Extracts from CH10 through CH14

- Applying Design Patterns to your design
- Deriving a class diagram
- User Interface design considerations
- State modeling in event-driven systems
- Activity modeling in transitional systems

Chapter 10: Applying Responsibility Assignment Patterns

Key Takeaway Points

- Design patterns are abstractions of proven design solutions to commonly encountered design problems.
- The controller, expert, and creator patterns are applicable to almost all objectoriented systems.

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.

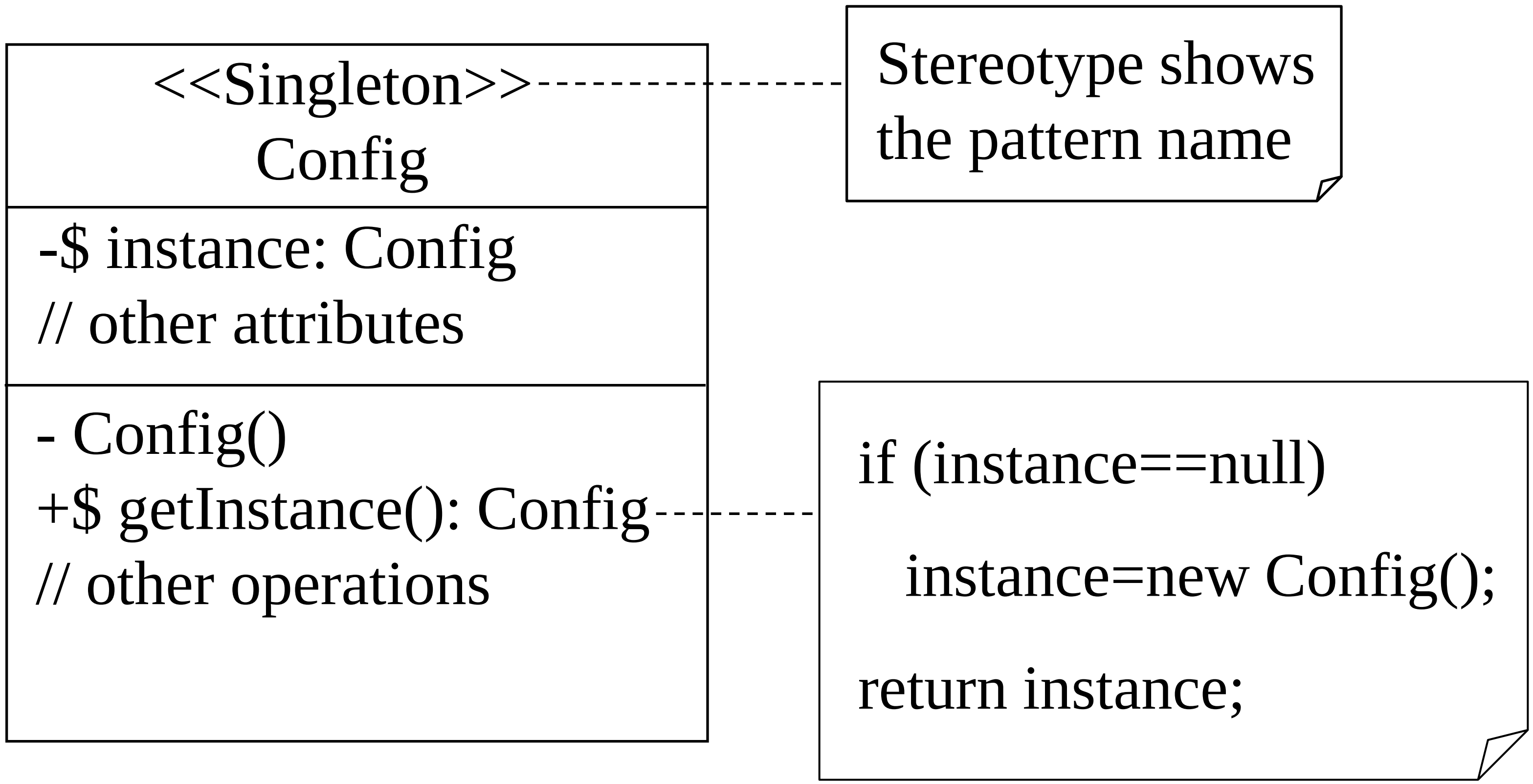
Example: The Singleton Pattern

- 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

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
}
```

Example: The Singleton Pattern



+: public   -: private $: static

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 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 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

http://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%29

(Provides comparison with GoF pattern set)

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
  - algorithmatic 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

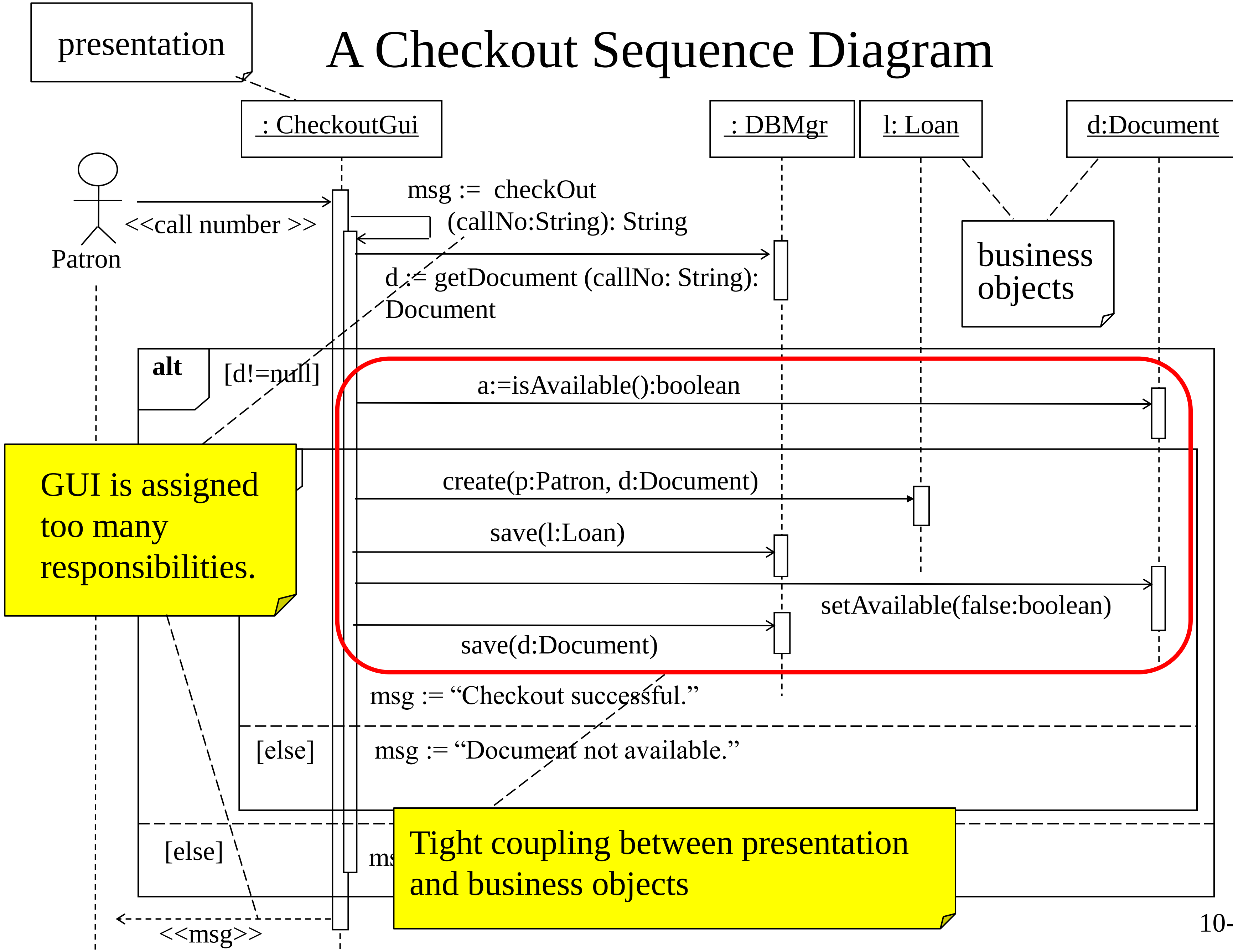
Behavioral Patterns

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

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



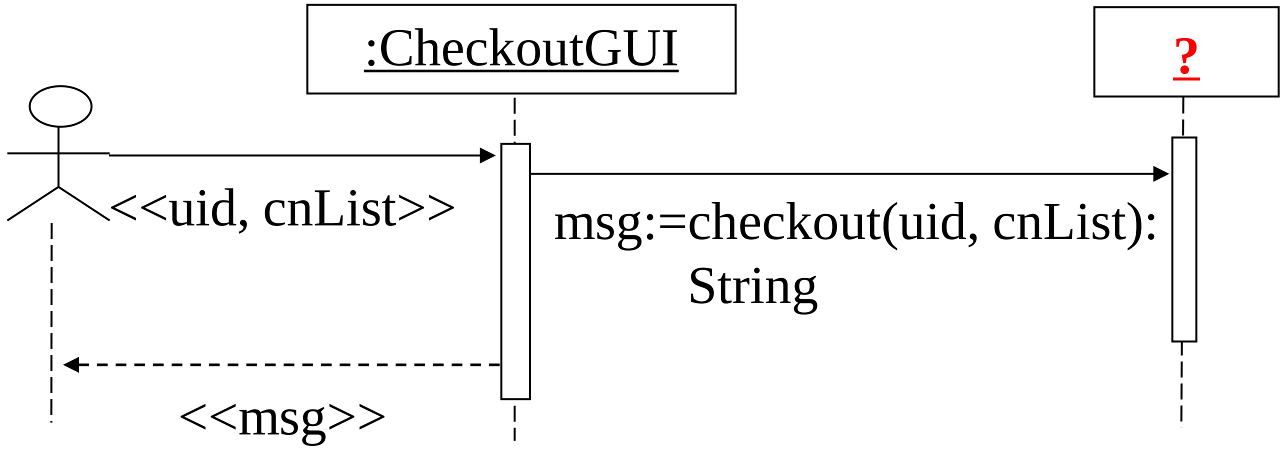
Problems with This Design

- 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.

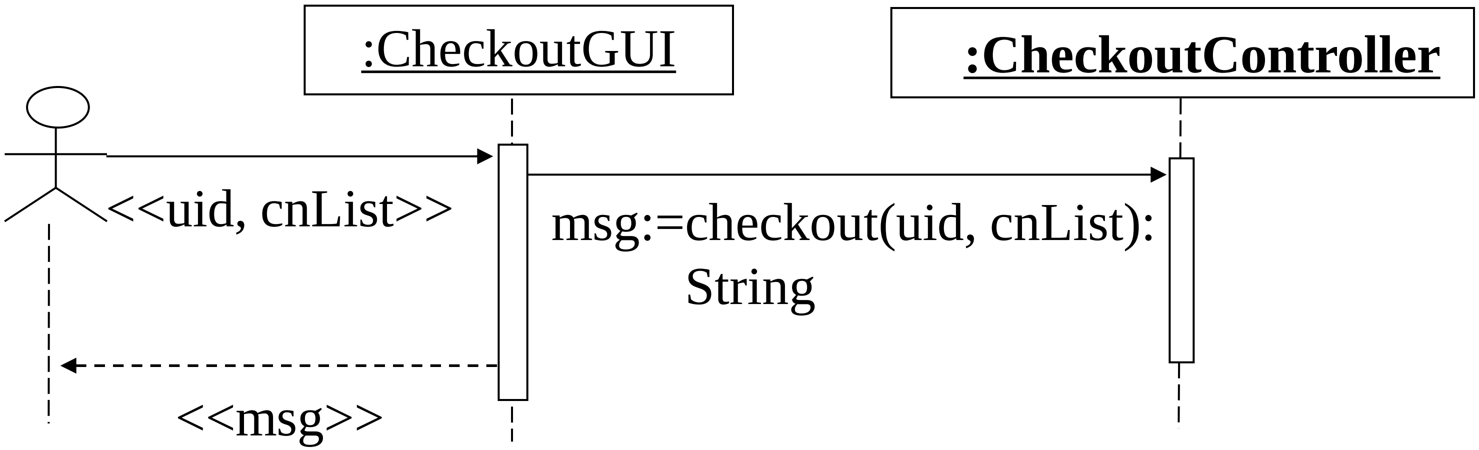
A Better Solution

- 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.

Benefits of The Controller Pattern

- Separation of concerns
- High cohesion
- Low coupling
- Supporting multiple interfaces
- Easy to change and enhance
- Improving software reusability
- It can keep track of use case state, and ensure that the correct 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 methodology is followed.
- If not designed properly, it may result in bloated controllers.

Bloated Controller

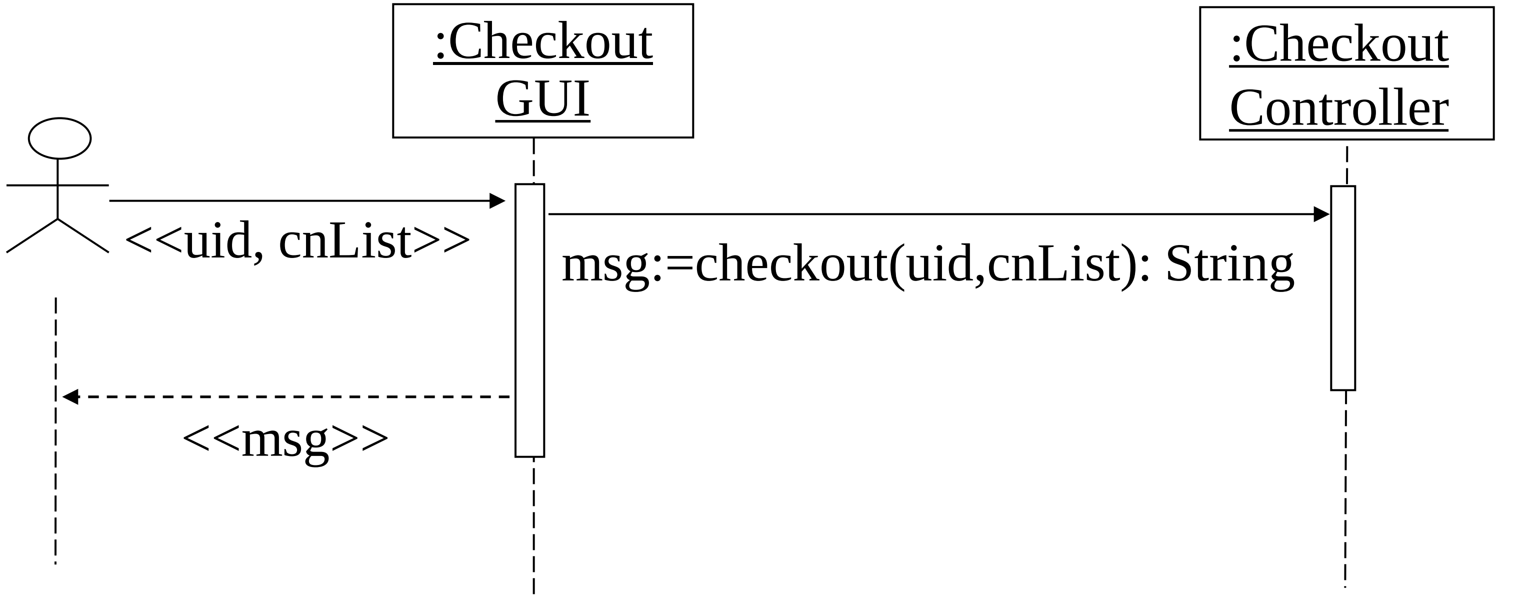
- A bloated controller is one that is assigned too many unrelated responsibilities.
- Symptoms
  - There is only one controller to handle many actor requests.
    - This is often seen with a role controller or a facade controller.
  - The controller does everything to handle the actor requests rather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

Cures to Bloated Controllers

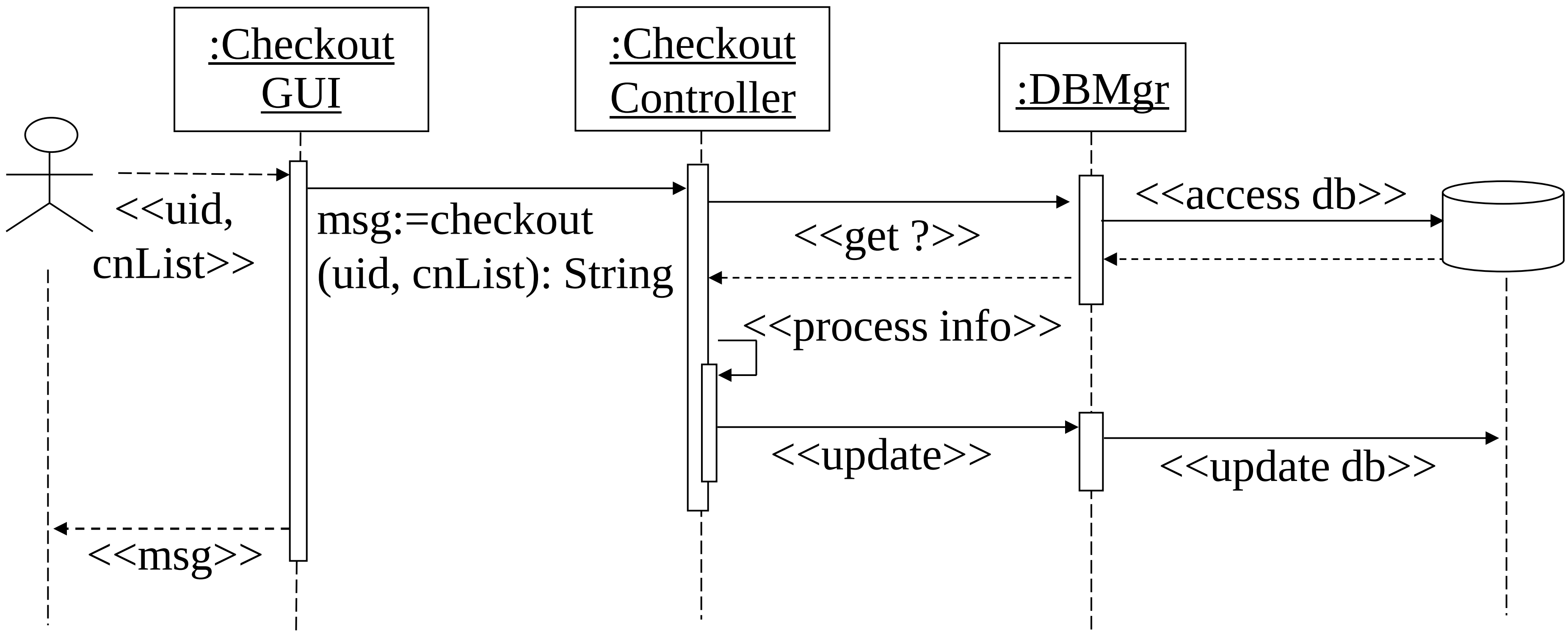
- Symptoms
  - only one controller to process many system events
  - the controller does all things rather than delegating them to business objects
  - the controller has many attributes to maintain system or domain information
- Cures
  - replace the controller with use case controllers to handle use 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

- Complete the sequence diagram for the "Checkout Document" use case.

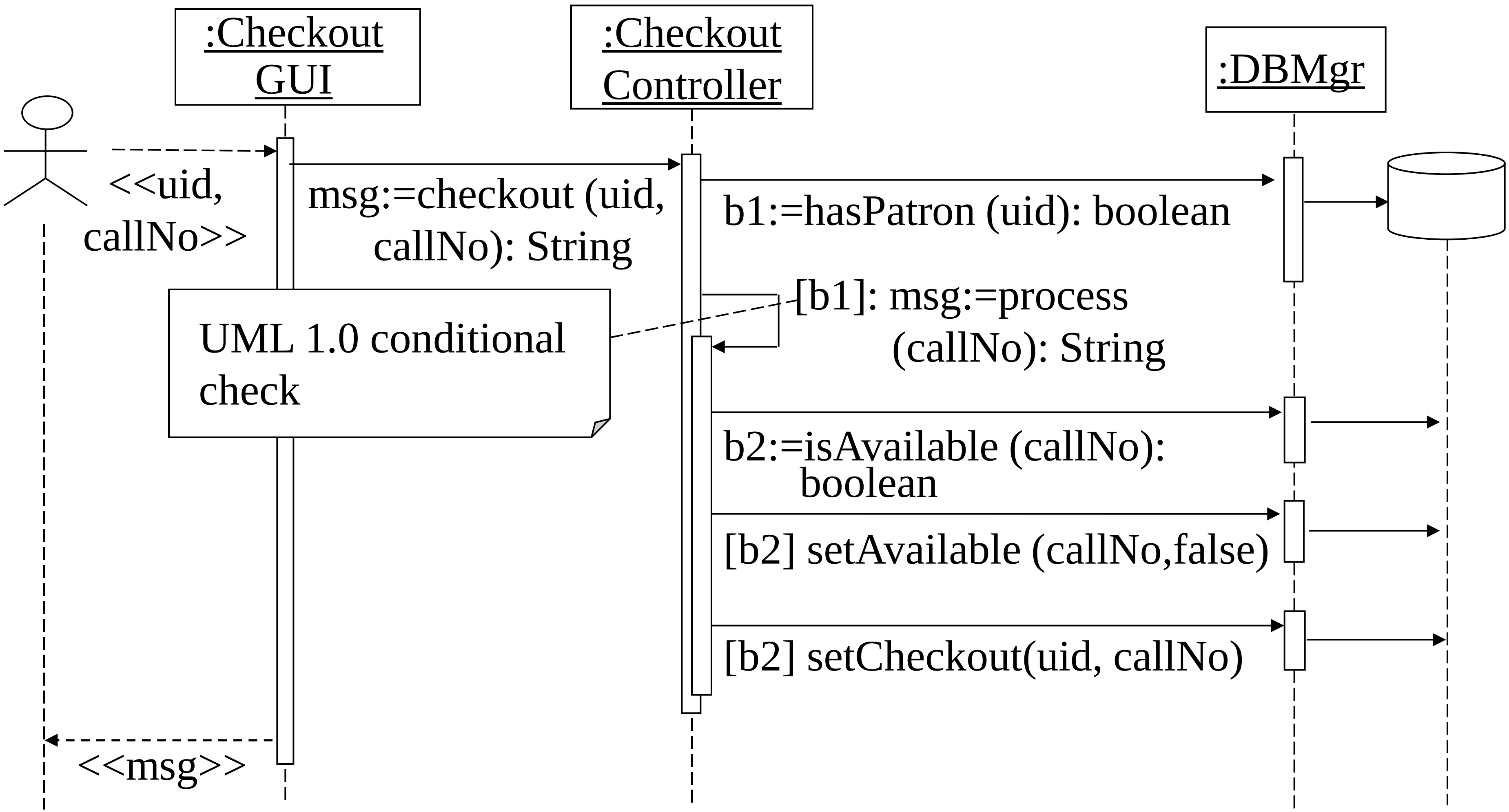


What Should the Checkout Controller Do?



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

Conventional Design



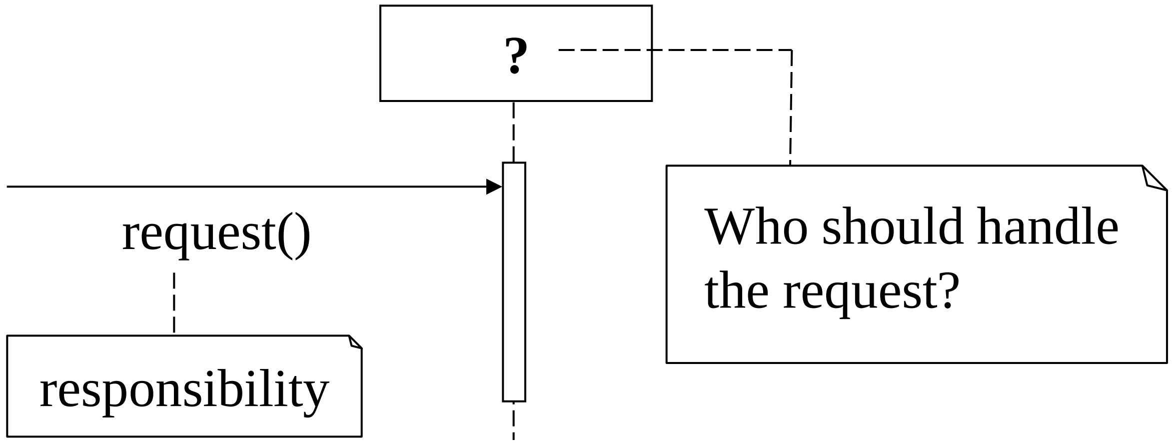
Problems with the Conventional Design

- 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!

The following slides are
for you to review on your own:

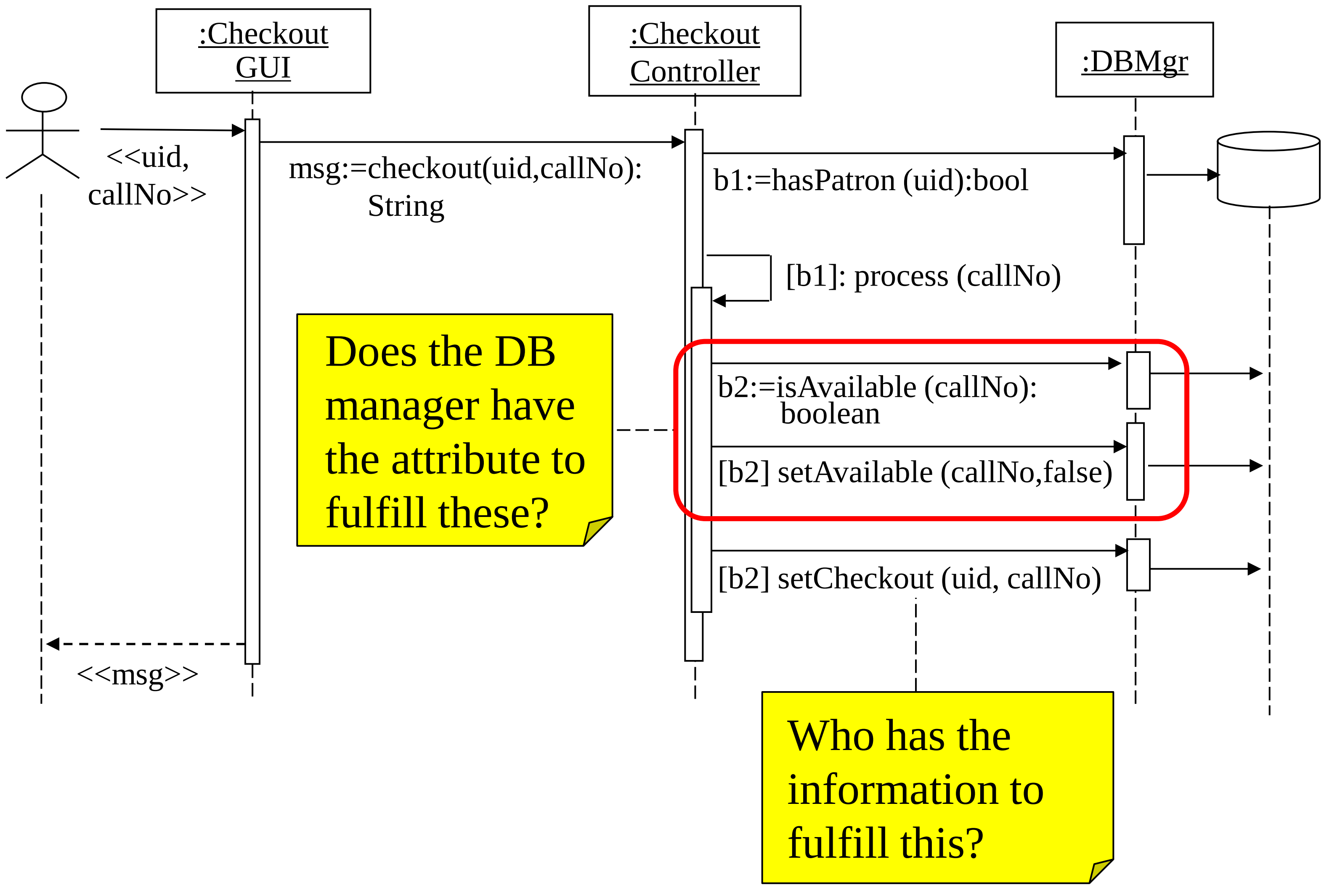
Applying The Expert Pattern

- Expert Pattern: Assign the request to the information expert.
  - *It is the object that stores the information needed to fulfill the request.*

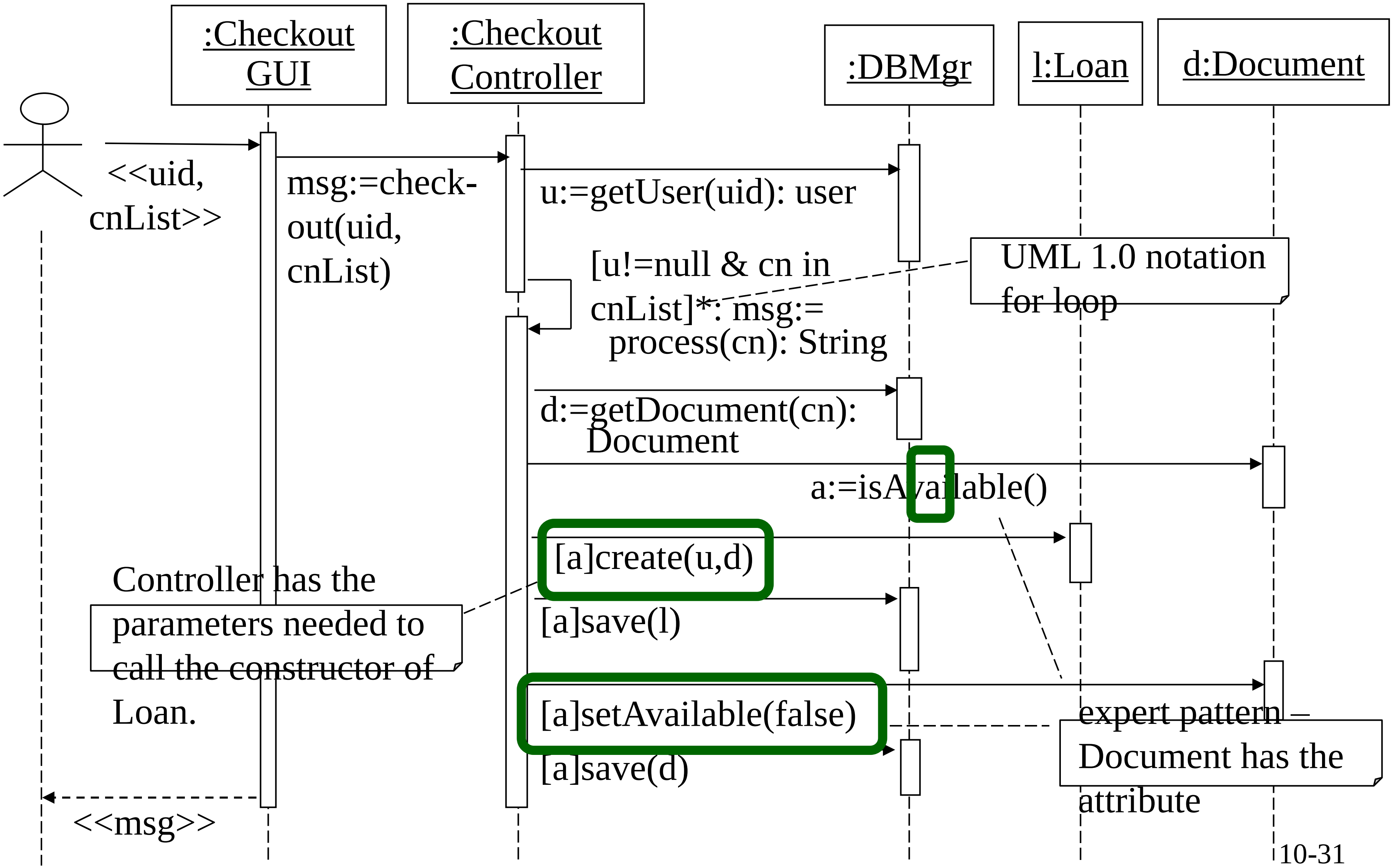


Assign the responsibility to the object that has the information to fulfill the request – the object that has an attribute that stores the information.

Applying The Expert Pattern



Applying The Expert Pattern



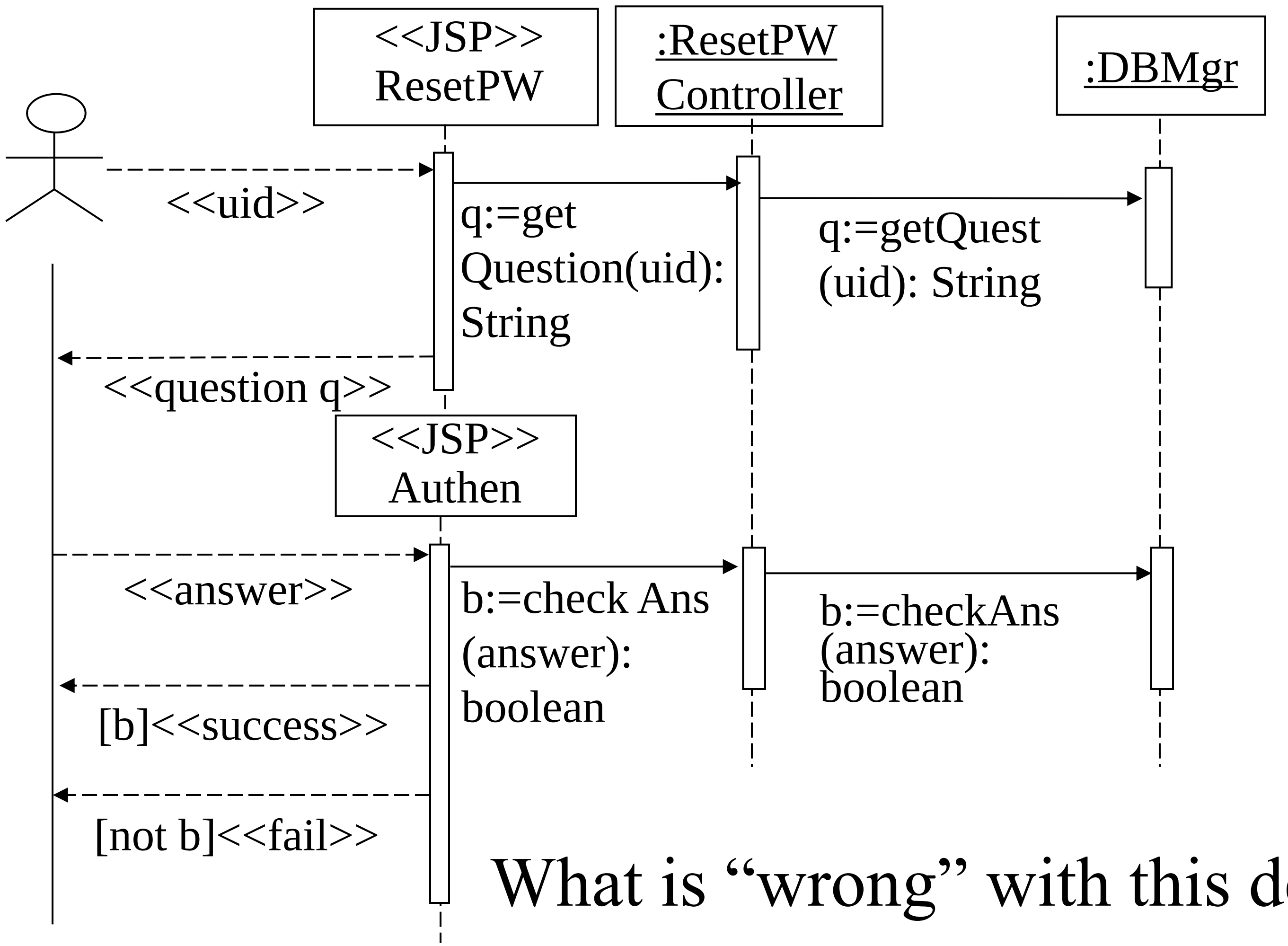
The Expert Pattern

- 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”

A Reset Password Sequence Diagram

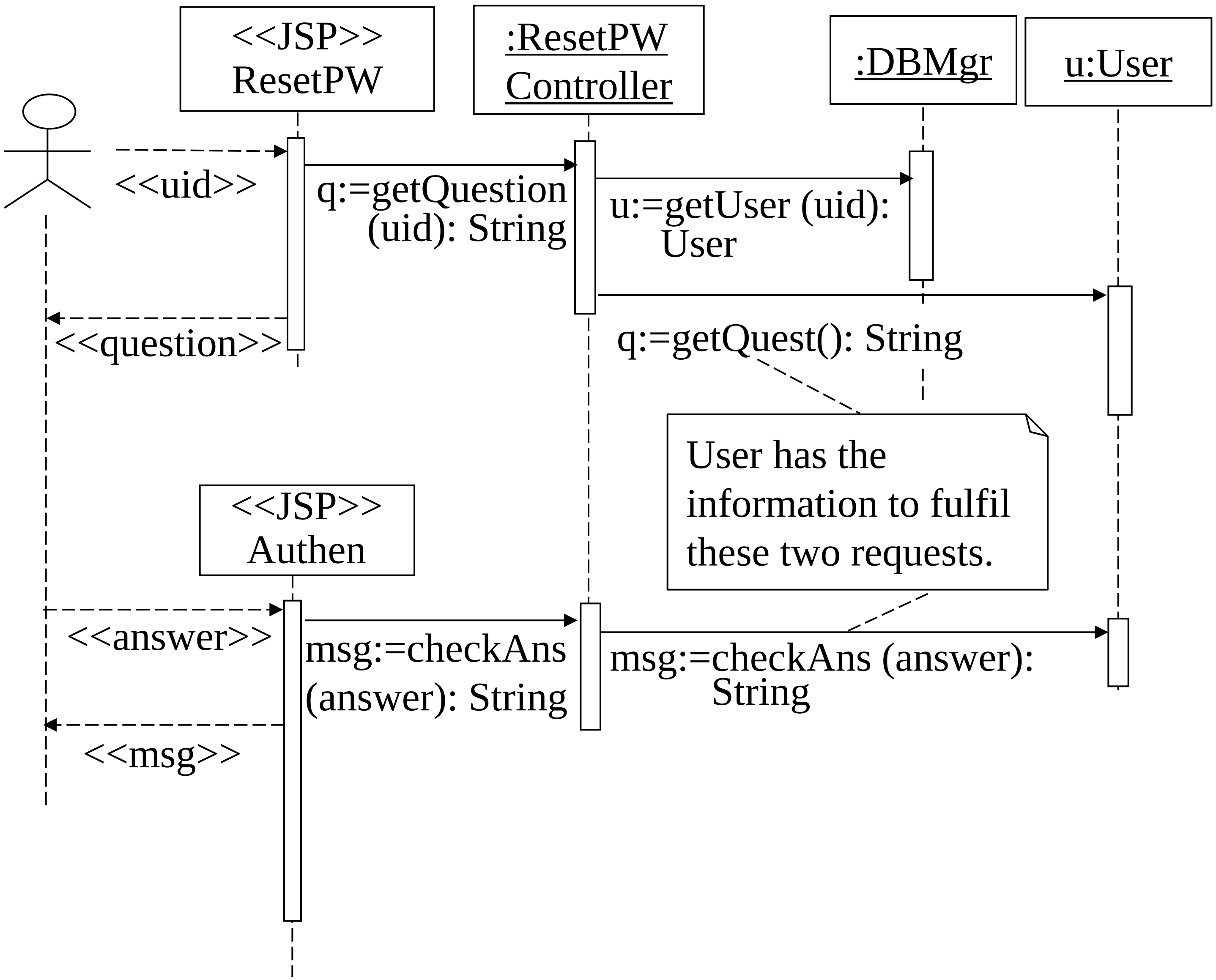


What is “wrong” with this design?

Problems with the Design

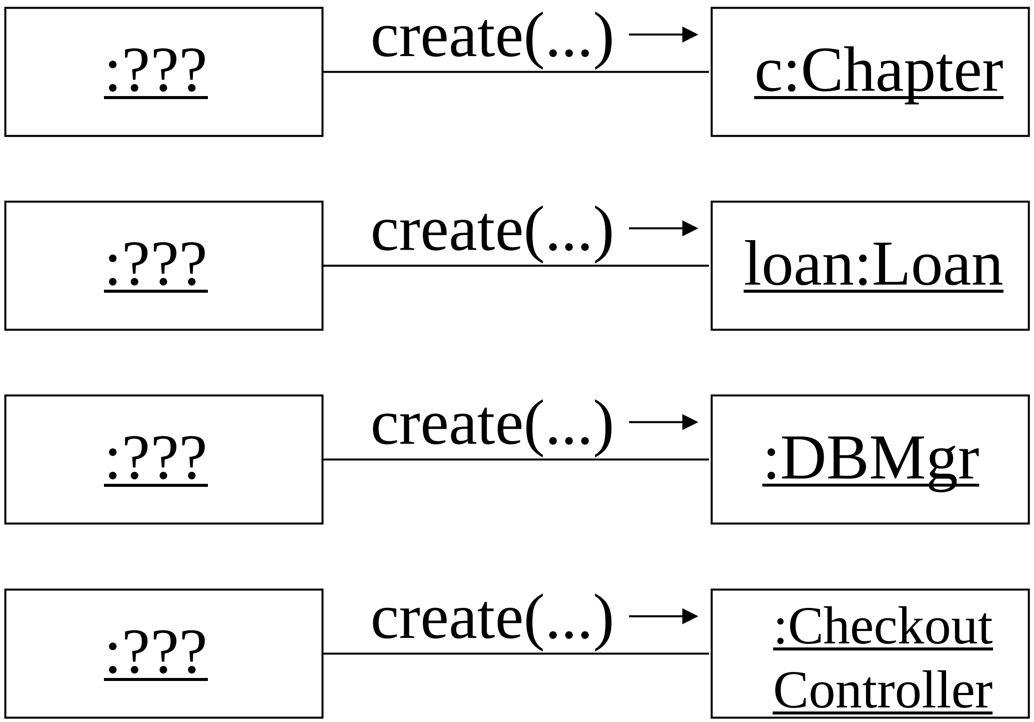
- It assigns getQuest() and checkAns() to the wrong object – DBMgr, which does not have the attributes to fulfill the requests.
- It does not design "stupid objects."
- It violates the expert pattern.
- It is designed with a conventional mindset.

Applying The Expert Pattern



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?

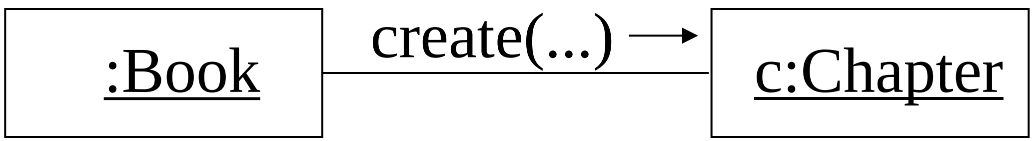
Who should create a checkout controller?

The Creator Pattern

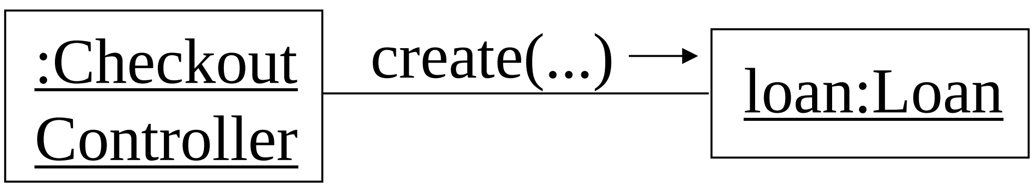
- Object creation is a common activity in OO design – it is useful to have a general principle for assigning the responsibility.
- Assign class B the responsibility to create an object of class A if
  - B is an aggregate of A objects.
  - B contains A objects, for example, the dispenser contains vending 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.

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

Chapter 11: Deriving a Design Class Diagram

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.

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.
- It should contain only classes appear in the sequence diagrams, and a few classes from the domain model.

Deriving Design Class Diagram

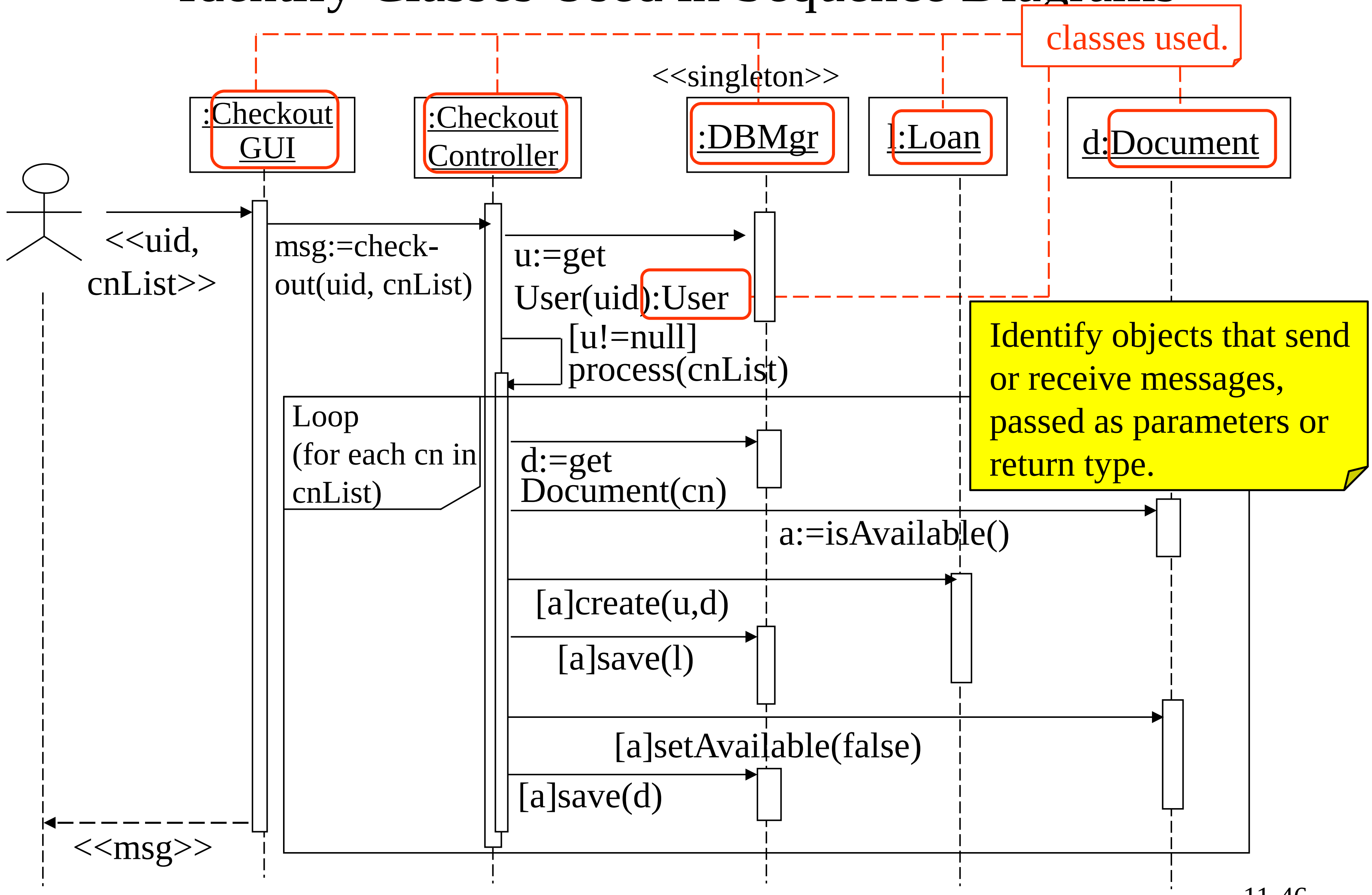
- It is derived from the domain model (DM) and the sequence diagrams:
  - The domain model provides a few classes, the attributes and some relationships.
  - The sequence diagrams determines the classes, methods, some attributes, and dependence relationships.
- DCD may contain design classes like controller, command, GUI classes. Domain model only contains application classes.
- DCD must be carefully specified. DM is more liberal.

Steps for Deriving DCD

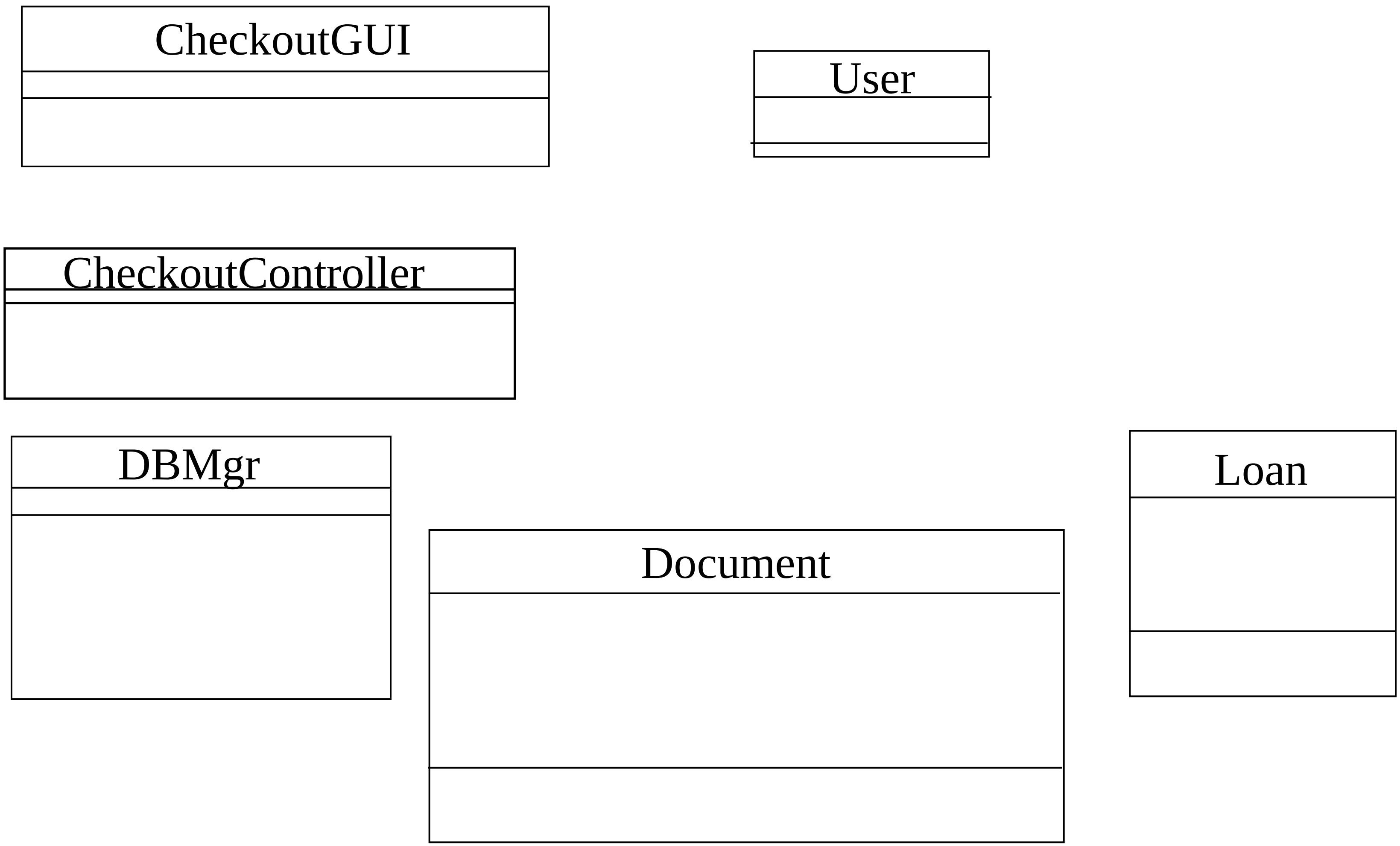
1) Identify all classes used in each of the sequence diagrams and put them down in the DCD:

   – classes of objects that send or receive messages
   – classes of objects that are passed as parameters or return types/values

Identify Classes Used in Sequence Diagrams



Classes Identified

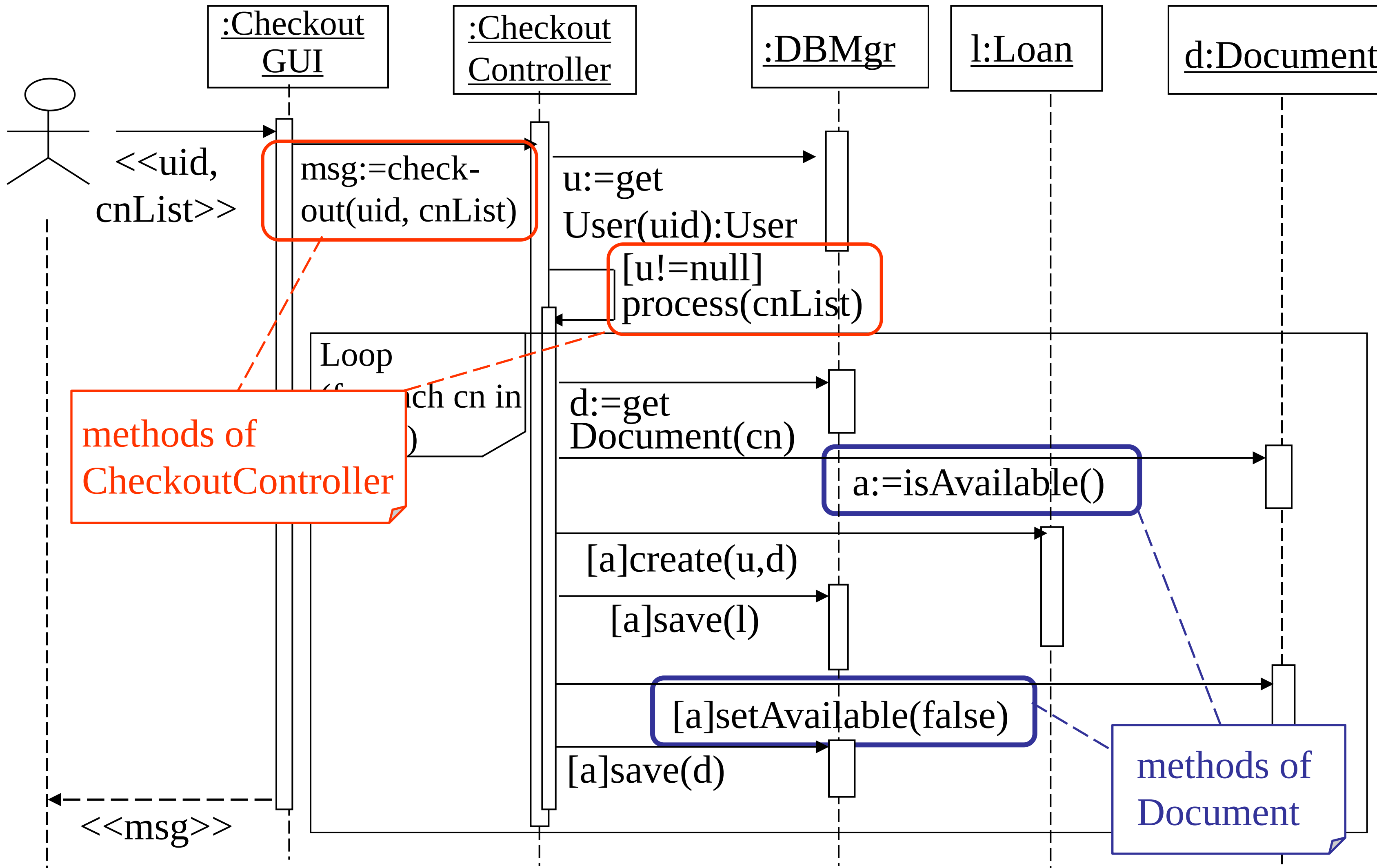


Steps for Deriving DCD

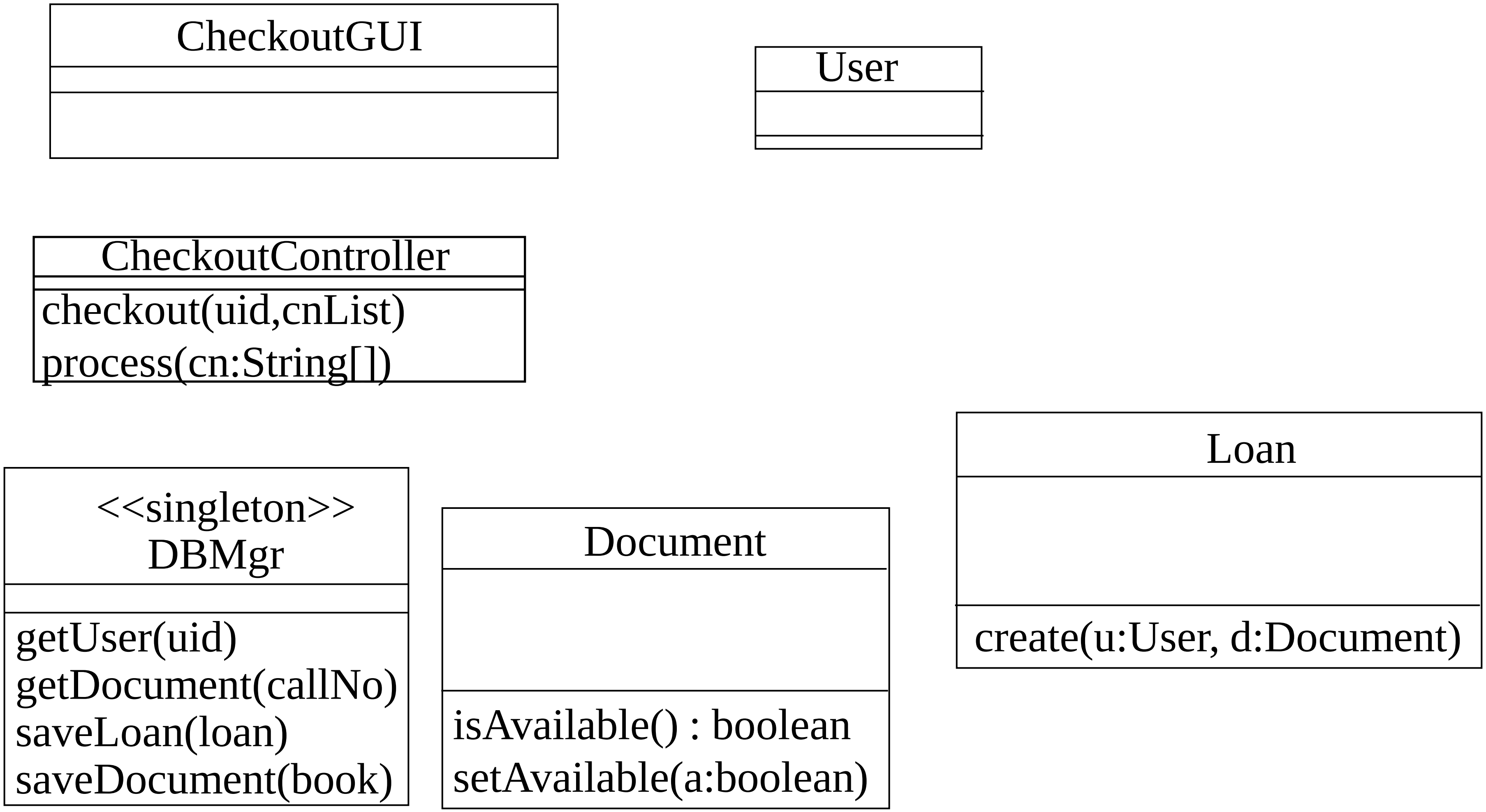
2) Identify methods belonging to each class and fill them in the DCD:

- 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.

Identify Methods



Fill In Identified Methods

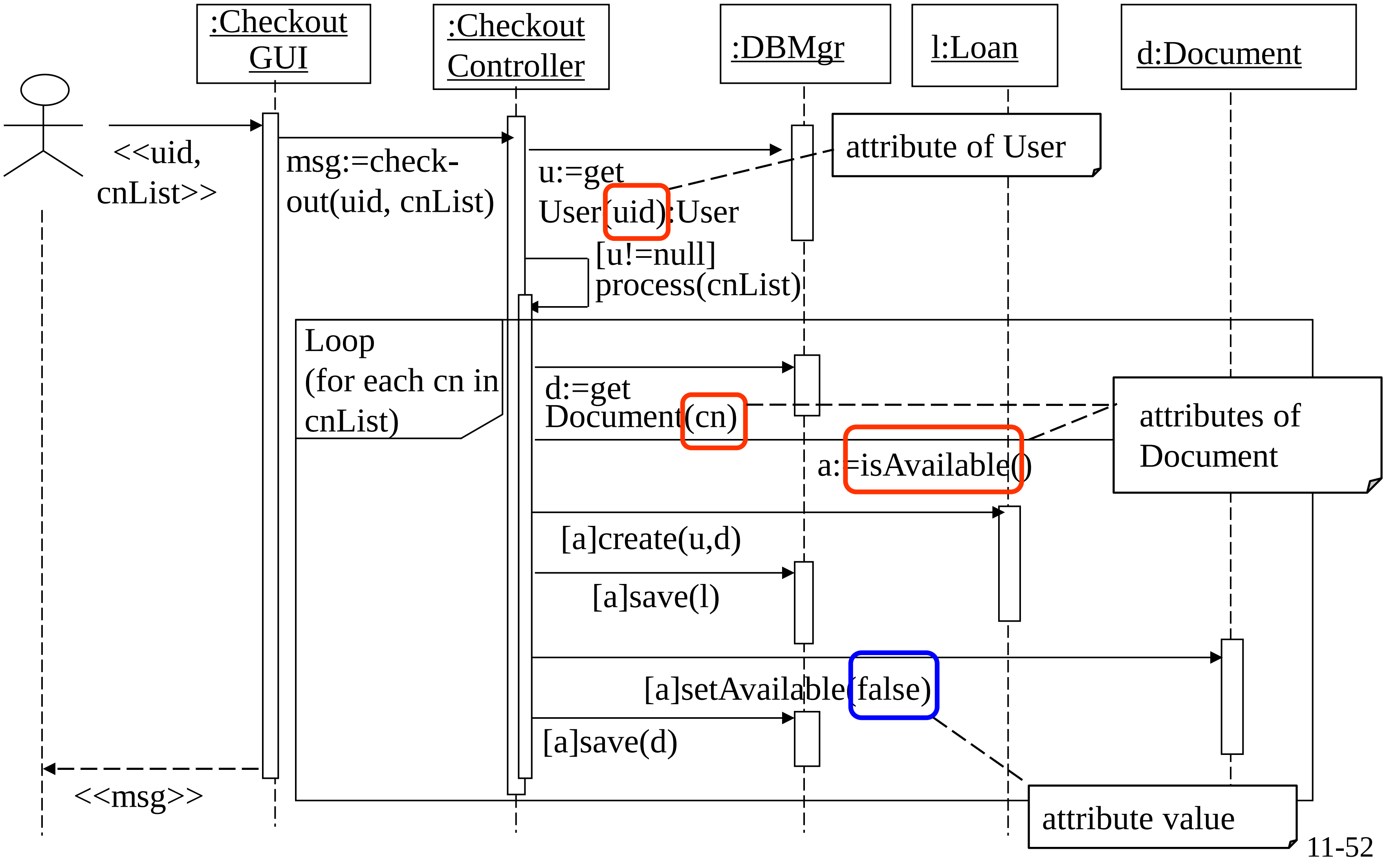


Steps for Deriving DCD

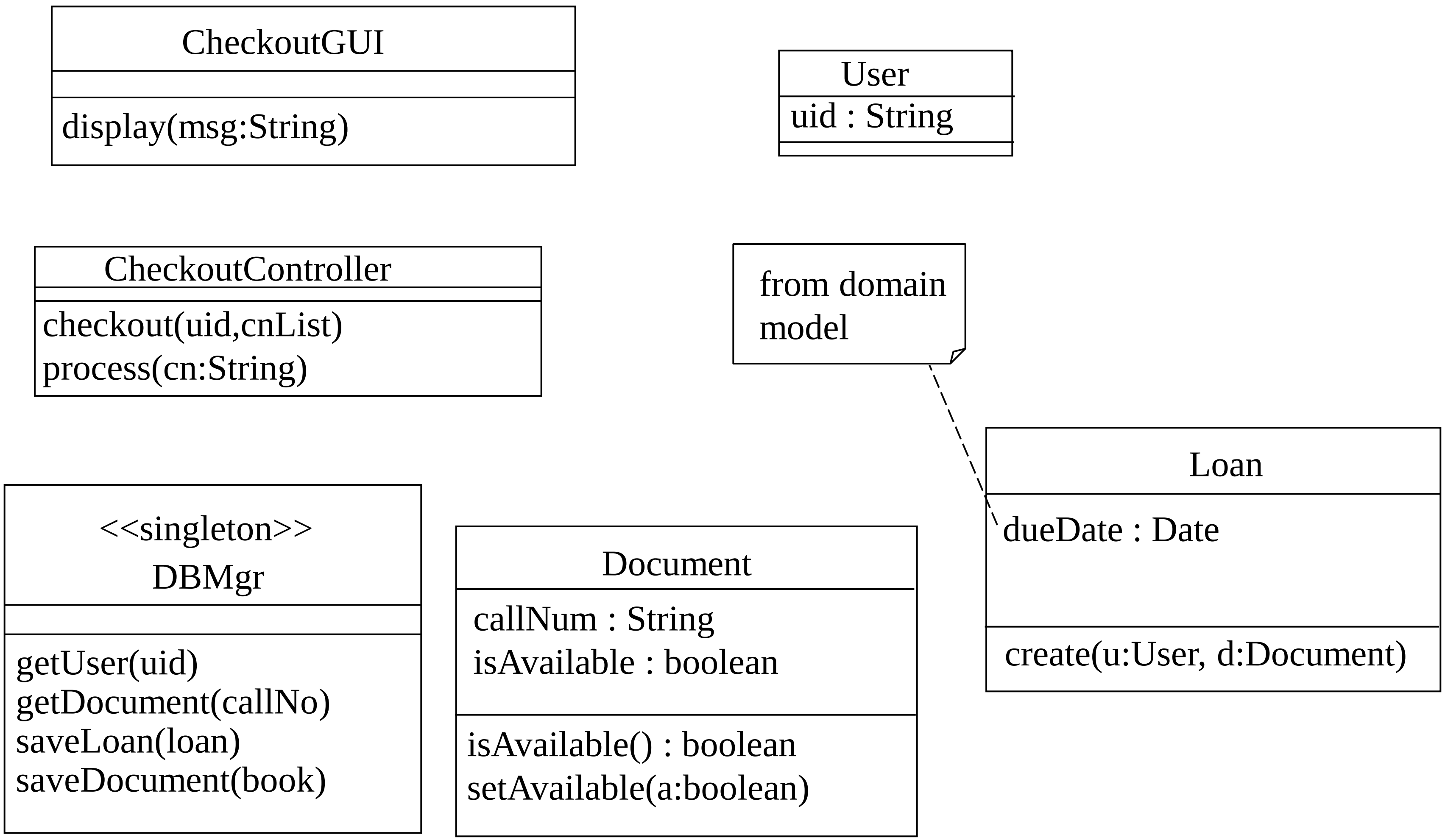
3) Identify and fill in attributes from sequence diagrams and domain model:

- Attributes are not objects and have only scalar types.
- Attributes may be used to get objects.
- Attributes may be identified from getX() and setX(...) methods.
- Needed attributes may also be found in the domain model.

Identify Attributes



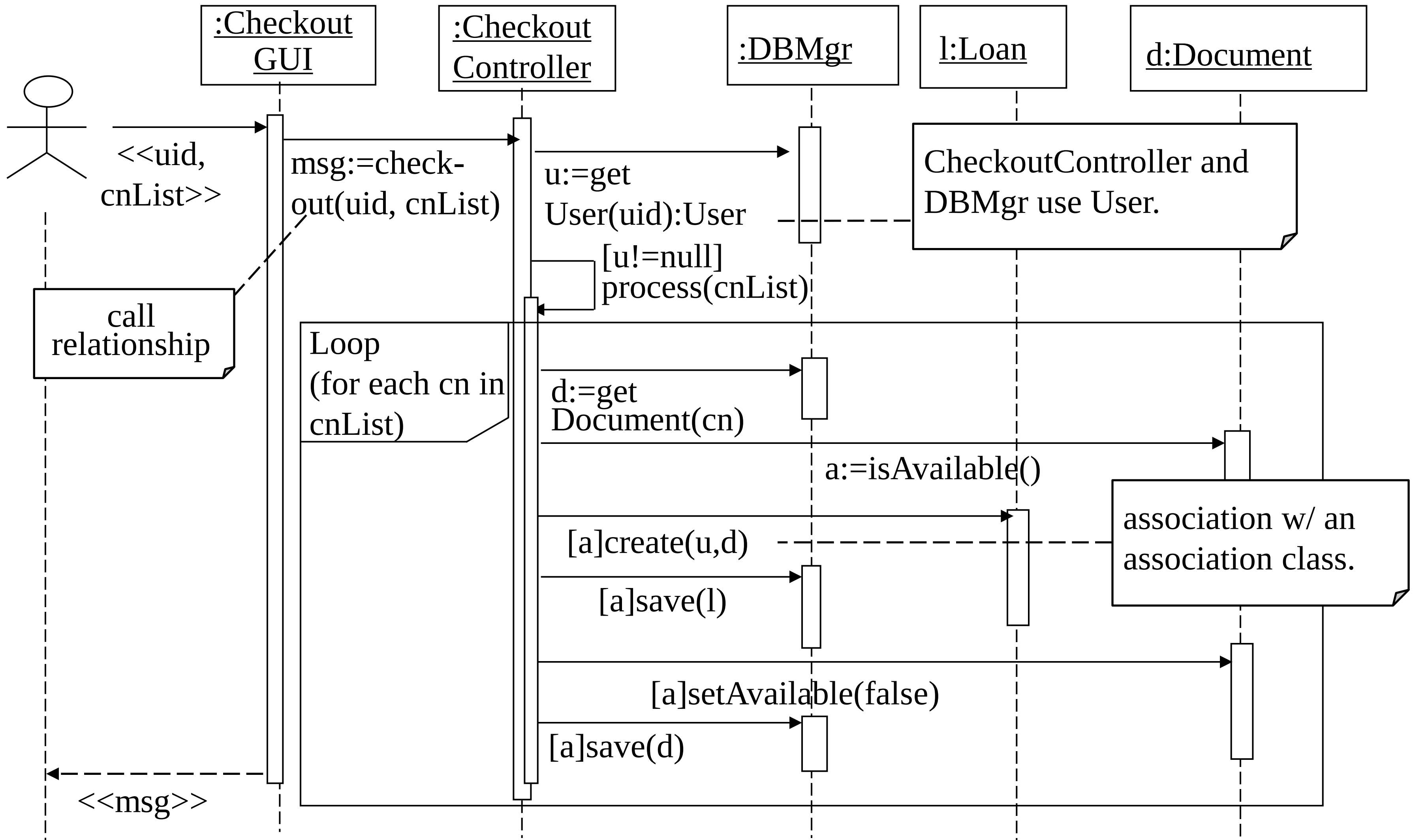
Fill In Attributes



Steps for Deriving DCD

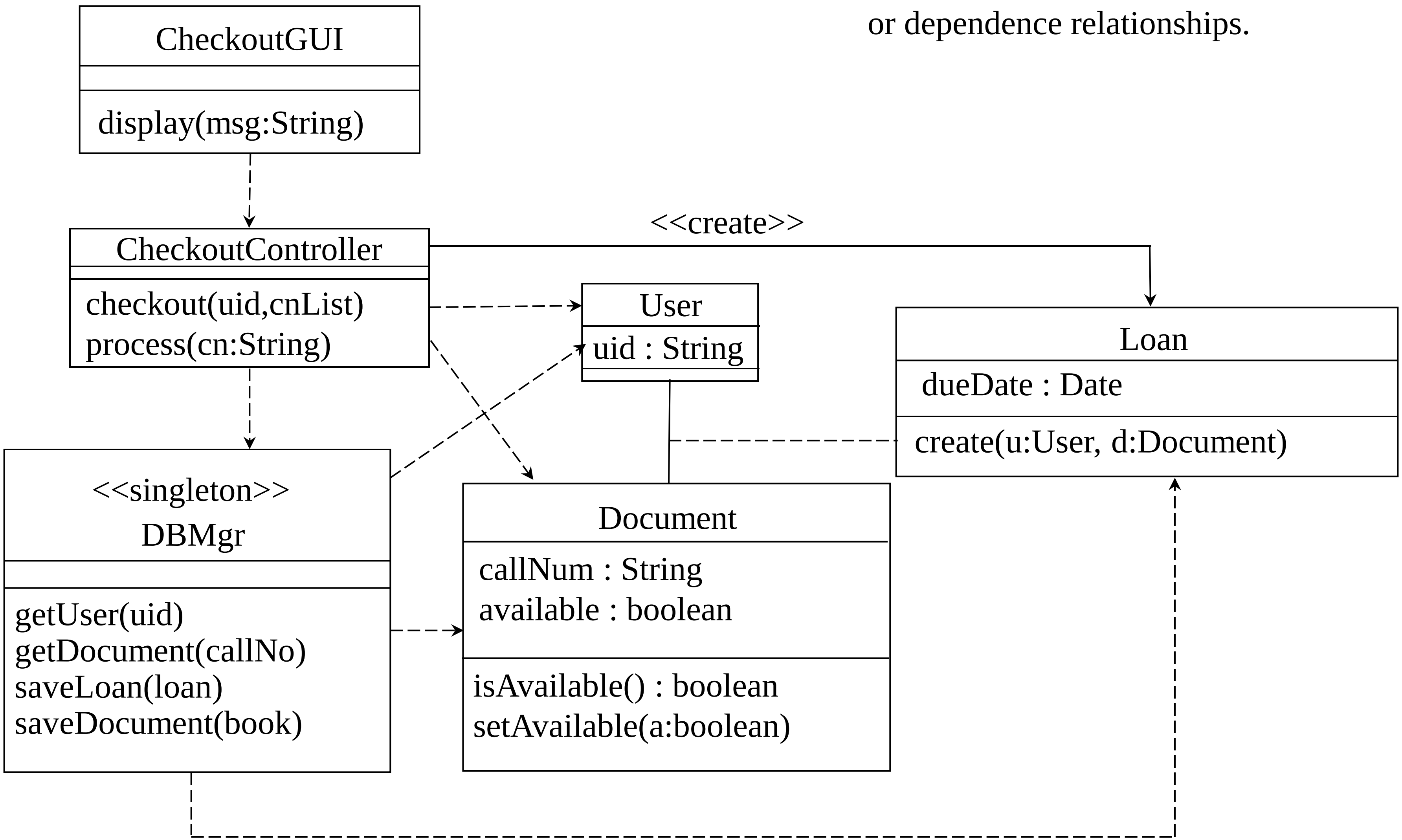
4) Identify and fill in relationships from sequence diagram and domain model:
   - An arrow from one object to another is a call and hence it indicates a dependence relationship.
   - An object passed as a parameter or return type/value indicates an association or uses relationship.
   - Two or more objects passed to a constructor may indicate an association and an association class.
   - The domain model may contain useful relationships as well.

Identify Relationships

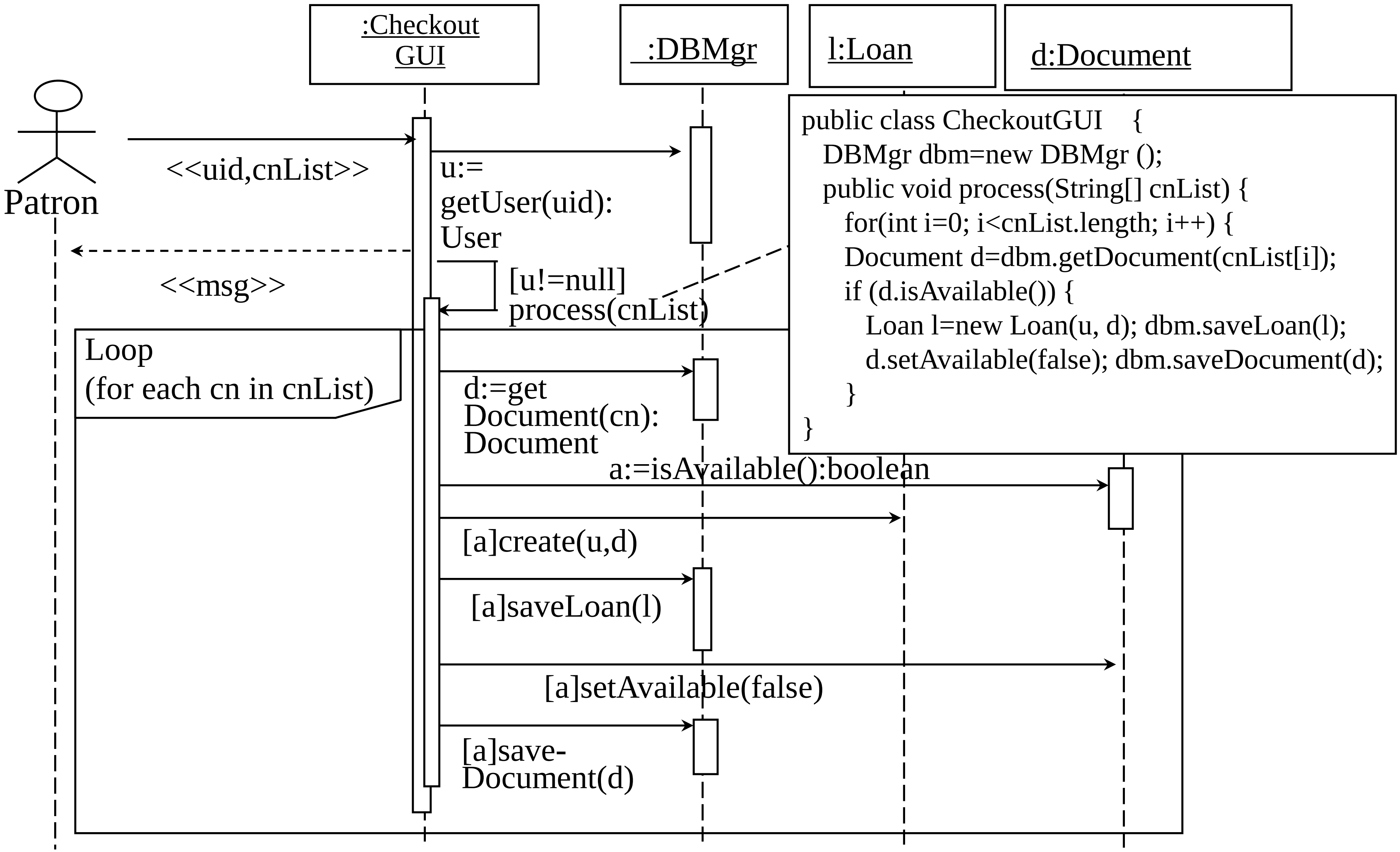


Fill In Relationships

The dashed arrow lines denote uses or dependence relationships.



From Sequence Diagram to Implementation




```
public class CheckoutGUI    {
   DBMgr dbm=new DBMgr ();
   public void process(String[] cnList) {
      for(int i=0; i<cnList.length; i++) {
      Document d=dbm.getDocument(cnList[i]);
      if (d.isAvailable()) {
         Loan l=new Loan(u, d); dbm.saveLoan(l);
         d.setAvailable(false); dbm.saveDocument(d);
      }
}
```

Applying Agile Principles

1. *Value working software over comprehensive documentation.*
2. *Good enough is enough.*

Chapter 12: User Interface Design

Key Takeaway Points

- User interface design is concerned with the design of the look and feel of the user interfaces.
- The design for change, separation of concerns, information-hiding, high-cohesion, low-coupling, and keep-it-simple-and-stupid software design principles should be applied during user interface design.

User Interface Design Activities

- Layout design for windows and dialog boxes.
- Design of interaction behavior.
- Design of information presentation schemes.
- Design of online support.

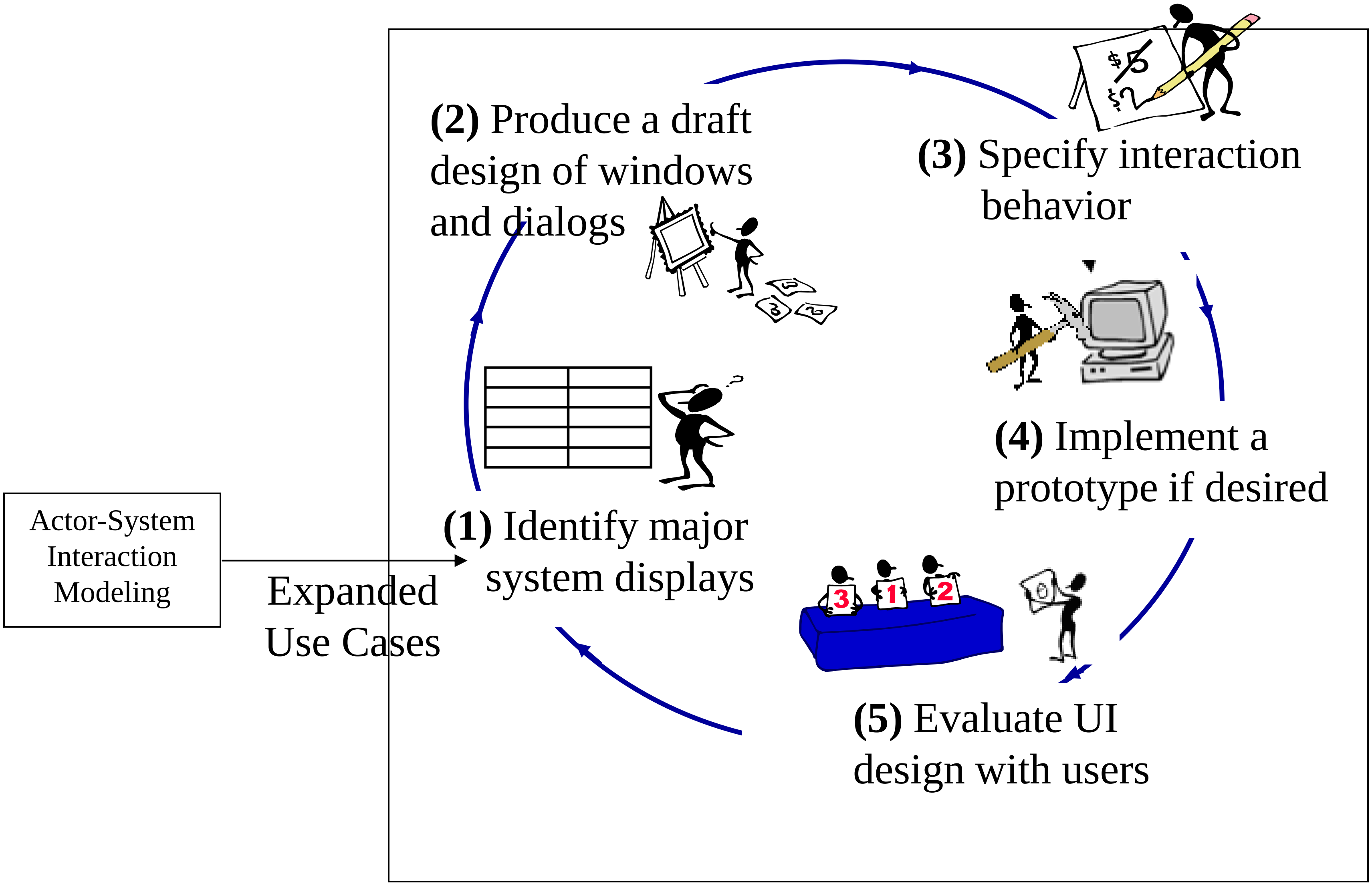
Importance of User Interface Design

- The user interface is the sole communication channel between the user and the system.
- Users' feeling about the interface greatly influences the acceptance of the system and success of the project.
- User-friendly interfaces may improve an organization's productivity and work quality, and reduce operating costs.
- *Technology Acceptance Model (TAM)* is used in IS research all the time!

Graphical User Interface Widgets

- Container widgets
  - window, dialog box, scroll pane, tabbed pane, and layered pane, and others.
- Input, output and information presentation widgets
  - text-oriented I/O widgets, selection-oriented input widgets, featured widgets

User Interface Design Process



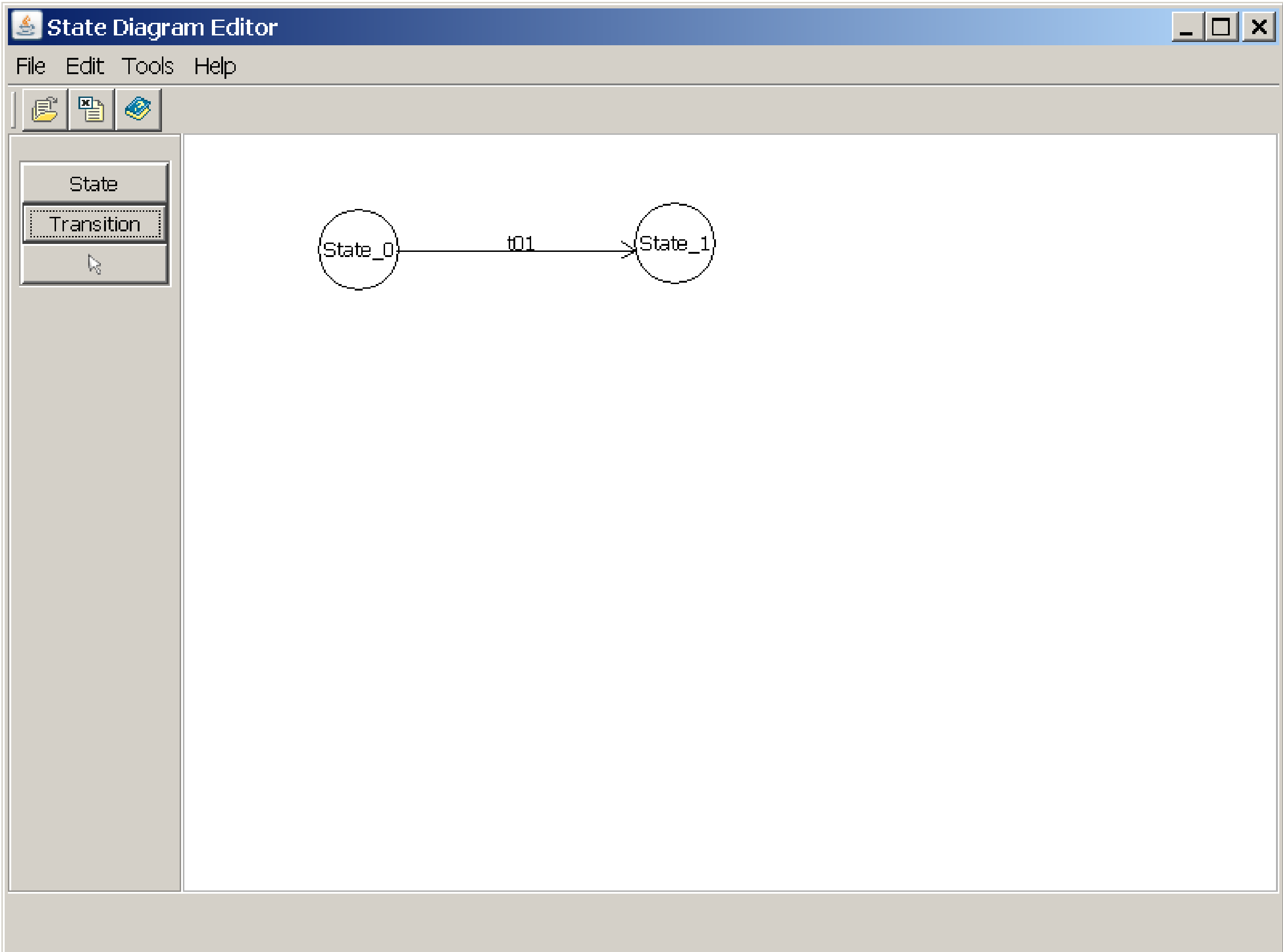
Edit State Diagram System Displays

Expanded Use Case Step	System Display	Information Displayed	User Input	User Actions
0)	Editor Main Window			
2)	• blank diagram • diagram selected • Selection Dialog	• files & directories (inferred)	• diagram file selected	• clicks File on menu bar and selects New Diagram • clicks File on menu bar and selects Open Diagram • locates the diagram and clicks the OK button
4)	• Edit State Dialog • Edit Transition Dialog	• state information • transition information	• edited state information • edited transition information	• clicks State button • clicks Transition button • double-clicks a state or transition • clicks Edit on menu bar and selects Undo or Redo • clicks OK or Cancel button
6)	Save State Diagram As Dialog	• "Diagram Saved" or "Diagram Saved As ..." in status bar	• requested information	• clicks File on menu bar and selects Save or Save As • clicks OK or Cancel button (Cancel button is inferred)

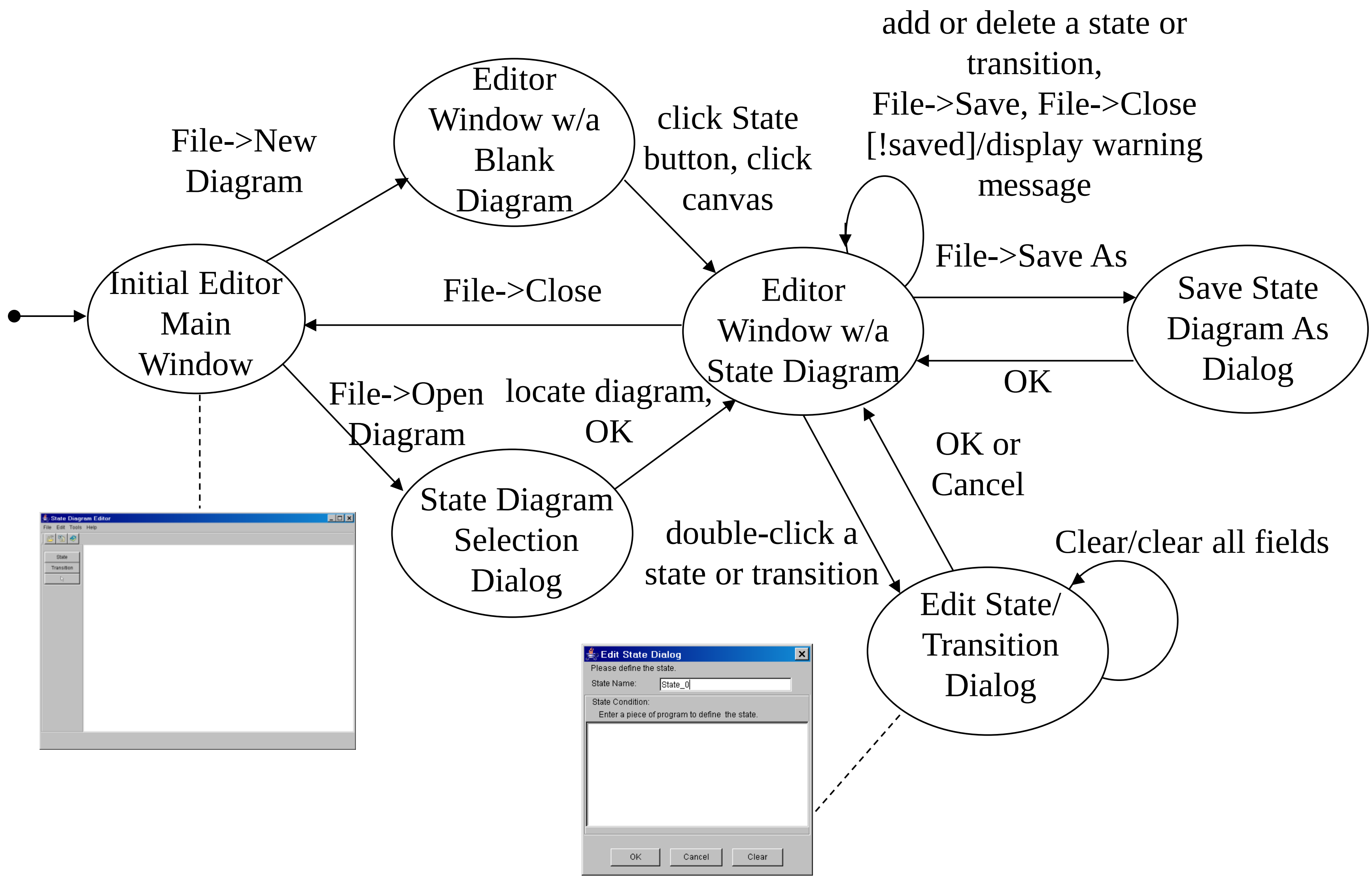
Windows, Dialogs and Widgets

Window/Dialog	GUI Component/Widget
Editor Main Window	Diagram Canvas (for blank diagram and selected diagram) Status bar Menu bar File (New Diagram/Open Diagram/Save/Save As) Edit (Undo/Redo) Buttons: State, Transition, Pointer (inferred)
State Diagram Selection File Chooser	File browser Buttons: OK, Cancel
Edit State Dialog	Text fields State Name // other text fields for editable state attributes as shown in the domain model Text areas State Condition ...
Edit Transition Dialog	Text fields Transition Name // other text fields for editable transition attributes as shown in the domain model Text areas Transition Code ...
Save State Diagram As File Chooser	File browser Buttons: Save, Cancel

Layout Design of State Diagram Editor



State Diagram Editor Behavior (partial)



Using Prototypes

- Prototypes are useful for obtaining user feedback.
- Types of prototypes
  - Static approaches generate nonexecutable prototypes.
  - Dynamic approaches generate executable prototypes.
  - Hybrid approaches construct static prototypes during the initial stage of prototype development and switch to dynamic prototyping later.

Evaluating User Interfaces with Users

- User interface presentation.
- User interface demonstration.
- User interface experiment.
- User interface review meeting.
- User interface survey.

User Support Capabilities

- User support capabilities include online documentation, context-dependent help, error messages, and recovery.
- Online help should let the user find the needed information easily.
- Context-dependent help is a user-friendly design technique. Chain of responsibility supports this.
- Error messages should be user-oriented, rather than developer-oriented, and be easy to understand.

Recover from Undesired State

- Undo and redo operations (command pattern)
- Automatic backup and restore system states (memento pattern)
- Exception handling
- Software fault tolerance

Guidelines for User Interface Design

- User interface design should be user-centric.
- The user interface should be consistent.
- Minimize switching between mouse mode and keyboard mode.
  - Provide keyboard shortcuts
  - Especially for text-intensive applications
- A "nice feature" may not turn out to be that "nice".
- *Eat your own cooking*.

Applying Agile Principles

- Active user involvement is imperative. A collaborative and cooperative approach between all stakeholders is essential.
- Requirements evolve but the timescale is fixed.
- Develop small, incremental releases and iterate. In addition, focus on frequent delivery of software products.
- A good enough user interface design is enough. Value the working software over the design.
- Capture requirements at a high level; lightweight and visual.

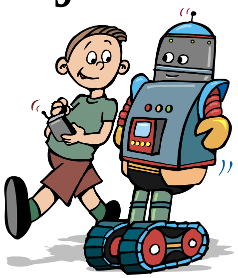
Chapter 13: Object State Modeling for Event-Driven Systems

Key Takeaway Points

- Object state modeling is concerned with the identification, modeling, design, and specification of state-dependent, reactive behavior of objects.
- The state pattern reduces the complexity of state behavior design and implementation, and makes it easy to change.

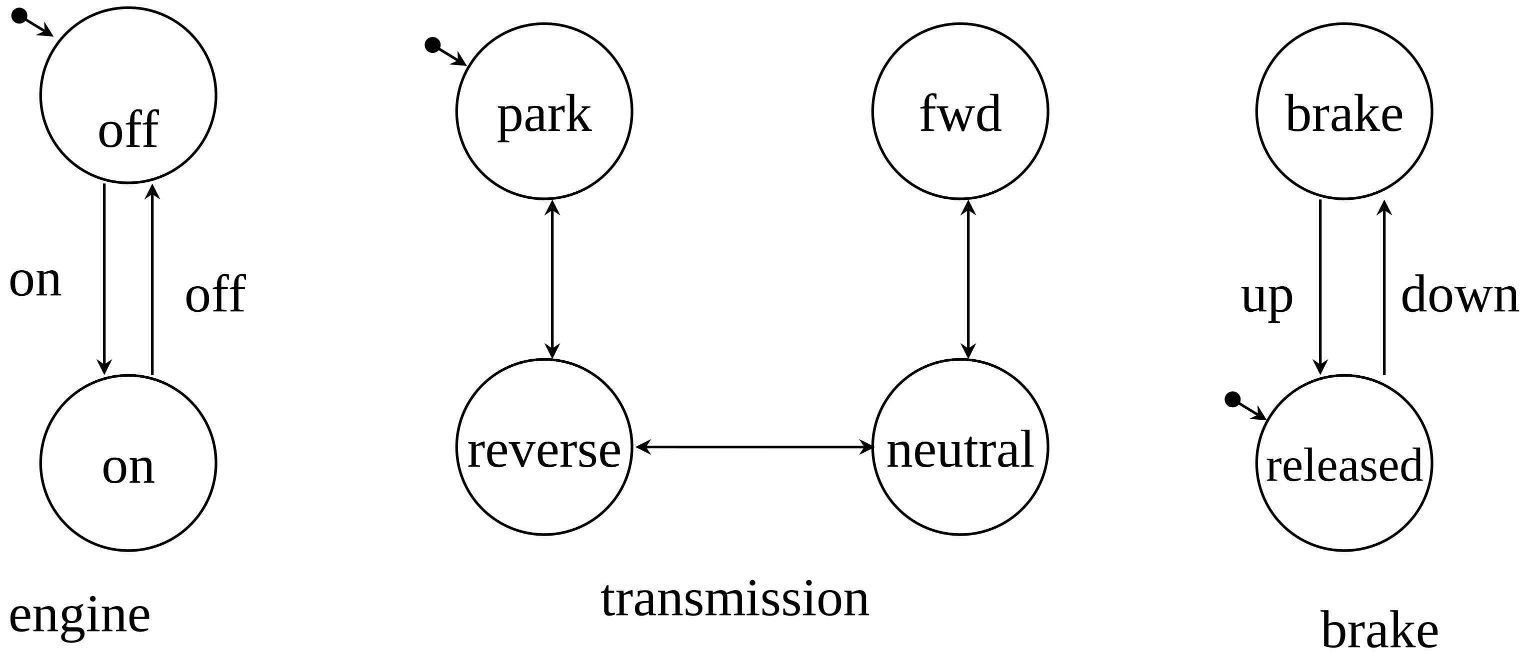
Object State Modeling

- Identification, modeling, analysis, design, and specification of state-dependent reactions of objects to external stimuli.
  - What are the external stimuli of interest?
  - What are the states of an object?
  - How does one characterize the states to determine whether an object is in a certain state?
  - How does one identify and represent the states of a complex object?
  - How does one identify and specify the state-dependent reactions of an object to external stimuli?
  - How does one check for desired properties of a state behavioral model?



State Behavior Modeling

- State dependent behavior is common in software systems.

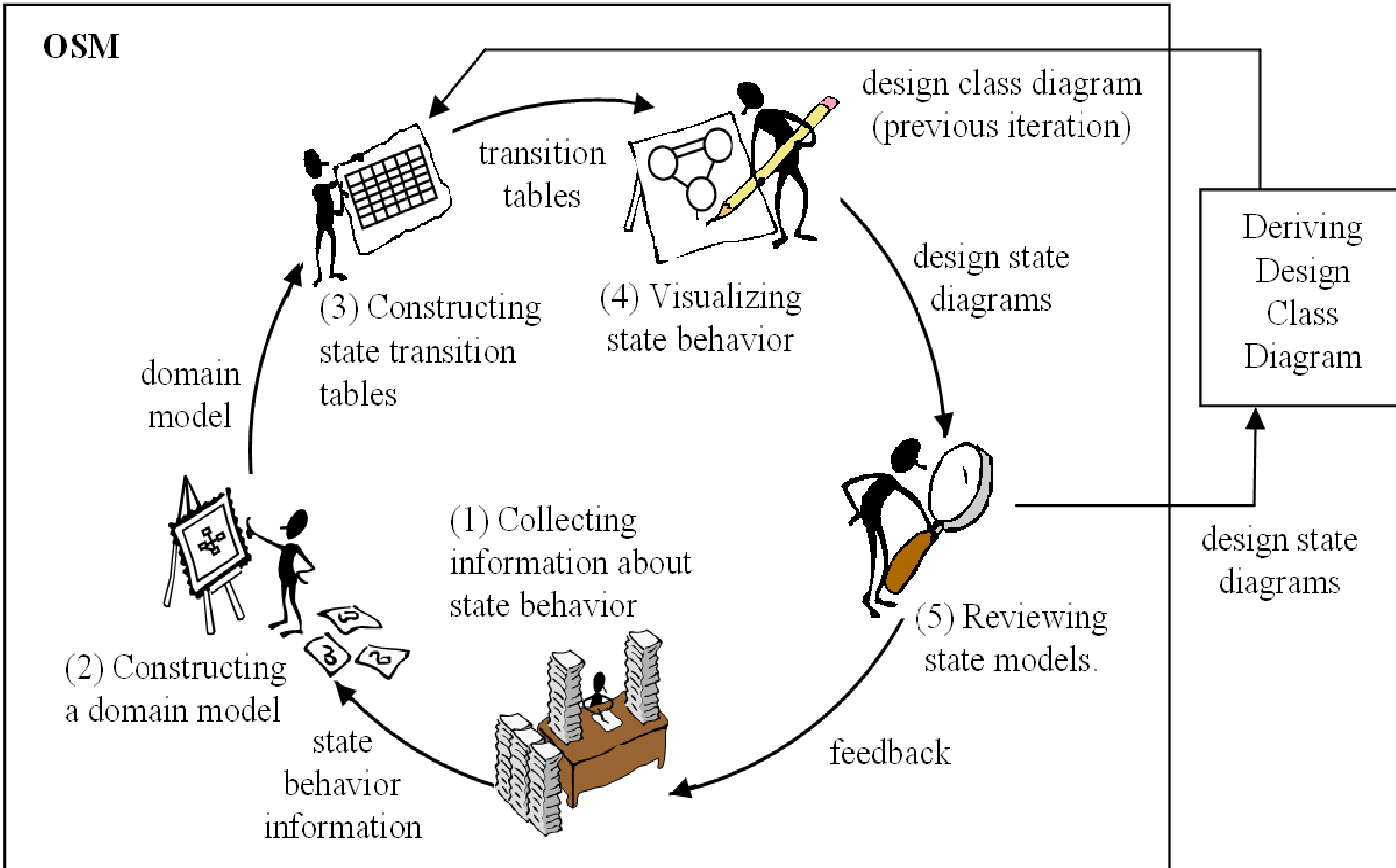


What is the state dependent behavior of a car?

Basic Definitions

- An *event* is some happening of interest or a request to a subsystem, object, or component.
- A *state* is a named abstraction of a subsystem/ object condition or situation that is entered or exited due to the occurrence of an event.

Object State Modeling Steps



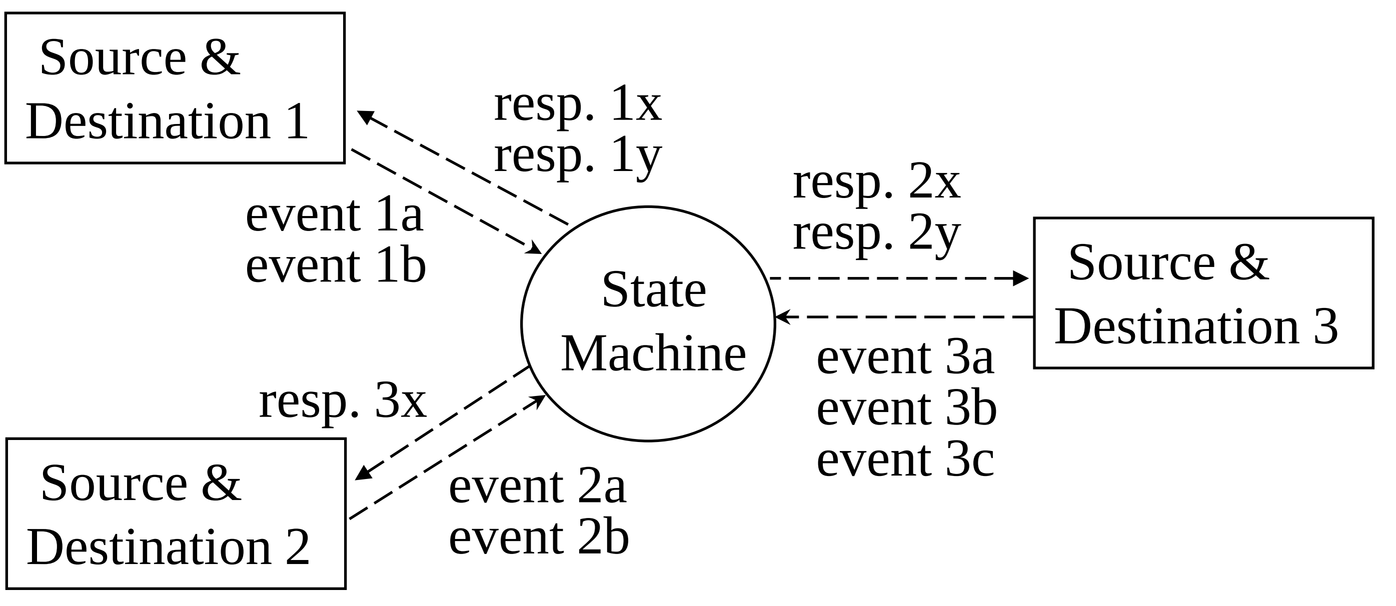
Collecting & Classifying State Behavior Information

What to look for	Example	Classification	Rule
Something of interest happened	An online application submitted.	Event	E1
Mode of operation	A cruise control operates in activated/ deactivated modes.	State	S2
Conditions that govern the processing of an event	Turn on AC if room temperature is high	Guard condition	G1
An act associates with an event	Push the lever down to set the cruising speed.	Response	R1

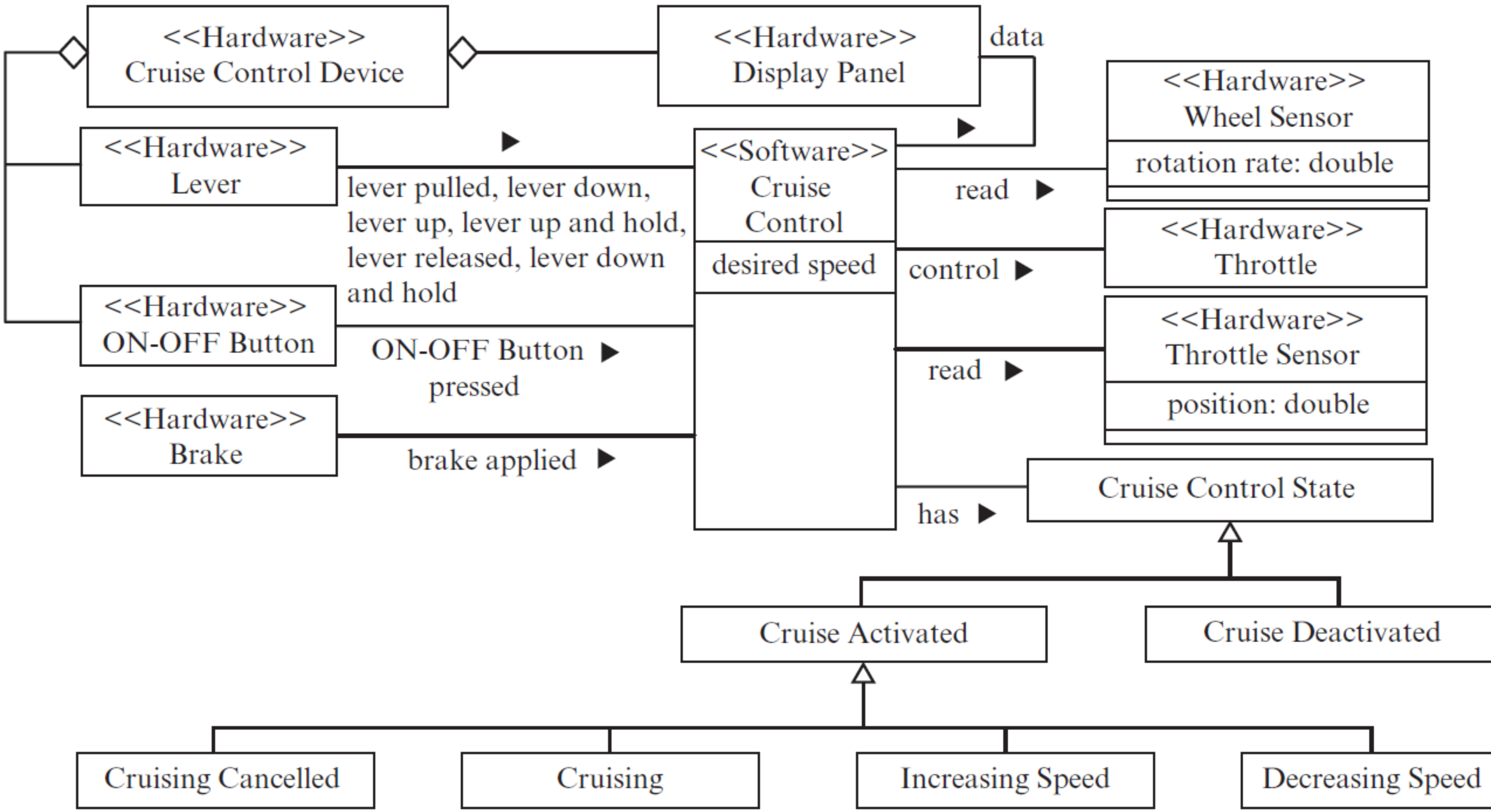
More rules are found in the textbook (p 322).

Constructing a Domain Model

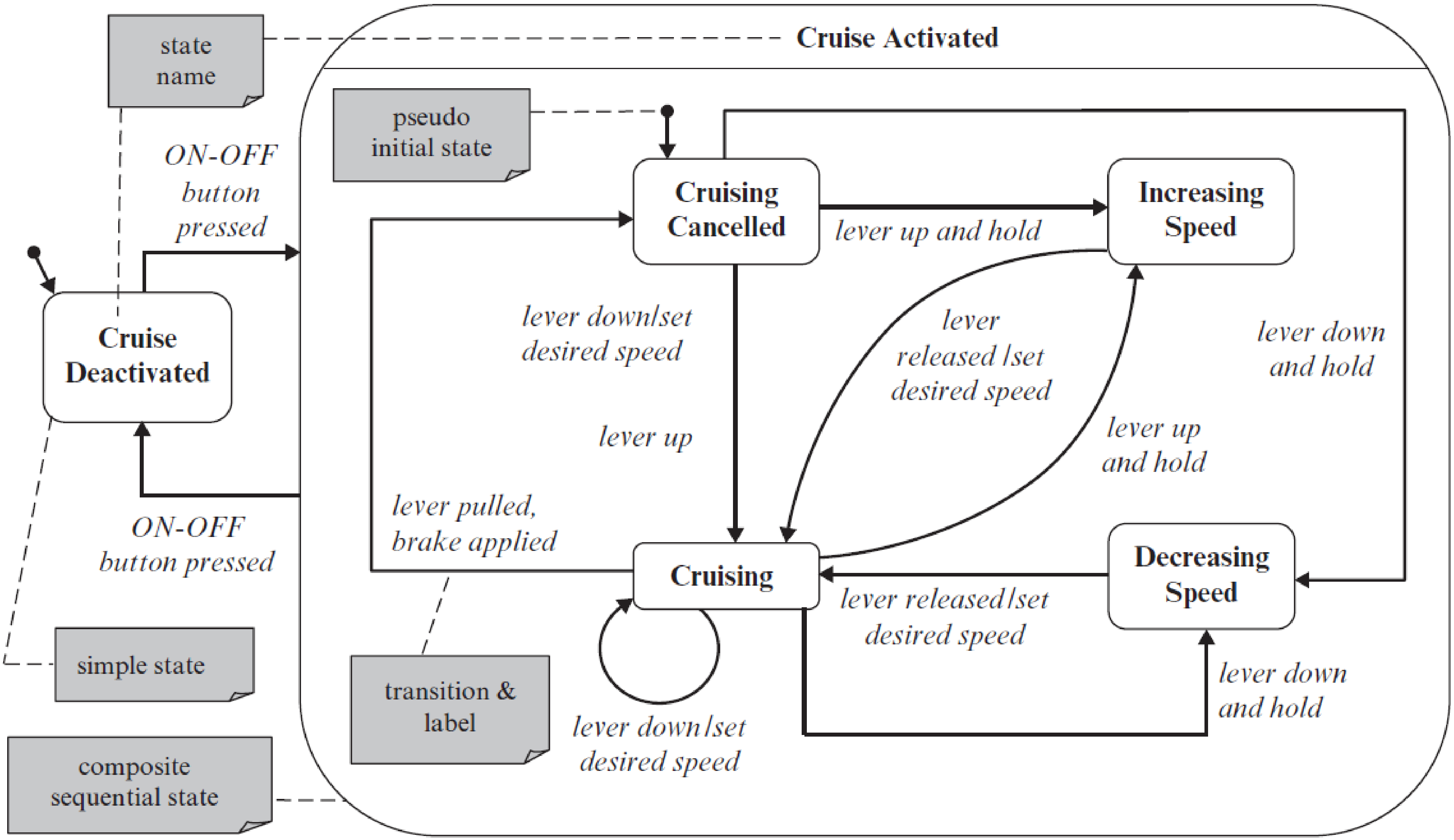
- The domain model shows relationships between the state dependent software and its context.



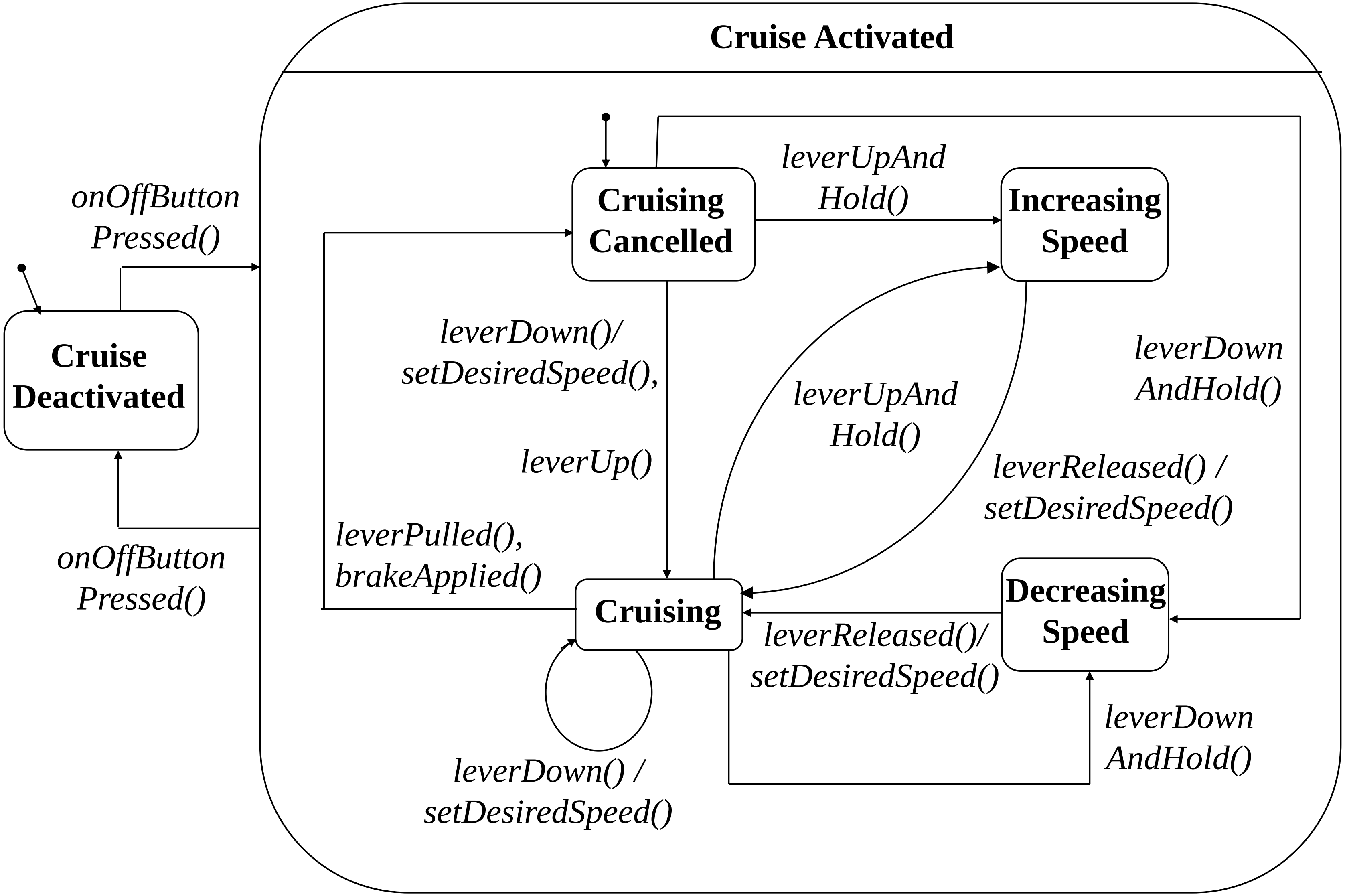
Cruise Control Domain Model



Converting to State Diagram



Converting Events/Transitions to Function Calls



Update Design Class Diagram

- Add methods labeling the transitions to the subject class:

CruiseControl
onOffButtonPressed() leverDown() leverUp() brakeApplied() leverDownAndHold() leverUpAndHold() leverPulled() leverReleased() setDesiredSpeed()

Implementing State Behavior

- Conventional approaches:
  - nested switch approach
  - using a state transition matrix
  - using method implementation

Conventional Implementation: Nested Switches

- Using nested switch statements

```
switch (STATE) {
  case Init: switch (EVENT) {
        case State button clicked:
              set cursor to crosshair, STATE=AddState;
              break;
        case Trans button clicked:
              set cursor to crosshair; STATE=AddTransition;
              break; }
  case AddState: switch (EVENT) { ... }
  case AddTransition: switch (EVENT) { ... }
  case ...
}
```

- Using a state transition matrix

Conventional State Transition Matrix

State \ Event	State button clicked	mouse clicked	Trans button clicked	mouse pressed	mouse dragged	mouse released	Select button pressed
Init	set cursor to crosshair/ Add State		set cursor to crosshair/ Add Transition				
Add State		add state to state diagram; repaint state diagram; reset cursor/ Init					reset cursor/ Init
Add Transition				[source found] save transition source/ Trans Source Selected			reset cursor/ Init
Trans Source Selected					show rubber-band line/ Trans Source Selected	add transition to state diagram; repaint state diagram; reset cursor/ Init	reset cursor/ Init

Using Method Implementation

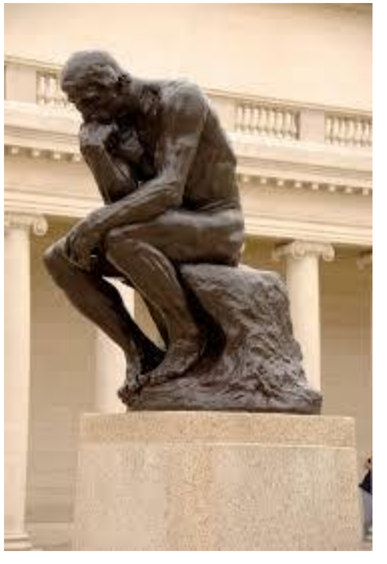
- State behavior of a class is implemented by member functions that denote an event in the state diagram.
- The current state of the object is stored in an attribute of the class.
- A member function evaluates the state variable and the guard condition, executes the appropriate response actions, and updates the state variable.

Problems with Conventional Approaches

- High cyclomatic complexity (number of states multiplied by number of transitions).
- Nested case statements make the code difficult to comprehend, modify, test, and maintain.
- Difficult to introduce new states (need to change every event case).
- Difficult to introduce new events (need to change every state case).
- Solution: applying state pattern

For your personal edification…

…Thinking on your own!



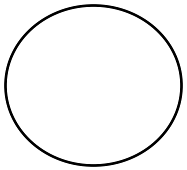
Transformation Schema for Real Time Systems

- Ward and Mellor extended DFD with control flows and control processes.
- Control processes are modeled by Mealy type state machines.
- Control processes control the ordinary data transformational processes.
- Control flows represent events or triggers to control processes and responses of control processes to transformational processes.

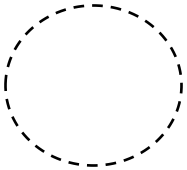
Transformation Schema for Real Time Systems

- Real time data flows, which must be processed quickly enough to prevent losing the data.
- Data flows and control flows may be related using logical connectors.
- Timing may be specified for state transitions and data transformation processes.

Real Time Systems Design



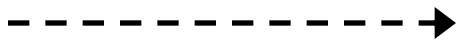
transformational processes, representing computations or information processing activities



control processes, representing system’s state dependent behavior, which is modeled by a Mealy type state machine

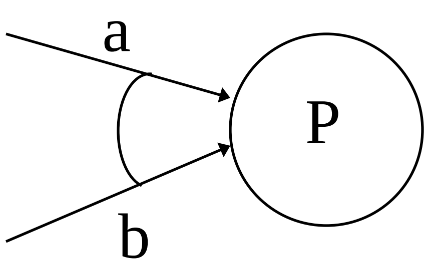
continuous data flow, which must be processed in real time

ordinary or discrete data flow

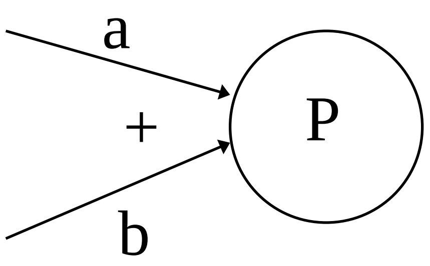


event flow or control flow that triggers a transition of the state machine of a control process, or a command from a control process to a transformational process

Real Time Systems Design



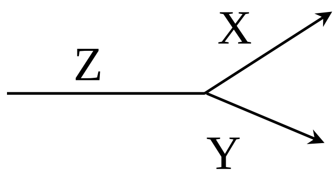
indicates that both data flow a and data flow b are required to begin executing process P



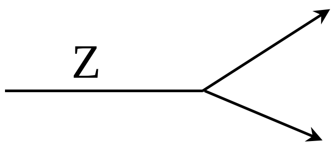
indicates that either data flow a or data flow b is required to begin executing process P

These logical connector can be applied to both data flow and control flow and transformation process and control process.

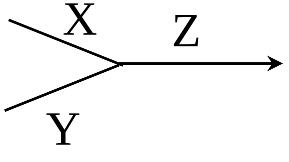
Real Time Systems Design



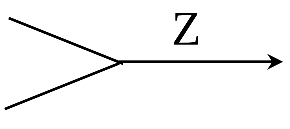
Two subsets of Z are used by two different successor processes.



All of Z is used by two different successor processes.

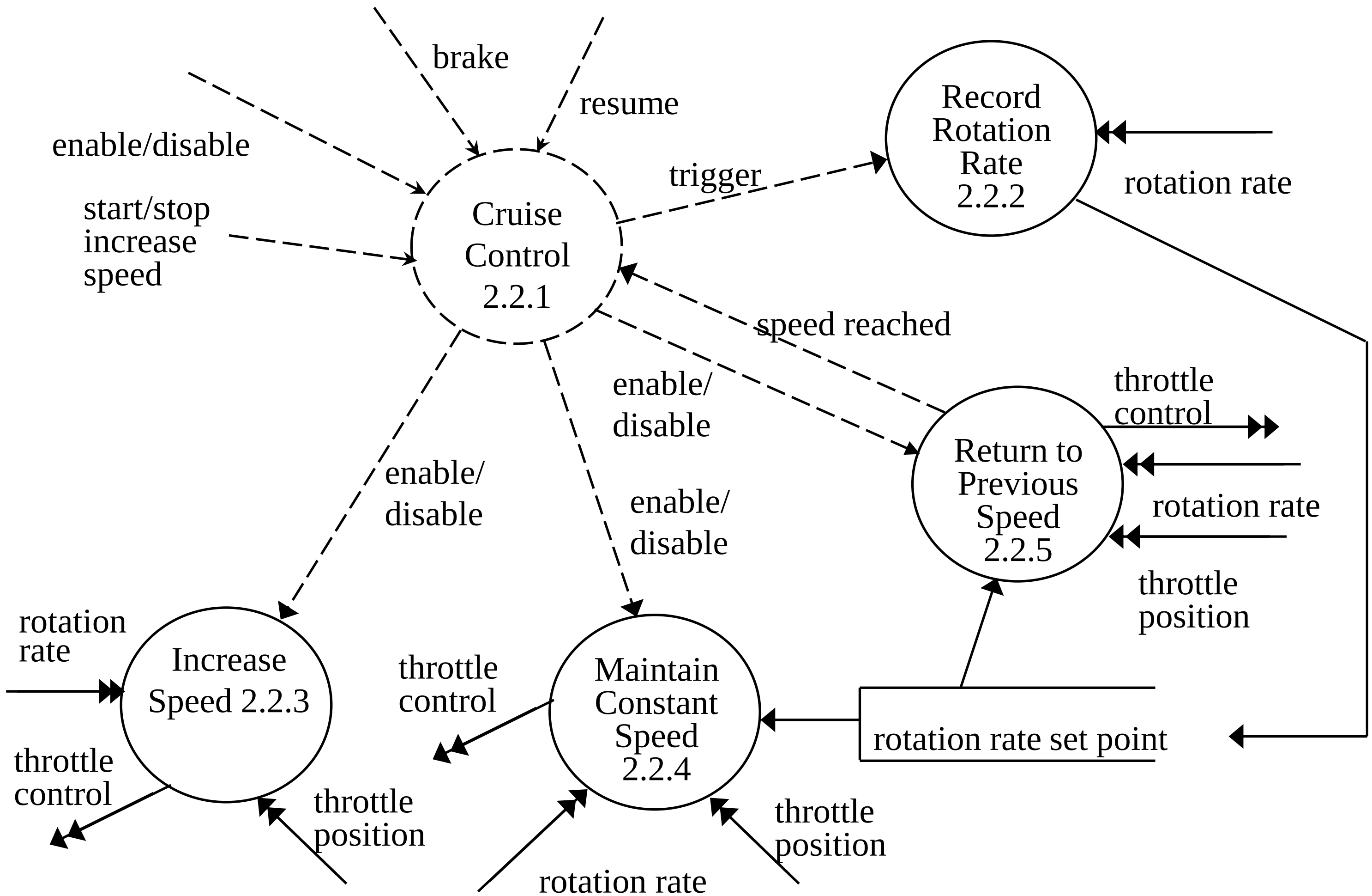


Z is composed of Two subsets provided by two different predecessor processes.

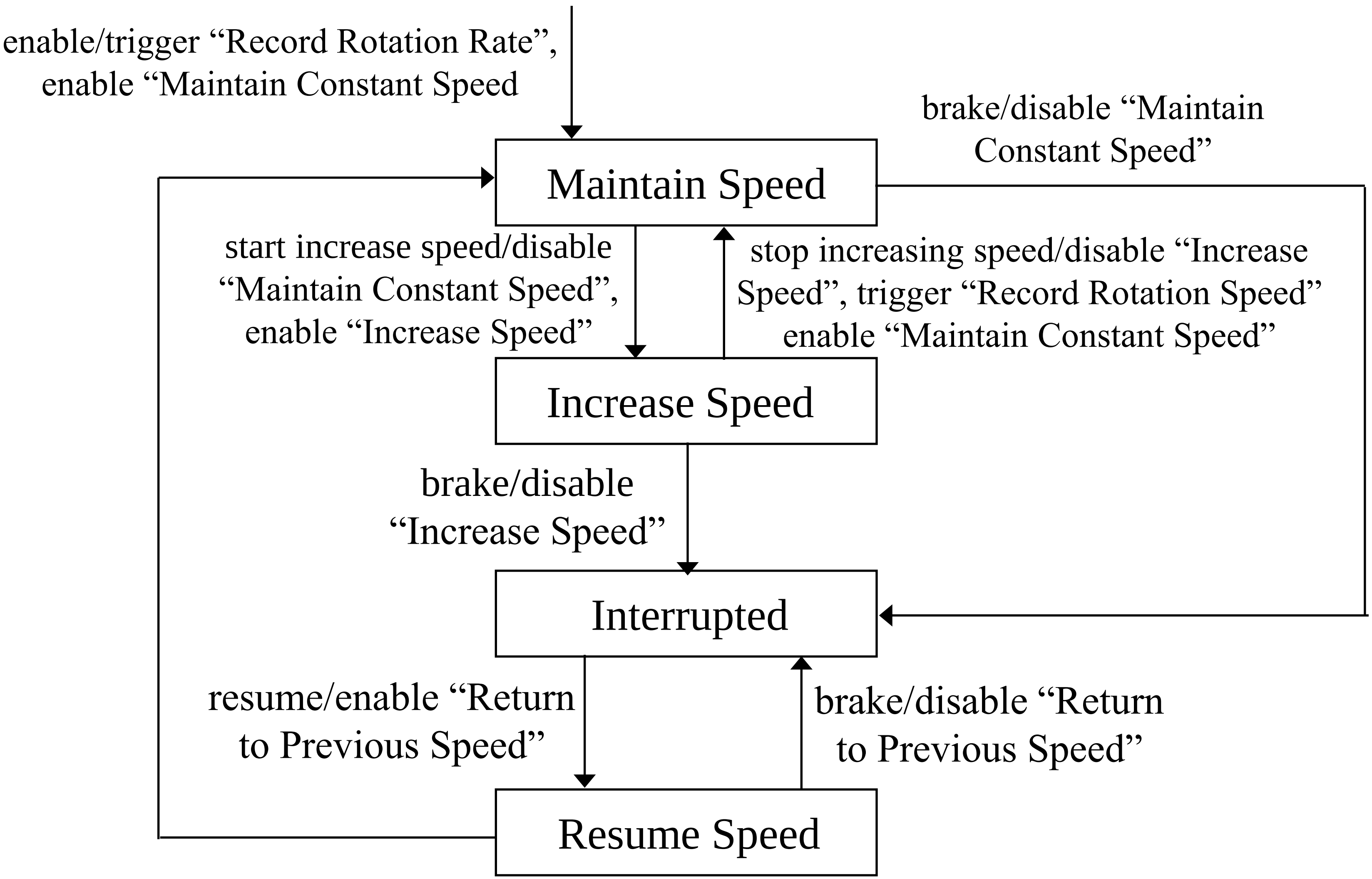


All of Z is provided by either one of two predecessor processes.

Cruise Control Example



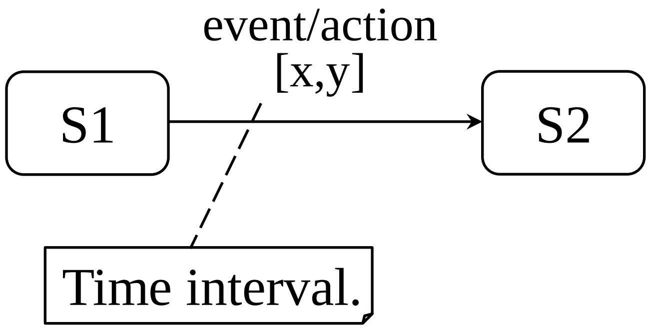
Cruise Control State Machine



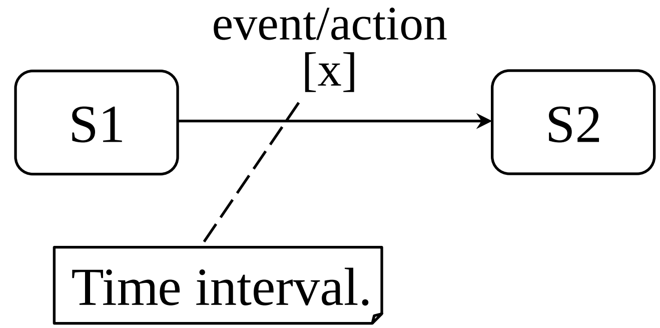
Timed State Machine

- Time intervals can be used to label the state transitions.
- The time intervals define the timing lower bounds and upper bounds allowed for processing the event and executing the list of actions.
- The time interval can be decomposed to define the allowed times for processing the event, and executing each of the actions in the action list.
- Similarly, time can be defined for state entrance action, state exit action, and state activity.

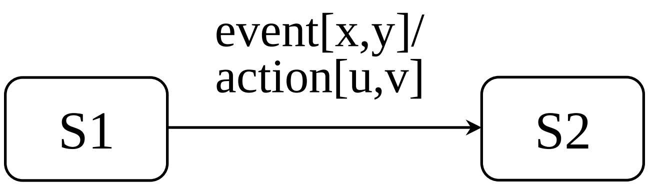
State Diagram for Real Time Systems



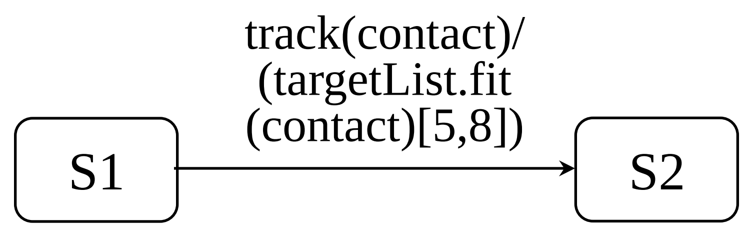
The time allowed for processing the event and executing the action is x to y time units.



The time allowed for processing the event and executing the action is x time units.



The time allowed for processing the event is x to y time units and executing the action is u to v time units.



The time allowed for targetList object to fit the contact with a tracked target is 5 to 8 time units.

Applying Agile Principles

- Work closely with the customer and users to identify and model the state behavior.
- Capture the state behavior at a high level, lightweight, and visual.
- Value working software over comprehensive documentation—do barely enough modeling.

Chapter 14. Activity Modeling for Transformational Systems

Key Takeaway Points

- Activity modeling deals with the modeling of the information processing activities of an application or a system that exhibits sequencing, branching, concurrency, as well as synchronous and asynchronous behavior.
- Activity modeling is useful for the modeling and design of transformational systems.

What Is Activity Modeling

- Activity modeling focuses on modeling and design for
  - complex information processing activities and operations
  - information flows and object flows among the activities
  - branching according to decisions
  - synchronization, concurrency, forking, and joining control flows
  - workflow among the various departments or subsystems

Why Activity Modeling

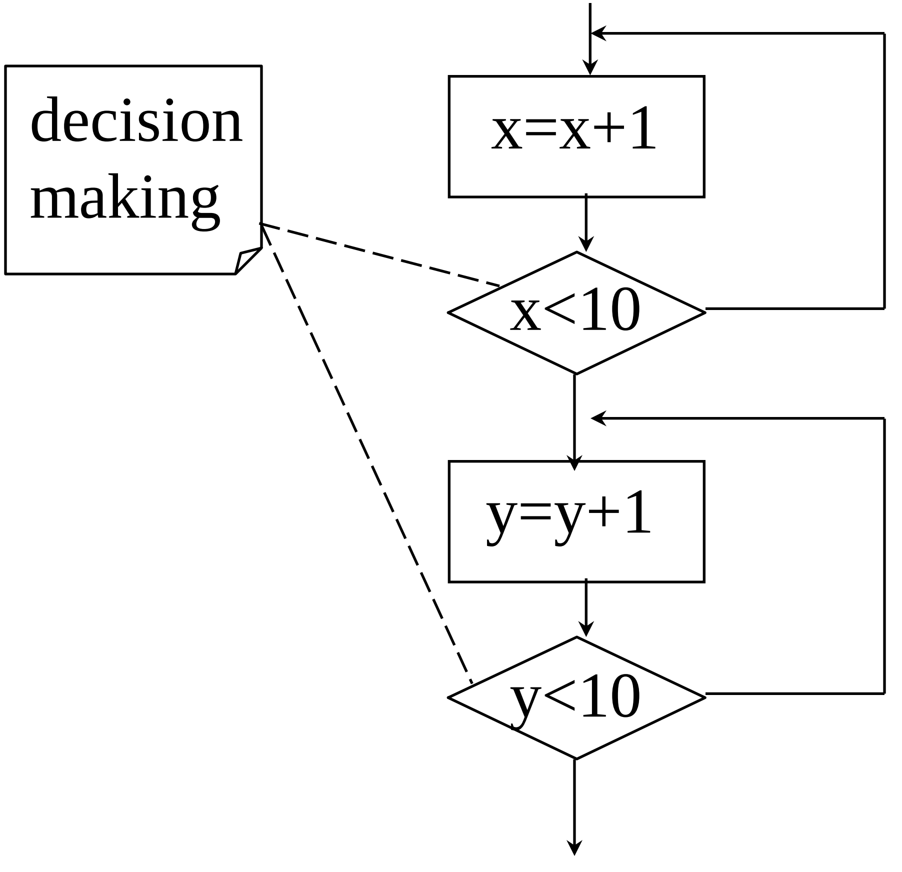
Systems analysis and design activities need to:

- describe current information processing activities in the organization or existing system (modeling of the existing system) to help the development team understand the existing business
- describe information processing with the proposed solution (system design)

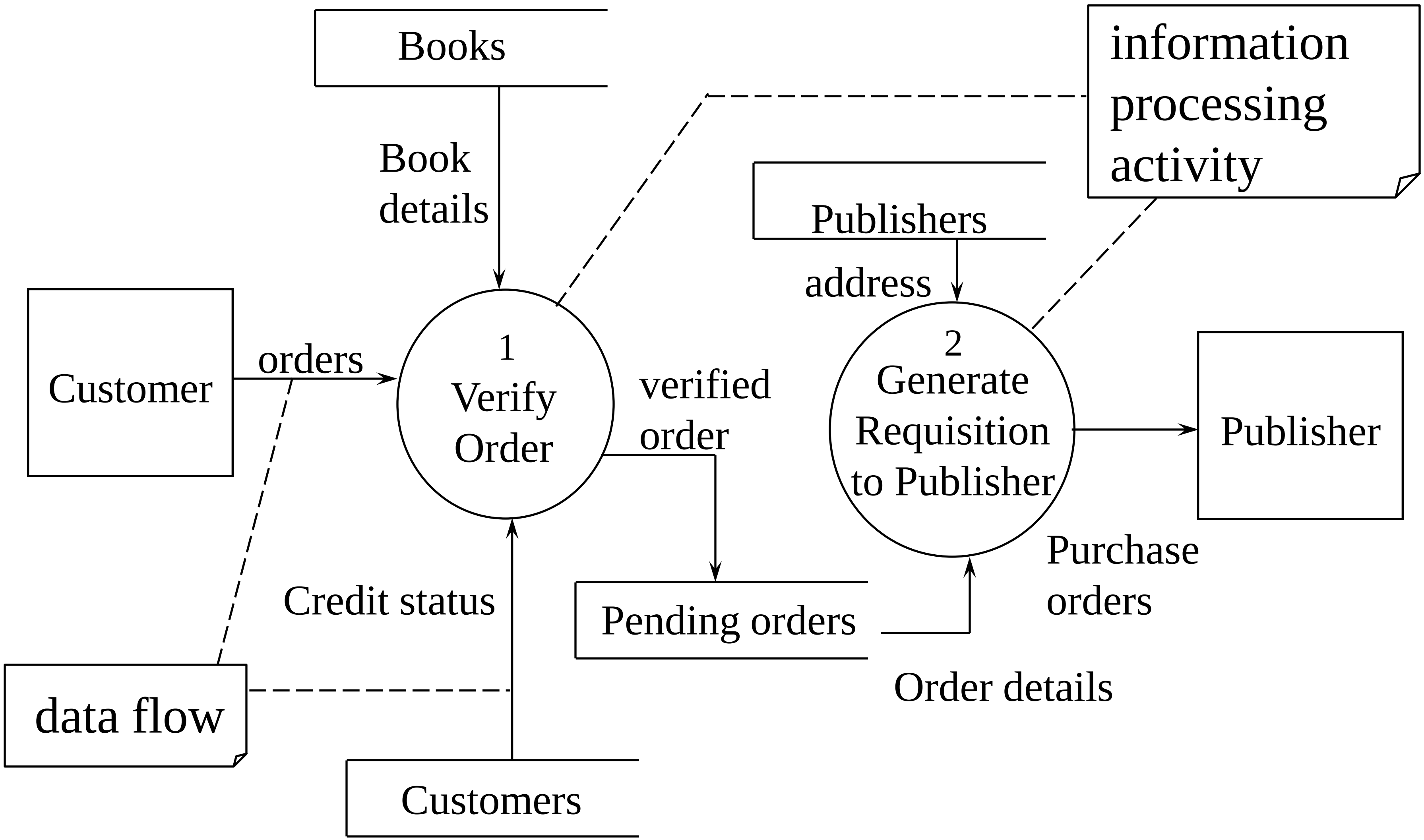
Activity Diagram

- An activity diagram models the information processing activity in the real world (analysis model) or the system (design model).
- A UML activity diagram is a combination of
  – flowchart diagram
    - for decision making or branching
  – data flow diagram
    - for information processing and data flows
  – Petri net diagram
    - for various control flows
    - for synchronization, concurrency, forking, and joining

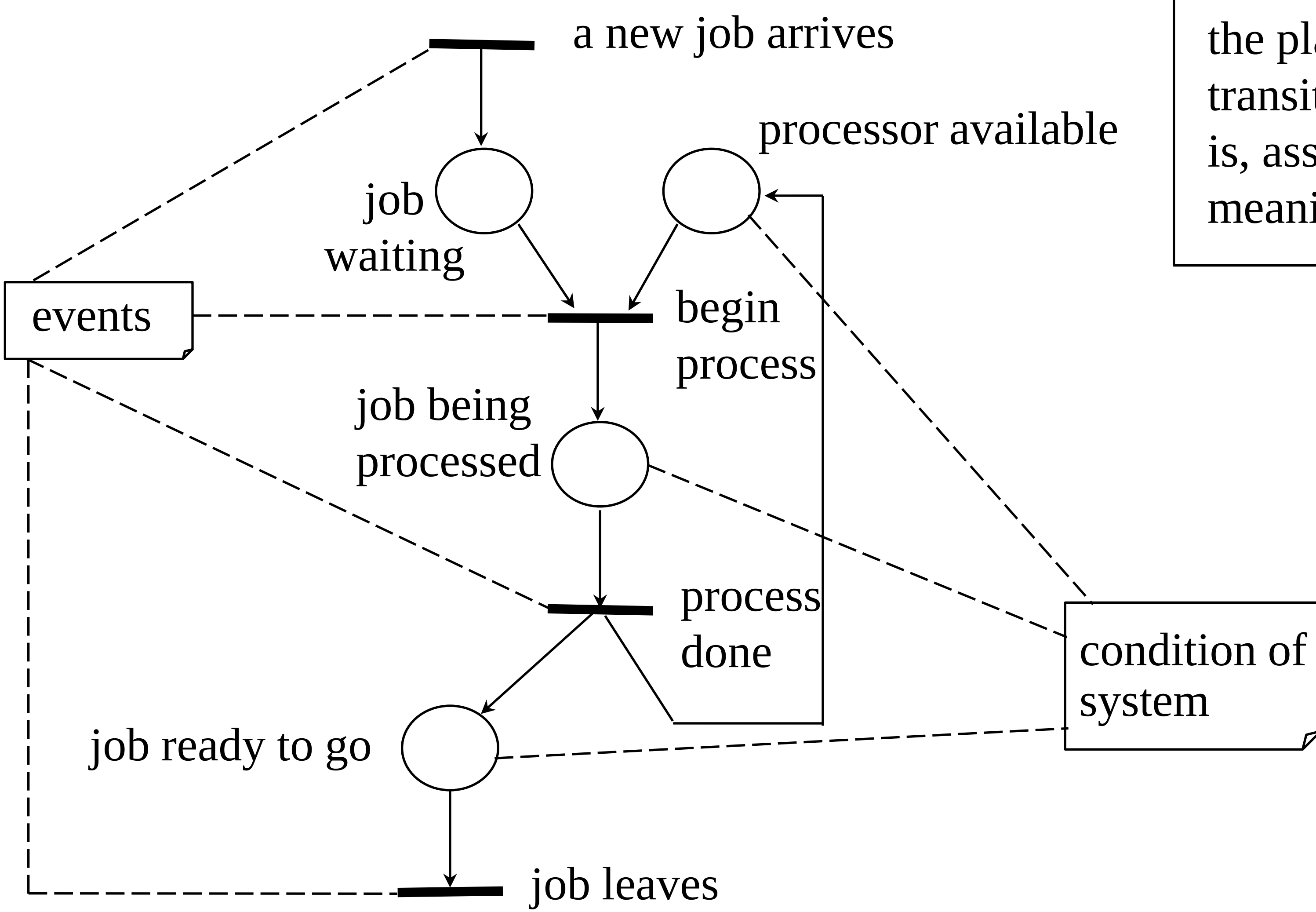
A Flowchart



A Data Flow Diagram



A Petri Net Example

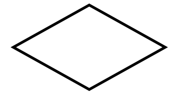


You can interpret the places and transitions. That is, assigning meanings to them.

Activity Diagram Notions and Notations

Activity or action

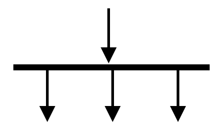
Conditional branching



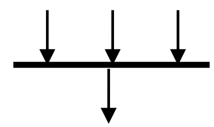
Control flow

Object flow

Forking

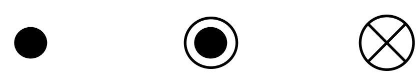


Joining or synchronization

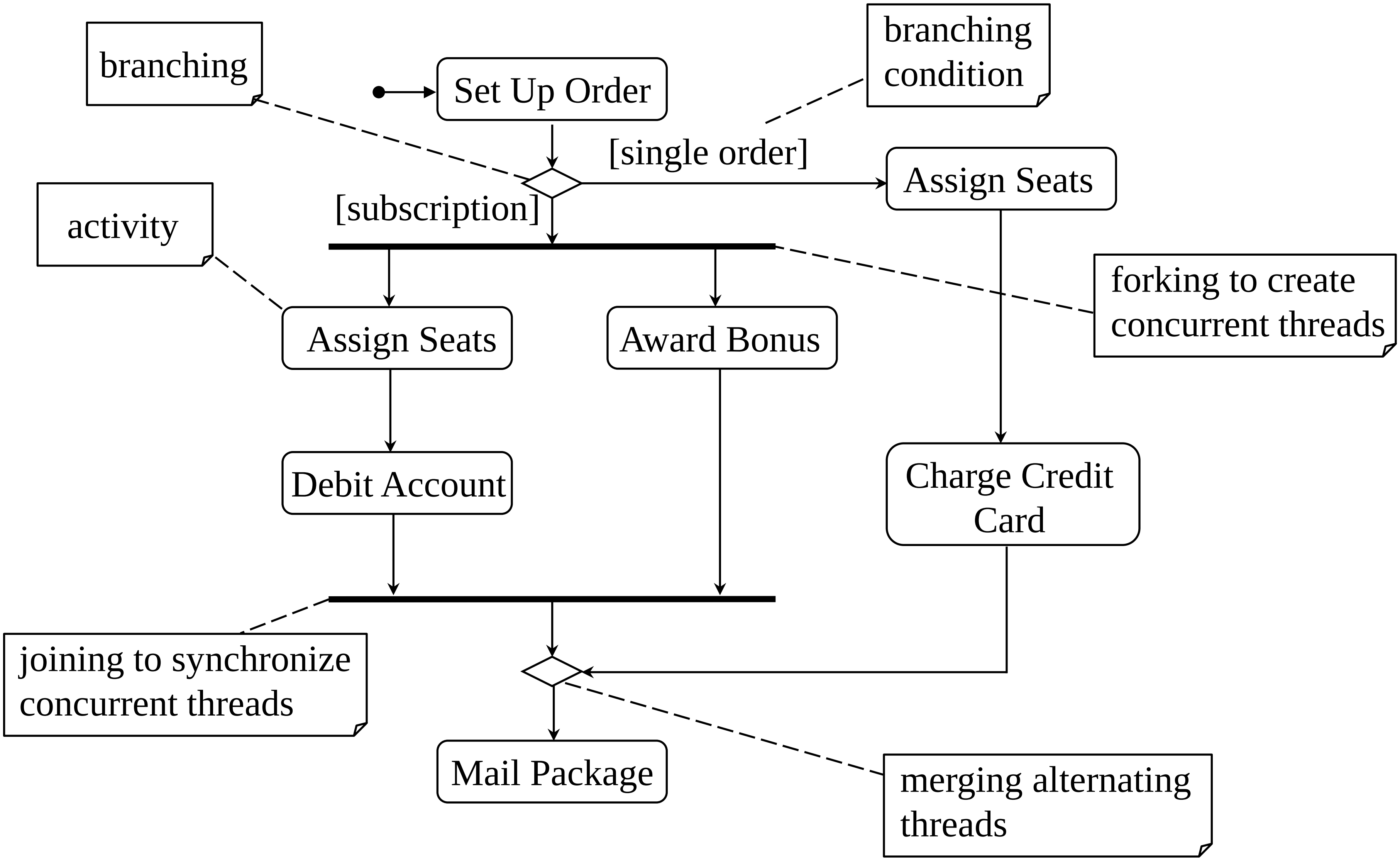


Swim lane to represent info and control flow between departments/subsystems

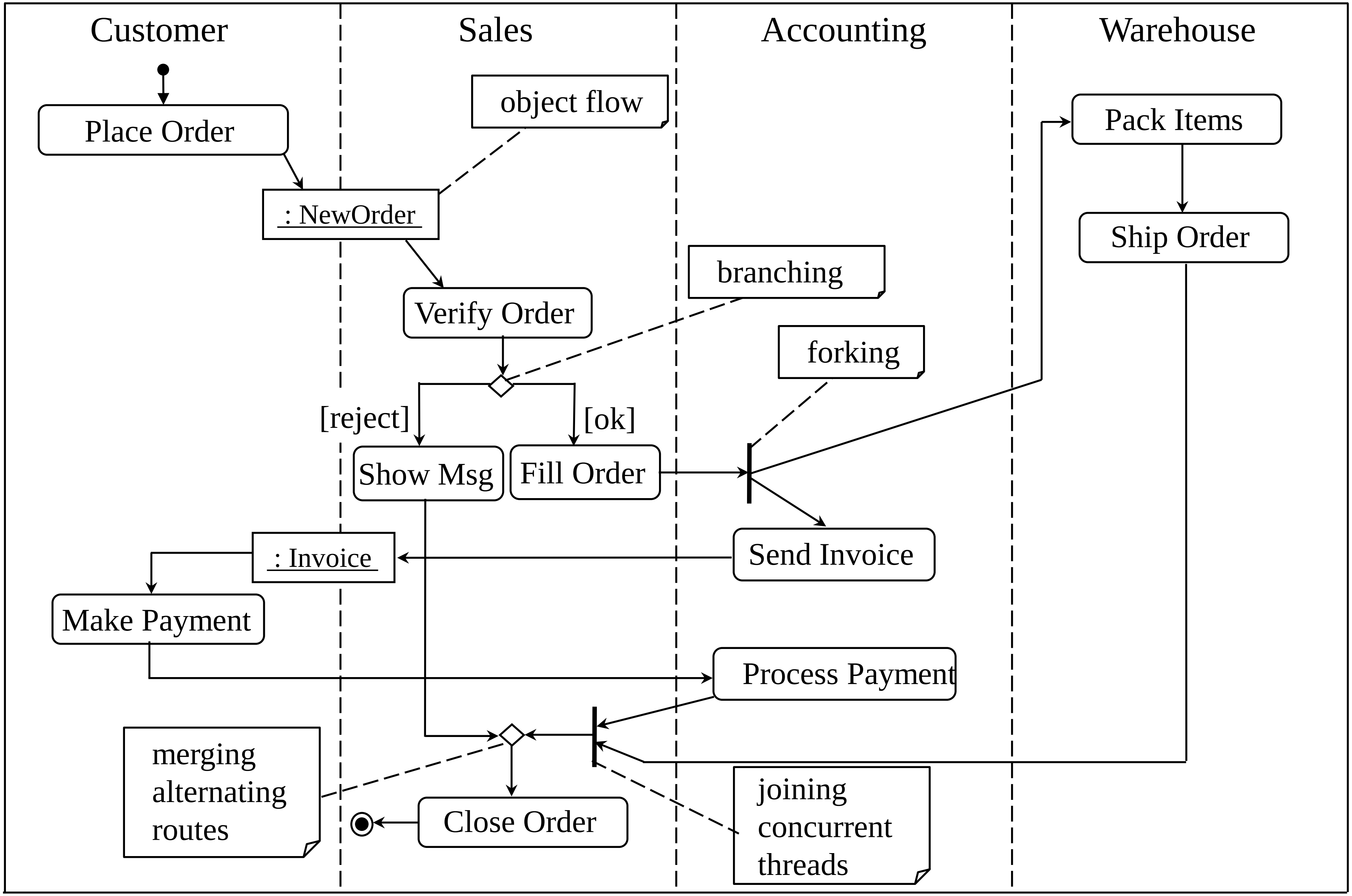
Initial node, final node, and flow final node



Box Office Order Processing

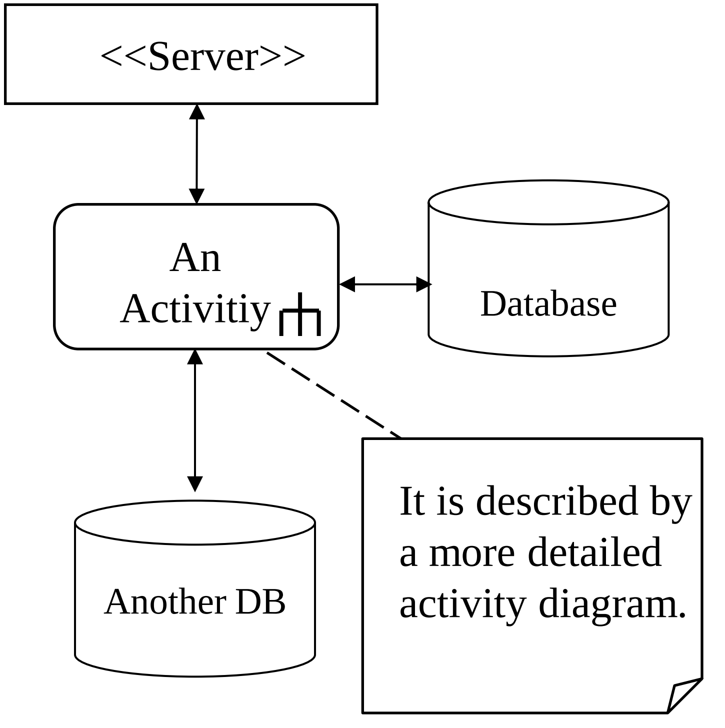


Activity Diagram: Swim Lane



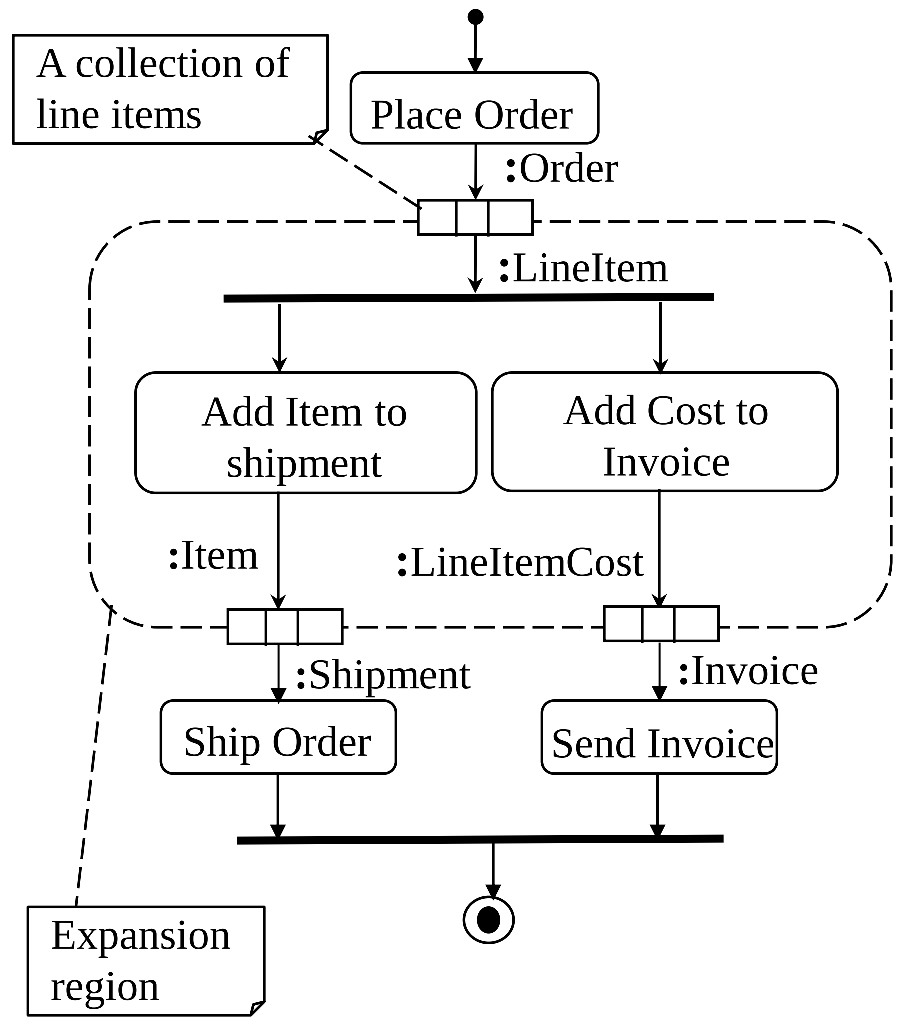
Activity Decomposition and Invocation

- A complex activity can be decomposed and represented by another activity diagram.
- The rake-style symbol is used to signify that the activity has a more detailed activity diagram.



Expansion Region

- An expansion region is a subset of activities or actions that should be repeated for each element of a collection.
- The repeated region may produce one or more collections.



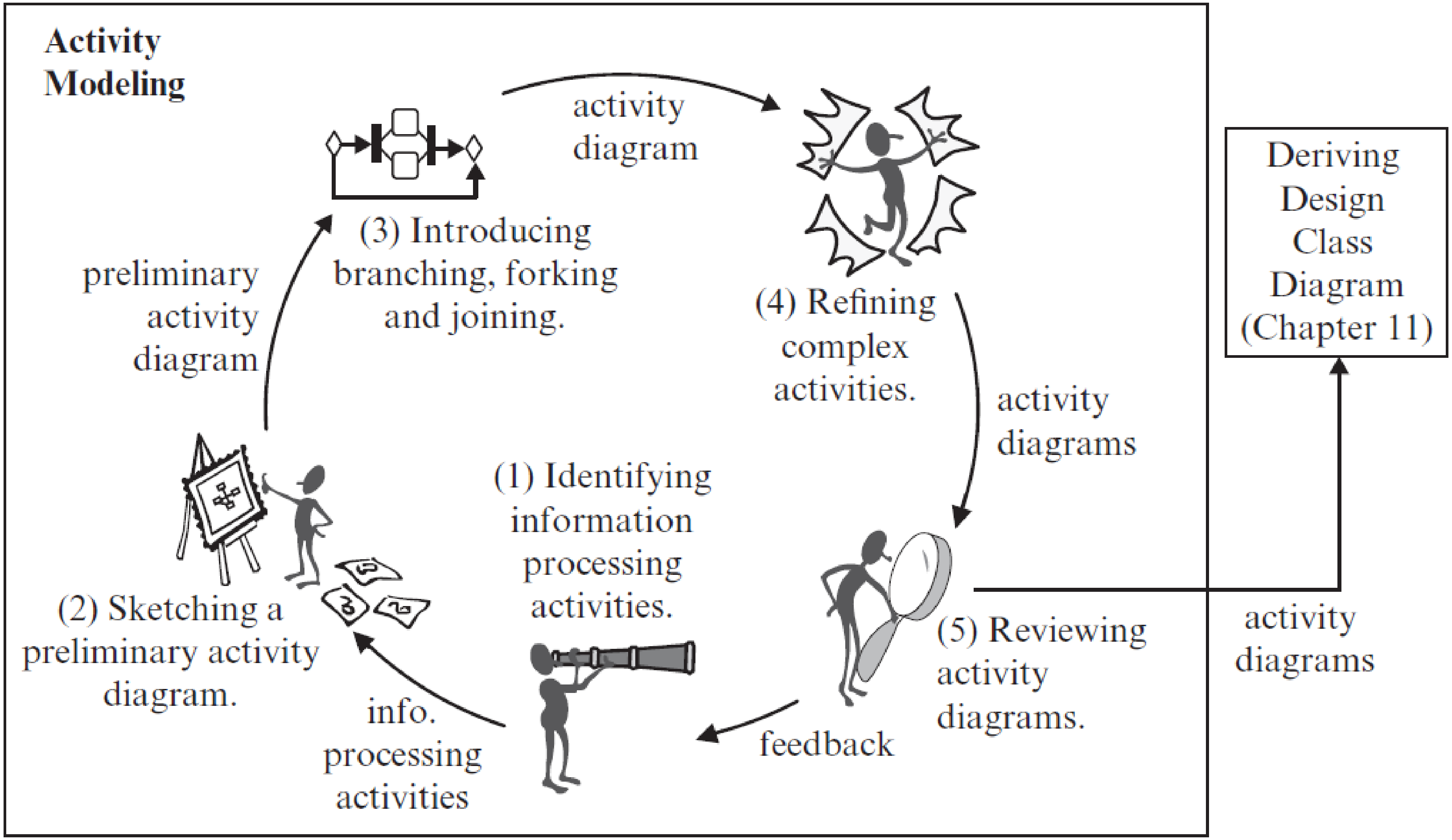
Using Activity Diagram

- Modeling, analysis and design of complex information processing activities involving one or all of the following:
  - control flows
  - object flows or data flows
  - access to databases or data depositories
  - conditional branching
  - concurrent threads
  - synchronization
- Work flow among multiple organizational units and/or subsystems.

Using Activity Diagram

- Activity diagram can be used alone or used with the other diagrams.
- Activity diagram can be used to model the information processing activity of a system, subsystem or component, or a method of a class.

Steps for Activity Modeling



Relation to Other Diagrams

- An activity may be a use case, or suggest a use case. Therefore, activity modeling is useful for identifying use cases.
- Activity diagrams are useful for showing workflows and control flows among use cases.
- Activity modeling is useful for processing complex requests in a sequence diagram.
- An activity may exhibit state-dependent behavior, which can be refined by state modeling.

Relation to Other Diagrams

- A state may represent a complex process, which can be modeled by an activity diagram.
- Each object sent from one activity to another should appear in the design class diagram (DCD), or the domain model.
- Swim lanes may suggest object classes in the domain model, or the DCD. The activities of the swim lane identify operations for the class.
- A complex activity may decompose into lower-level activities. Some of these may be operations of the class.

Class Exercise

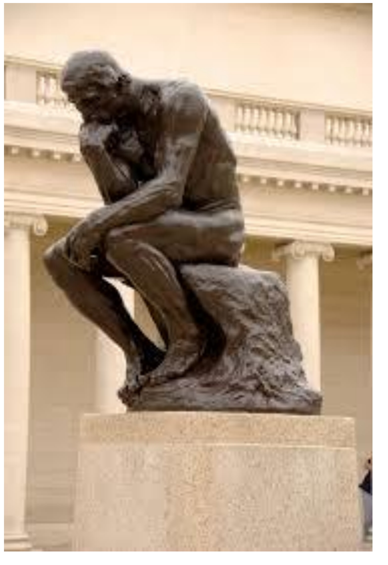
- Describe the activities for doing one of the following:
  - preparing and submitting a project proposal in your organization
  - buying a new or used car
  - workflow for configuration management
- Convert the description into an activity diagram.
- Review the diagram and identify potential problems.

Applying Agile Principles

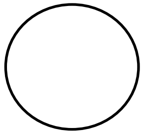
- Value working software over comprehensive documentation.
- Active user involvement is imperative.
- A collaborative and cooperative approach between all stakeholders is essential.
- Capture requirements at a high level; make them lightweight and visual.
- Do barely enough activity modeling.

For your personal edification…

…Thinking on your own!

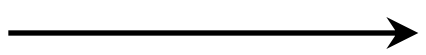


Petri Nets



places, representing an abstract condition

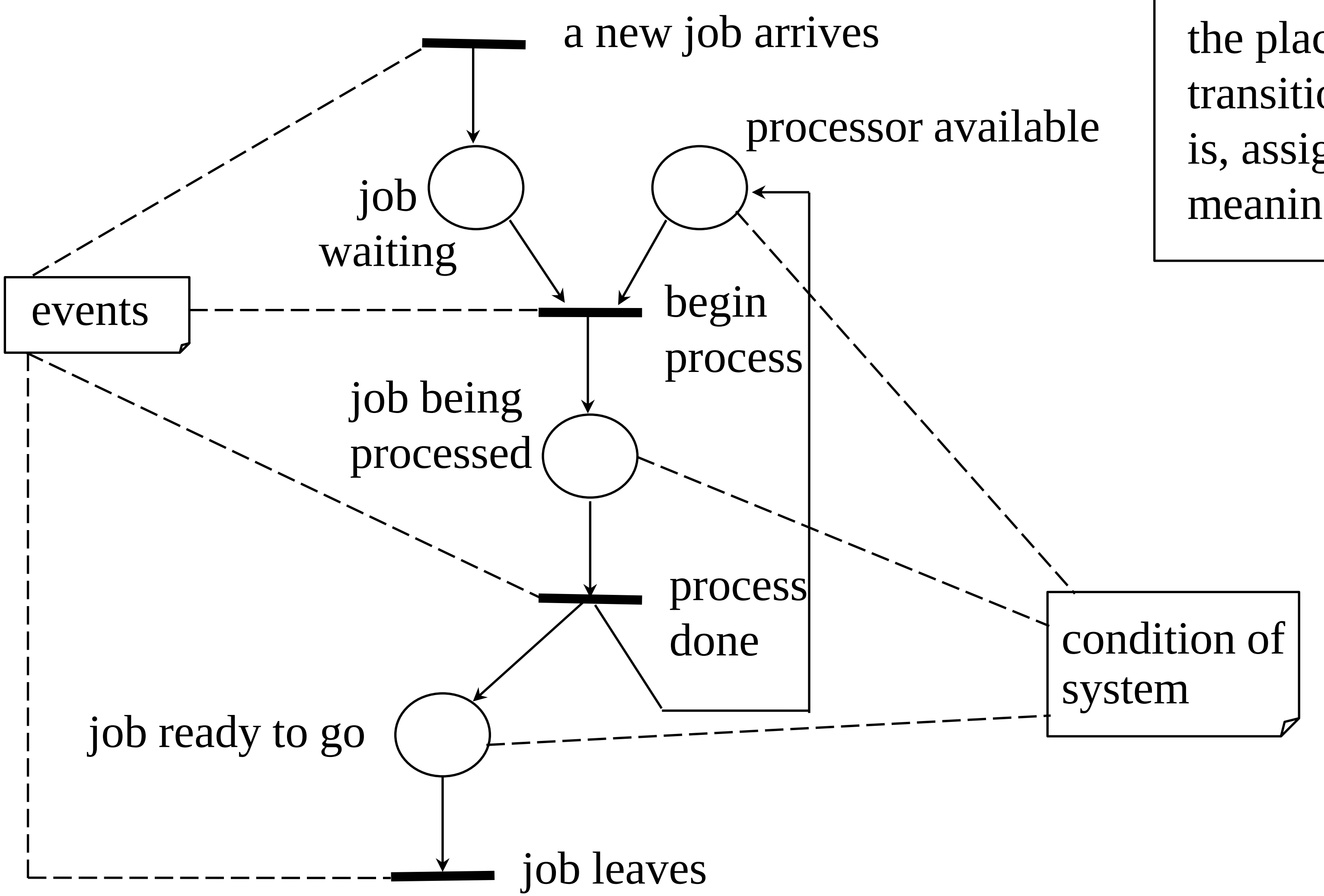
transitions, representing an event or happing



relationship between a place and a transition, it can only come from a place to a transition or from a transition to a place

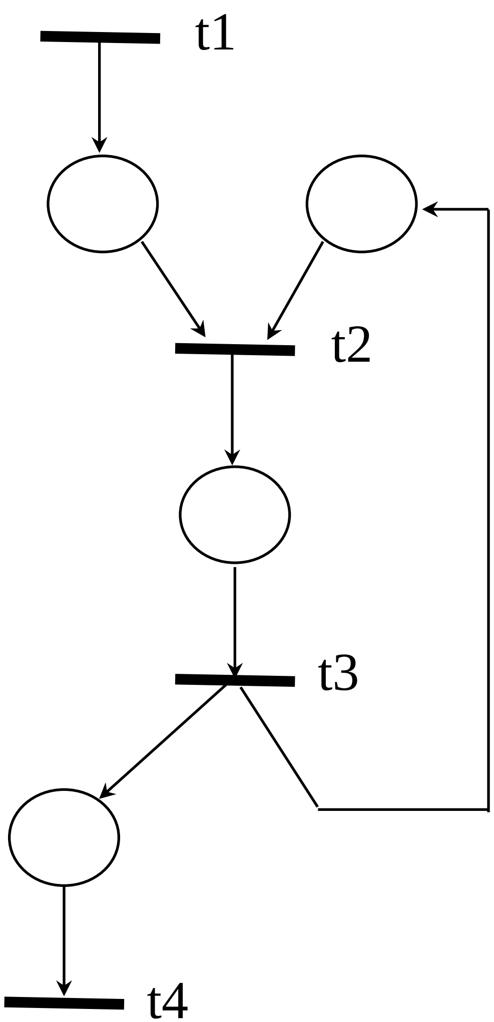
tokens, which can be placed in places to indicate that the condition is true

A Petri Net Example



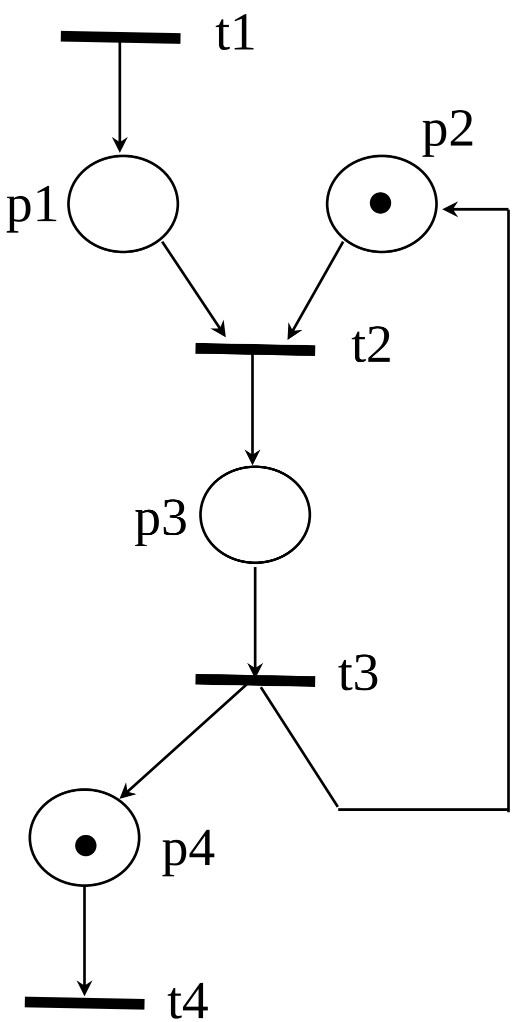
You can interpret the places and transitions. That is, assigning meanings to them.

Petri Net Execution



- A transition is enabled if and only if each of its input places contains a token.
- A transition can be fired sooner or later if it is enabled.
- Firing a transition
  - removes a token from each of its input places AND
  - places a token into each of its output places

Petri Net Marking



An initial marking is an assignment of tokens to places.

t1 is always enabled, because it does not have an input place.

firing t1 places a token in p1

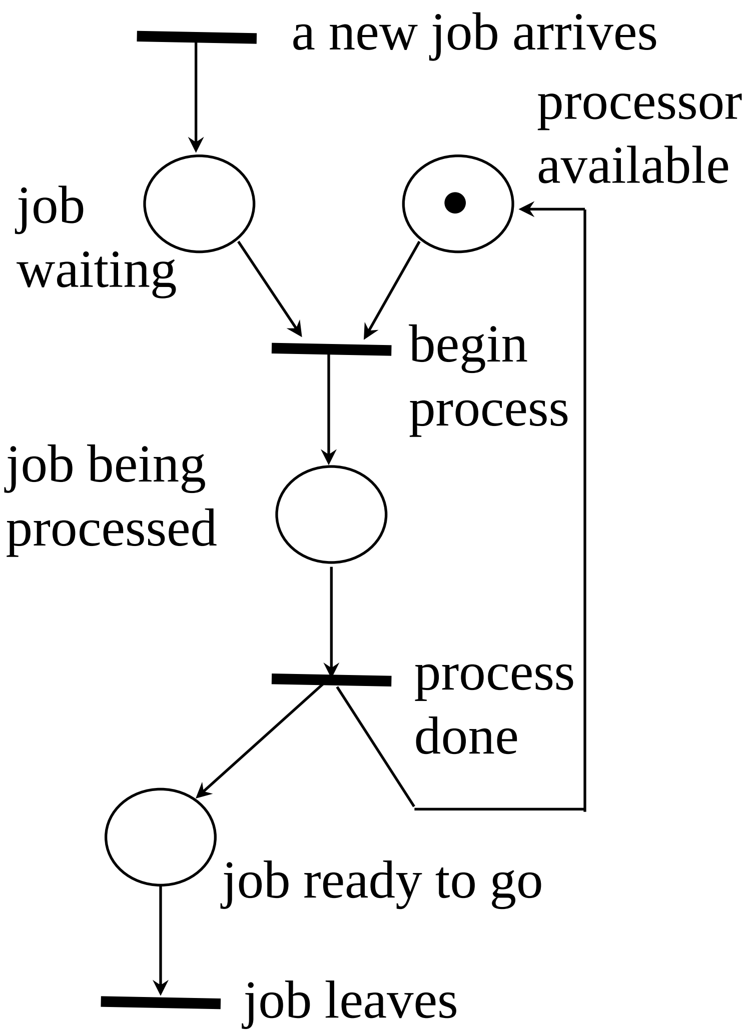
t2 is now enabled

firing t2 removes one token from each of p1 and p2 and places one token in p3

t3 is now enabled

firing t3 removes one token from p3 and places one token in p4 and p2

Petri Net Marking



"a new job arrives" is always enabled, meaning a new job can arrive anytime.

fire "a new job arrives"

"begin process" is now enabled

fire "begin process"

job is being processed

"process done" is now enabled

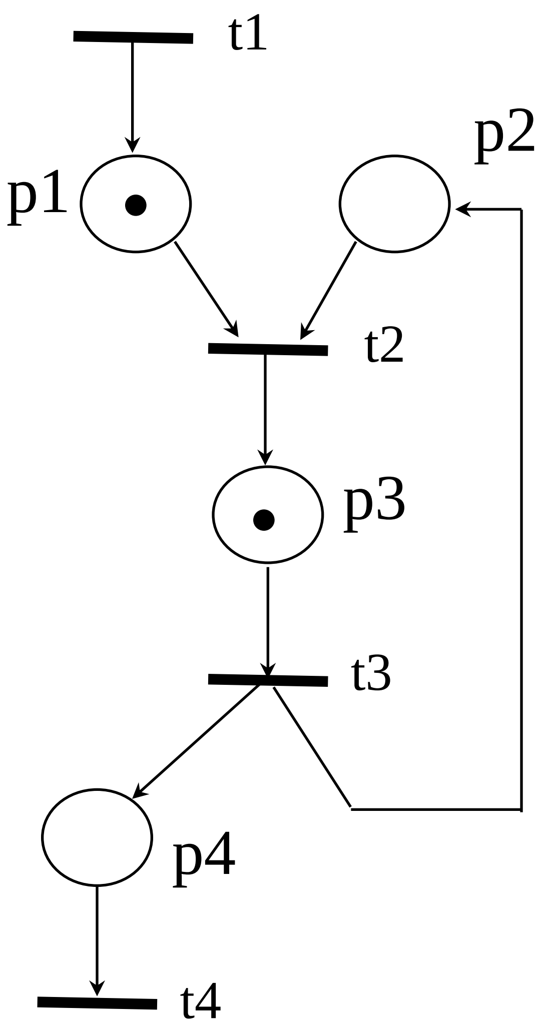
fire "process done"

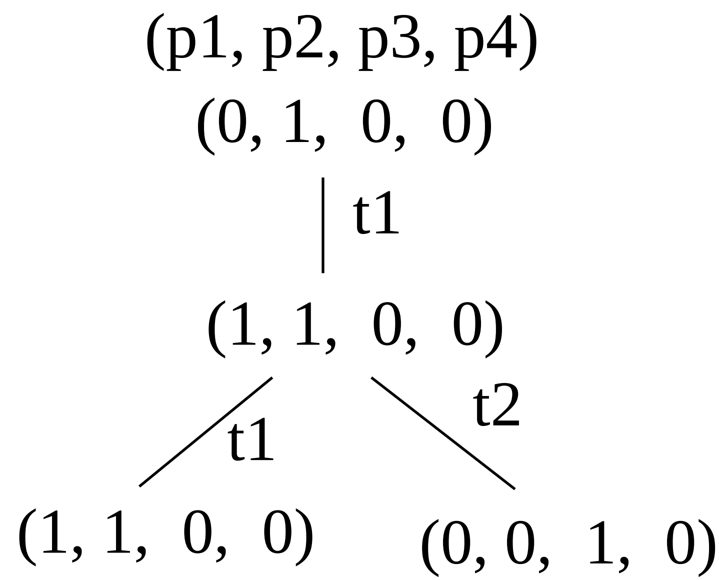
job is ready to leave & processor is available again

"job leaves" is now enabled

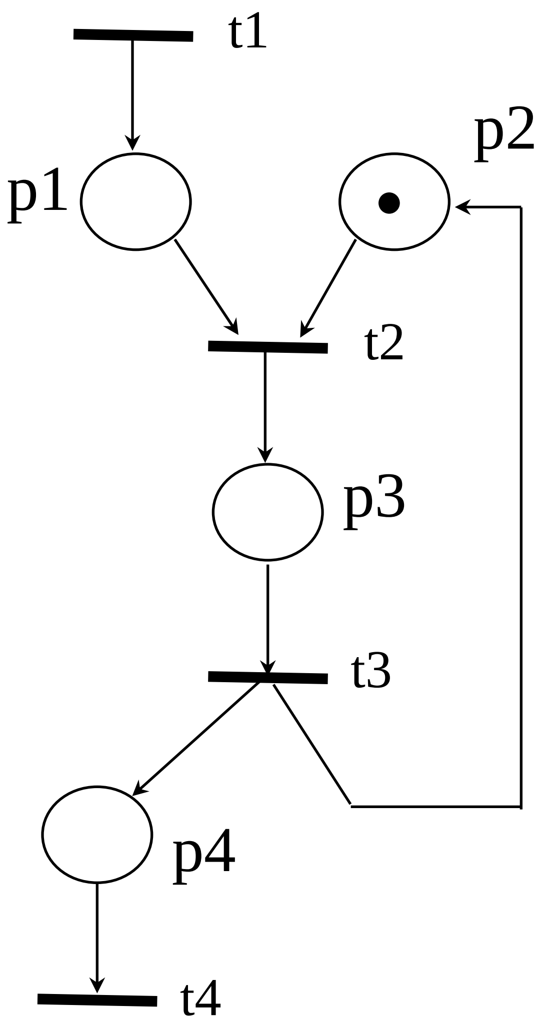
fire "job leaves"

Analysis of Petri Net





Petri Net Analysis Tree



- A marking is presented by a list of "0" and "1":

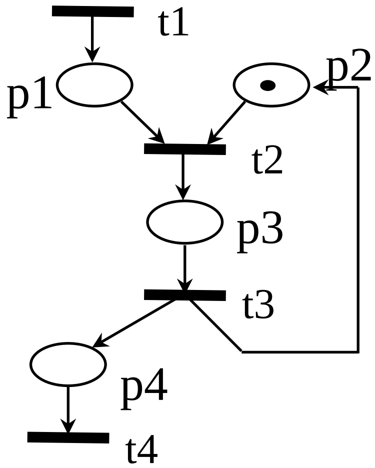
  (n1, n2, ..., nk)

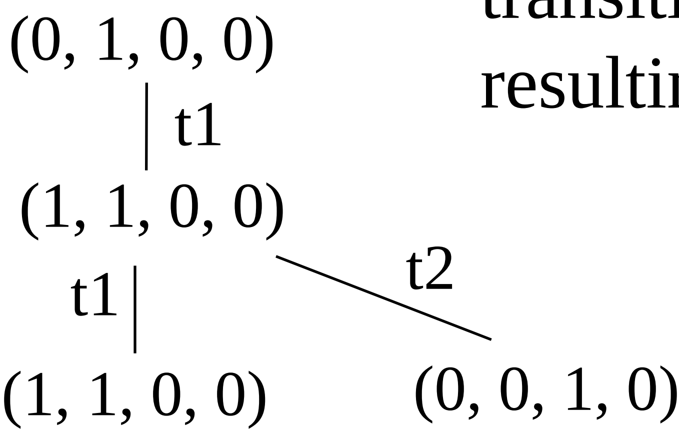
  where ni = 1 if place i contains a token
- The root of the tree denotes the initial marking (initial system state).
- Thus, the initial marking and root of tree for the Petri net on left is:

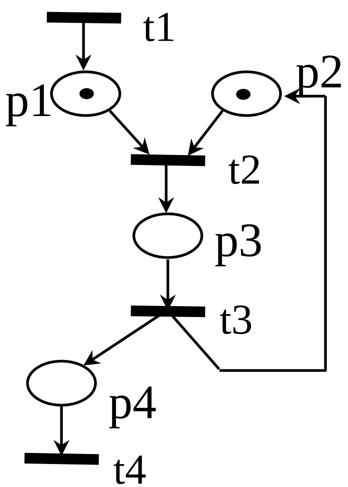
  (0, 1, 0, 0).

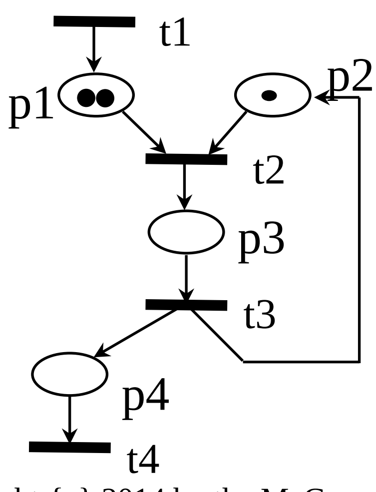
Petri Net Analysis Tree

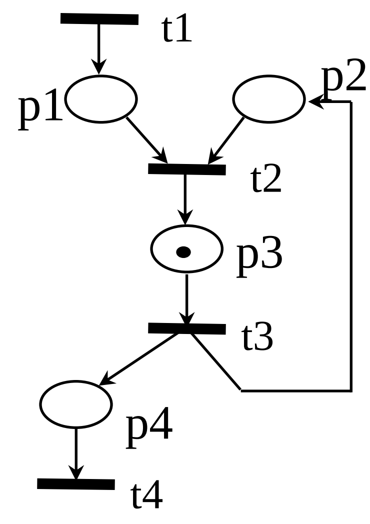
- Firing a transition grows the tree with a new branch, labeled by the transition fired and the resulting new marking.



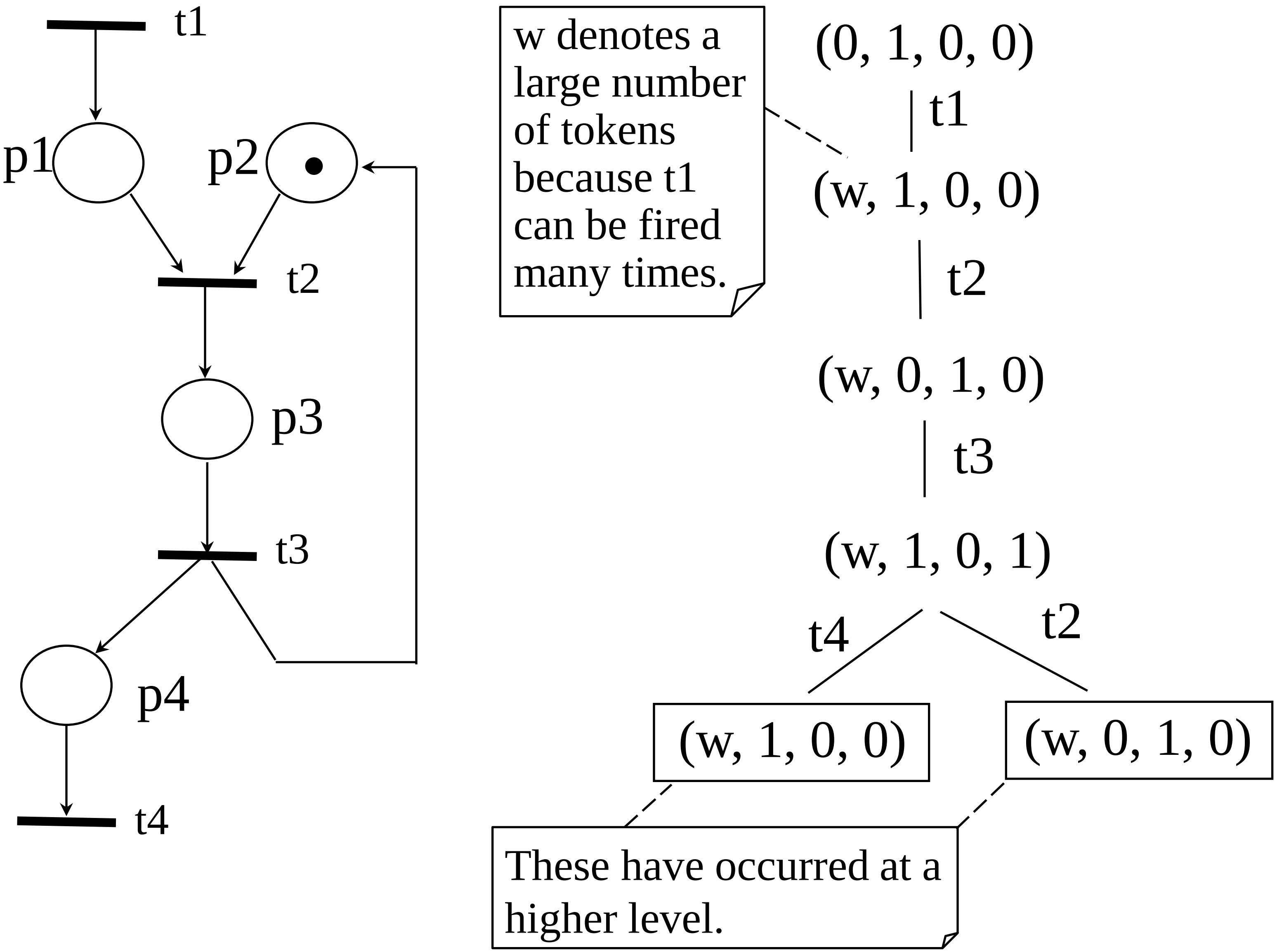




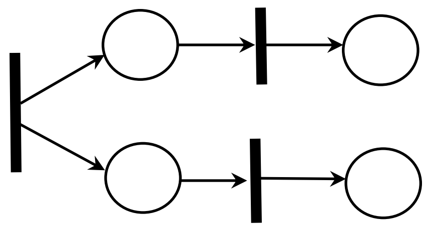




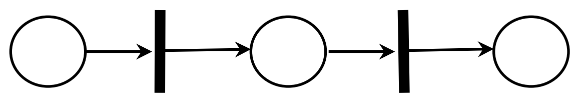
Petri Net Analysis



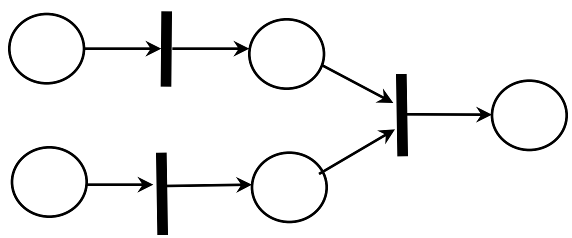
Petri Net Expressiveness



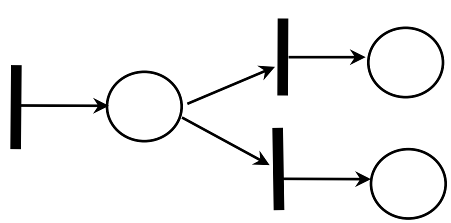
parallelism



sequencing



synchronization



exclusion