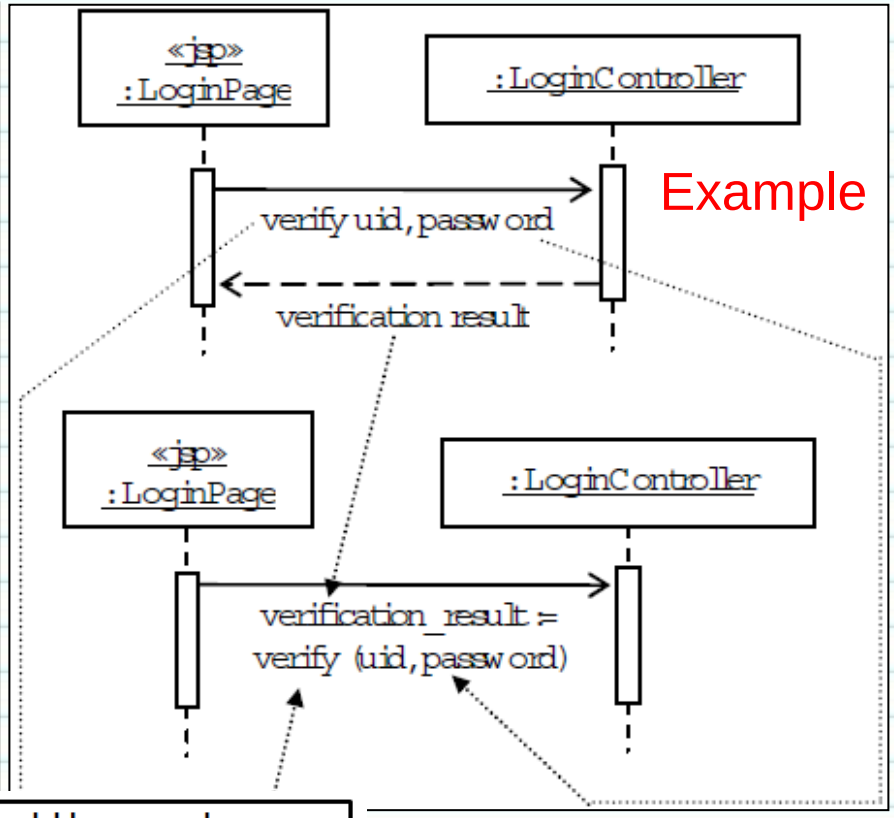
- Recall that when a scenario table is converted into a sequence diagram, the instance names and instance types are not specified.

- Deciding on instance names
  1. Give parameters or return values in the sequence diagram a name
  2. If the instance has a connotation outside the sequence diagram (Java Server Page) then give it a name and make it a stereotype instance e.g., <<html>>WelcomePage:

Formal Parameter Types on Sequence Diagrams

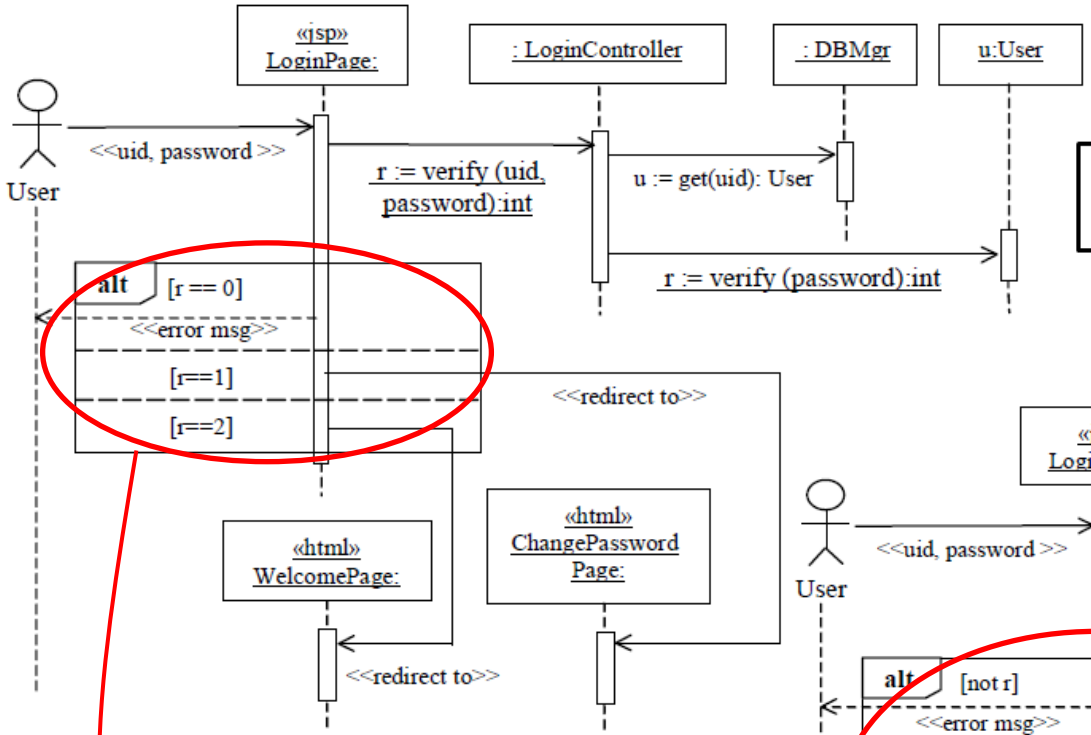
#	Subject	Subject Action	Other Data/Objects	Object Acted Upon
2	LoginPage	verifies	uid, password	LoginController



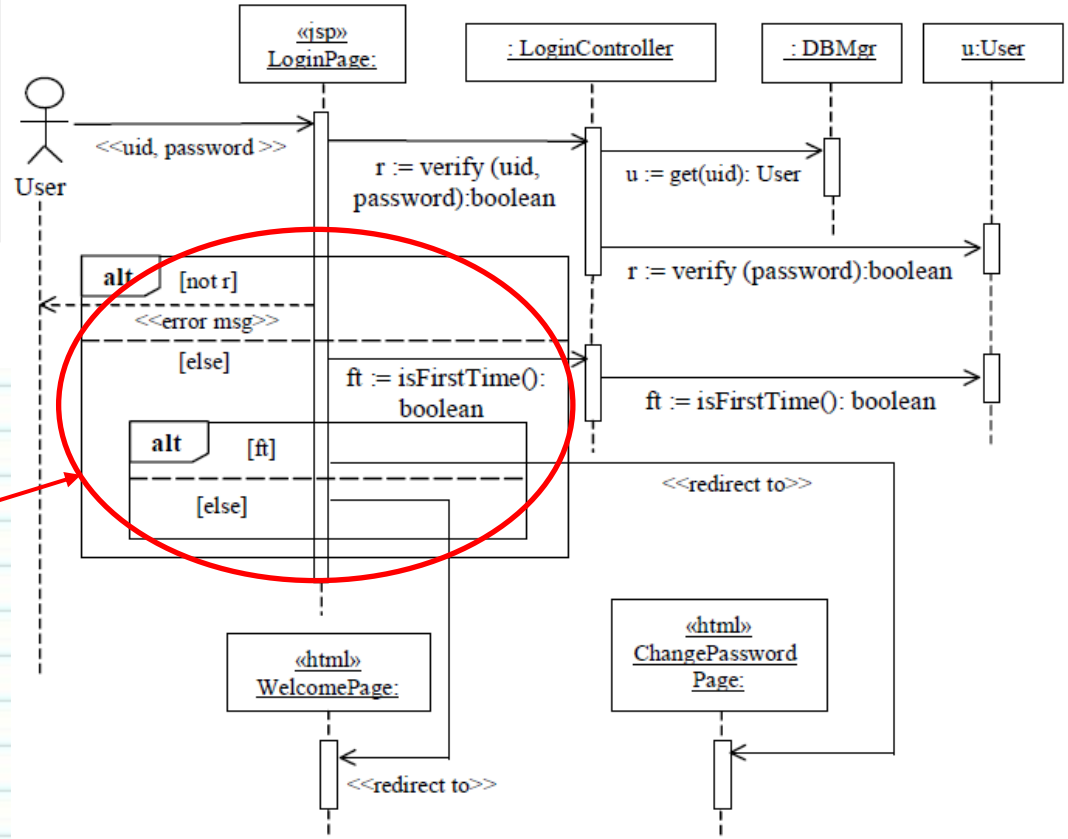


Note: all messages should be synchronous →

Object Interaction Coupling



Note: all messages should be synchronous

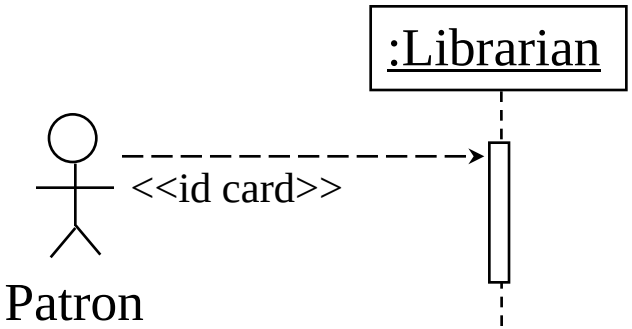


Please only put fragments on Controller

Much better coupling

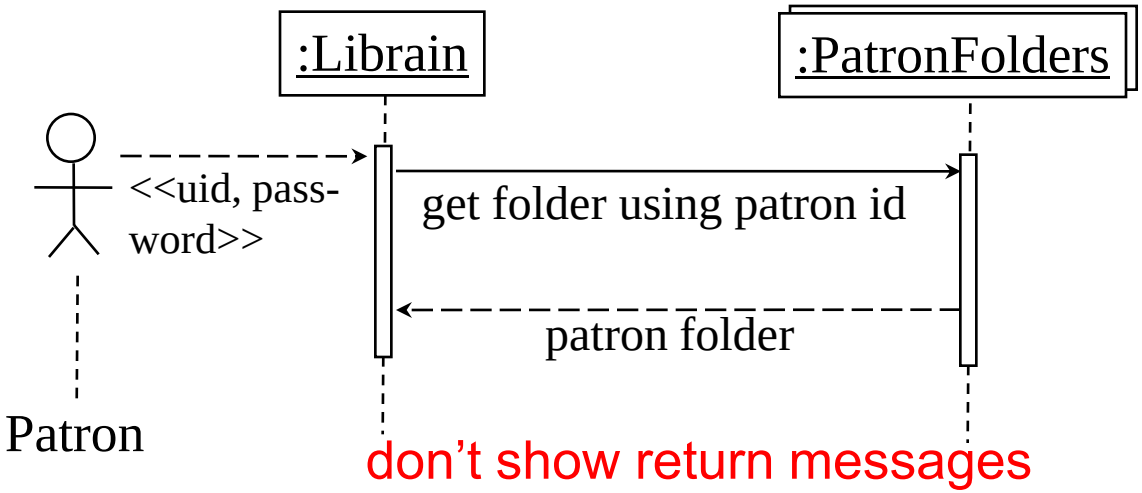
Modeling a Manual Library System

Patron presents id card to librarian.

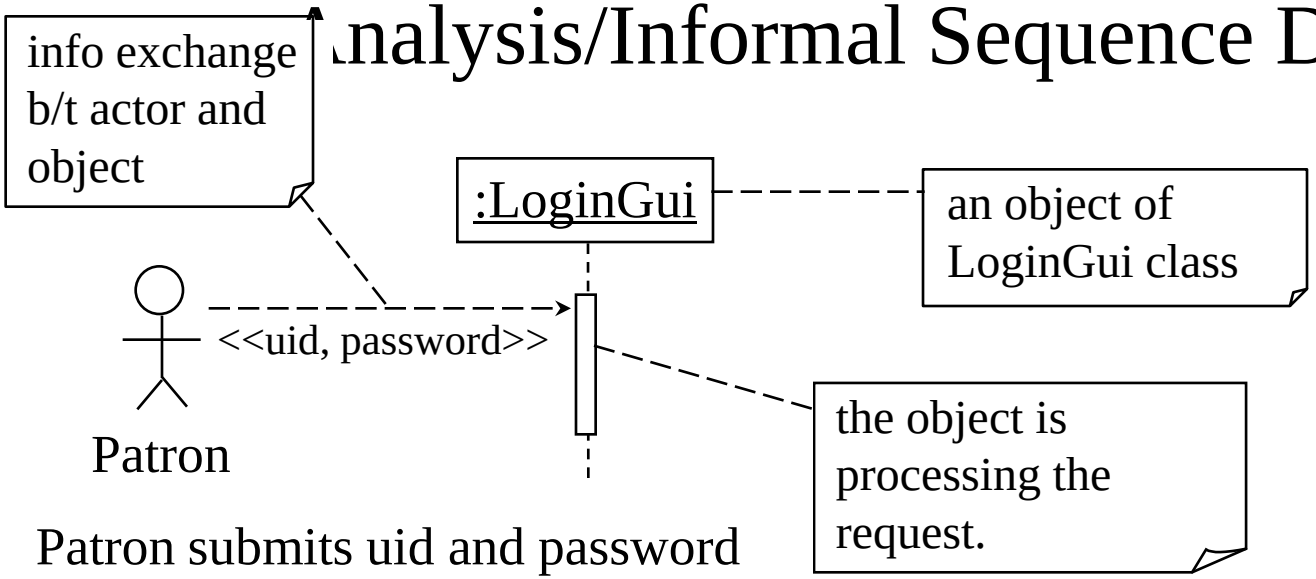


These are analysis sequence diagrams that model an existing system.

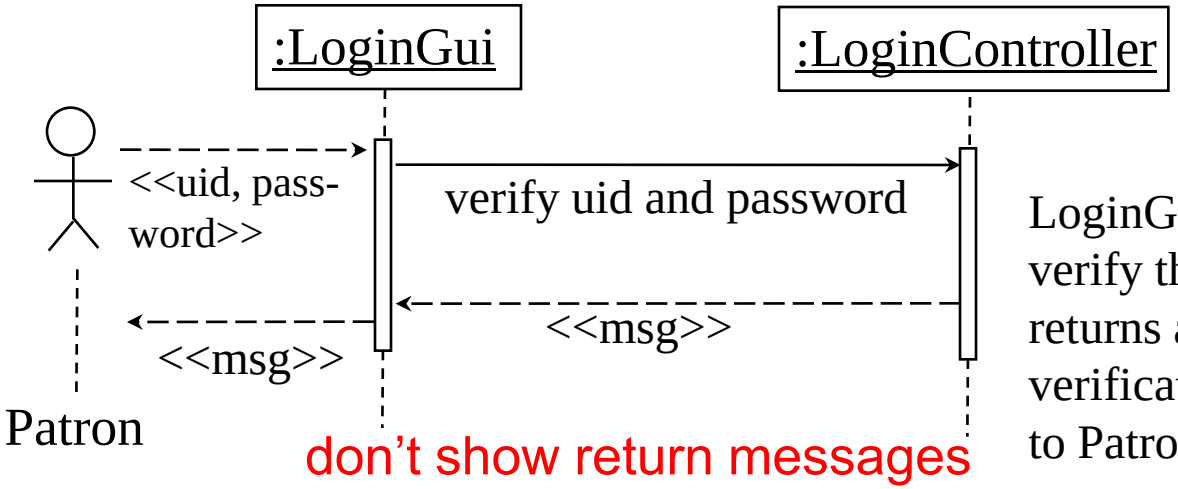
Librarian pulls out patron's folder using id number.



Analysis/Informal Sequence Diagram

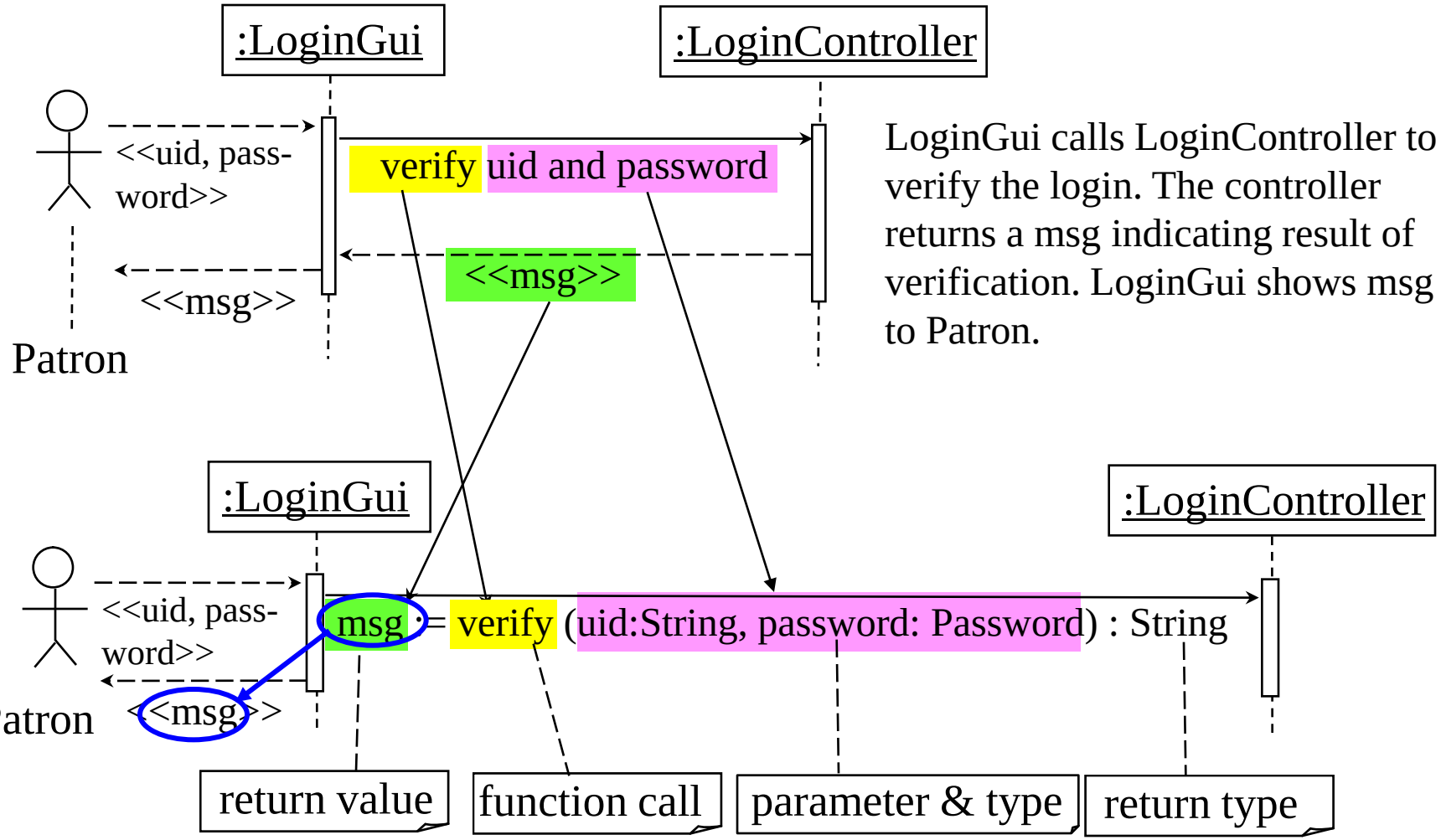


Patron submits uid and password to LoginGui.



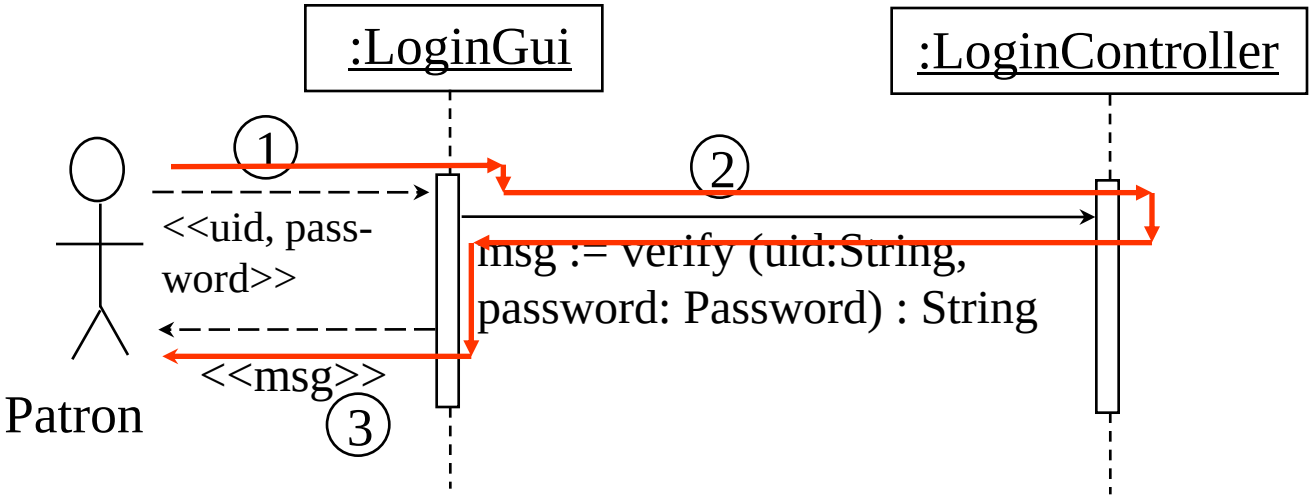
LoginGui calls LoginController to verify the login. The controller returns a msg indicating result of verification. LoginGui shows msg to Patron.

From Analysis to Design



LoginGui calls LoginController to verify the login. The controller returns a msg indicating result of verification. LoginGui shows msg to Patron.

Sequence Diagram: Flow of Control

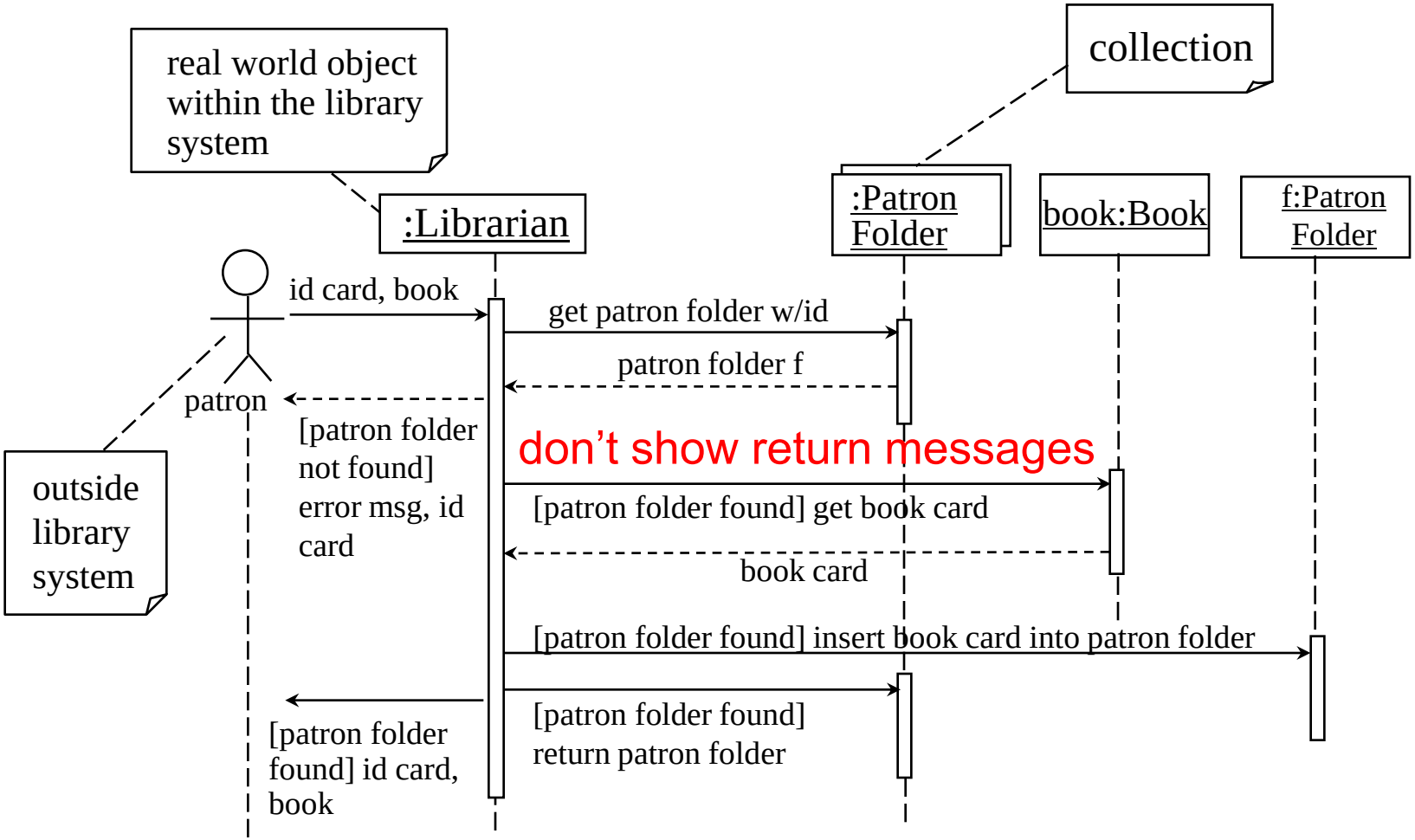


(1) Patron submits uid and password to LoginGui object.

(2) LoginGui object calls the verify function of a LoginController object. The called function returns msg of type String.

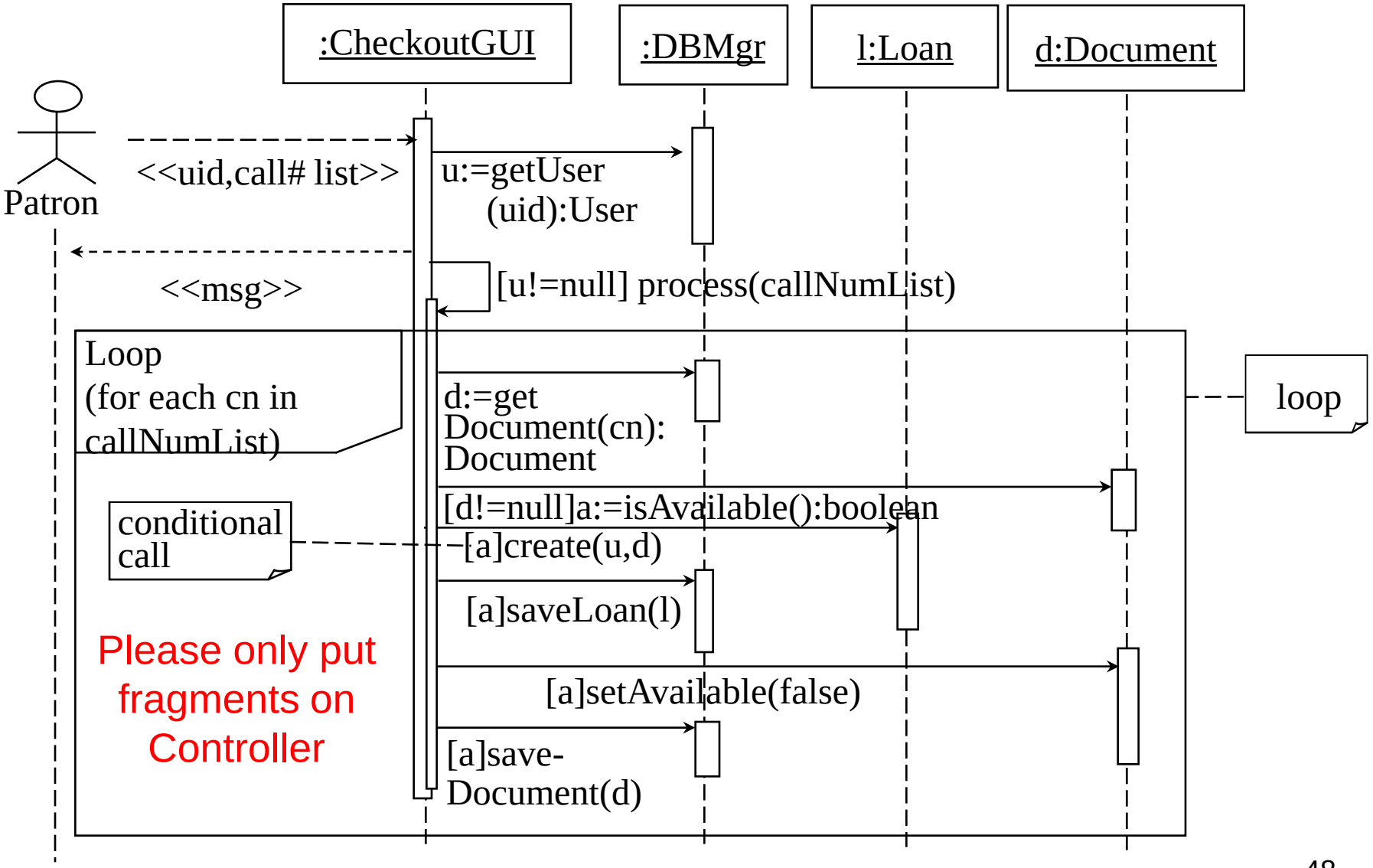
(3) The LoginGui object shows msg to patron.

An Analysis Sequence Diagram

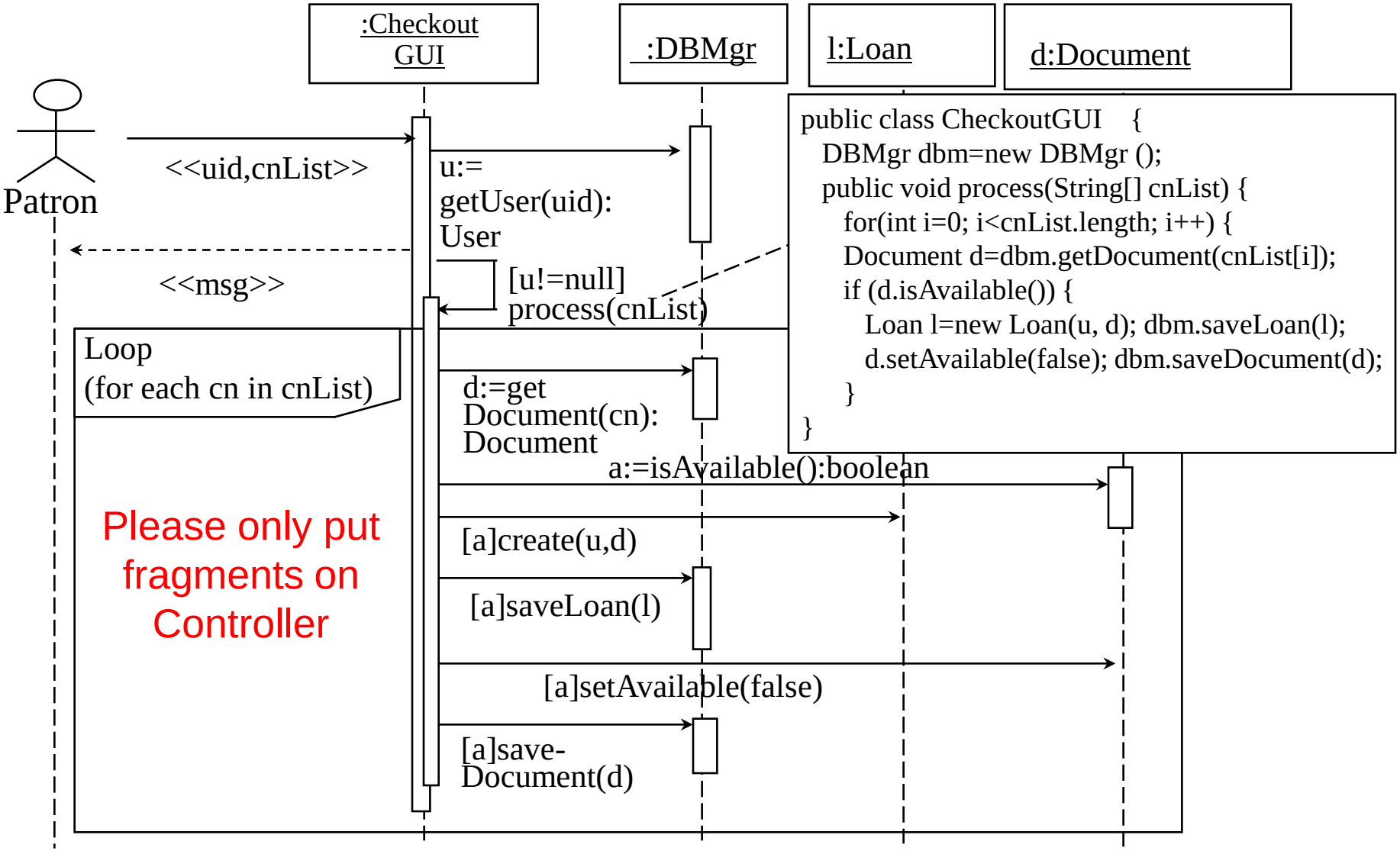


This an analysis sequence diagram models the current, manual operation, little design decision is made.

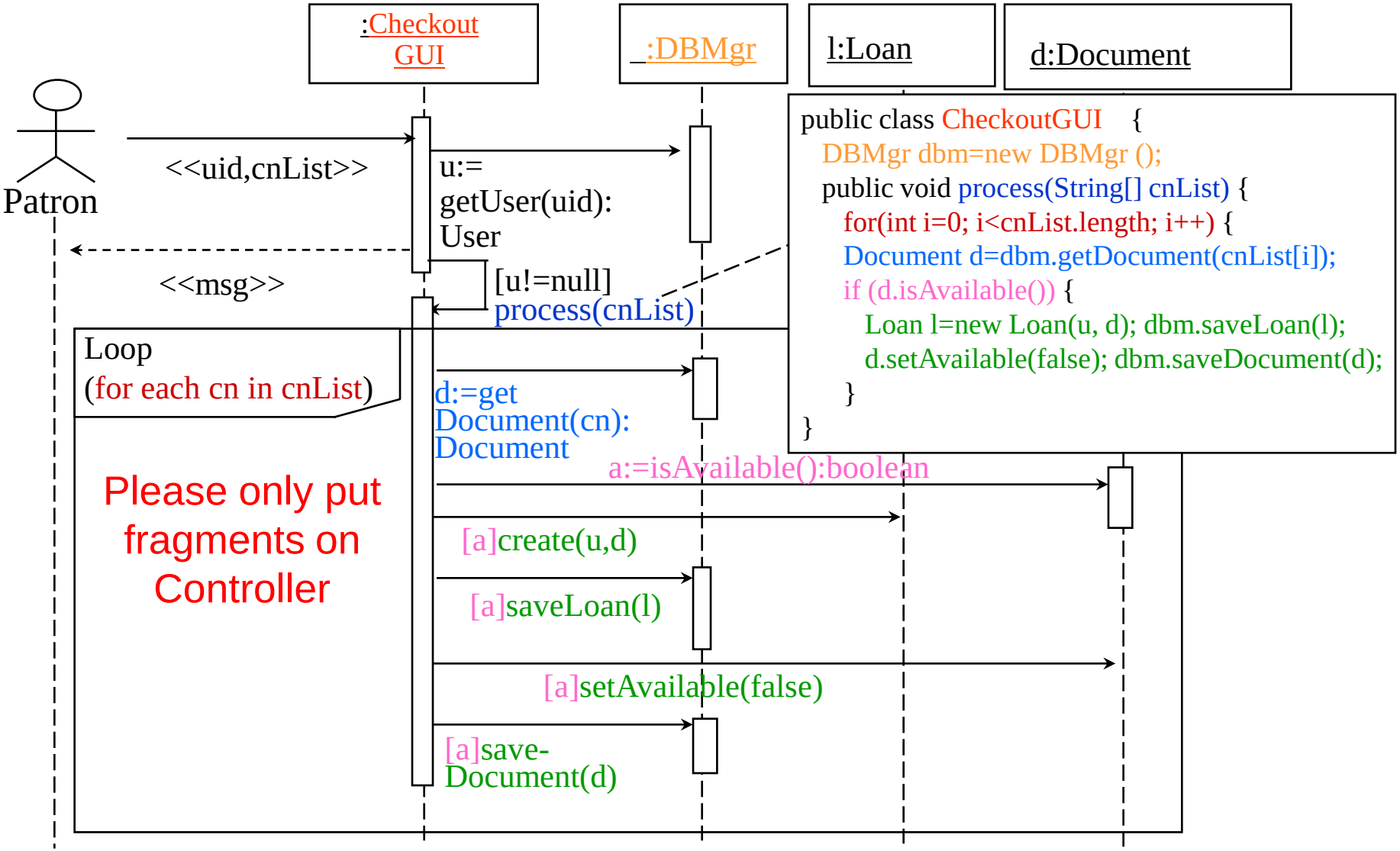
A Design Sequence Diagram



From Sequence Diagram to Implementation

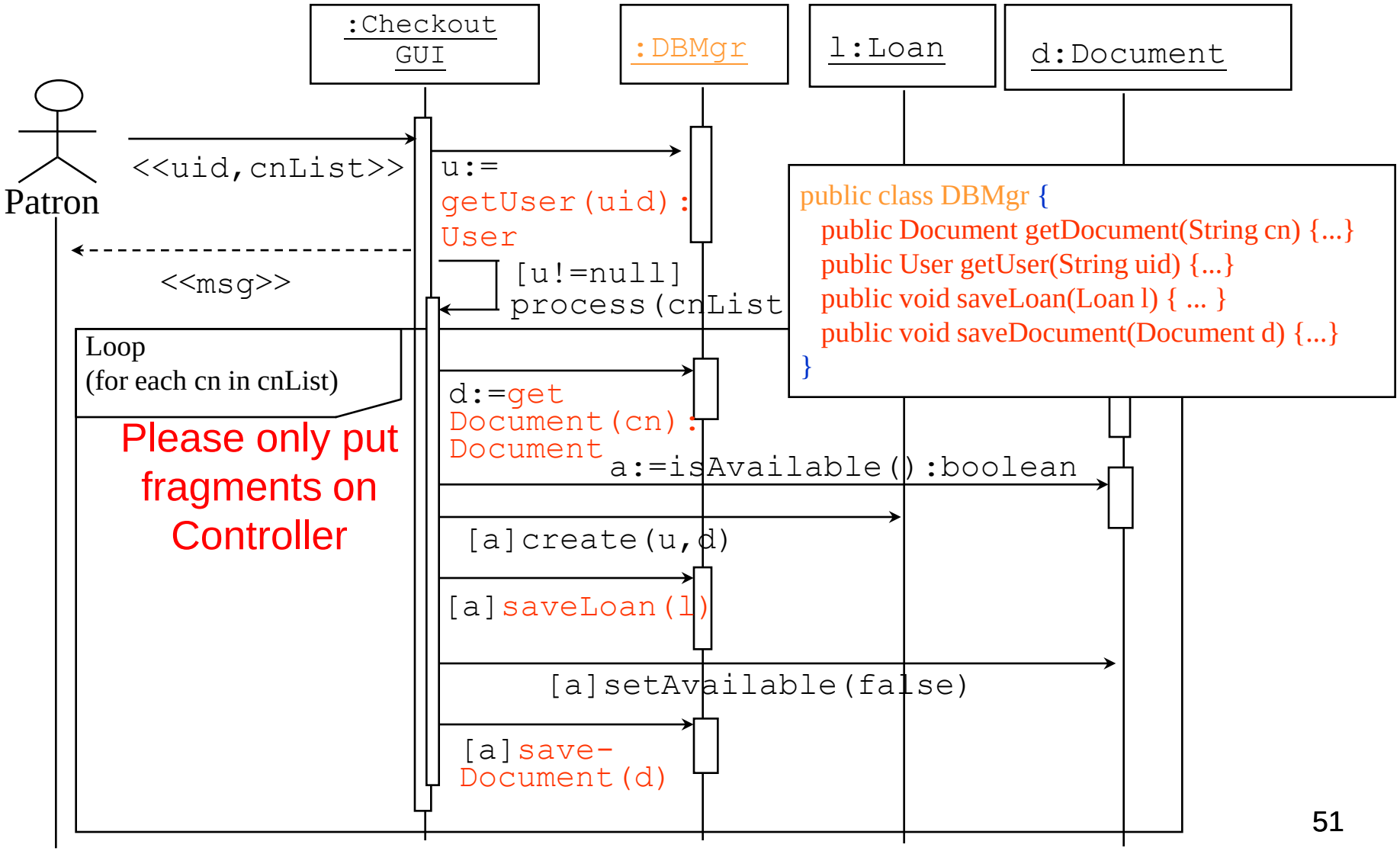


From Sequence Diagram to Implementation

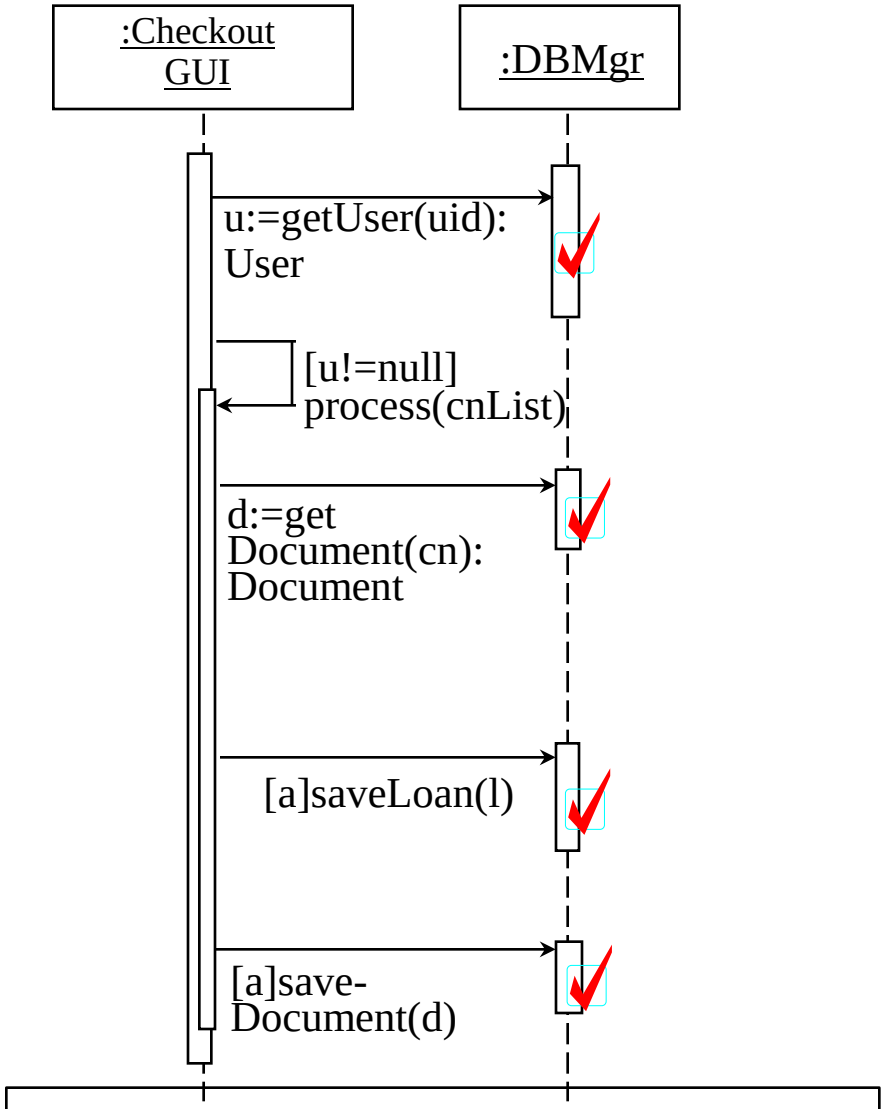


Note the color correspondence in SD and code.

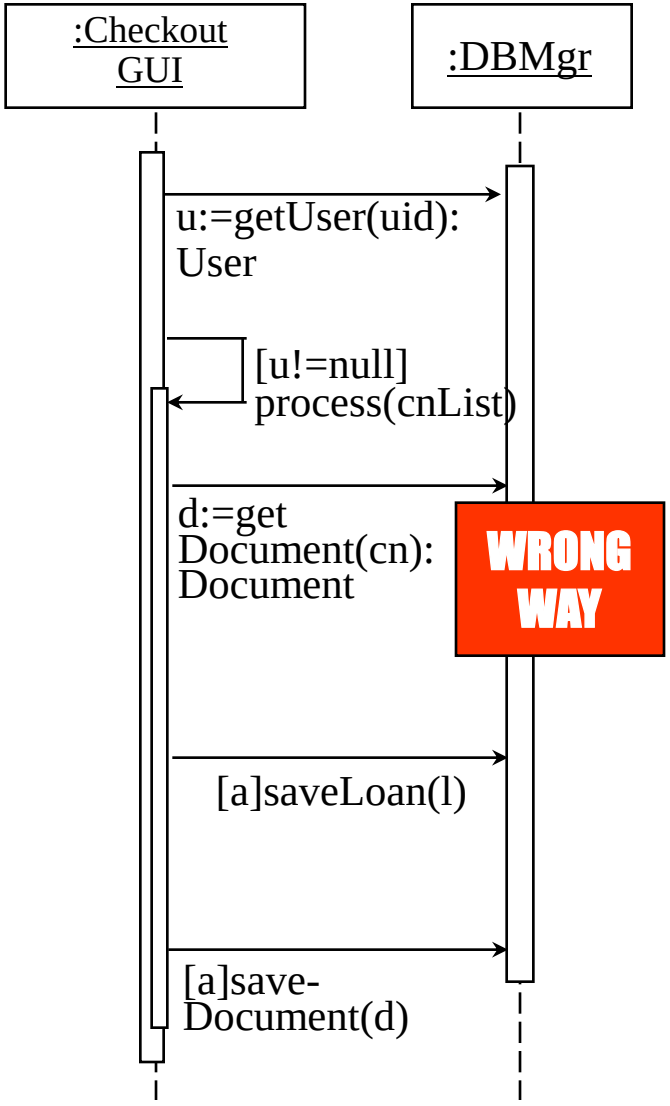
From Sequence Diagram to Implementation



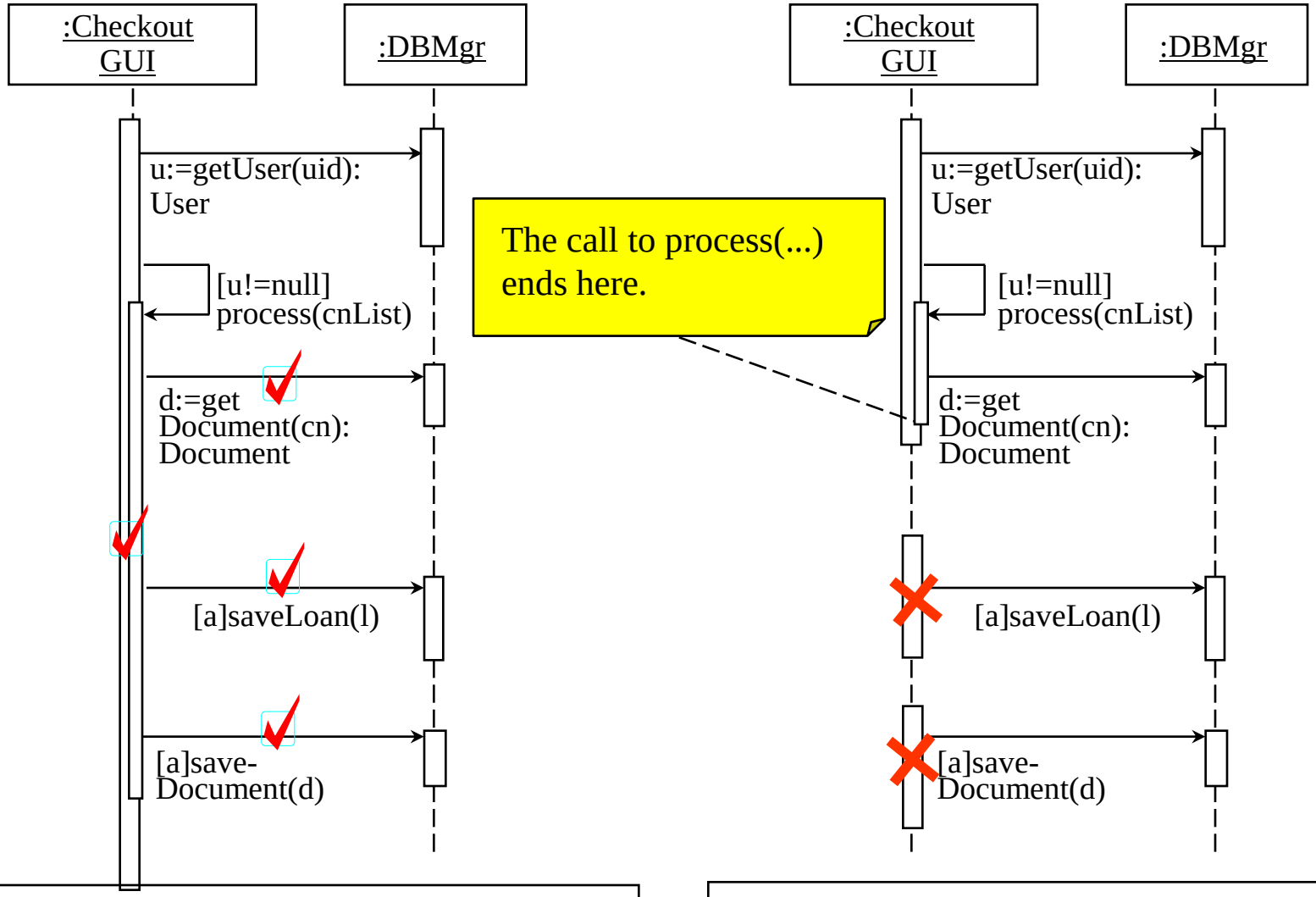
Commonly Seen Mistakes



Correct: 4 function calls to DBMgr; the functions are executed during 4 time intervals.



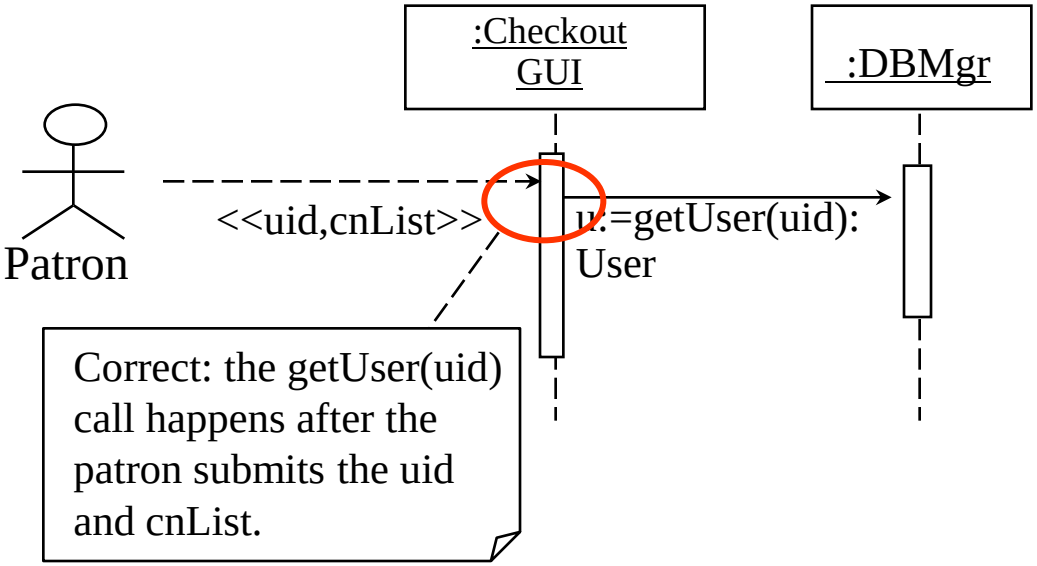
Commonly Seen Mistakes

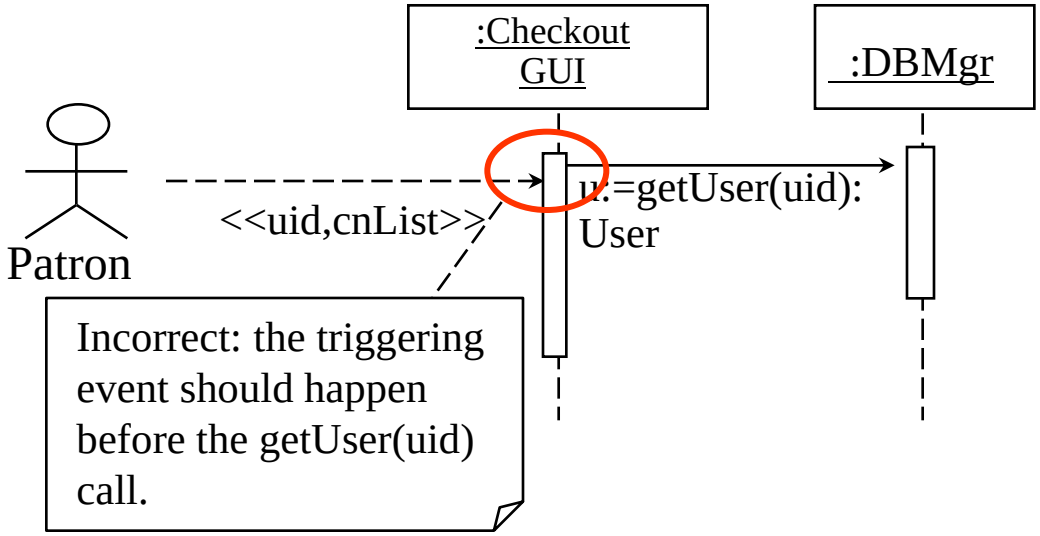


Correct: during execution of process(...), 3 function calls are made.

The last two calls need to be initiated. But even though the semantics is still incorrect.

Commonly Seen Mistakes

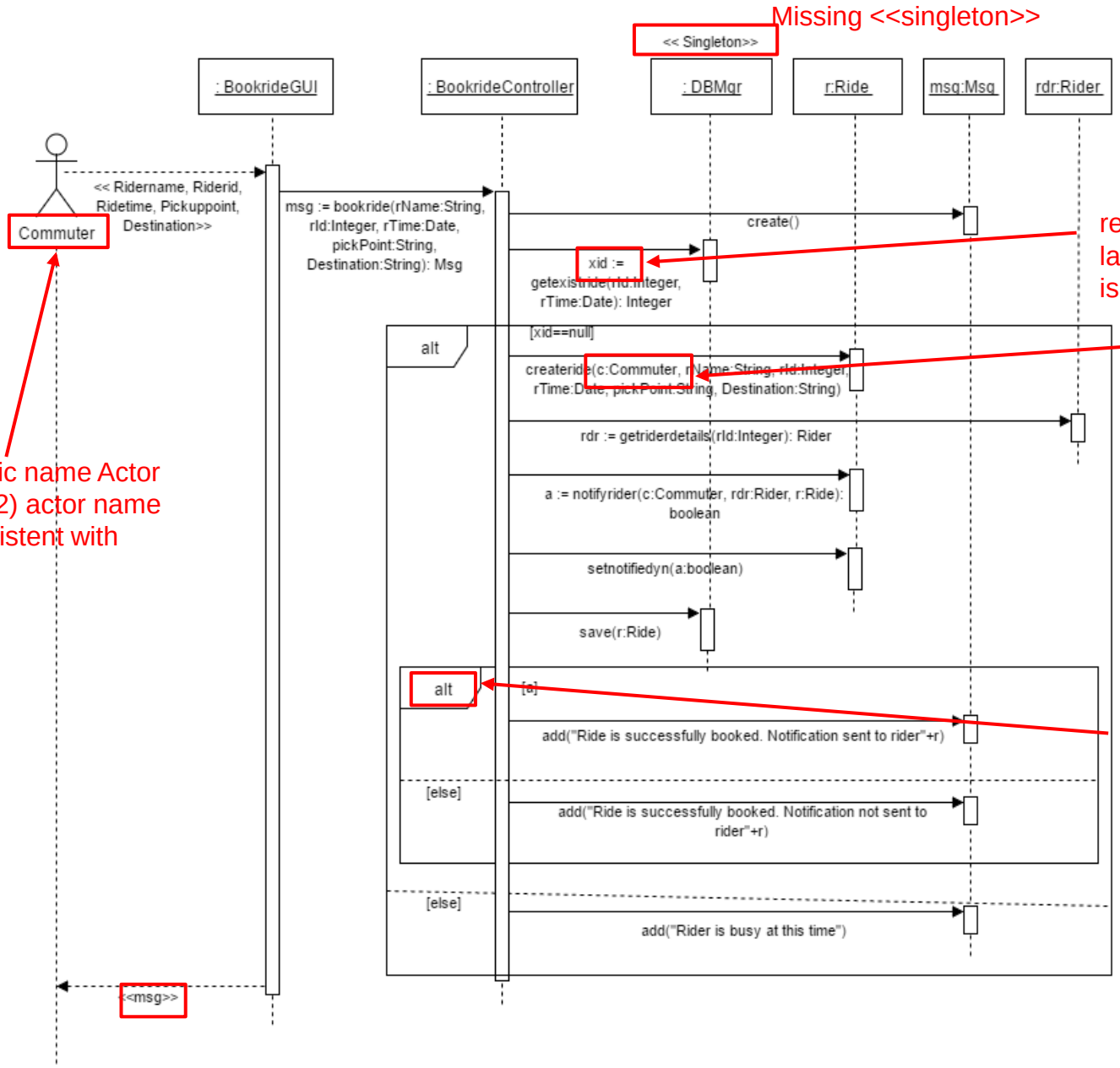




Example Student Project

Book Ride

Common mistakes identified



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

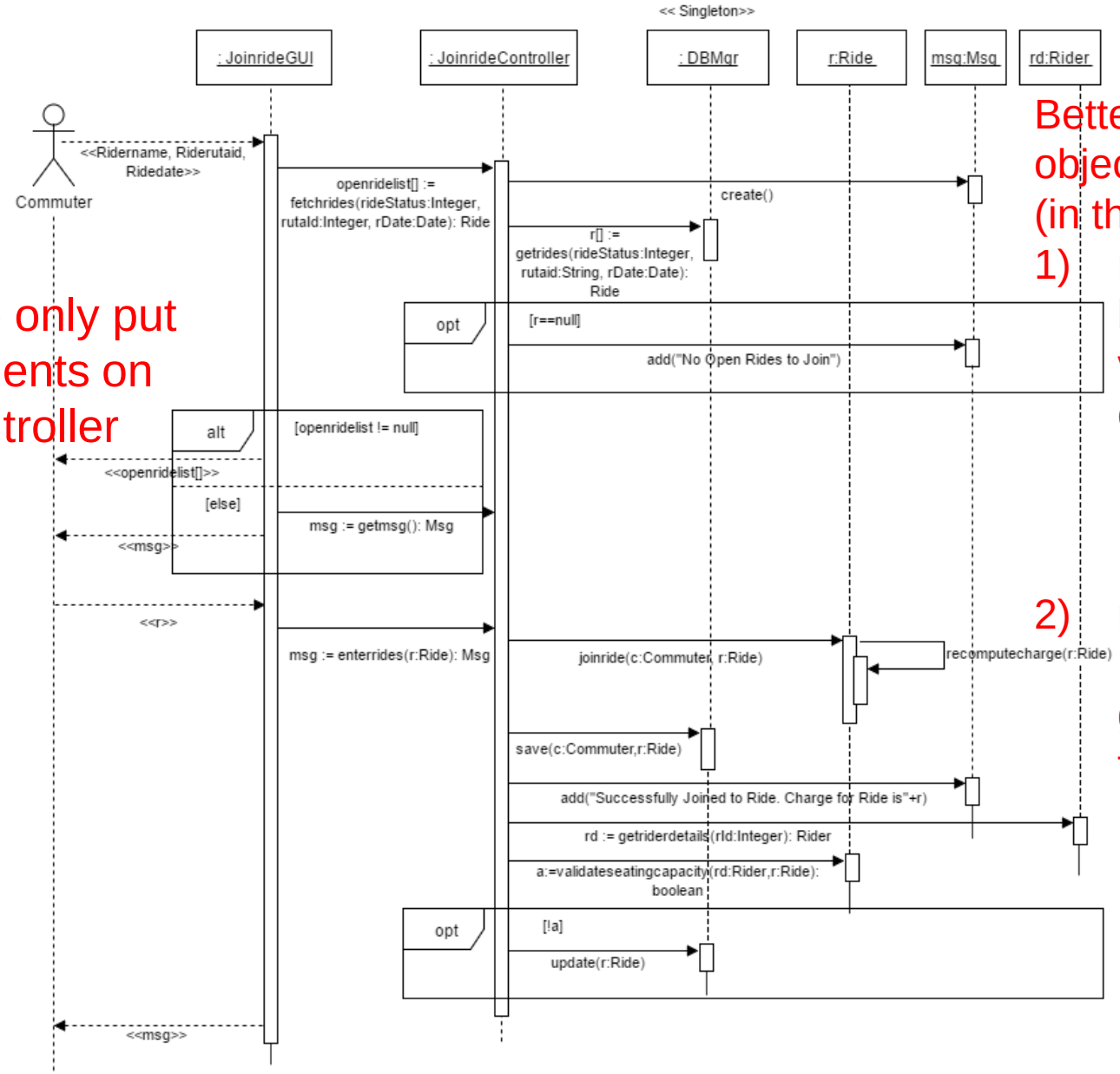
new parameter comes from nowhere! error

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



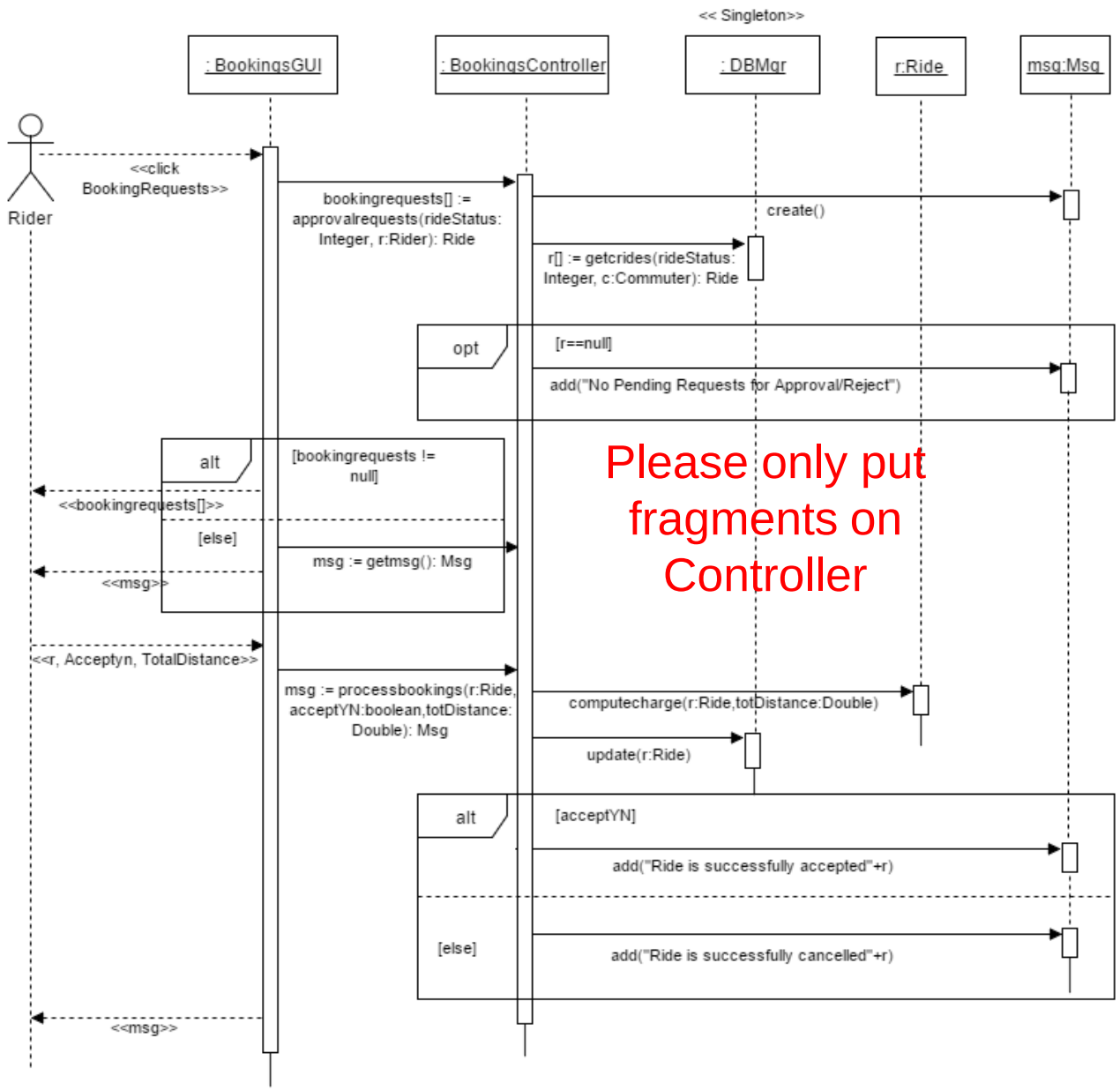
Please only put fragments on Controller

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

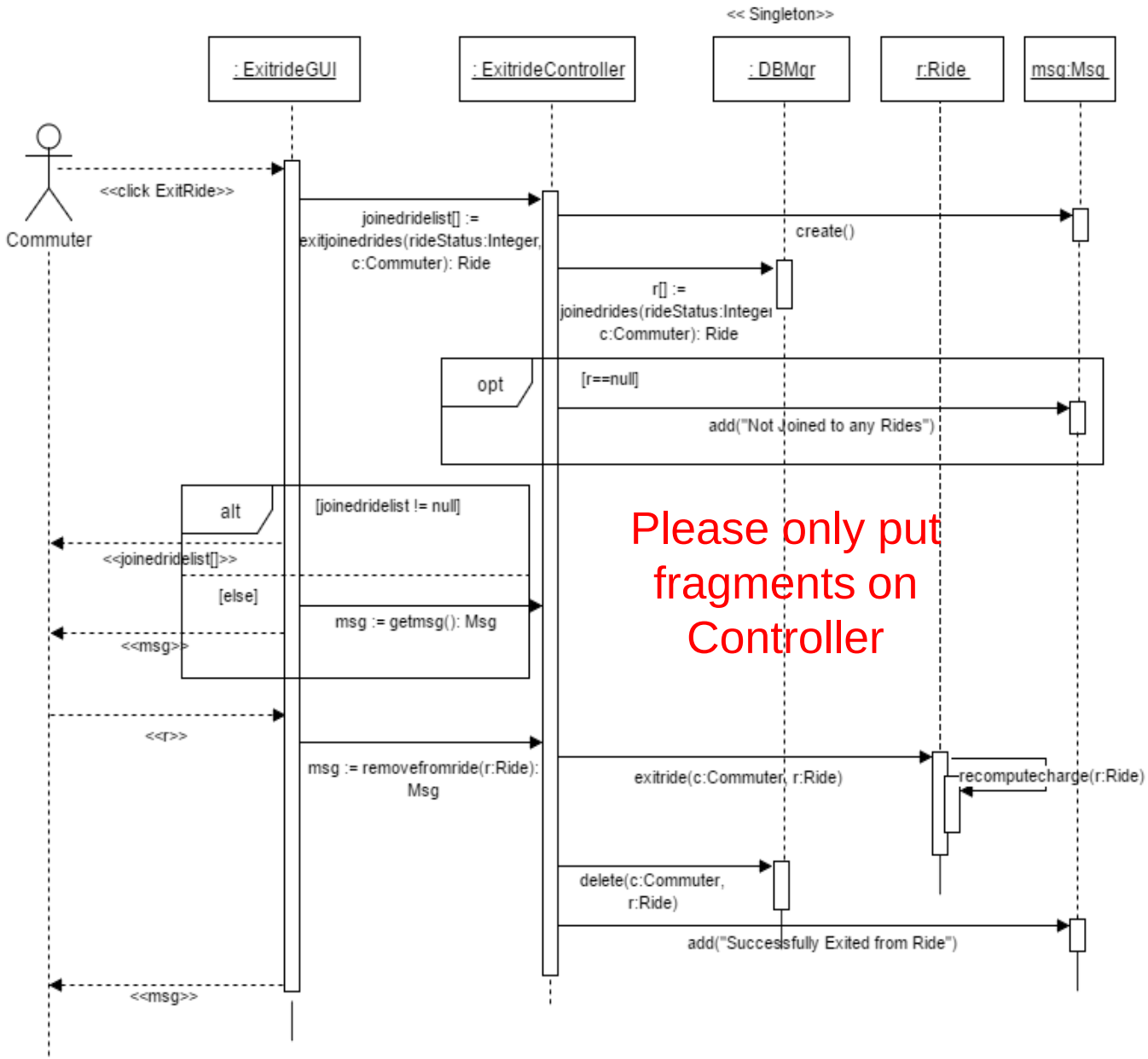
1) message with boolean return value to Ride object to validate Ride data from User or to check Ride rules

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

Accept/Reject Booking Request



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