Because learning changes everything.®

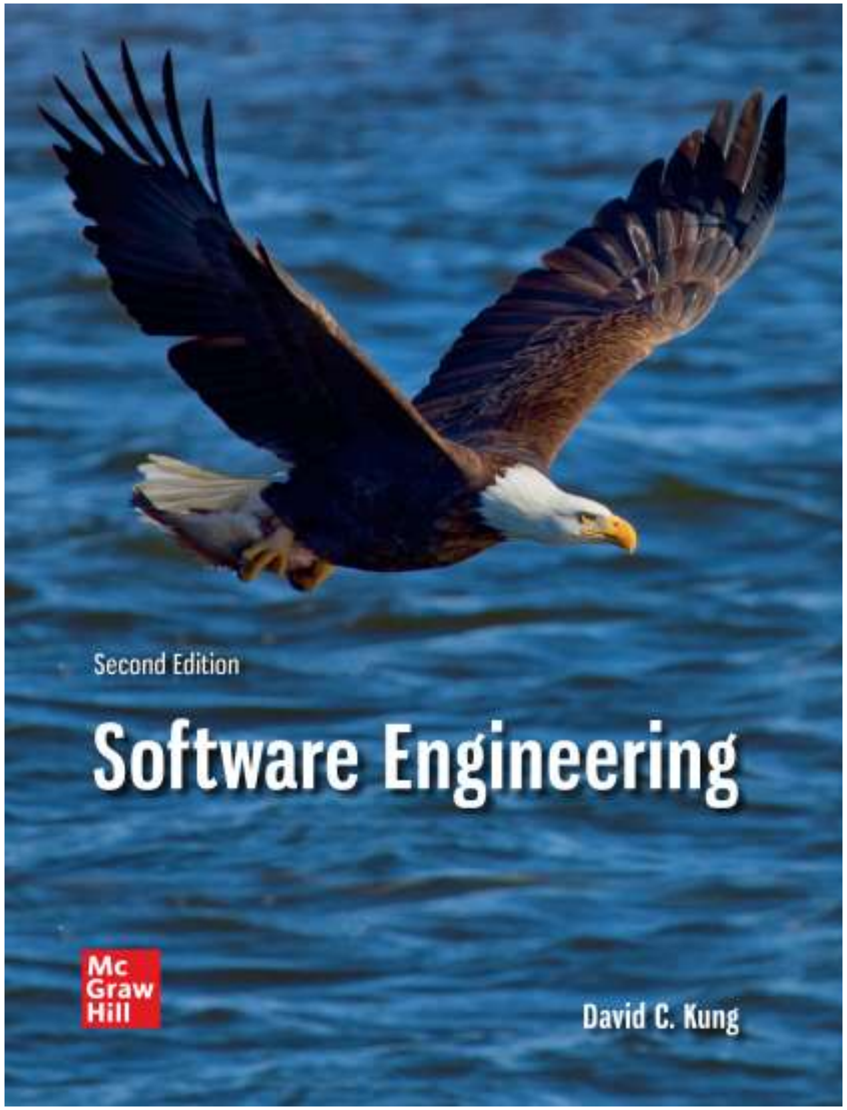
Chapter 10

Applying Responsibility Assignment Patterns

Software Engineering

Second Edition

David C. Kung



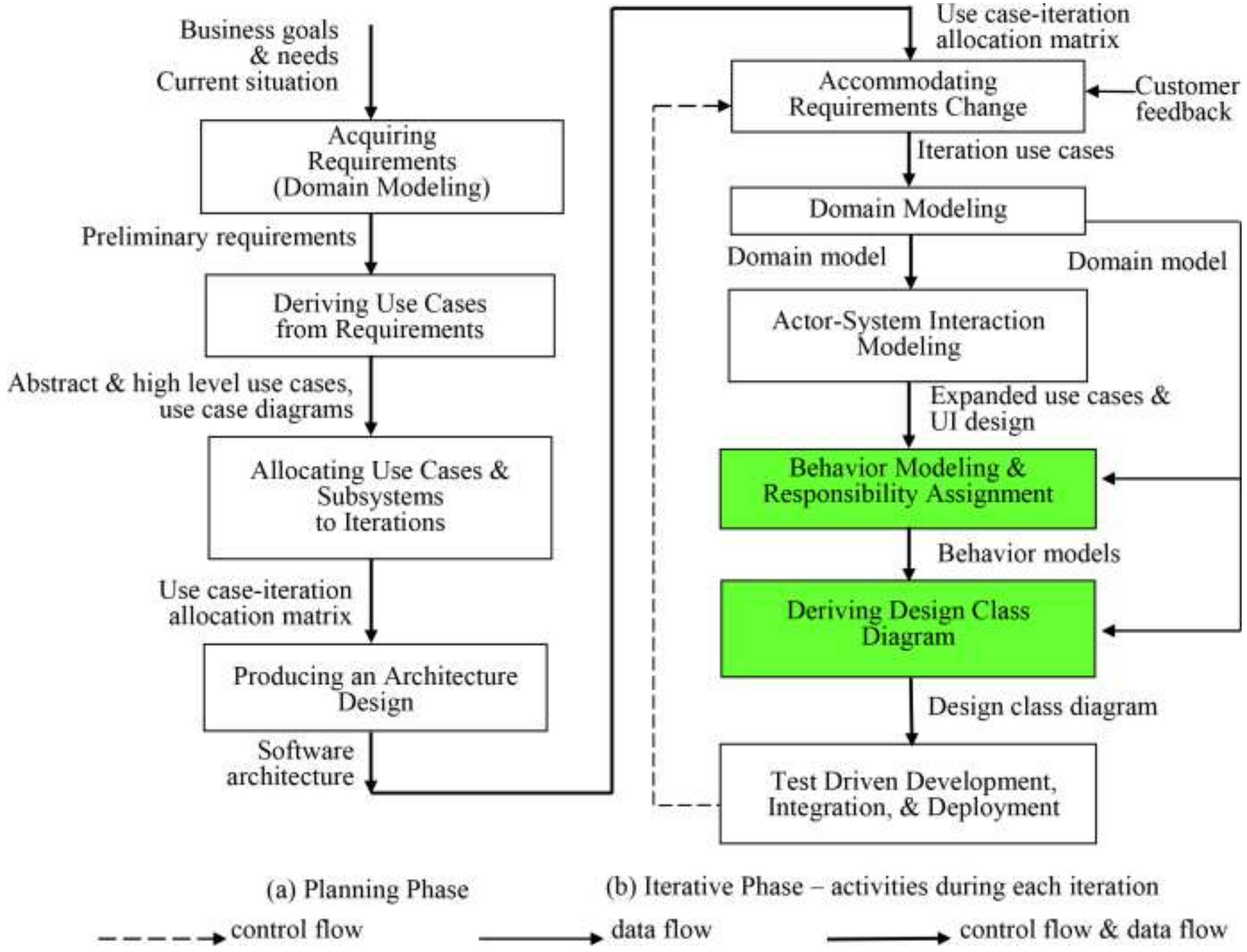
Key Takeaway Points

Design patterns are abstractions of proven design solutions to commonly encountered design problems.

The controller, expert, and creator patterns are

- Are general responsibility assignment software patterns (GRASP)
- They are applicable to almost all object-oriented systems.

Applying Patterns in the Methodology Context


Business goals & needs
Current situation
Acquiring Requirements (Domain Modeling)
Preliminary requirements
Deriving Use Cases from Requirements
Abstract & high level use cases, use case diagrams
Allocating Use Cases & Subsystems to Iterations
Use case-iteration allocation matrix
Producing an Architecture Design
Software architecture
Use case-iteration allocation matrix
Accommodating Requirements Change
Customer feedback
Iteration use cases
Domain Modeling
Domain model
Domain model
Actor-System Interaction Modeling
Expanded use cases & UI design
Behavior Modeling & Responsibility Assignment
Behavior models
Deriving Design Class Diagram
Design class diagram
Test Driven Development, Integration, & Deployment
(a) Planning Phase
(b) Iterative Phase – activities during each iteration
control flow
data flow
control flow & data flow

Access the text alternative for these images

What Are Design Patterns?

Design patterns are proven design solutions to commonly encountered design problems.

Patterns codify software design principles and idiomatic solutions.

Each pattern solves a class of design problems.

Patterns can be combined to solve a large complex design problem.

Design patterns improve communication among software developers.

- Face-to-face, across space, across time.

Design patterns empower less experienced developers to produce high-quality software.

Example: The Singleton Pattern 1

Pattern name: Singleton

Design Problem: How do we ensure that a class has at most one globally accessible instance?

Example uses:

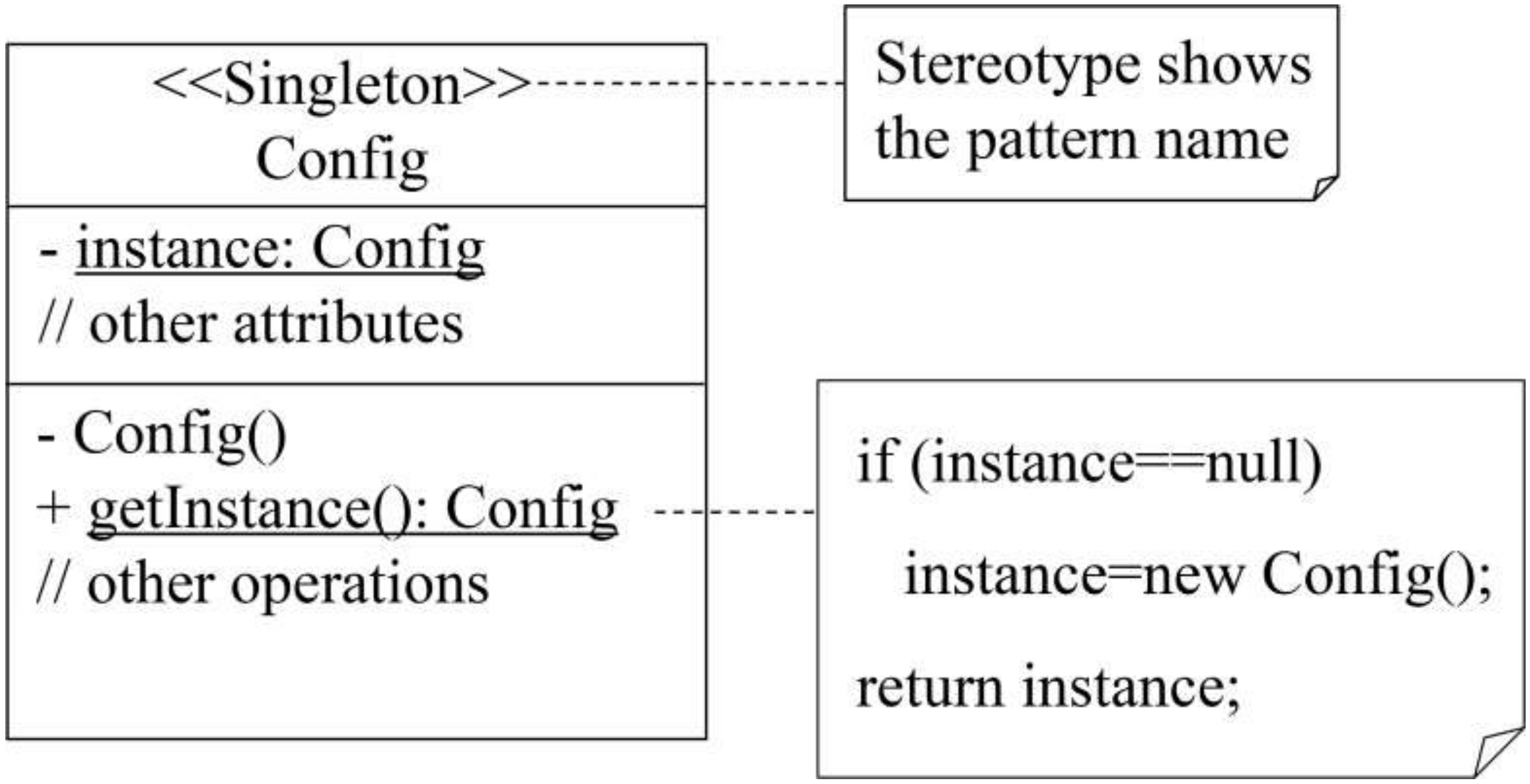
- System configuration class
- System log file
- Library, or product catalog

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
   }
```

Note: Singleton may be used to produce a limit number of instances, not just at most one instance.

Example: The Singleton Pattern 2

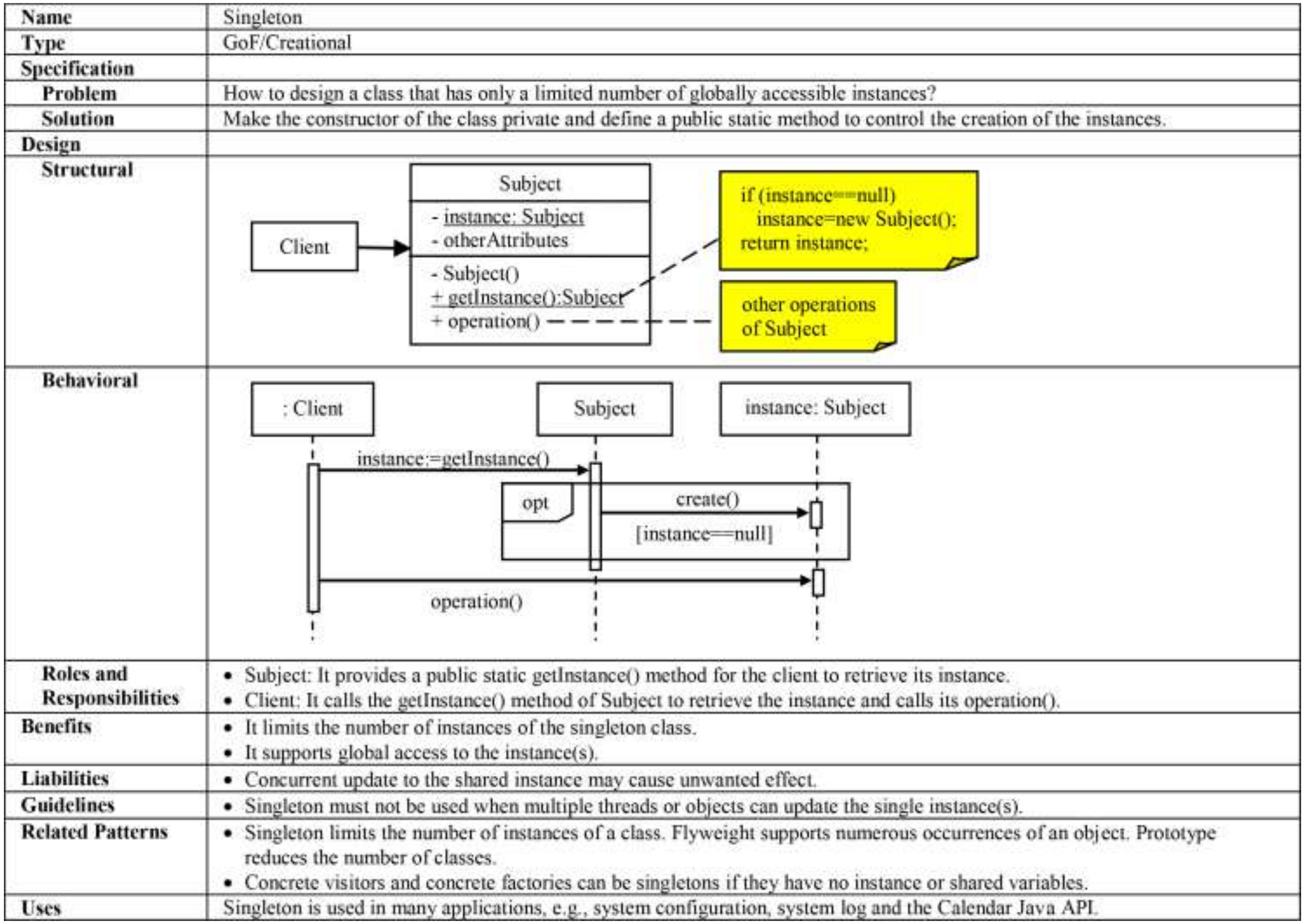


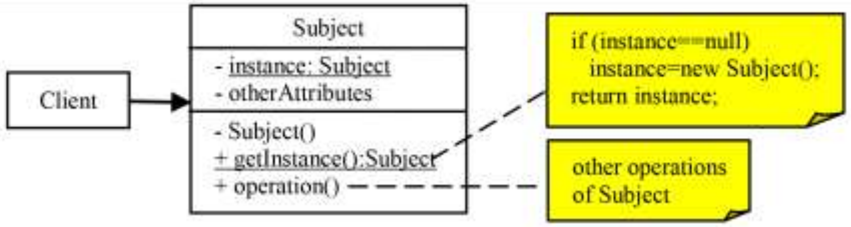
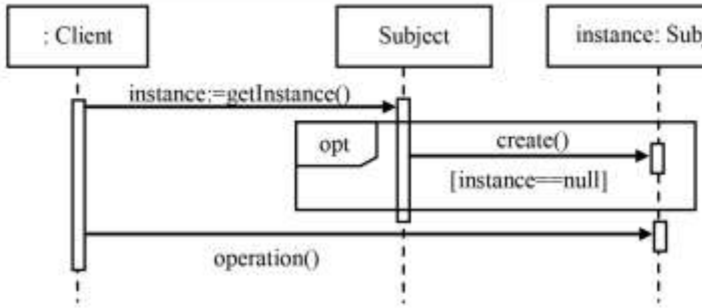
```
if (instance==null)
   instance=new Config();
return instance;
```

+: public   -: private   underlined: static

Access the text alternative for these images

Specification of Singleton



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	 Client Subject - instance: Subject - otherAttributes - Subject() + getInstance():Subject + operation() if (instance==null) instance=new Subject(); return instance; other operations of Subject
Behavioral	 : Client Subject instance: Subject instance:=getInstance() opt create() [instance==null] operation()
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 of the singleton class. • It supports global access to the instance(s).
Liabilities	• Concurrent update to the shared instance may cause unwanted effect.
Guidelines	• Singleton must not be used when multiple threads or objects can update the single instance(s).
Related Patterns	• Singleton limits the number of instances of a class. Flyweight supports numerous occurrences of an object. Prototype reduces the number of classes. • Concrete visitors and concrete factories can be singletons if they have no instance or shared variables.
Uses	Singleton is used in many applications, e.g., system configuration, system log and the Calendar Java API.

Access the text alternative for these images

Another Version of Singleton Pattern

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

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

More About Design Patterns

Patterns are recurring designs.

Patterns are not new designs.

Most patterns aim at improving the maintainability of the software system.

- easy to understand
- easy to test
- easy to change (significantly reduce change impact)

Some patterns also improve efficiency or performance.

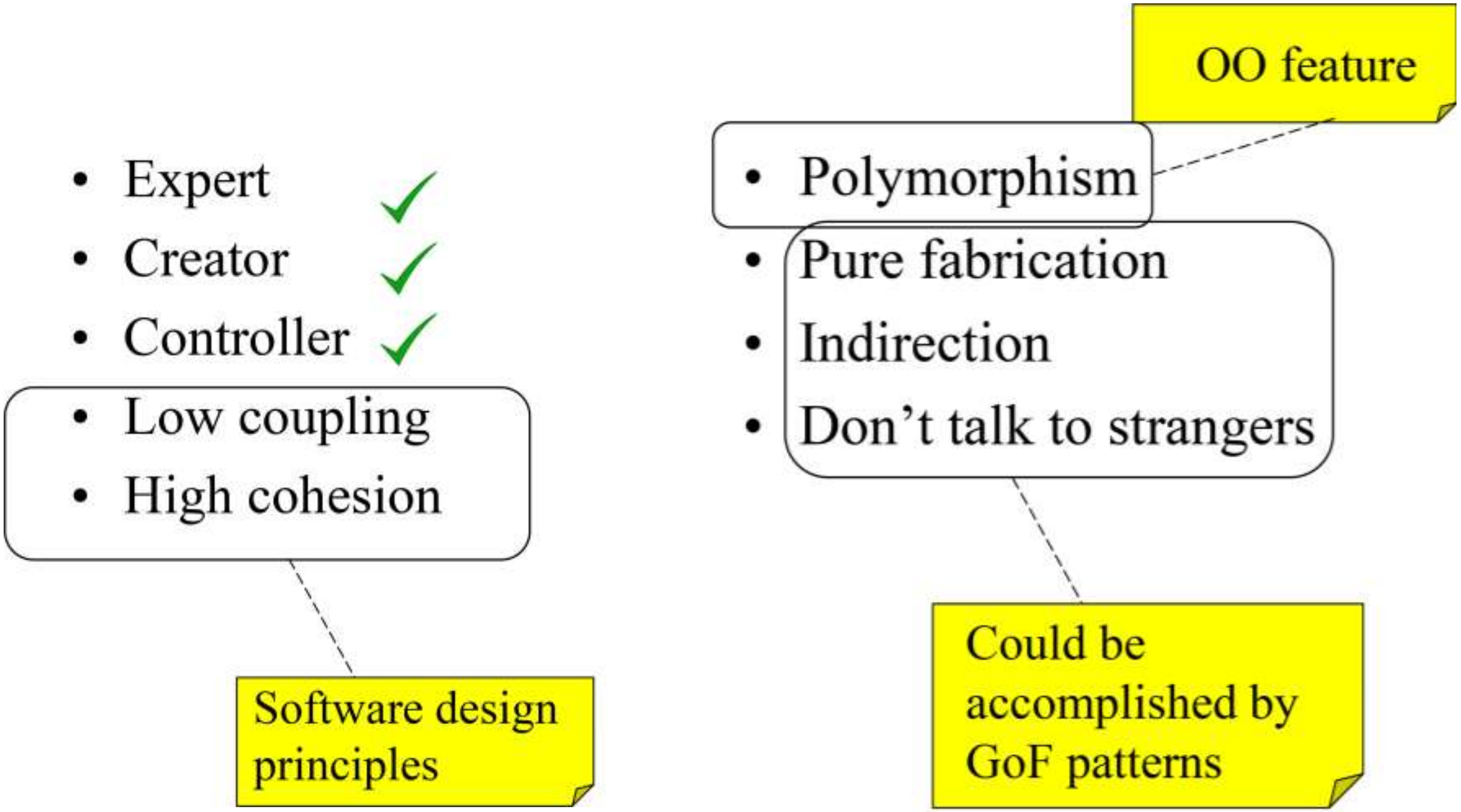
Commonly Used Design Patterns

- The General Responsibility Assignment Software Patterns (GRASP)
- The Gang of Four (GoF) Patterns due to the fact that the classical design patterns book has four authors.

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Access the text alternative for these images

GRASP Patterns



Access the text alternative for these images

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, and/or
- communication between objects

The GoF Patterns

Creational Patterns

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

Structural Patterns

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

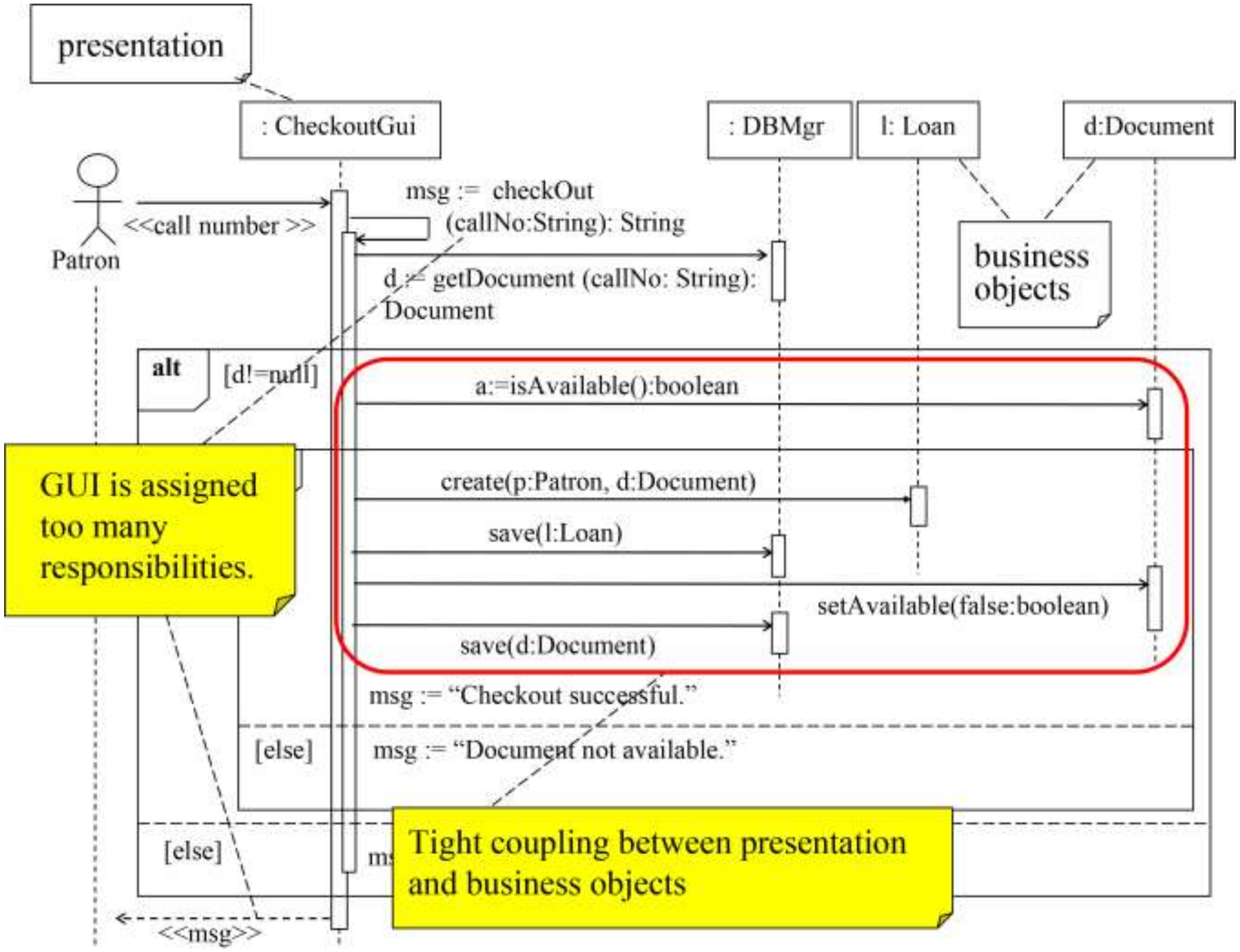
Behavioral Patterns

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

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


presentation
: CheckoutGui
: DBMgr
l: Loan
d:Document
business objects
Patron
<<call number>>
msg := checkOut (callNo:String): String
d := getDocument (callNo: String): Document
alt
[d!=null]
a:=isAvailable():boolean
create(p:Patron, d:Document)
save(l:Loan)
setAvailable(false:boolean)
save(d:Document)
msg := "Checkout successful."
[else]
msg := "Document not available."
[else]
<<msg>>
GUI is assigned too many responsibilities.
Tight coupling between presentation and business objects

Access the text alternative for these images

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 presentation technologies is difficult and costly.

A Better Solution

Reduce or eliminate the coupling between presentation and business objects.

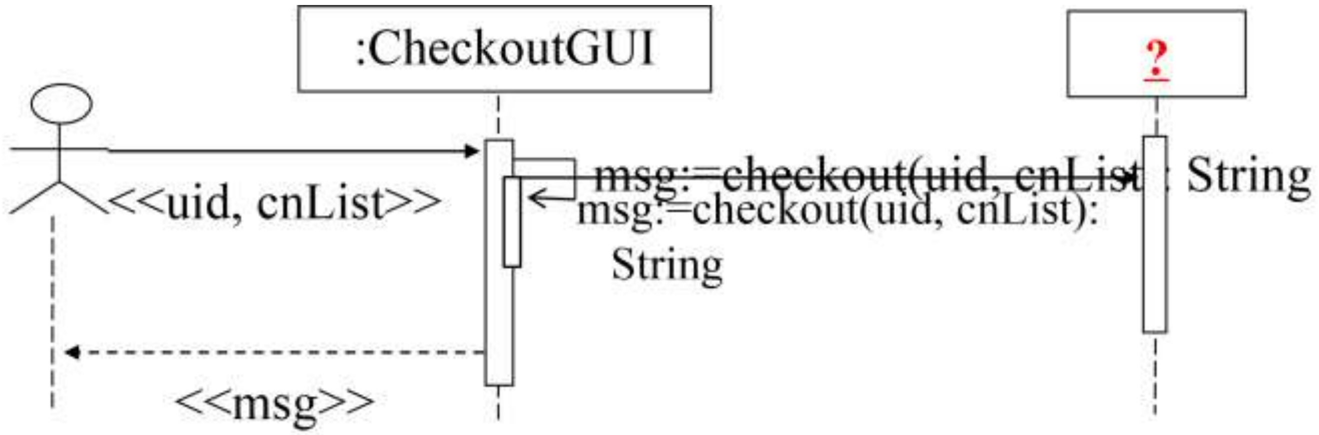
- Low coupling design principle

Remove irrelevant responsibilities from the presentation.

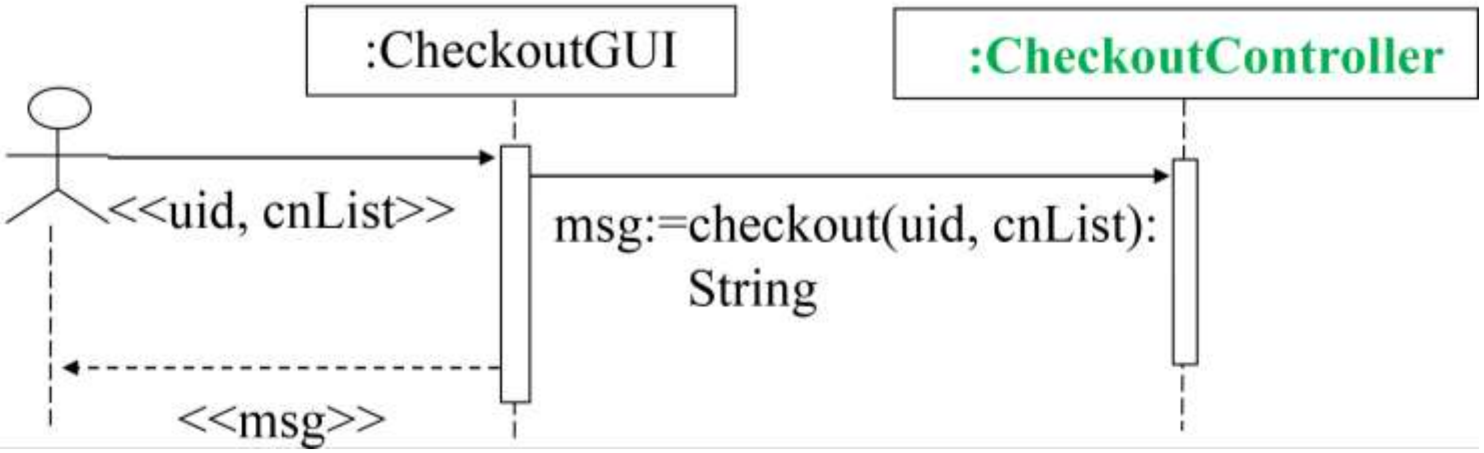
- Separation of concerns design principle.
- High cohesion design principle, and
- Designing “stupid objects.”

Have another object (class) to handle actor requests (system events).

Who Should Handle an Actor Request?


:CheckoutGUI
?
<<uid, cnList>>
~~msg:=checkout(uid, cnList~~ : String
msg:=checkout(uid, cnList):
String
<<msg>>

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



Access the text alternative for these images

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.

Types of Controller

Use case controller (most used)

- It handles all actor requests relating to a use case.
    - A checkout controller handles all actor requests to checkout a document.

Role controller

- It represents a role played by an actor (Librarian, Bank Teller).

Facade controller

- It represents the overall system (Library System, Banking System).
- It represents the organization (Library, Bank).

Applying the Controller Pattern

Descending Preference:

- use case controller
- role controller
- system controller
- organization 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 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.

Benefits of The Controller Pattern

- Separation of concerns
- High cohesion
- Low coupling
- Supporting multiple presentation technologies
- Easy to change and enhance
- Improving software reusability
- It can keep track of use case state, and ensure that the right sequence of events is being handled.

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 AUM methodology is followed.

If not designed properly, it may result in bloated controllers.

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

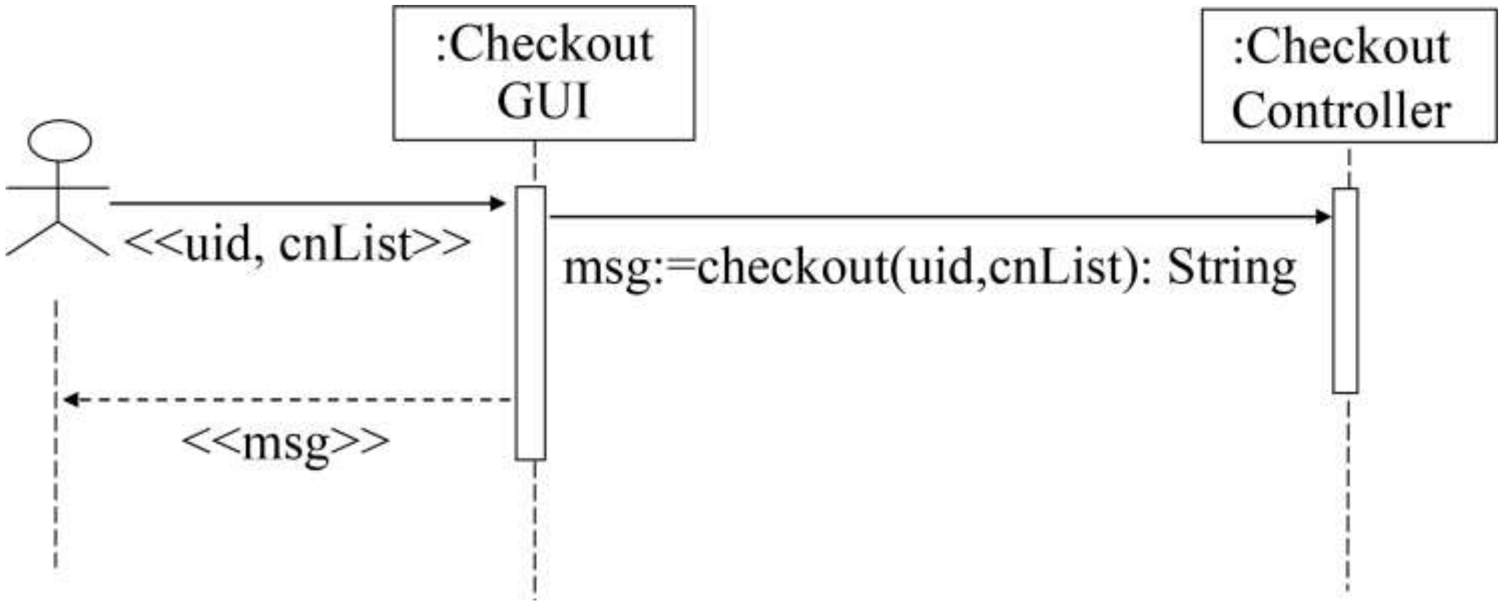
- only one controller to process many system events
- the controller does all things rather than delegating them to business objects
- the controller has many attributes to maintain system or domain information

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

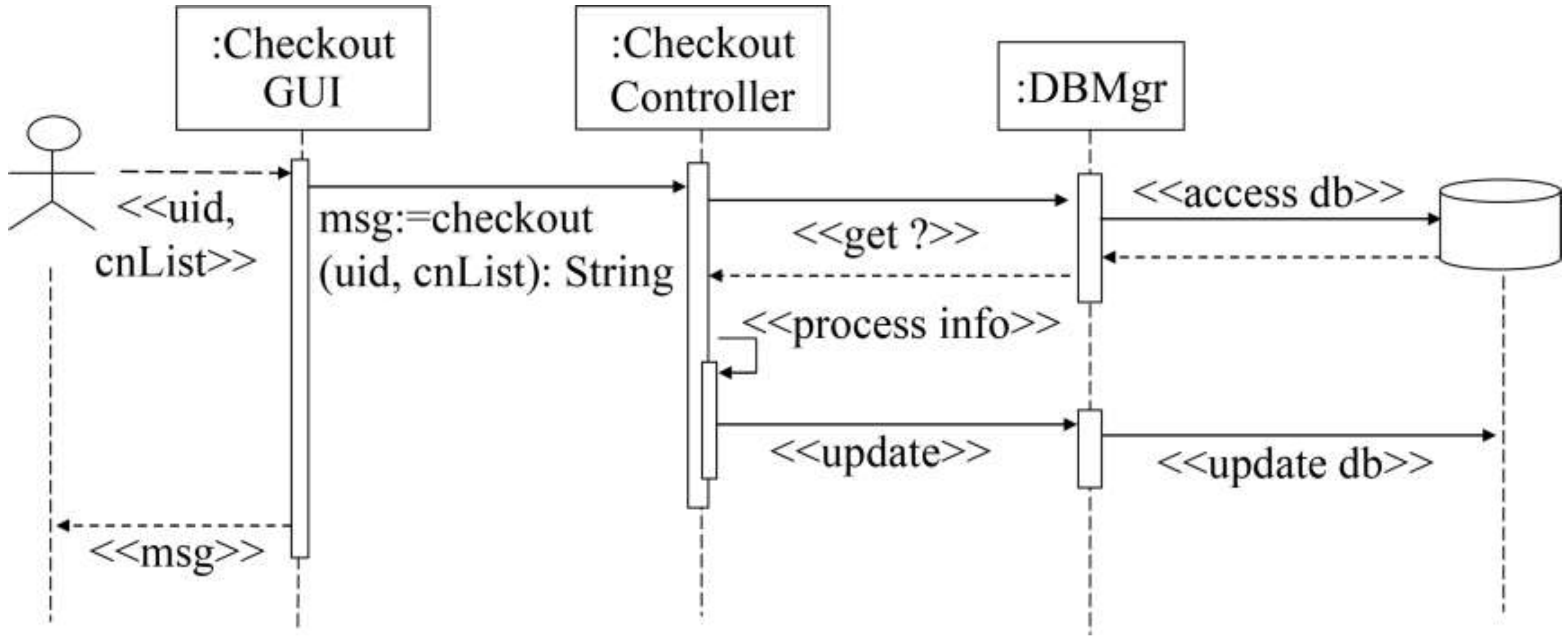
Class Exercise 1

Complete the sequence diagram for the “Checkout Document” use case.



Access the text alternative for these images

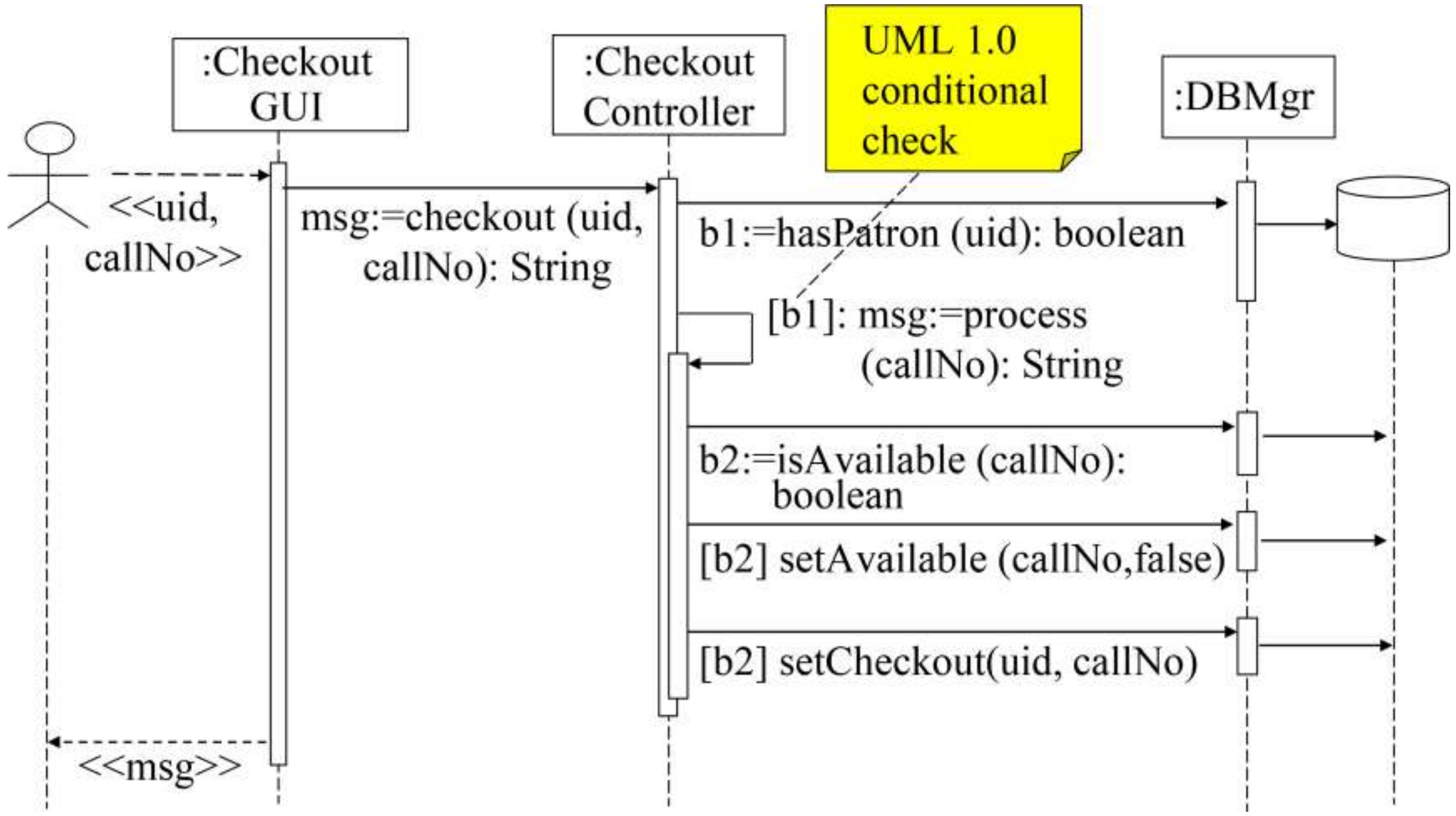
What Should the Checkout Controller Do?



- What should the controller get from the DBMgr?
- How should the controller process the result?
- How should the controller update the database?

Access the text alternative for these images

Conventional Design


:Checkout GUI
:Checkout Controller
UML 1.0 conditional check
:DBMgr
<<uid, callNo>>
msg:=checkout (uid, callNo): String
b1:=hasPatron (uid): boolean
[b1]: msg:=process (callNo): String
b2:=isAvailable (callNo): boolean
[b2] setAvailable (callNo,false)
[b2] setCheckout(uid, callNo)
<<msg>>

Access the text alternative for these images

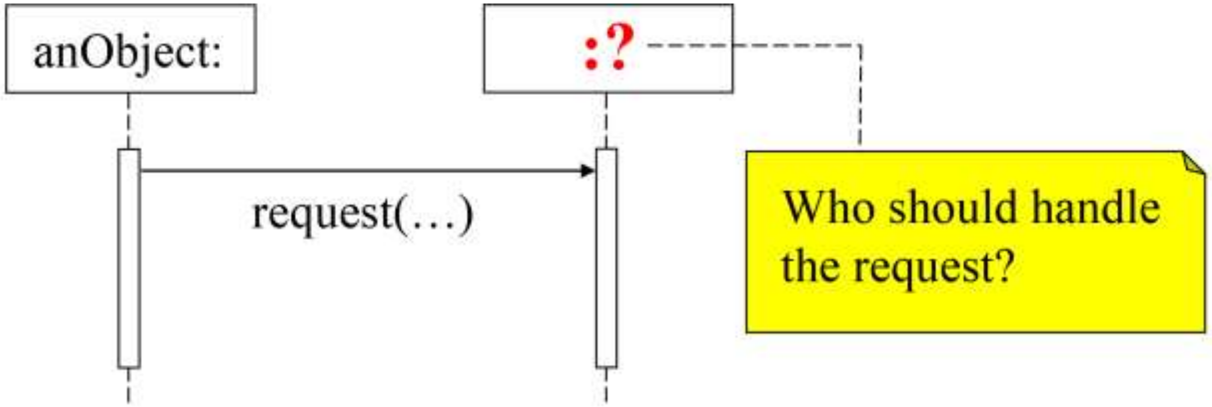
Problems with the Conventional Design

- The database manager has to know a lot of database detail.
- The database manager is not “stupid.”
- Responsibilities are not correctly assigned.
- It is designed with a procedural programming mindset!
- It is not an object-oriented design!

Applying The Expert Pattern 1

Expert Pattern: Assign the request from an object to the information expert.

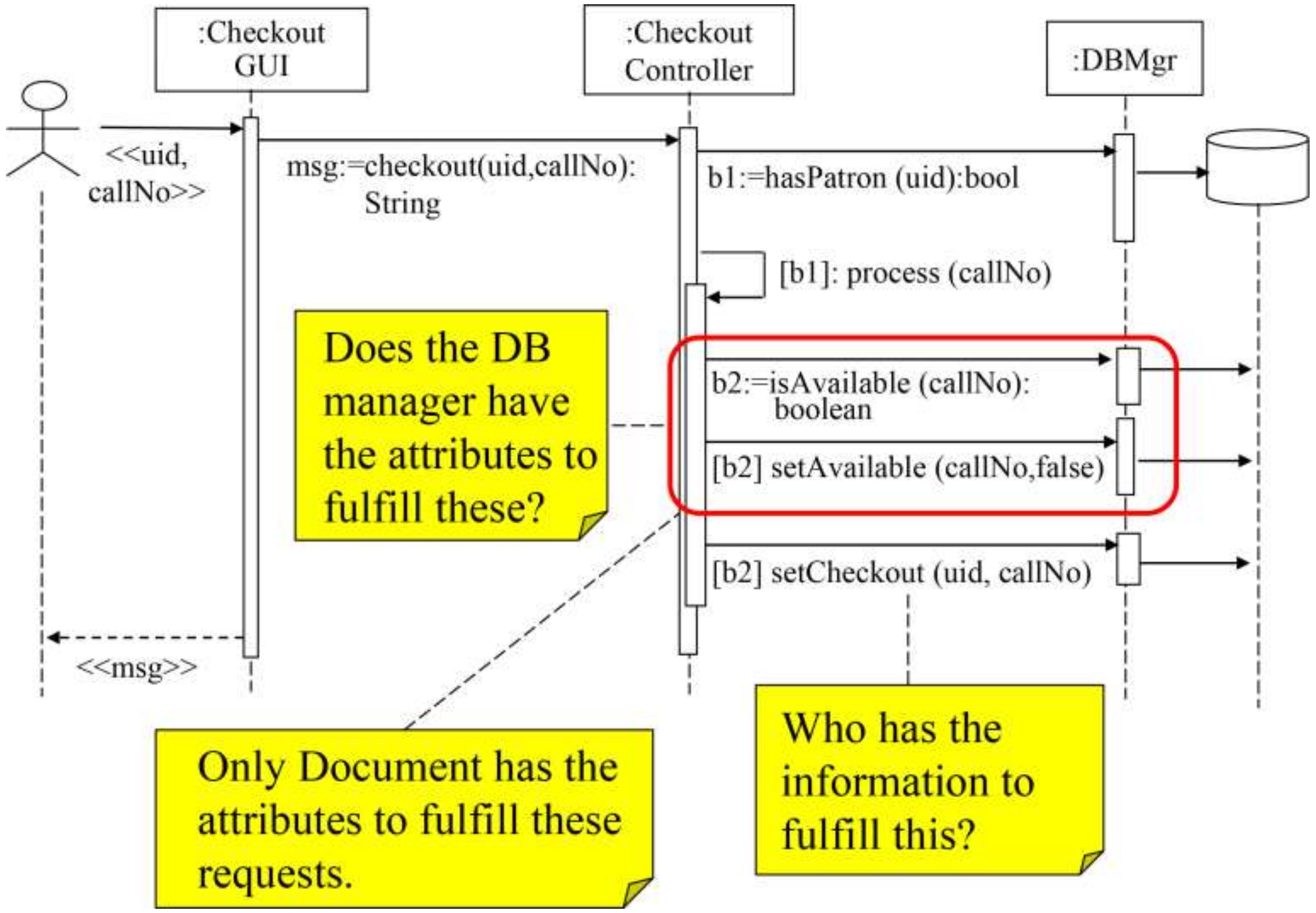
- *It is the object that stores the information needed to fulfill the request.*



Assign the responsibility to the object that has the information to fulfill the request – the object that has an attribute that stores the information.

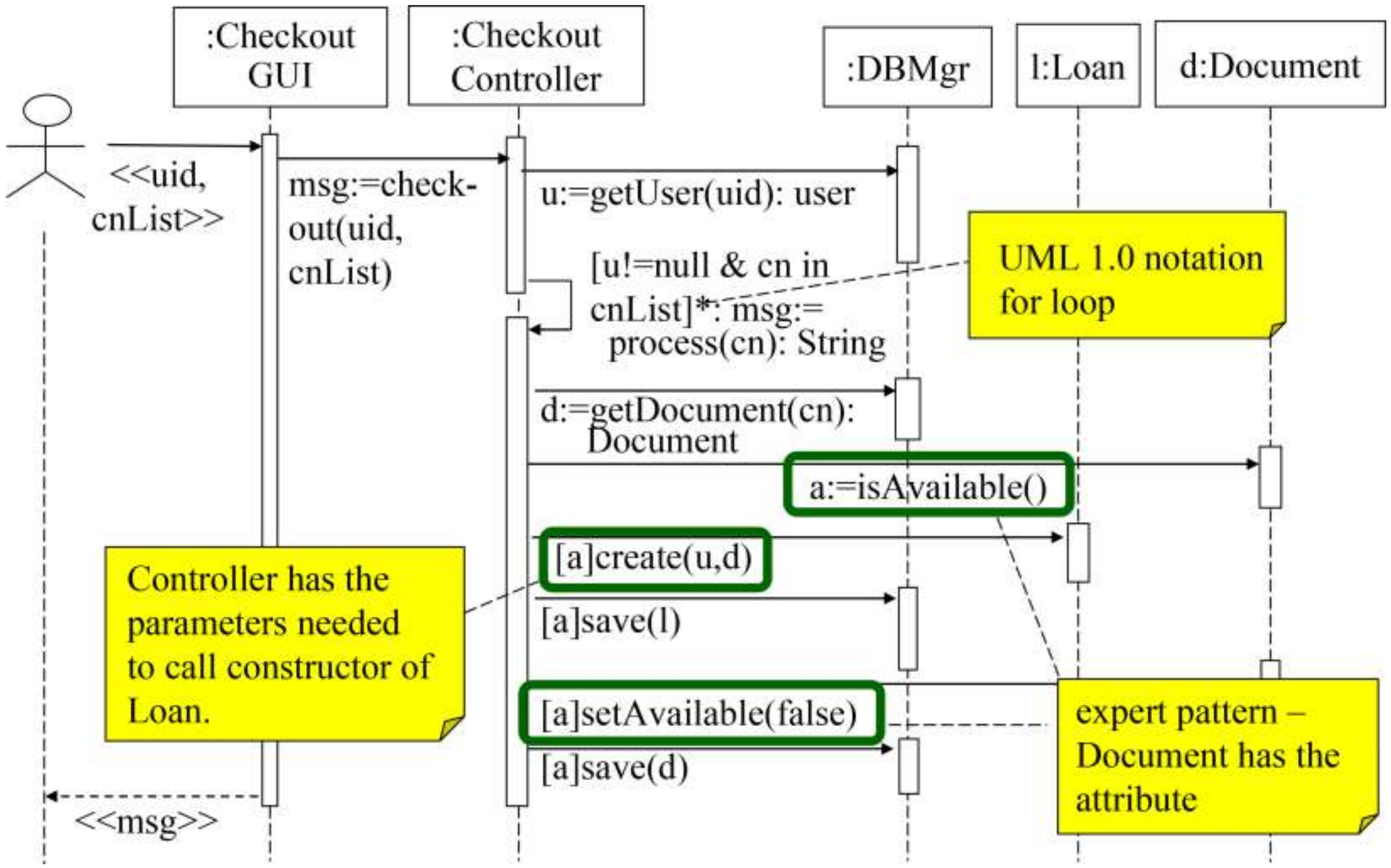
Access the text alternative for these images

Applying The Expert Pattern 2


:Checkout GUI
:Checkout Controller
:DBMgr
<<uid, callNo>>
msg:=checkout(uid,callNo): String
b1:=hasPatron (uid):bool
[b1]: process (callNo)
b2:=isAvailable (callNo): boolean
[b2] setAvailable (callNo,false)
[b2] setCheckout (uid, callNo)
<<msg>>
Does the DB manager have the attributes to fulfill these?
Only Document has the attributes to fulfill these requests.
Who has the information to fulfill this?

Access the text alternative for these images

Applying The Expert Pattern 3


:Checkout GUI
:Checkout Controller
:DBMgr
l:Loan
d:Document
<<uid, cnList>>
msg:=check-out(uid, cnList)
u:=getUser(uid): user
[u!=null & cn in cnList]*: msg:= process(cn): String
UML 1.0 notation for loop
d:=getDocument(cn): Document
a:=isAvailable()
[a]create(u,d)
Controller has the parameters needed to call constructor of Loan.
[a]save(l)
[a]setAvailable(false)
expert pattern – Document has the attribute
[a]save(d)
<<msg>>

Access the text alternative for these images

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.

Benefits of The Expert Pattern

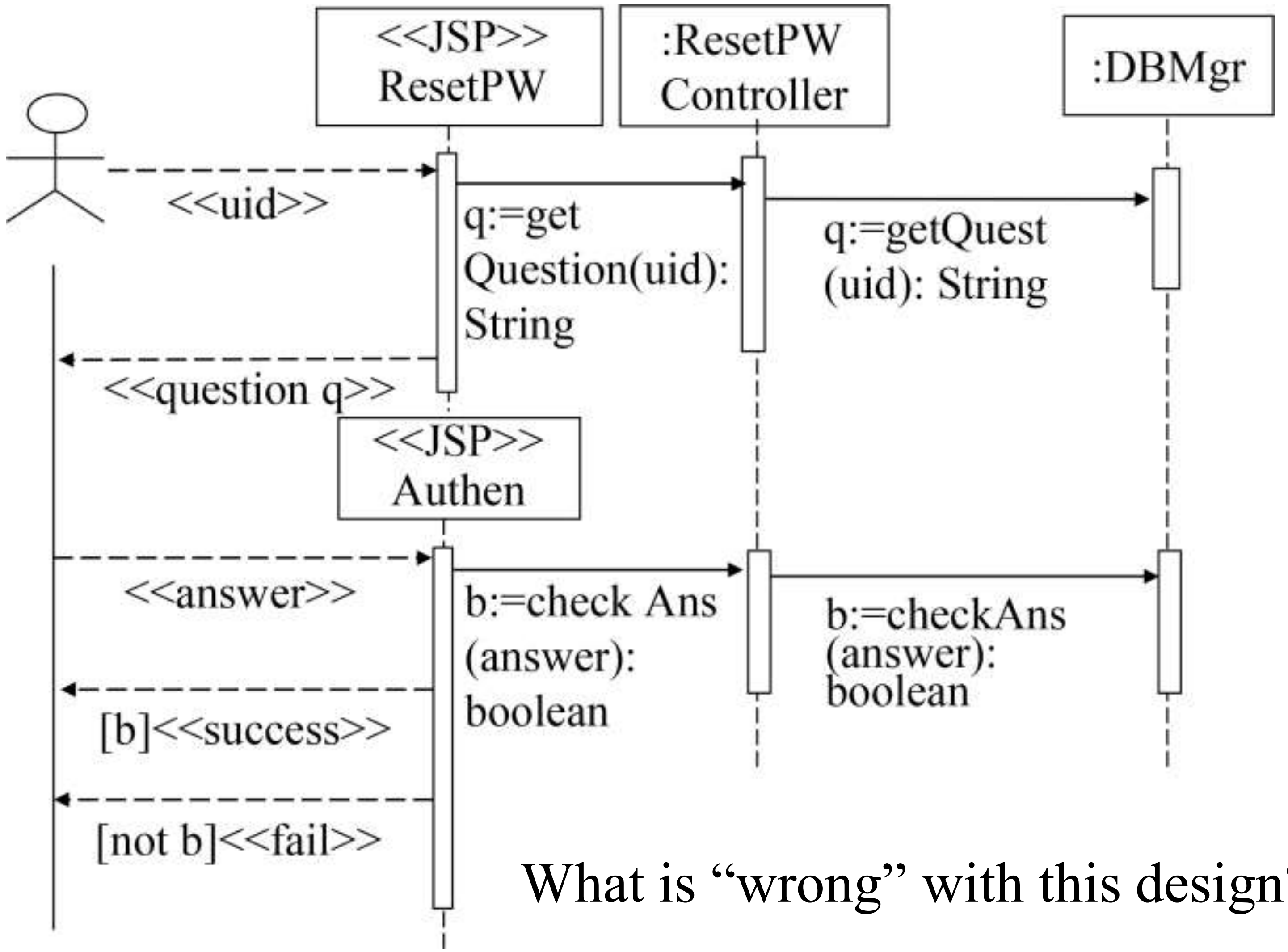
- Low coupling
- High cohesion
- Easy to comprehend and change
- Tend to result in “stupid objects”

Class Exercise 2

Draw a sequence diagram to realize a web-based “reset password” use case:

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) The user sees the confirmation message.	

A Reset Password Sequence Diagram



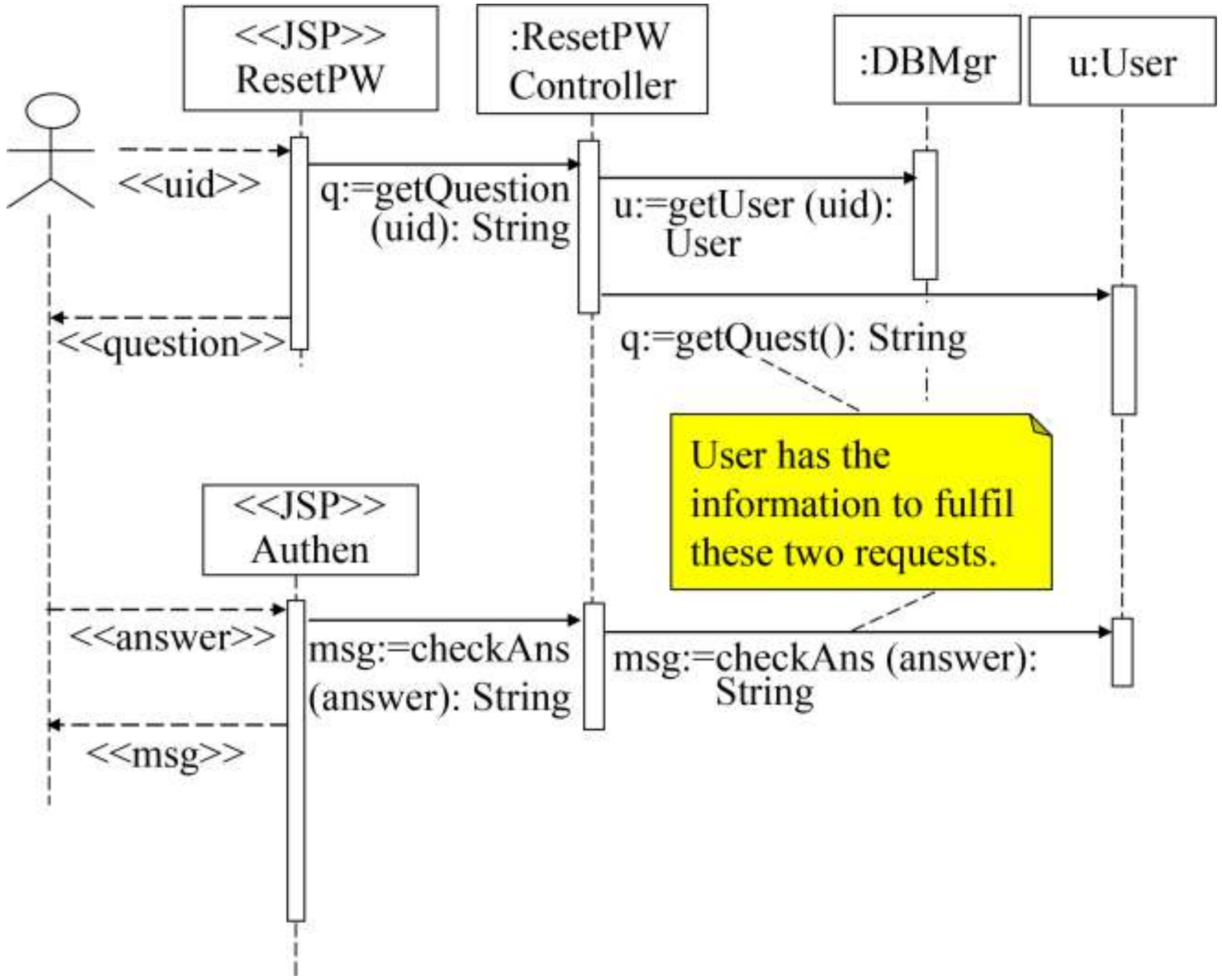
What is “wrong” with this design?

Access the text alternative for these images

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.

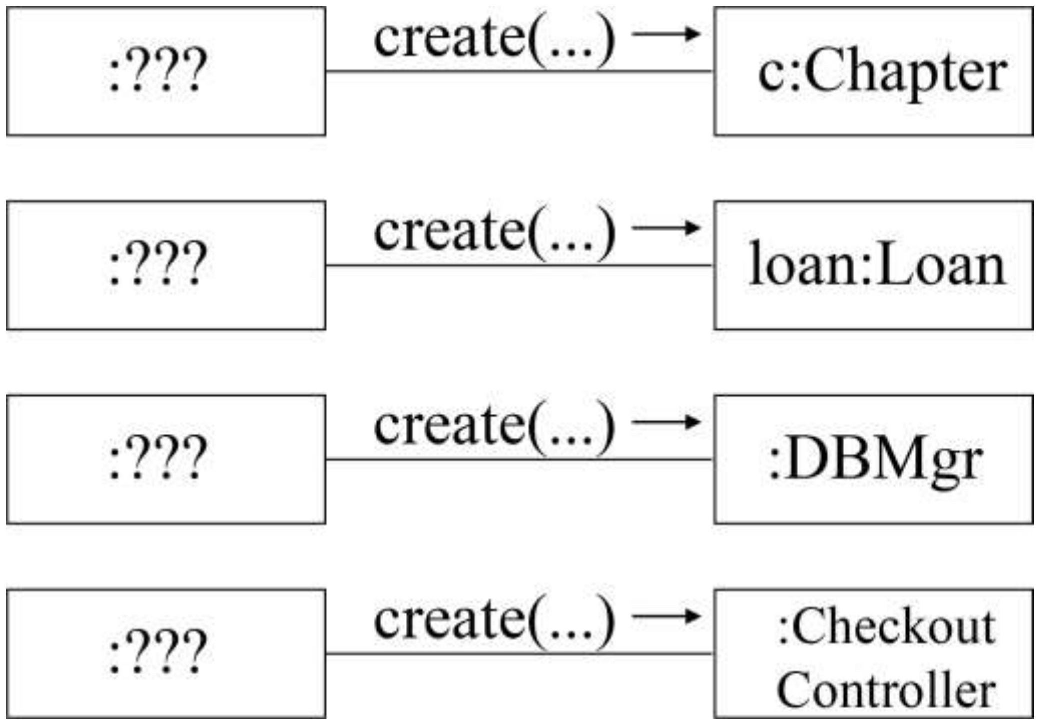
Applying The Expert Pattern 4


<<JSP>> ResetPW
:ResetPW Controller
:DBMgr
u:User
<<uid>>
q:=getQuestion (uid): String
u:=getUser (uid): User
<<question>>
q:=getQuest(): String
User has the information to fulfil these two requests.
<<JSP>> Authen
<<answer>>
msg:=checkAns (answer): String
msg:=checkAns (answer): String
<<msg>>

Access the text alternative for these images

The Creator Pattern 1

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?

Access the text alternative for these images

The Creator Pattern 2

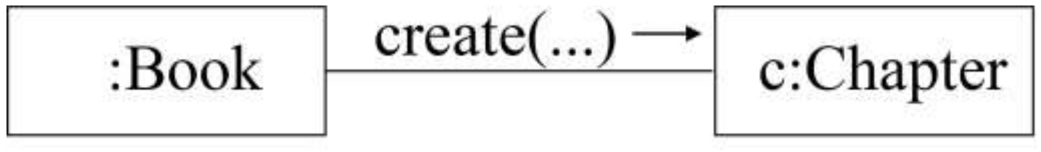
Object creation is a common activity in OO design – it is useful to have a general principle for assigning the responsibility.

Assign class B the responsibility to create an object of class A if

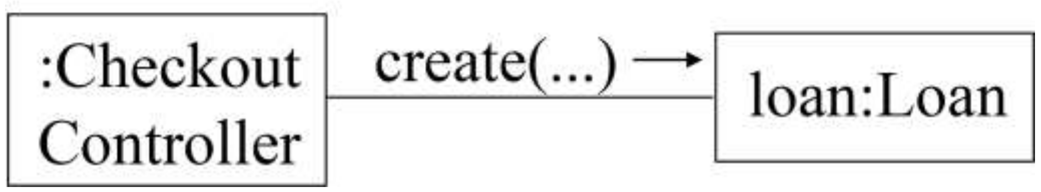
- B is an aggregate of A objects.
- B contains A objects, for example, the dispenser contains vending items.
- B records A objects, for example, the dispenser maintains a count for each vending item.
- B closely uses A objects.
- B has the information to create an A object.

The Creator Pattern 3

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.

Access the text alternative for these images

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

Summary

- Design patterns improve communication and software quality.
- GRASP patterns are “general” – they can/should be applied to design almost all systems.
- Controller deals with who should be responsible for handling an actor request?
- Expert deals with who should be responsible for handling an object request?
- Creator deals with who should be responsible for creating an object of a class?

McGraw Hill

Because learning changes everything.®

www.mheducation.com