Software Engineering I
CPTS 322

LUIS DE LA TORRE

WASHINGTON STATE UNIVERSITY

Modeling with UML
841-859

Overview: modeling with UML

- What is modeling?
- What is UML?
- Use case diagrams
- Class diagrams
- Sequence diagrams
- Activity diagrams

What is modeling?

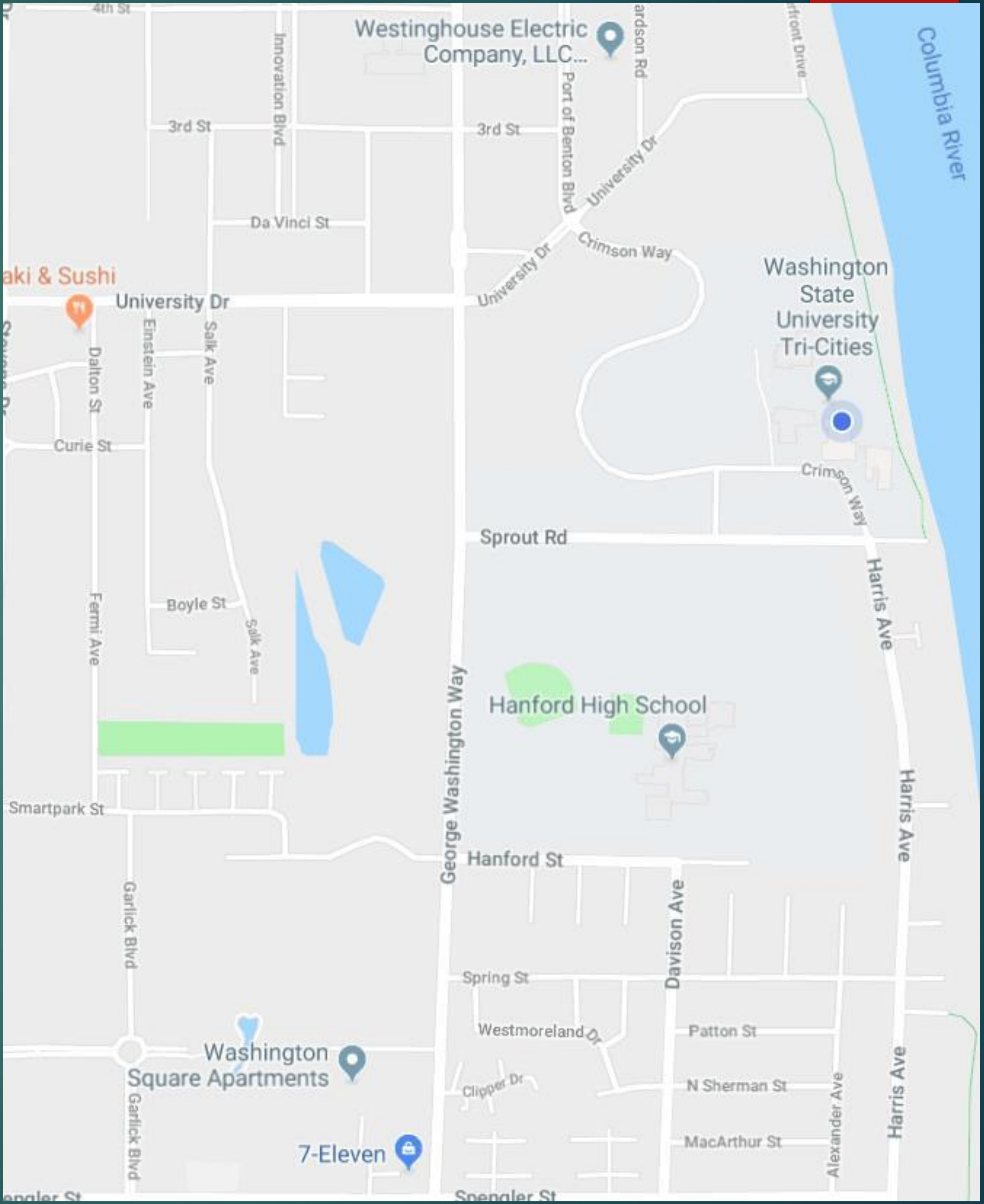
Modeling consists of building an abstraction of reality.

Abstractions are simplifications

They ignore irrelevant details and They only represent the relevant details.

What is *relevant* or *irrelevant* depends on the purpose of the model.

Example: street map



Why model software?

Why model software?

- Software is getting increasingly more complex
  - Windows XP > 40 million lines of code
  - A single programmer cannot manage this amount of code in its entirety.
- Code is not easily understandable by developers who did not write it
- We need simpler representations for complex systems
  - Modeling is a means for dealing with complexity

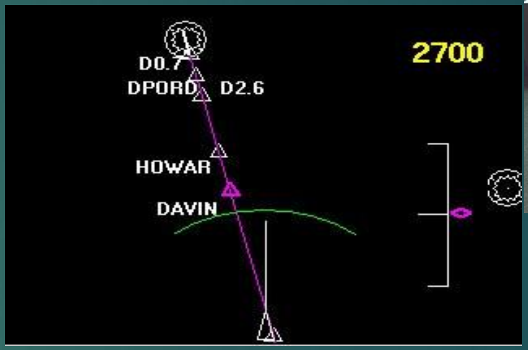
Application and Solution Domain

- Application Domain (Requirements Analysis):
  - The environment in which the system is operating

- Solution Domain (System Design, Object Design):
  - The available technologies to build the system

Object-oriented Modeling

Application Domain
(Phenomena)



Solution Domain
(Phenomena)

System Model (Concepts)*(Analysis)*

UML
Package

TrafficControl

Aircraft

TrafficController

Airport

FlightPlan

System Model (Concepts)*(Design)*

MapDisplay

Summary
Display

FlightPlanDatabase

TrafficControl

What is UML?

- UML (Unified Modeling Language)
  - Nonproprietary standard for modeling software systems, OMG
  - Convergence of notations used in object-oriented methods
    - OMT (James Rumbaugh and collegues)
    - Booch (Grady Booch)
    - OOSE (Ivar Jacobson)
- Current Version: UML 2.2
  - Information at the OMG portal http://www.uml.org/
- Commercial tools: Rational (IBM),Together (Borland), Visual Architect (business processes, BCD)
- Open Source tools: ArgoUML, StarUML, Umbrello
- Commercial and Opensource: PoseidonUML (Gentleware)

UML: First Pass

- You can model 80% of most problems by using about 20% UML
- We teach you those 20%

- 80-20 rule: Pareto principle (http://en.wikipedia.org/wiki/Pareto_principle)
  - 80% of your profits come from 20% of your customers
  - 80% of your complaints come from 20% of your customers
  - 80% of your profits come from 20% of the time you spend
  - 80% of your sales come from 20% of your products

UML First Pass

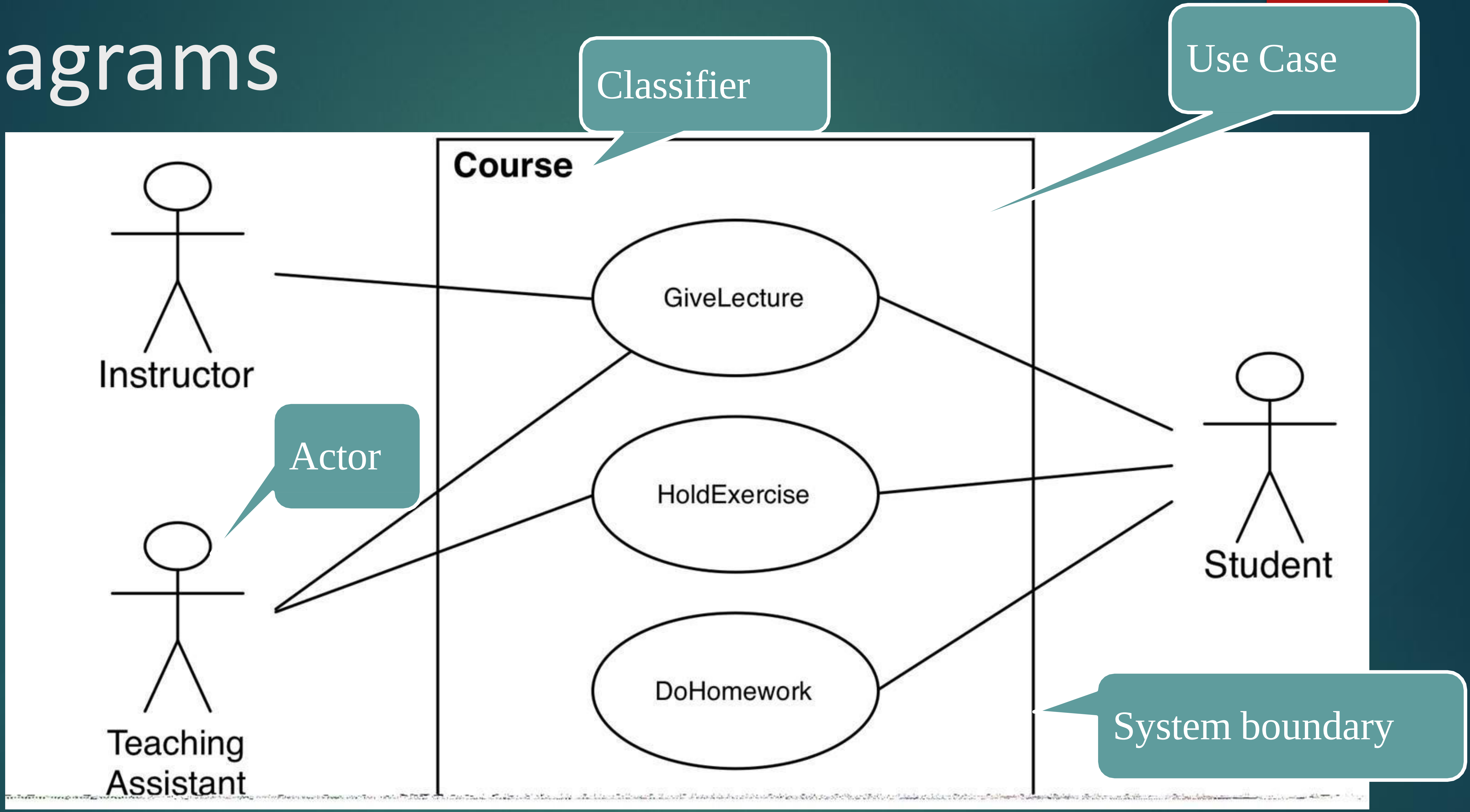
- Use case diagrams
  - Describe the functional behavior of the system as seen by the user
- Class diagrams
  - Describe the static structure of the system: Objects, attributes, associations
- Sequence diagrams
  - Describe the dynamic behavior between objects of the system
- State diagrams
  - Describe the dynamic behavior of an individual object

UML Core Conventions

- All UML Diagrams denote graphs of nodes and edges
  - Nodes are entities and drawn as rectangles or ovals
- Rectangles denote classes or instances
- Ovals denote functions

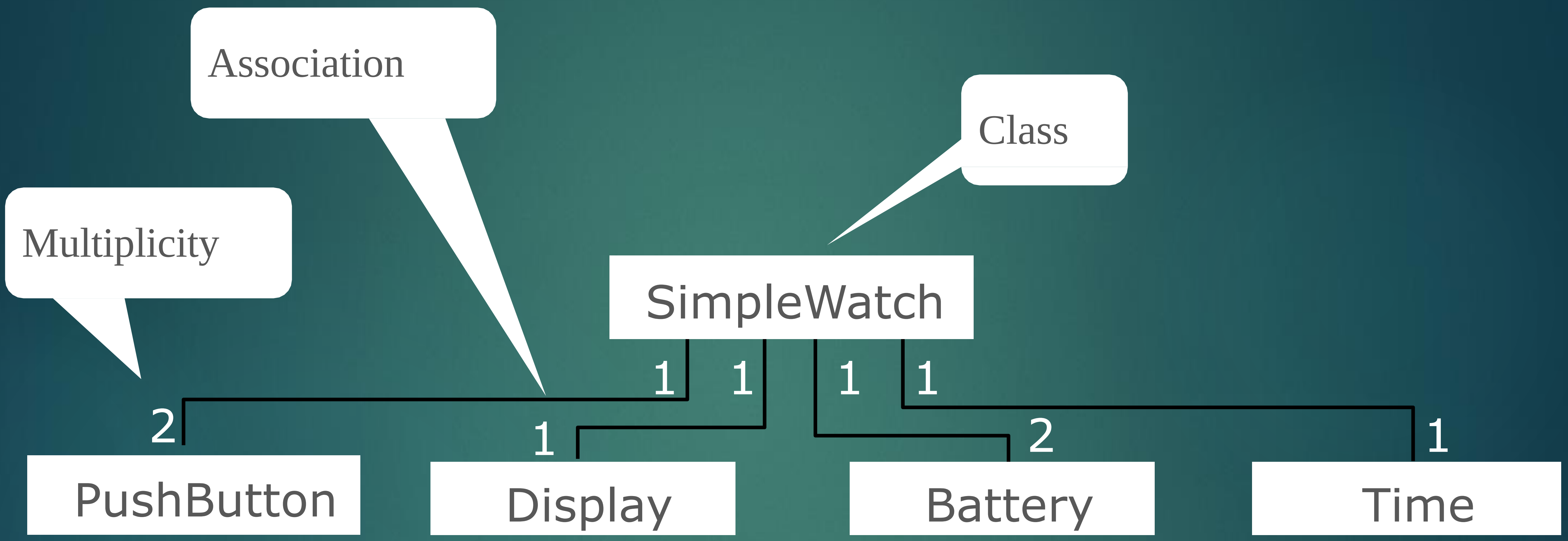
- Names of Classes are not underlined
  - SimpleWatch
  - Firefighter
- Names of Instances are underlined
  - myWatch:SimpleWatch
  - Joe:Firefighter
- An edge between two nodes denotes a relationship between the corresponding entities

UML first pass: Use case diagrams



Use case diagrams represent the functionality of the system from user's point of view

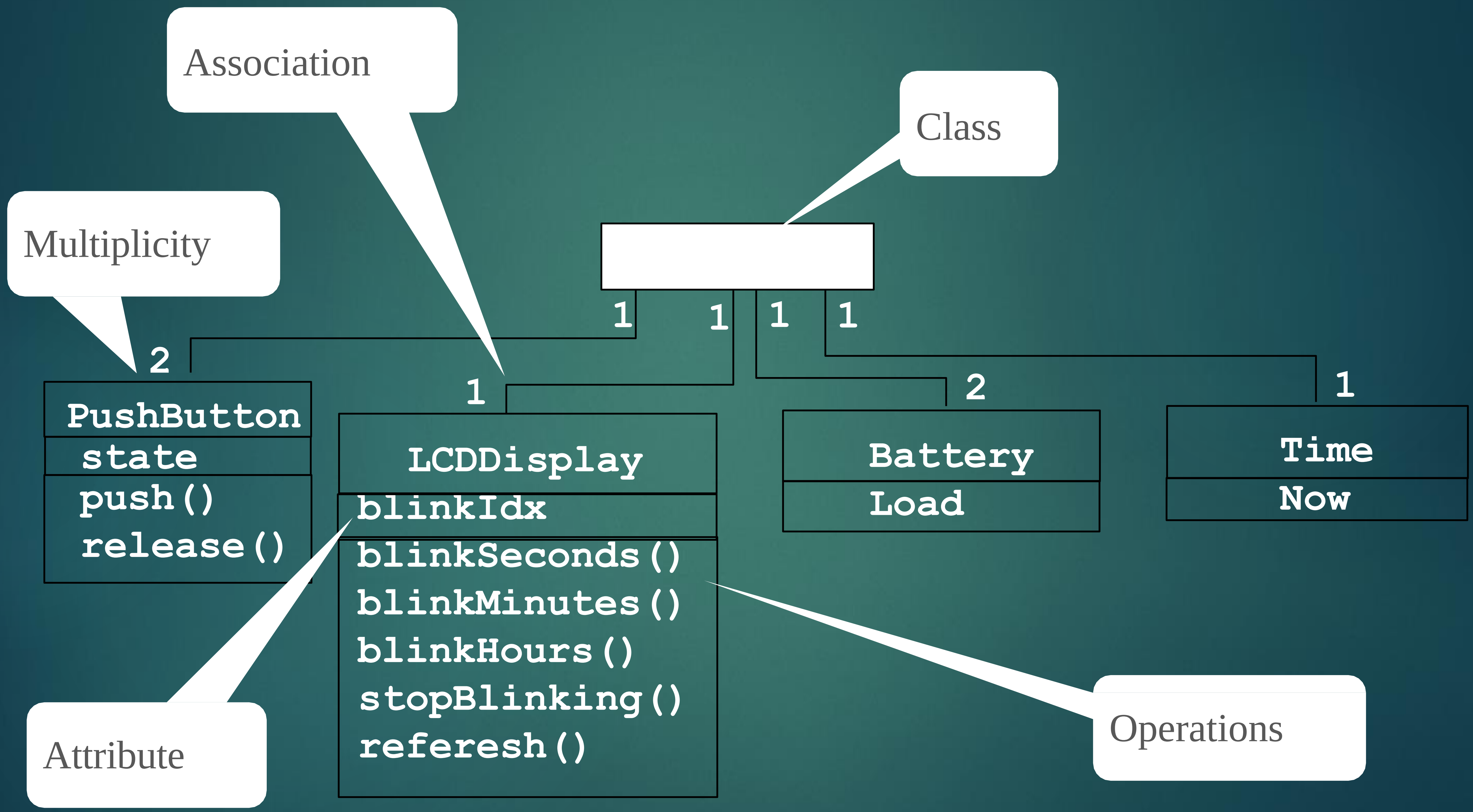
UML first pass: Class diagrams



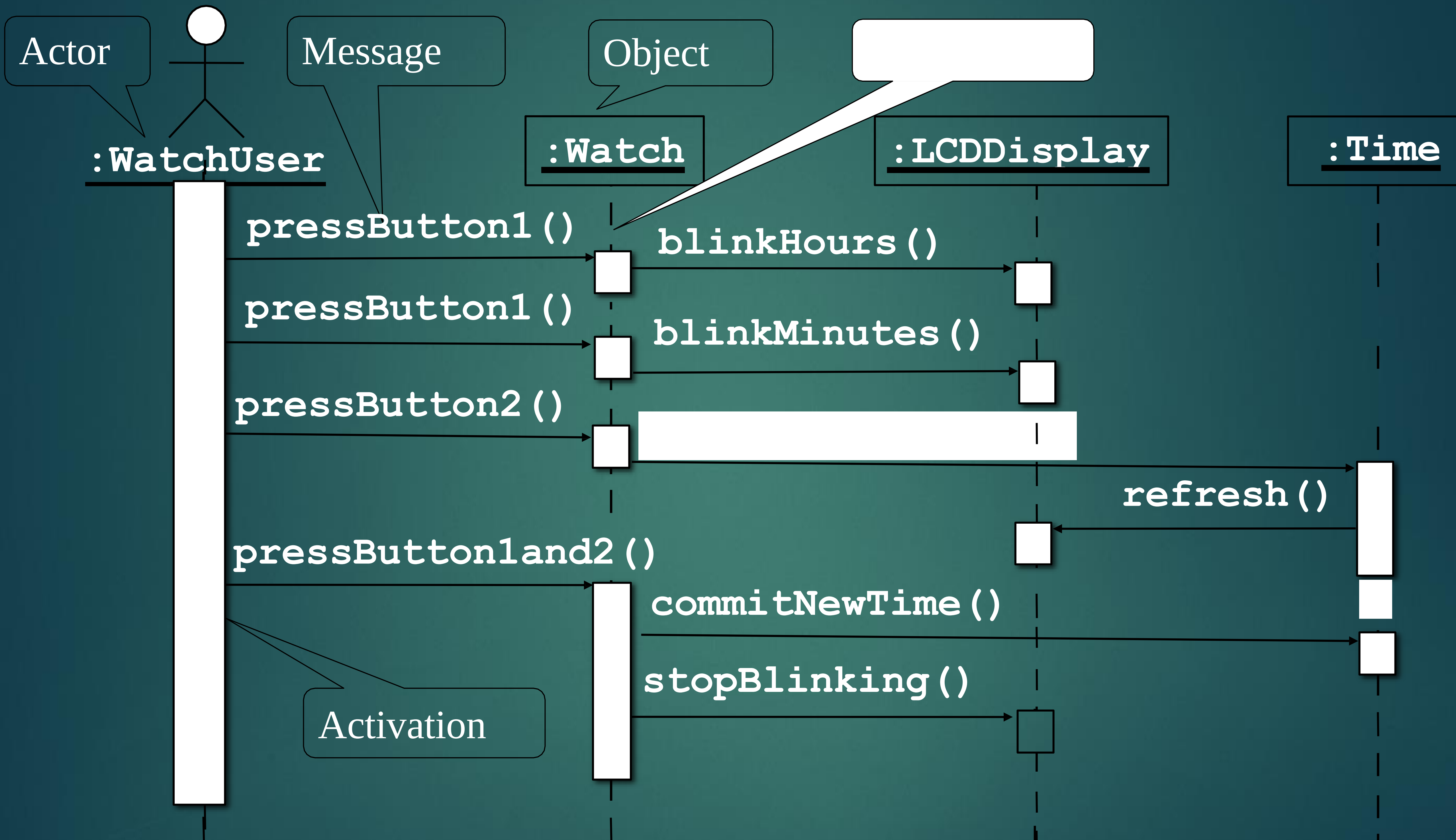
Class diagrams represent the structure of the system

UML first pass: Class diagrams

Class diagrams represent the structure of the system

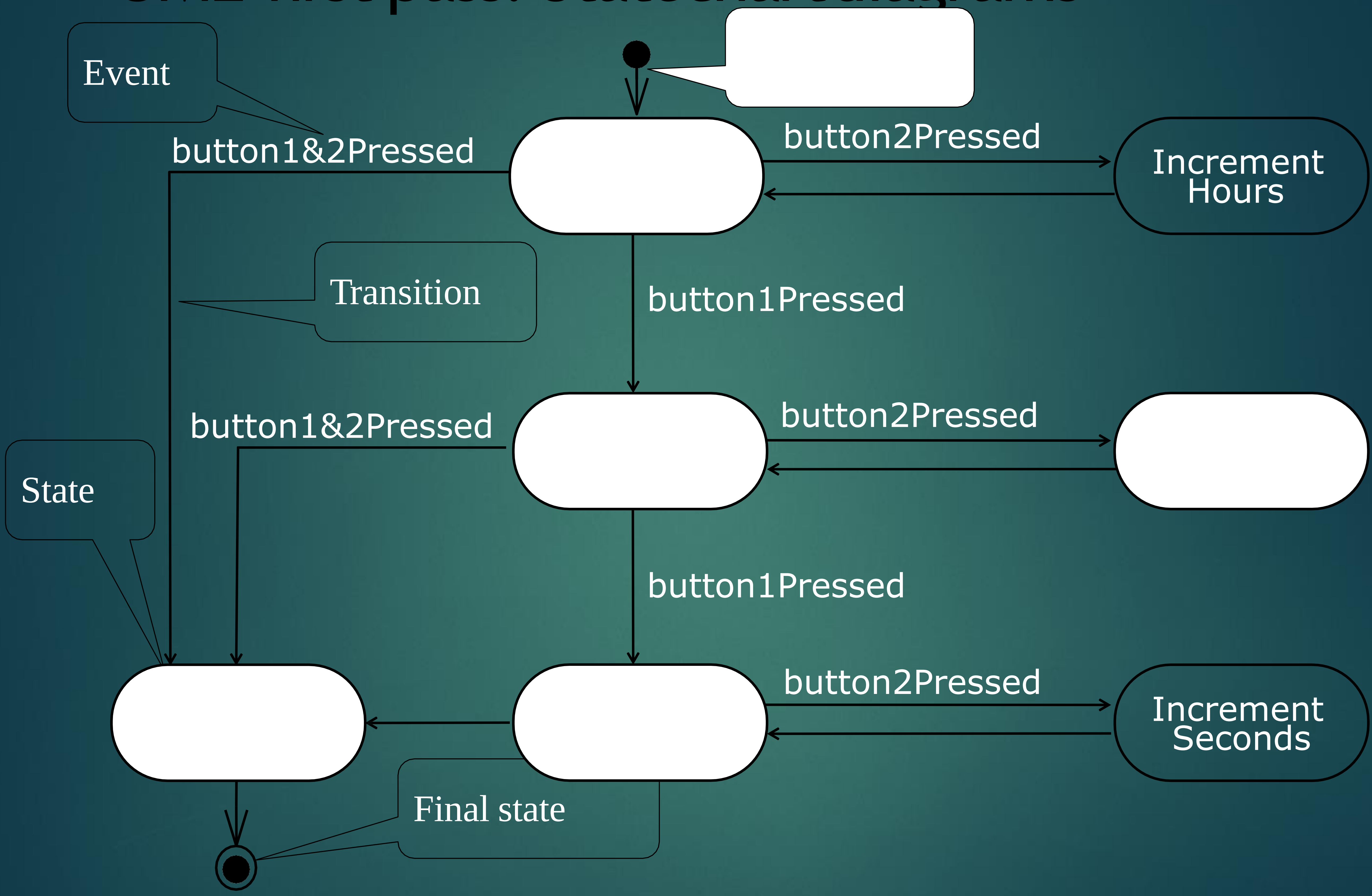


UML first pass: Sequence diagram



Sequence diagrams represent the behavior of a system as messages ("interactions") between *different objects*

UML first pass: Statechart diagrams



Represent behavior of *a single object* with interesting dynamic behavior.

Other UML Notations

UML provides many other notations, for example

- **Deployment diagrams** for modeling configurations
  - Useful for testing and for release management
- We introduce these and other notations as we go along in the lectures
  - OCL: A language for constraining UML models

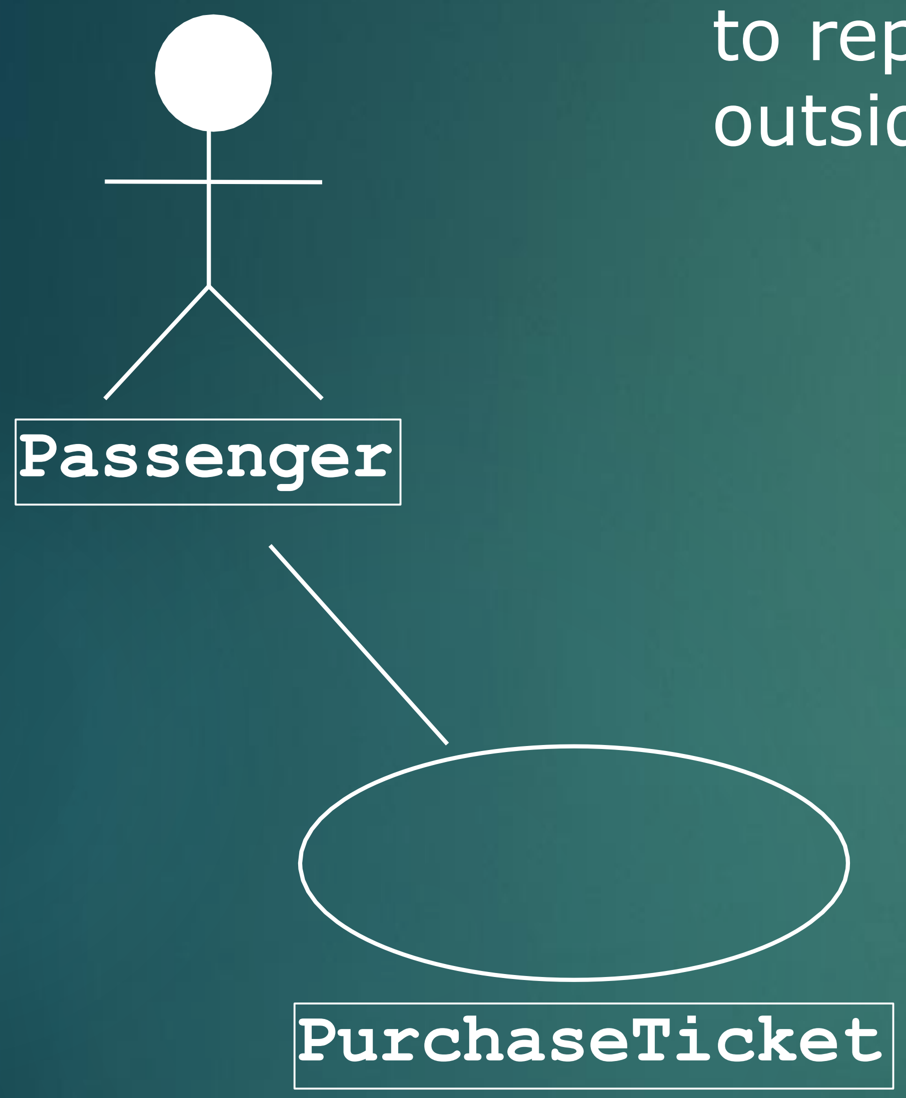
What should be done first? Coding or Modeling?

- It all depends….
- Forward Engineering
  - Creation of code from a model
  - Start with modeling
  - Greenfield projects
- Reverse Engineering
  - Creation of a model from existing code
  - Interface or reengineering projects
- Roundtrip Engineering
  - Move constantly between forward and reverse engineering
  - Reengineering projects
  - Useful when requirements, technology and schedule are changing frequently.

UML Second Pass

- Use case diagrams
  - Describe the functional behavior of the system as seen by the user
- Class diagrams
  - Describe the static structure of the system: Objects, attributes, associations
- Sequence diagrams
  - Describe the dynamic behavior between objects of the system
- **State diagrams**
  - Describe the dynamic behavior of an individual object
- **Activity diagrams**
  - Describe the dynamic behavior of a system, in particular the workflow.

UML Use Case Diagrams



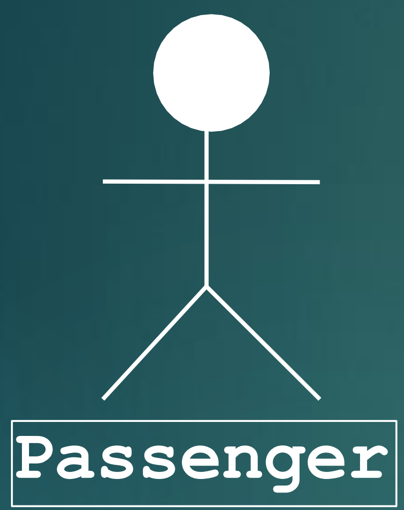
Used during requirements elicitation and analysis to represent external behavior ("visible from the outside of the system")

An ***Actor*** represents a role, that is, a type of user of the system

A ***use case*** represents a class of functionality provided by the system

Use case model:
The set of all use cases that completely describe the functionality of the system.

Actors



- An **actor** is a model for an external entity which interacts (communicates) with the system:
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)
- An actor has a unique name and an optional description
- Examples:
  - Passenger: A person in the train
  - GPS satellite: An external system that provides the system with GPS coordinates.

Use Case

- A use case represents a class of functionality provided by the system
- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of 6 parts:
    1. Unique name
    2. Participating actors
    3. Entry conditions
    4. Exit conditions
    5. Flow of events
    6. Special requirements.

Textual Use Case Description Example

1. *Name:* `Purchase ticket`

2. *Participating actor:* `Passenger`

3. *Entry condition:*

- `Passenger` stands in front of ticket distributor
- `Passenger` has sufficient money to purchase ticket

4. *Exit condition:*

- `Passenger` has ticket

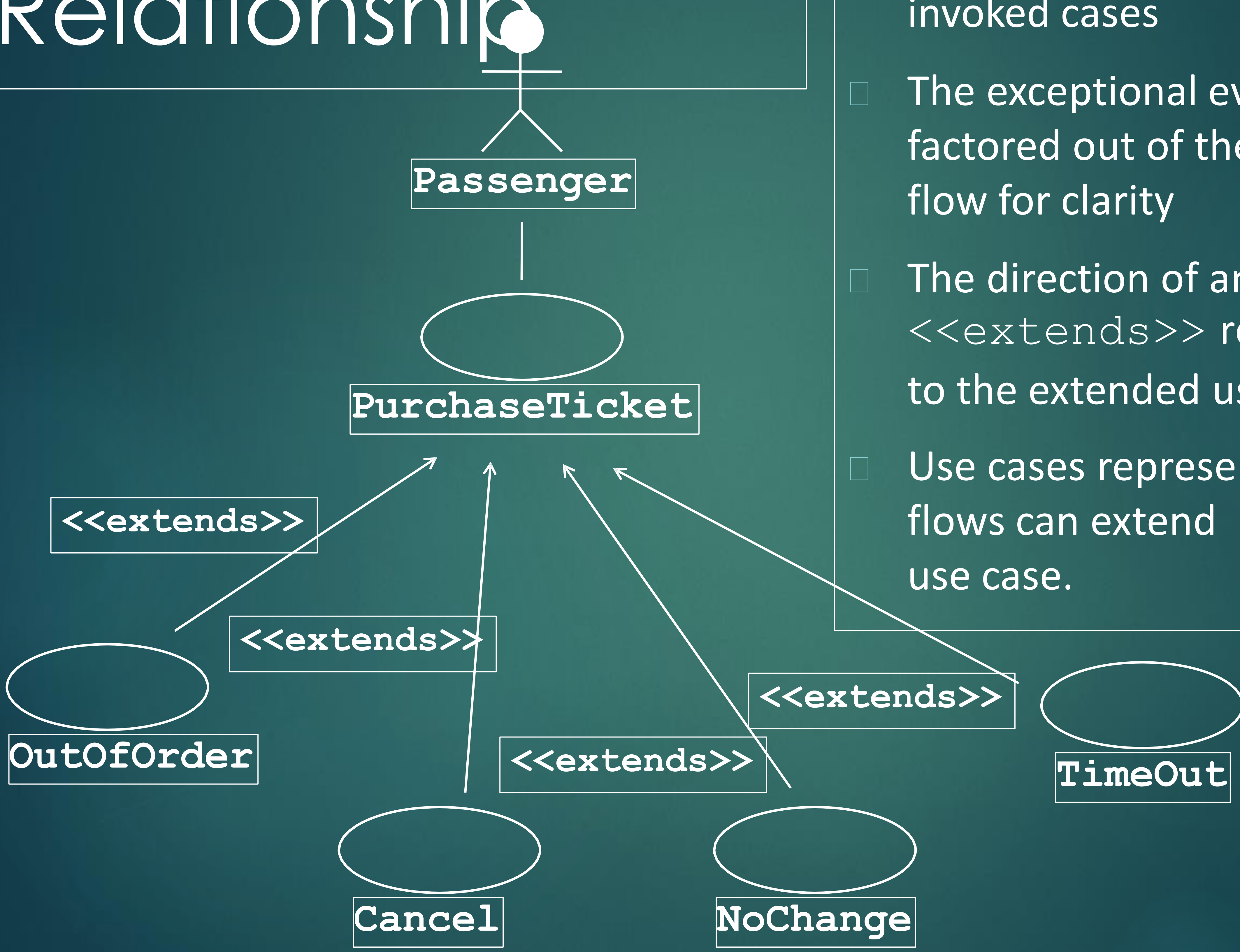
5. *Flow of events:*
    1. `Passenger` selects the number of zones to be traveled
    2. Ticket Distributor displays the amount due
    3. `Passenger` inserts money, at least the amount due
    4. Ticket Distributor returns change
    5. Ticket Distributor issues ticket

6. *Special requirements: None.*

Uses Cases can be related

- ExtendsRelationship
  - To represent seldom invoked use cases or exceptional functionality
- IncludesRelationship
  - To represent functional behavior common to more than one use case.

The <<extends>> Relationship



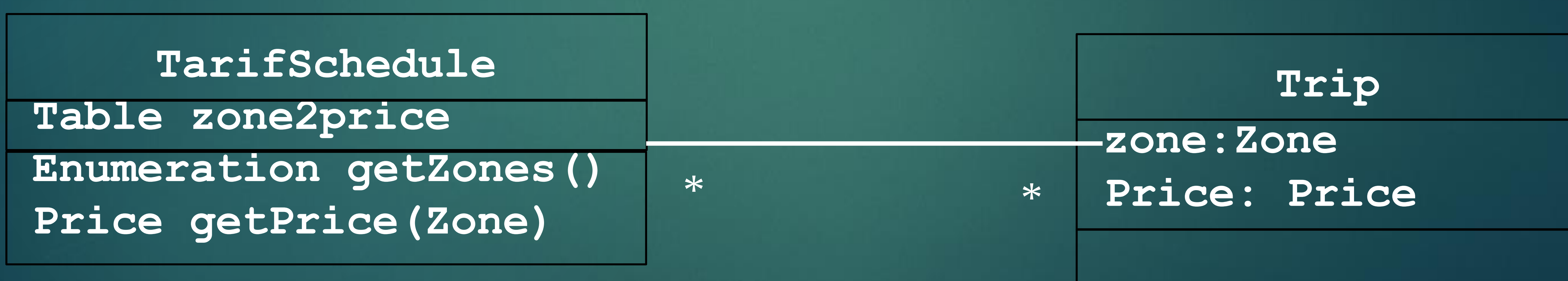
- `<<extends>>` relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an `<<extends>>` relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.

The <<includes>> Relationship

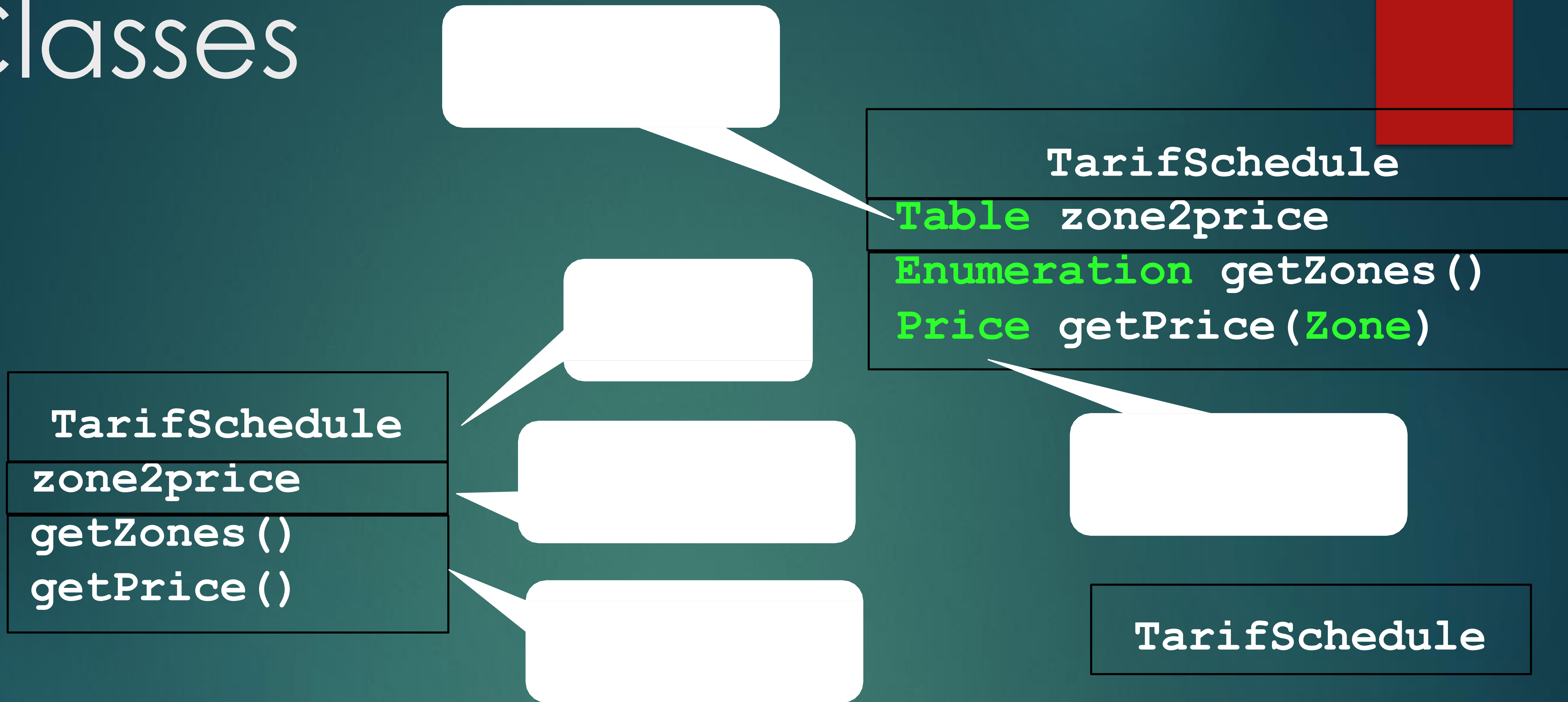
- <<includes>> relationship represents common functionality needed in more than one use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

Class Diagrams

- Class diagrams represent the structure of the system
- Used
  - during requirements analysis to model application domain concepts
  - during system design to model subsystems
  - during object design to specify the detailed behavior and attributes of classes.



Classes



- A ***class*** represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations)***

Each attribute has a ***type***
Each operation has a ***signature***

The class name is the only mandatory information

Instances

tarif2006:TarifSchedule

```
zone2price = {
{‘1’, 0.20},
{‘2’, 0.40},
{‘3’, 0.60}}
```

:TarifSchedule

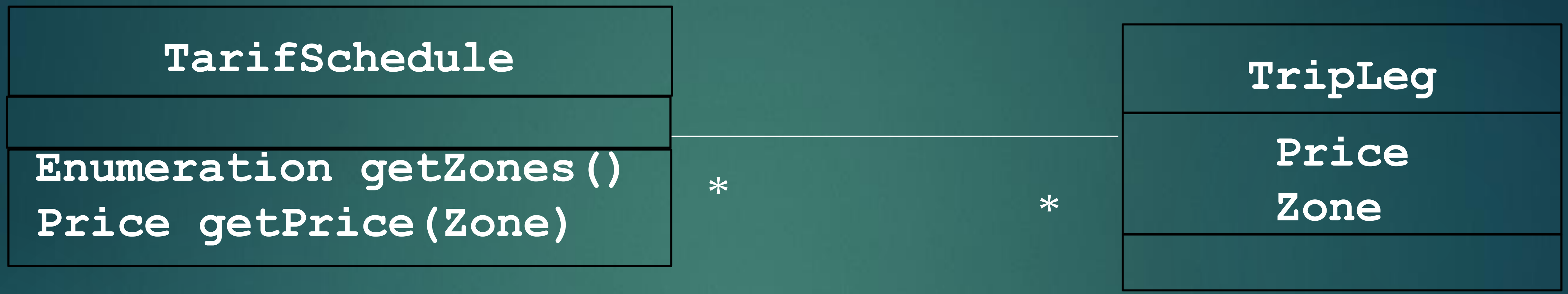
```
zone2price = {
{‘1’, 0.20},
{‘2’, 0.40},
{‘3’, 0.60}}
```

- An ***instance*** represents a phenomenon
- The attributes are represented with their ***values***
- The name of an instance is underlined
- The name can contain only the class name of the instance (anonymous instance)

Actor vs Class vs Object

- **Actor**
  - An entity outside the system to be modeled, interacting with the system (“Passenger”)
- **Class**
  - An abstraction modeling an entity in the application or solution domain
  - The class is part of the system model (“User”, “Ticket distributor”, “Server”)
- **Object**
  - A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

Associations



Associations denote relationships between classes

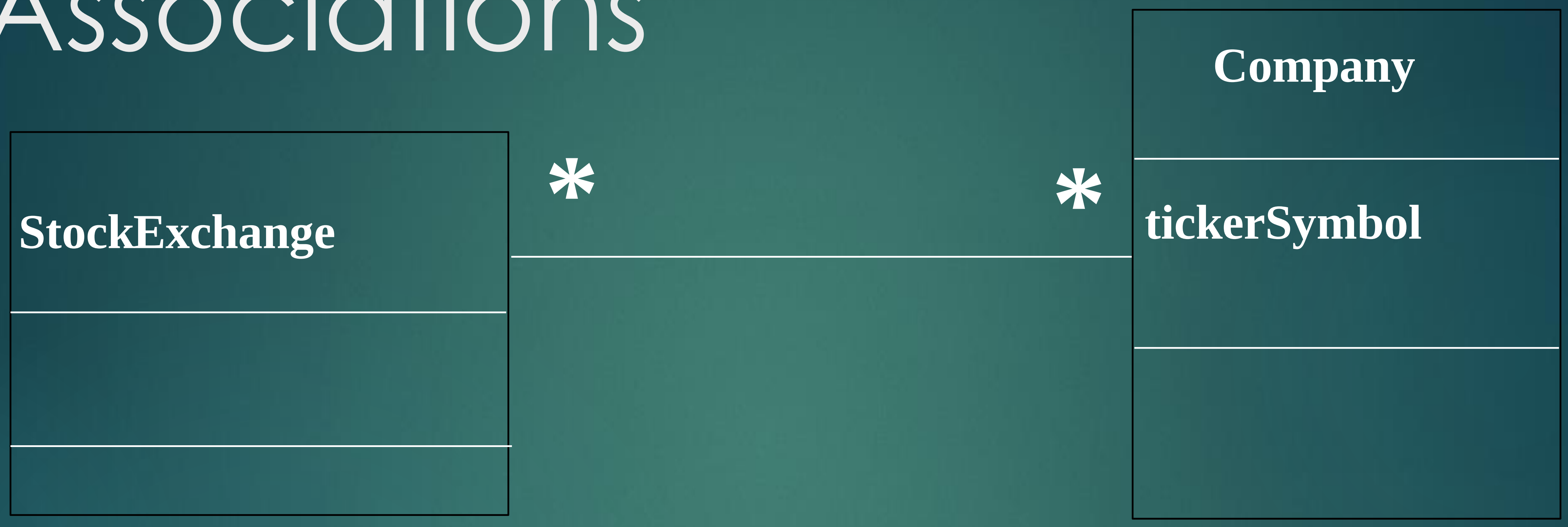
The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

1-to-1 and 1-to-many Associations

1-to-1 association

1-to-many association

Many-to-Many Associations



From Problem Statement To Object Model

Problem Statement: A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol

Class Diagram:

From Problem Statement to Code

Problem Statement : A stock exchange lists many companies. Each company is identified by a ticker symbol

Class Diagram:

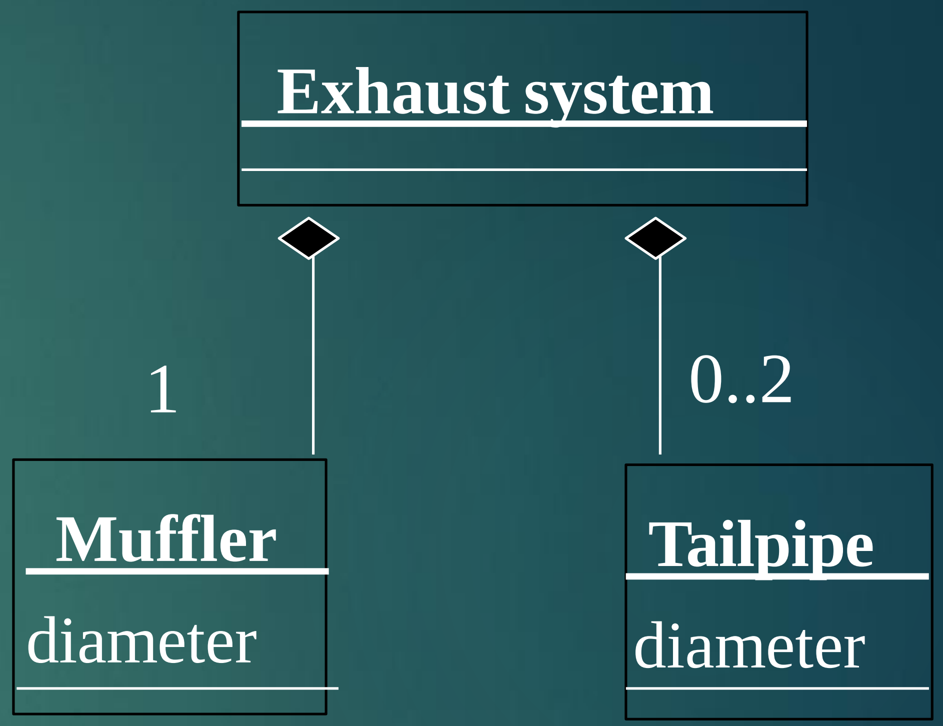
Java Code

```
public class StockExchange
{
 private Vector m_Company = new Vector();
};
public class Company
{
 public int m_tickerSymbol;
 private Vector m_StockExchange = new Vector();
};
```

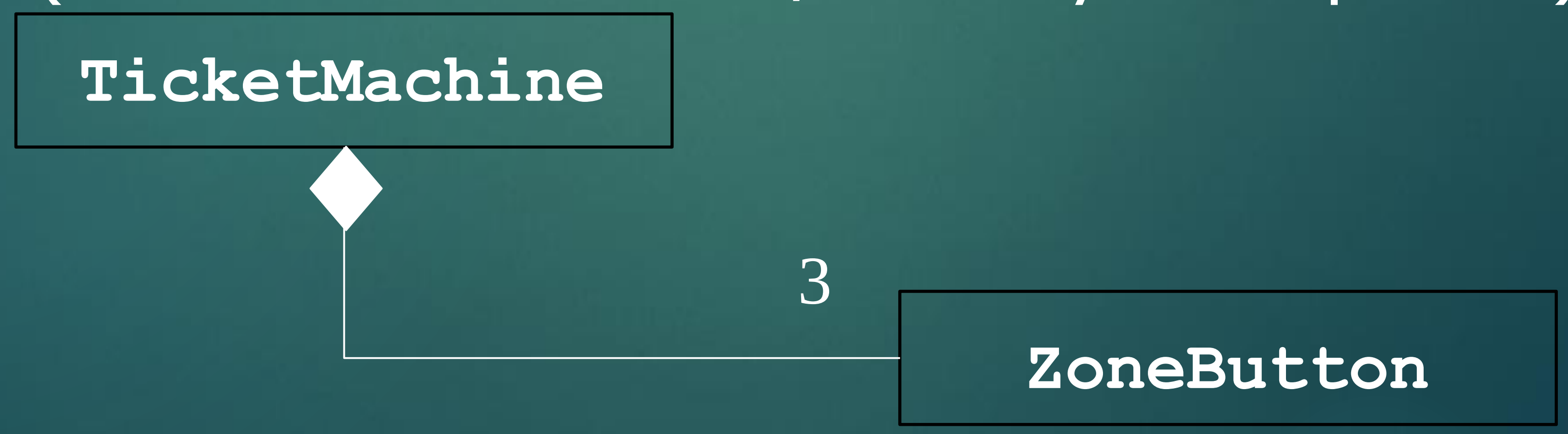
Associations are mapped to Attributes!

Aggregation

- An *aggregation* is a special case of association denoting a “consists-of” hierarchy
- The *aggregate* is the parent class, the components are the children classes



A solid diamond denotes *composition*: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, the parts don’t exist on their won (“the whole controls/destroys the parts”)



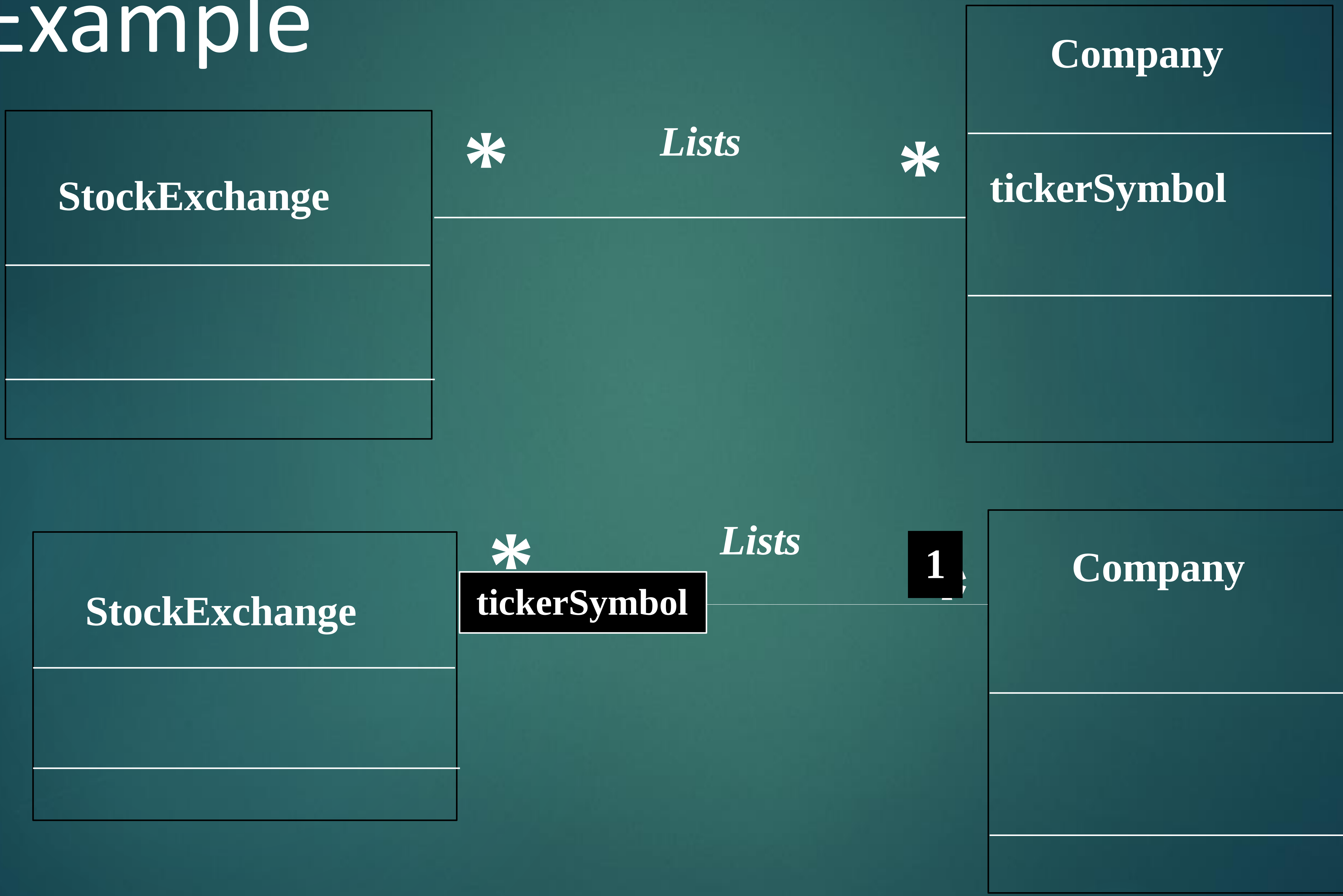
Qualifiers

Without qualification

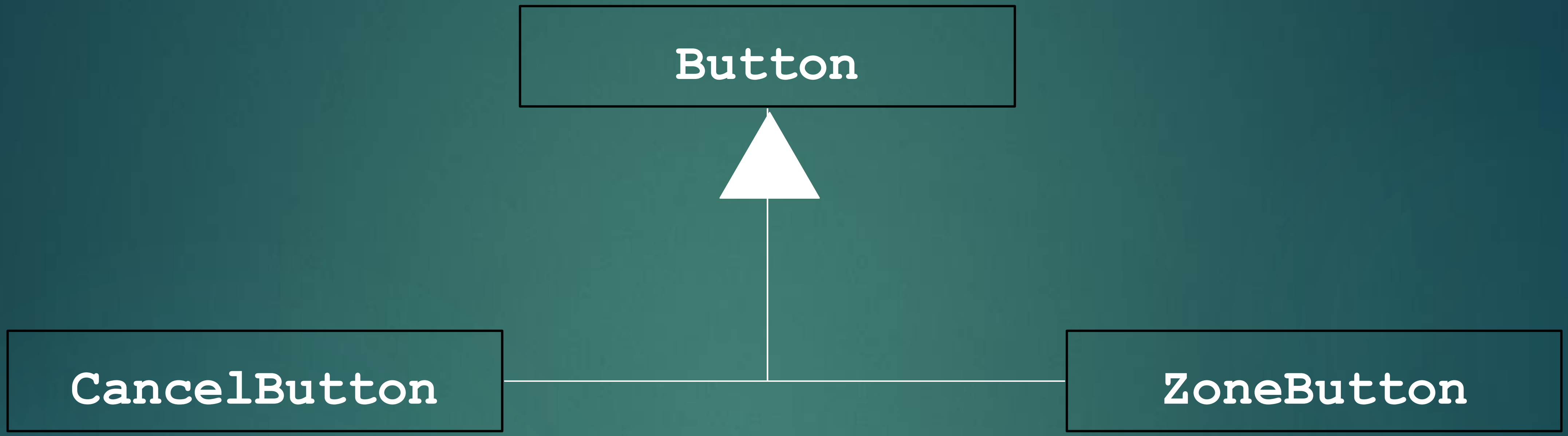
With qualification

- Qualifiers can be used to reduce the multiplicity of an association

Qualification: Another Example



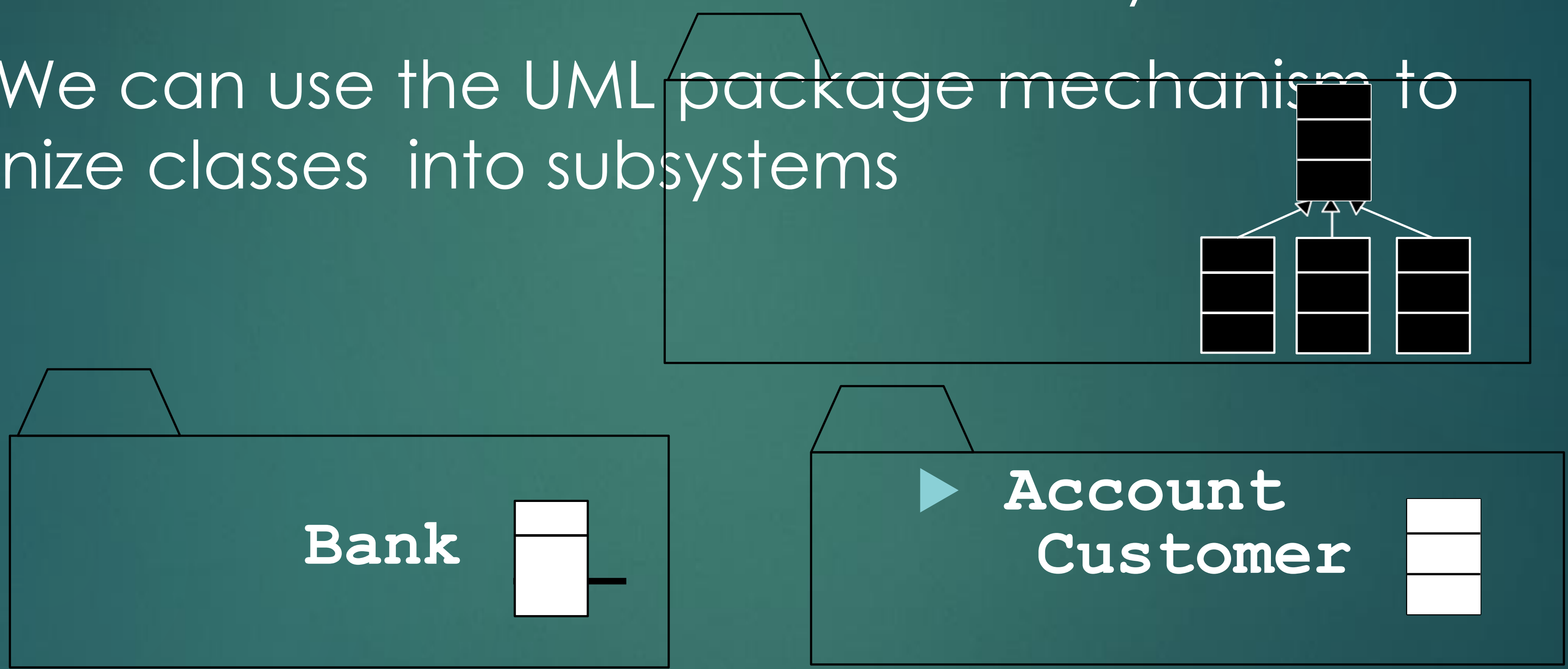
Inheritance



- *Inheritance* is another special case of an association denoting a "kind-of" hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The **children classes** inherit the attributes and operations of the **parent class.**

Packages

- Packages help you to organize UML models to increase their readability
- We can use the UML package mechanism to organize classes into subsystems



Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package.

Object Modeling in Practice

Foo
Amount CustomerId
Deposit() Withdraw() GetBalance()

Class Identification: Name of Class, Attributes and Methods

Is Foo the right name?

Object Modeling in Practice: Brainstorming

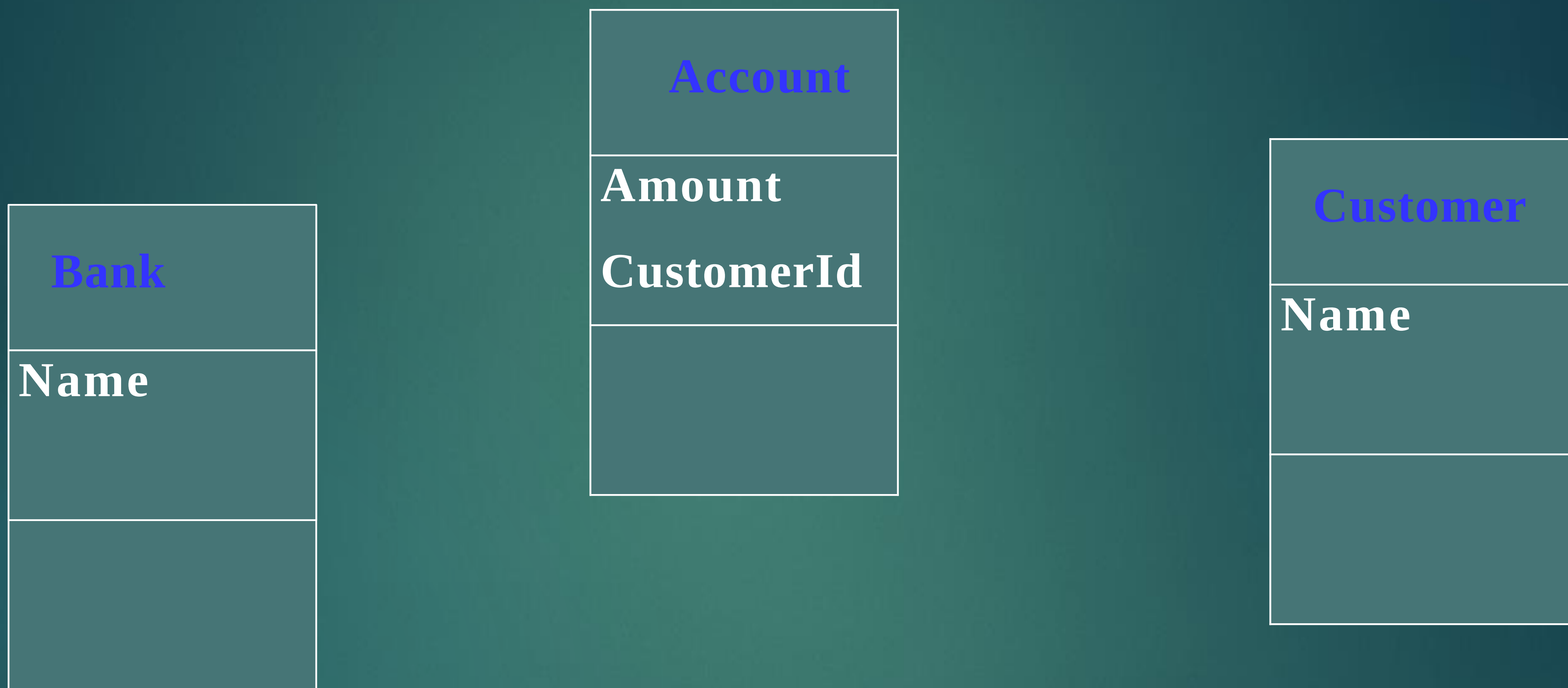
~~“Dada”~~
Amount CustomerId
Deposit() Withdraw() GetBalance()

~~Foo~~
Amount CustomerId
Deposit() Withdraw() GetBalance()

Is Foo the right name?

Account
Amount CustomerId
Deposit() Withdraw() GetBalance()

Object Modeling in Practice: More classes



1) Find New Classes

2) Review Names, Attributes and Methods

Object Modeling in Practice: More classes

Bank
Name

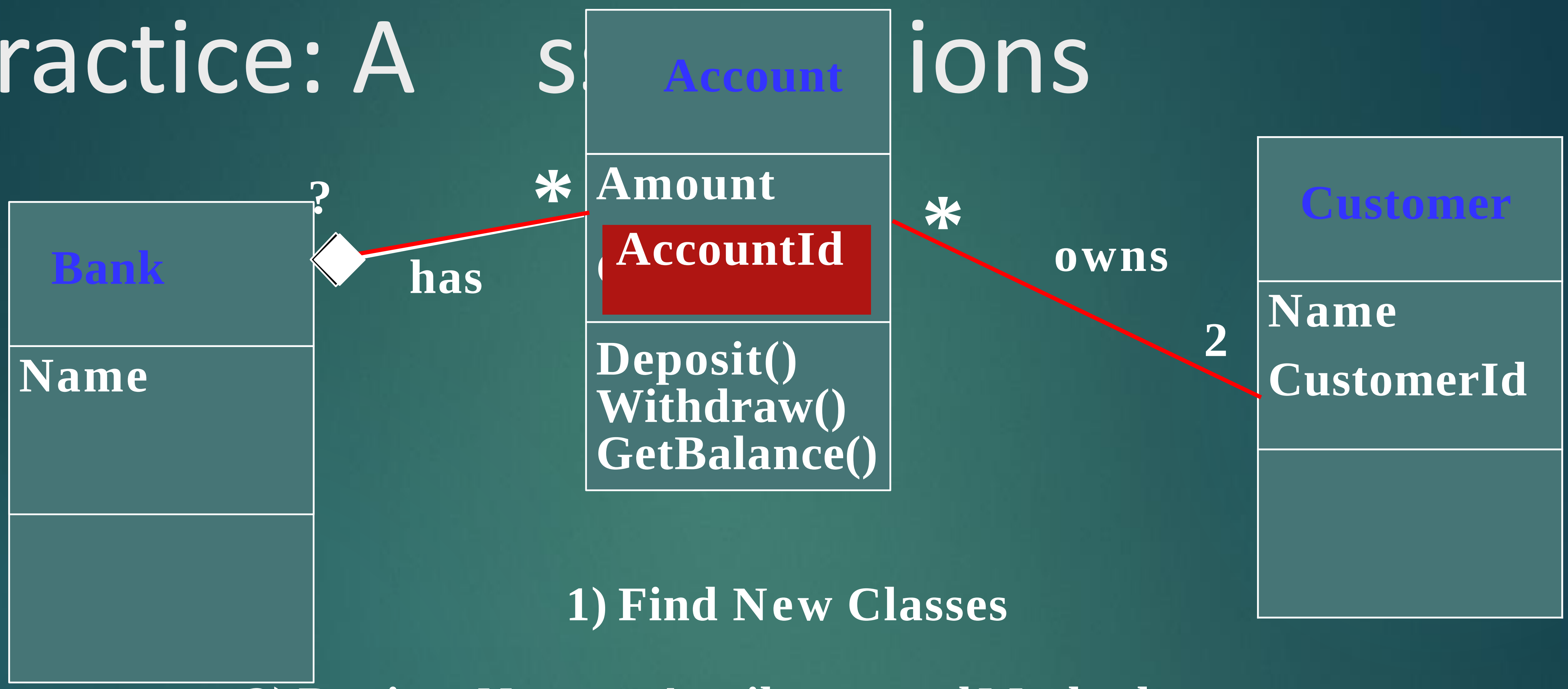
Account
Amount AccountId
Deposit() Withdraw() GetBalance()

Customer
Name CustomerId

1) Find New Classes

2) Review Names, Attributes and Methods

Object Modeling in Practice: A s ions



1) Find New Classes

2) Review Names, Attributes and Methods

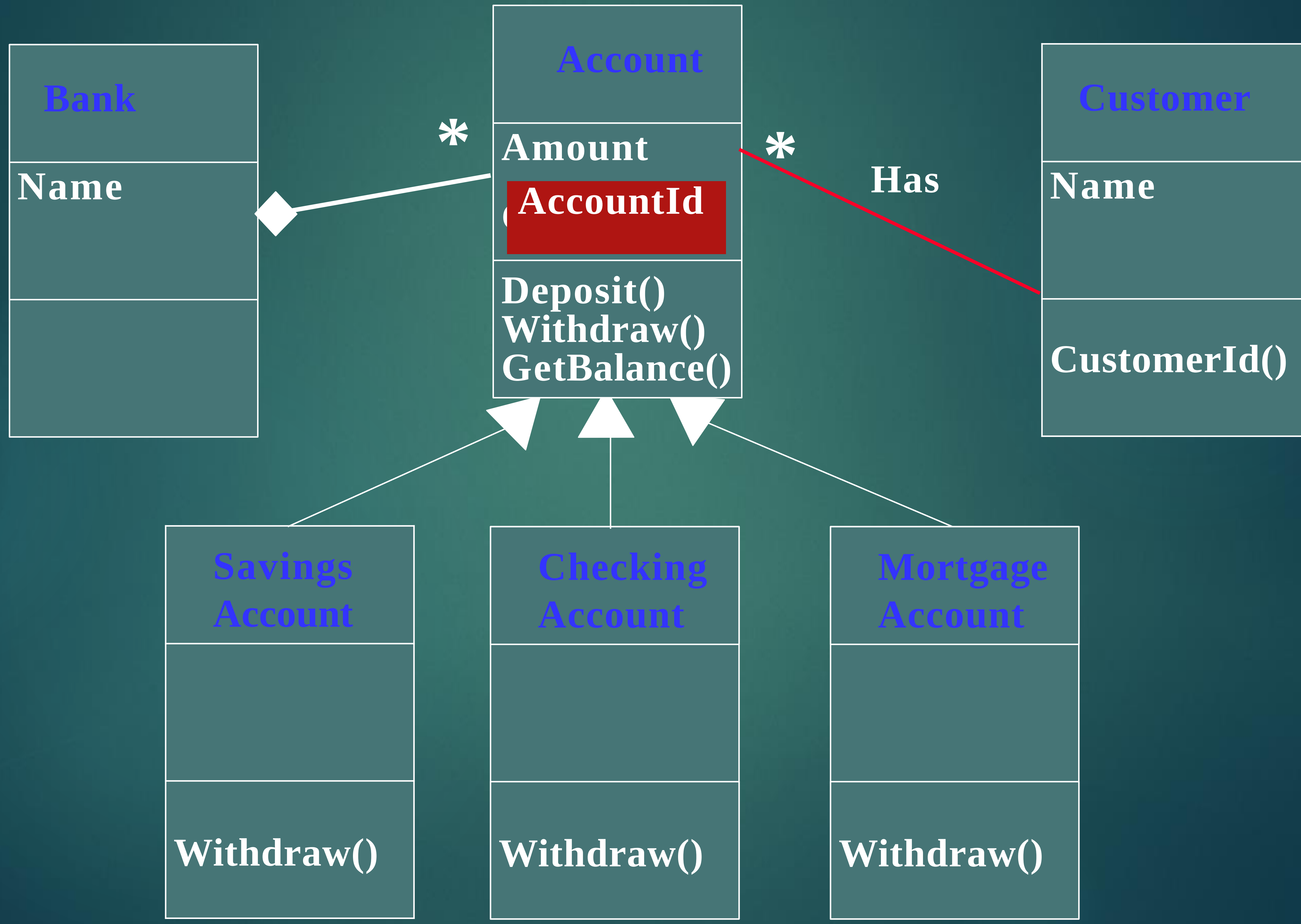
3) Find Associations between Classes

4) Label the generic assocations

5) Determine the multiplicity of the assocations

6) Review associations

Practice Object Modeling:
Find Taxonomies



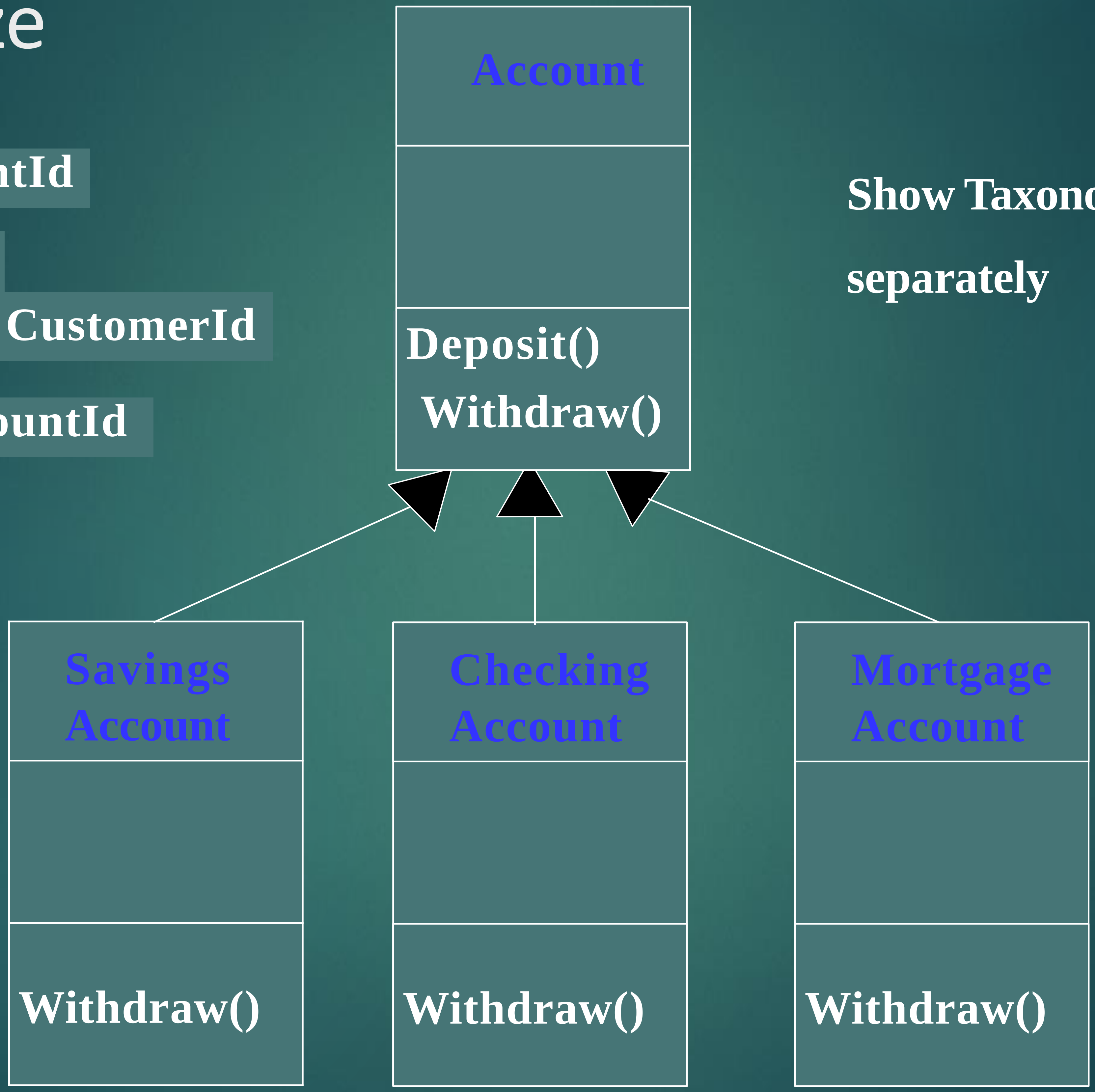
Practice Object Modeling: Simplify, Organize

AccountId

Amount

CustomerId

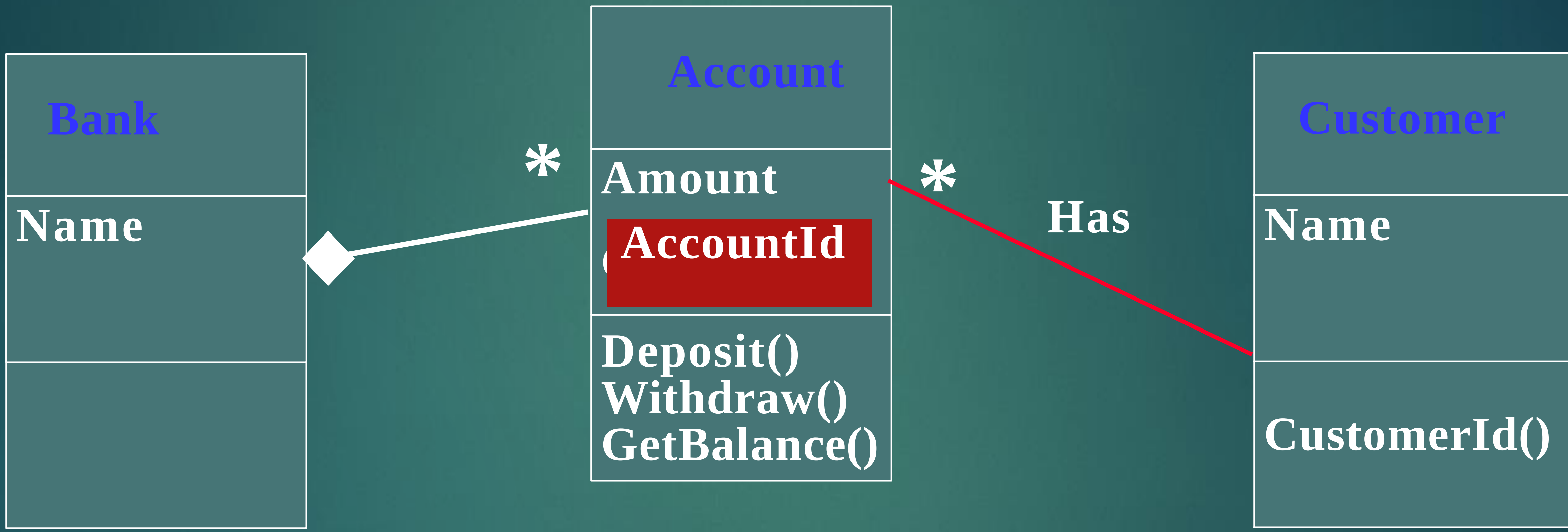
AccountId



Show Taxonomies

separately

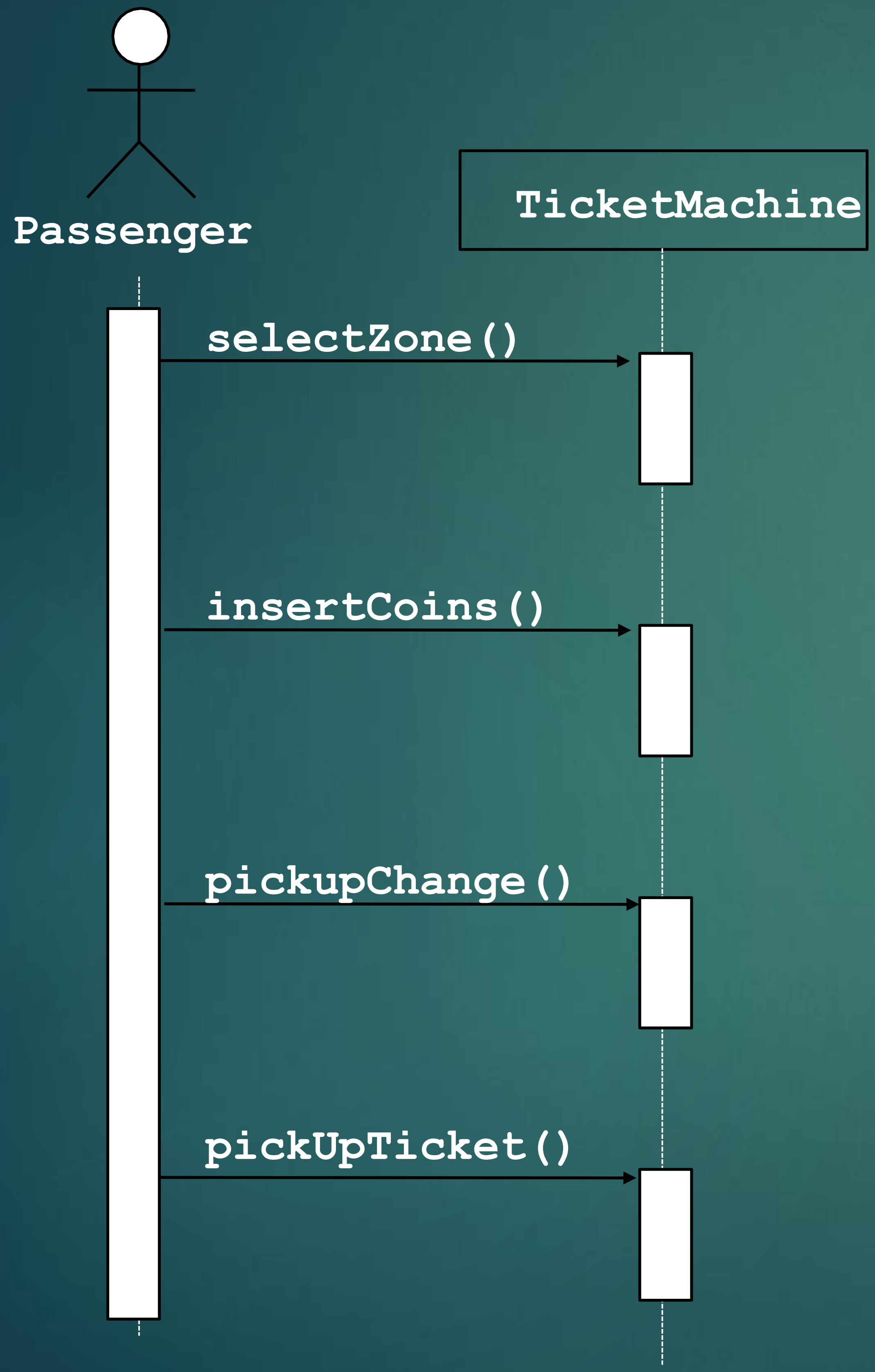
Practice Object Modeling: Simplify, Organize



Use the 7+-2 heuristics

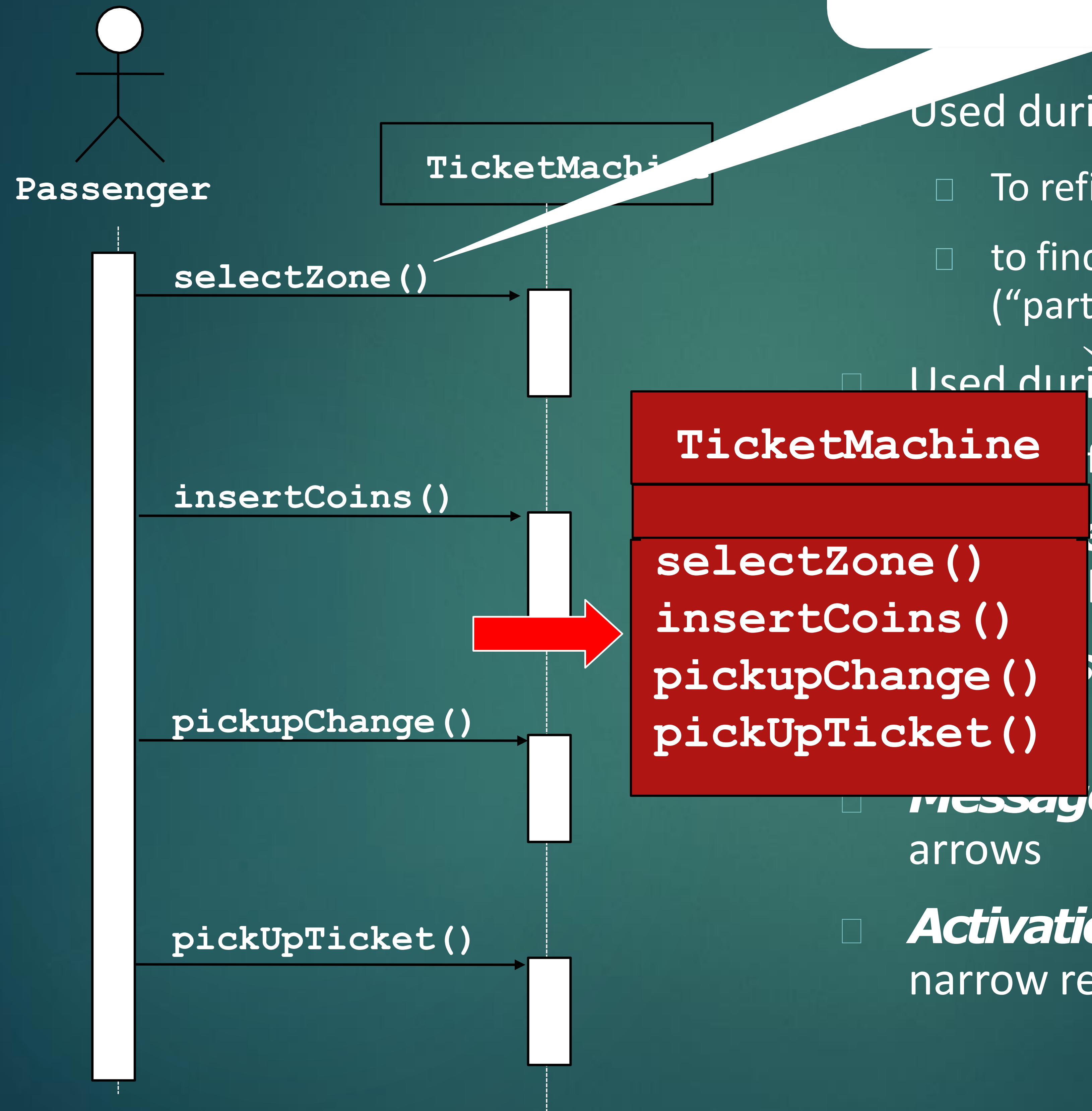
or better 5+-2!

Sequence Diagrams



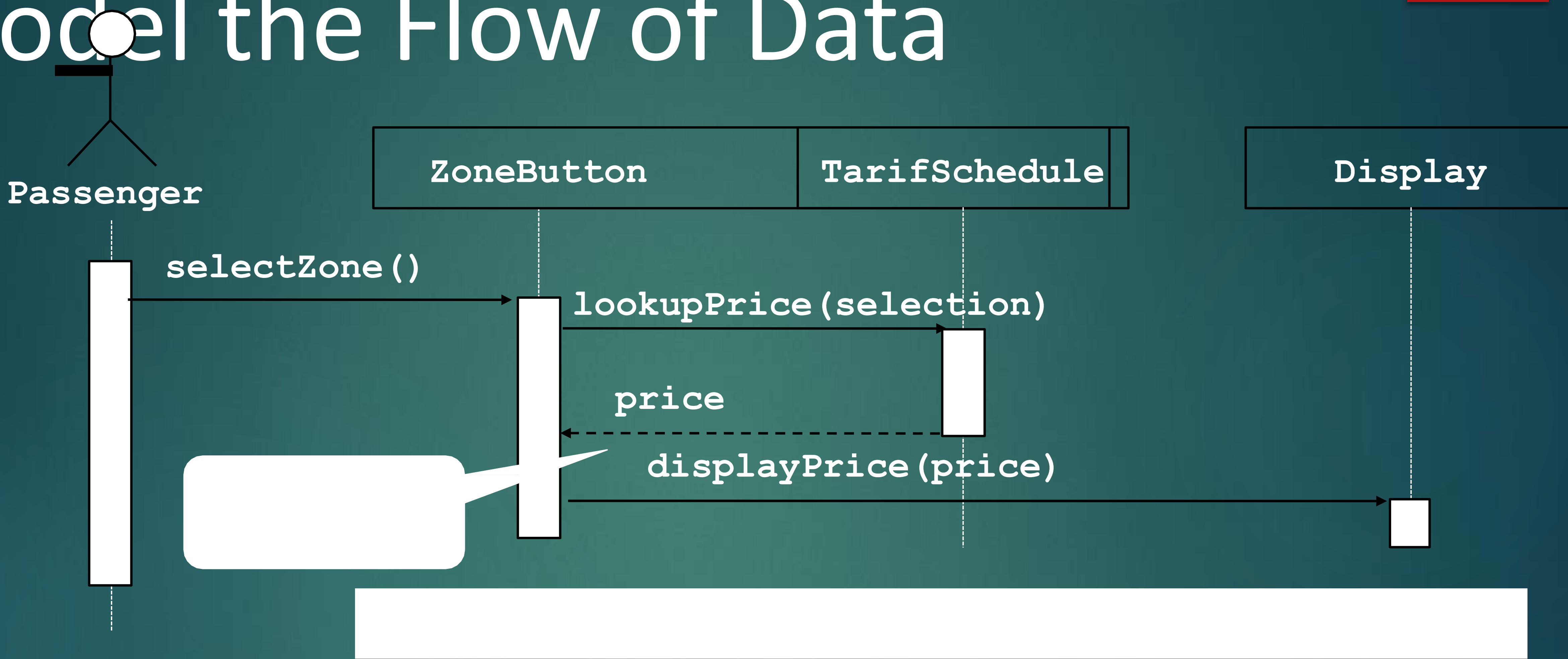
- Used during analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- ***Instances*** are represented by rectangles. ***Actors*** by sticky figures
- ***Lifelines*** are represented by dashed lines
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles.

Sequence Diag



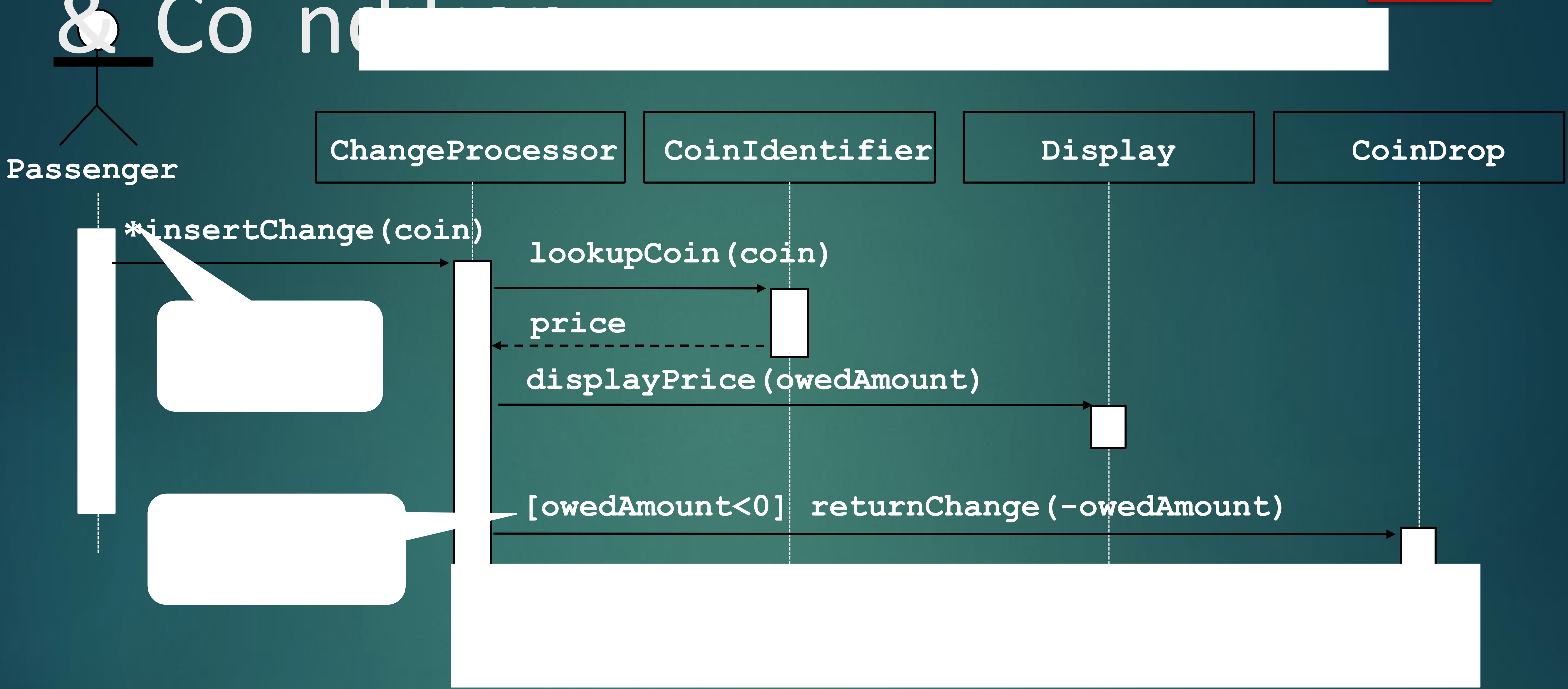
- Used during analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - fine su stem
- *es* are rep
  les. *Actors*
- are represe
- *Messages* are represented by arrows
- *Activations* are represented by narrow rectangles.

Sequence Diagrams can also model the Flow of Data



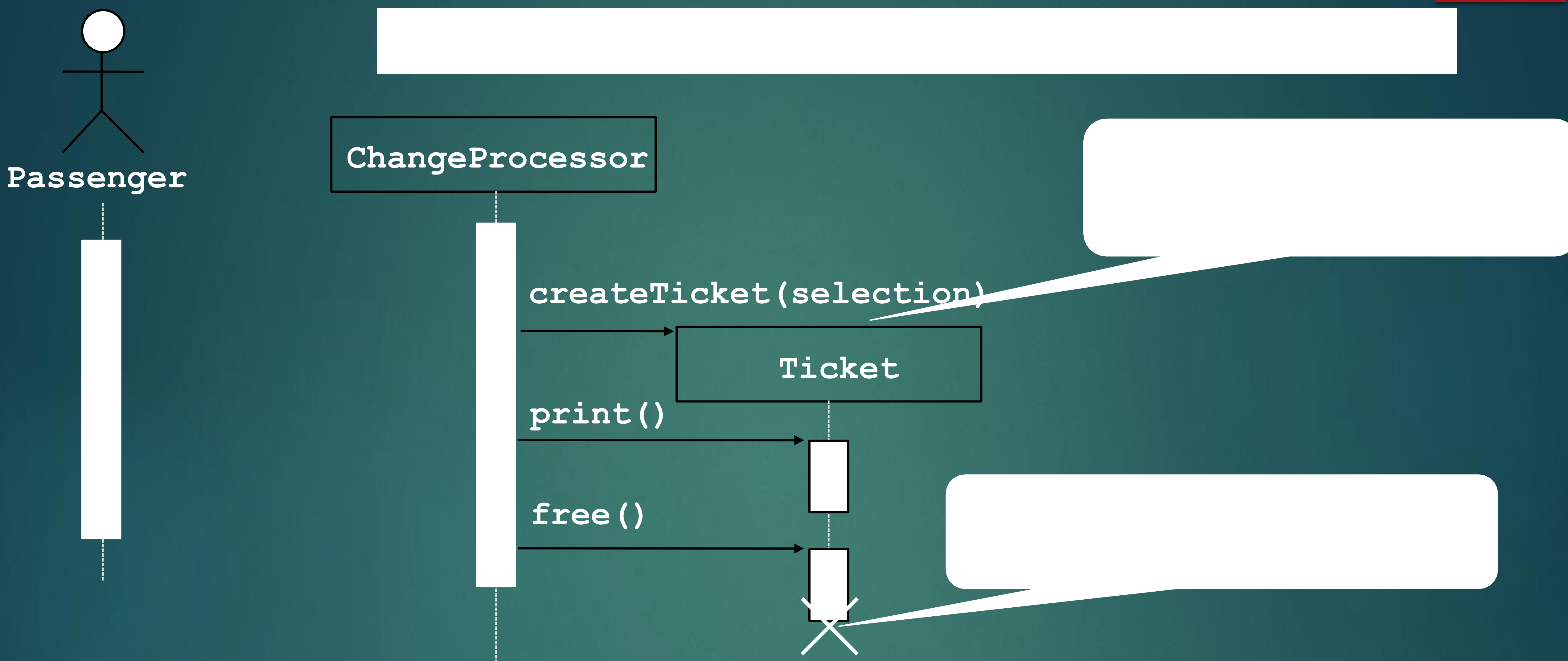
- The source of an arrow indicates the activation which sent the message
- Horizontal dashed arrows indicate data flow, for example return results from a message

Sequence Diagrams: Iteration & Co nd



- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction



- Creation is denoted by a message arrow pointing to the object
- Destruction is denoted by an X mark at the end of the destruction activation
  - In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

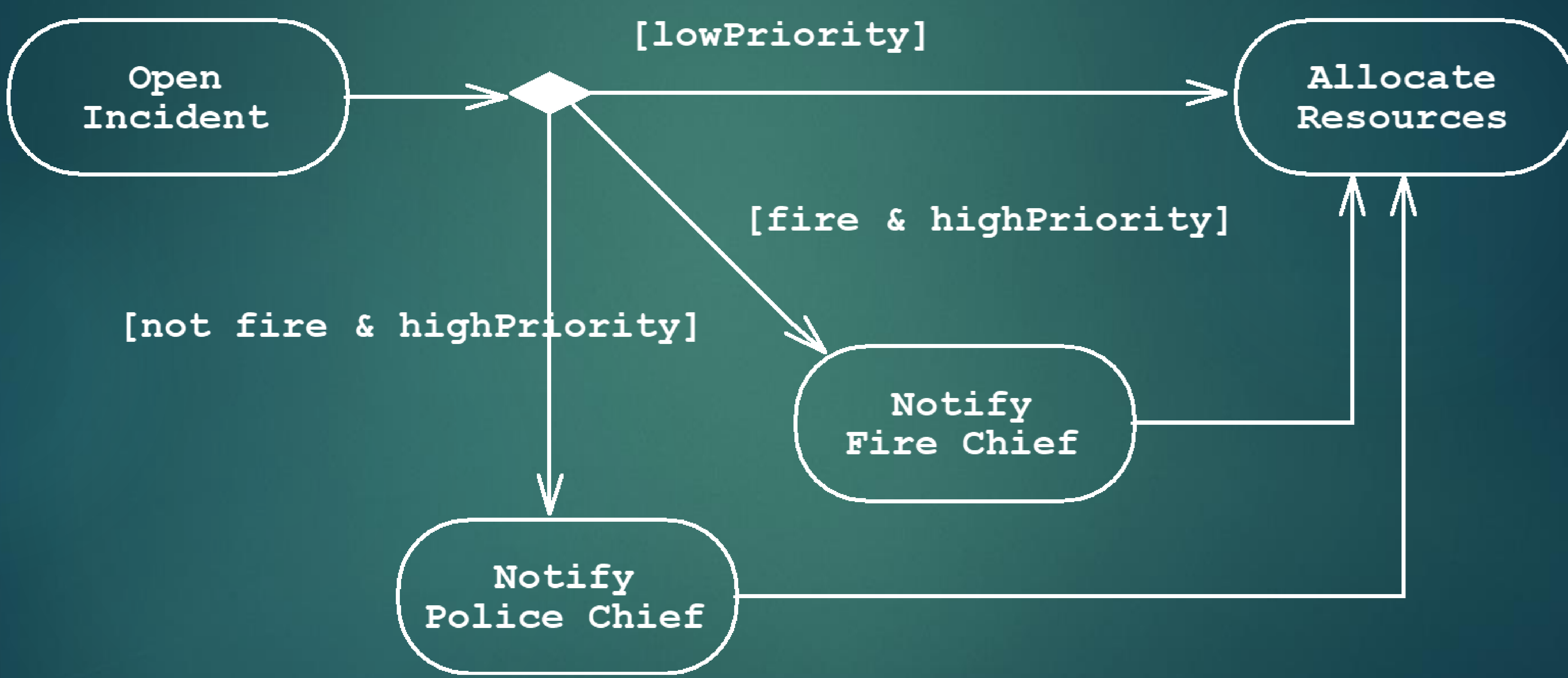
Sequence Diagram Properties

- UML sequence diagram represent *behavior in terms of interactions*
- Useful to identify or find missing objects
- Time consuming to build, but worth the investment
- Complement the class diagrams (which represent structure).

Activity Diagrams

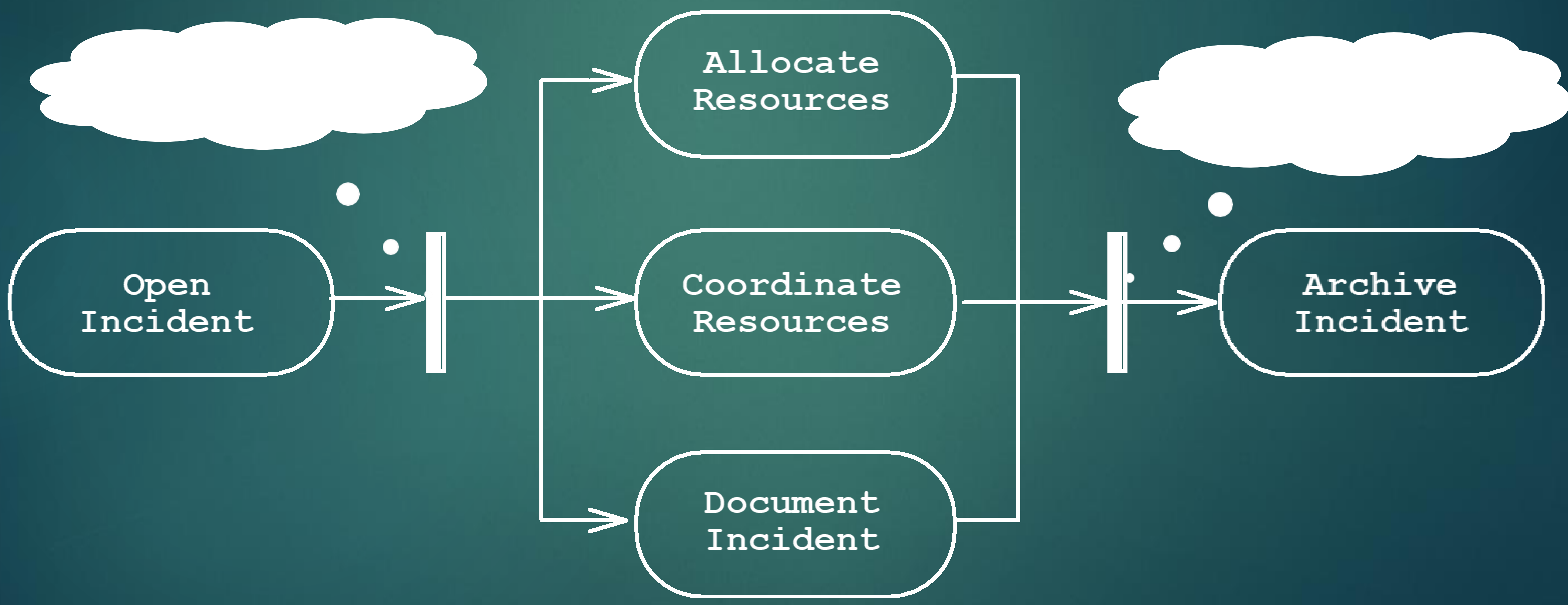
- An activity diagram is a special case of a state chart diagram
- The states are activities ("functions")
- An activity diagram is useful to depict the workflow in a system

Activity Diagrams allow to model Decisions



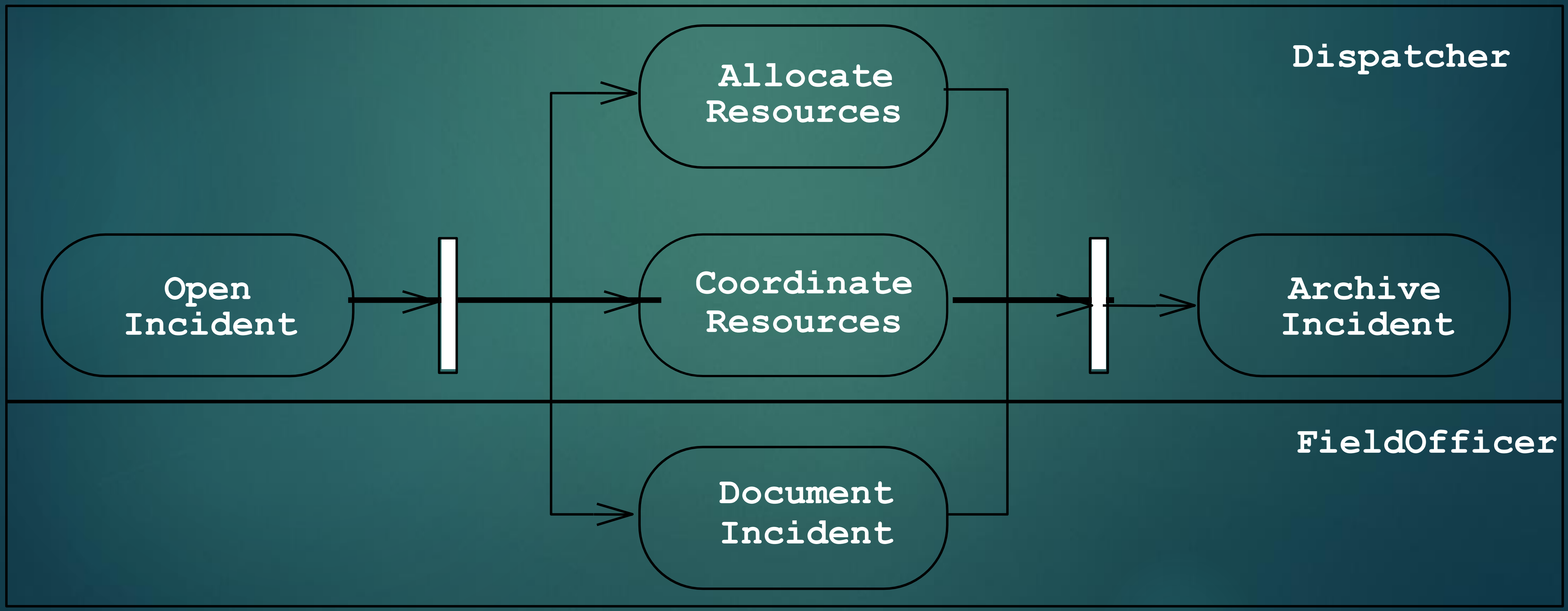
Activity Diagrams can model Concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



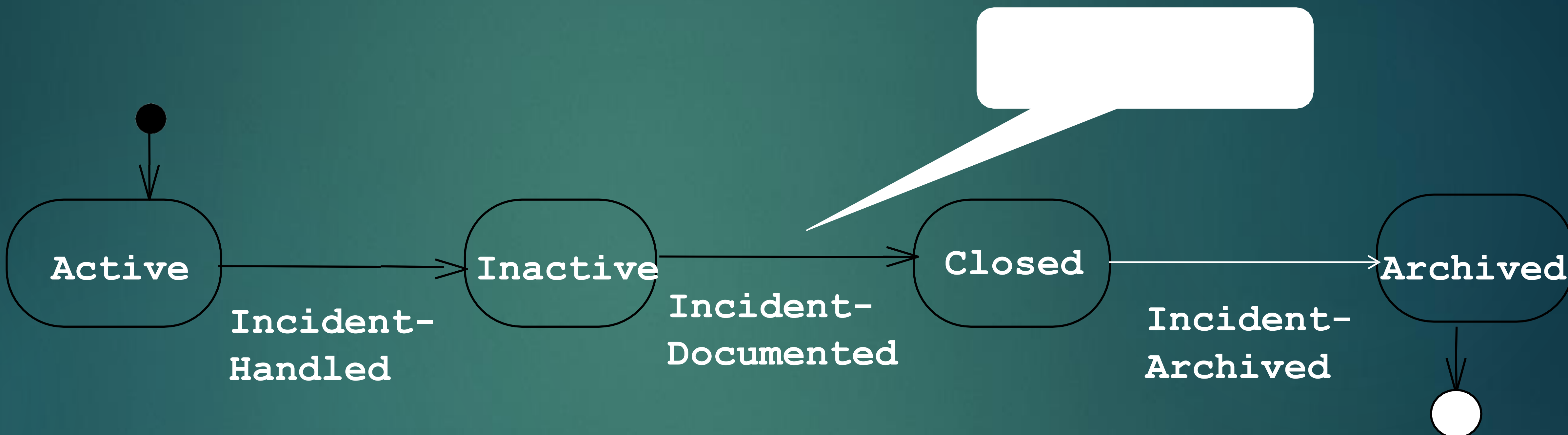
Activity Diagrams: Grouping of Activities

- Activities may be grouped into swimlanes to denote the object or subsystem that implements the activities.

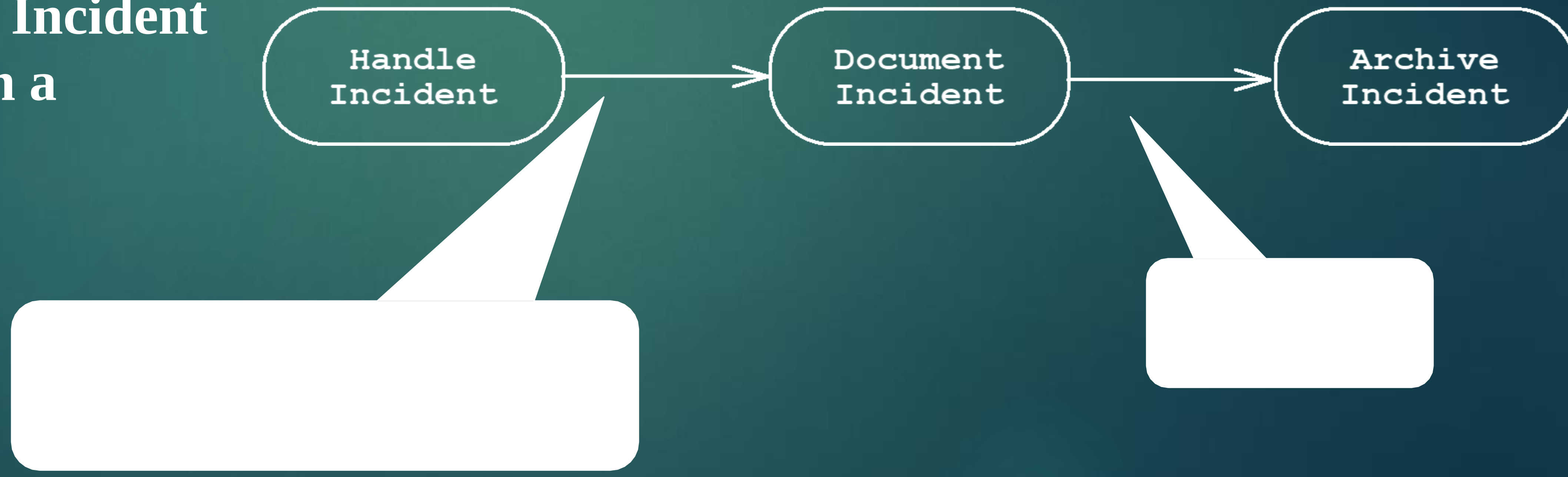


Activity Diagram vs. Statechart Diagram

Statechart Diagram for Incident
Focus on the set of attributes of a single abstraction (object, system)



Activity Diagram for Incident (Focus on dataflow in a system)



UML Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex
- UML is a programming language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features
- We concentrated on a few notations:
  - Functional model: Use case diagram
  - Object model: class diagram
  - Dynamic model: sequence diagrams, statechart and activity diagrams

Additional References

- Martin Fowler
  - UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd ed., Addison-Wesley, 2003
- Grady Booch,James Rumbaugh,Ivar Jacobson
  - The Unified Modeling Language User Guide, Addison Wesley, 2nd edition, 2005
- Commercial UML tools
  - Rational Rose XDE for Java
    - http://www-306.ibm.com/software/awdtools/developer/java/
  - Together (Eclipse, MS Visual Studio, JBuilder)
    - http://www.borland.com/us/products/together/index.html
- Open Source UML tools
  - http://java-source.net/open-source/uml-modeling
  - ArgoUML,UMLet,Violet, …