Review

Four Kinds of Visibility

OO programming languages may provide four levels of scope for names:

- Attribute visibility
- Parameter visibility
- Local visibility
- Global visibility

Motivation and Concept

- OO systems exploit recurring design structures that promote
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance

What Is a Design Pattern?

- A design pattern
  - **Is a common solution to a recurring problem in design**
  - Abstracts a recurring design structure
  - Comprises class and/or object
    - Dependencies
    - Structures
    - Interactions
    - Conventions
- A design pattern has 4 basic parts:
  - 1. Name
  - 2. Problem
  - 3. Solution
  - 4. Consequences and trade-offs of application
- Language- and implementation-independent

GRASP patterns: General Responsibility Assignment Software Patterns

- Expert - allocate a responsibility to a class that has the information. Look in both the design classes and the domain model to find a type of object that knows enough to handle the responsibility. Knowledge includes having the data, and also knowing who has got the data.
- Creator - creating an object should be the responsibility of a class that is closely related to the created object. Notice that you always need at least one way of creating an object that doesn't depend on already having an object of that type. So which class or object is close to the created object.
- Low Coupling - Assign a responsibilities so that class depends less on other classes using a "need to know basis". Organize responsibilities so that classes do not depend on each other two much.
- High Cohesion - Design elements to have strongly related and focused responsibilities

GRASP patterns: General Responsibility Assignment Software Patterns

- Controller - Assign the responsibility for handling system event messages to a class representing either the whole system, device, or subsystem, or representing the use case /scenario within which the system event occurs.
- Polymorphism - Give the same name to services in different objects when the services are similar or related. Classify objects, using inheritance/generalization, to allow the right version of a service to be executes for the object in question.
- Pure Fabrication - assigning a highly cohesive set of responsibilities to an artificial convenience class that doesn't represent a problem domain concept.
- Indirection - Assign a responsibility to a class that knows where to find an object that can complete the task

GoF Patterns: Three Types

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Classification of GoF Design Pattern

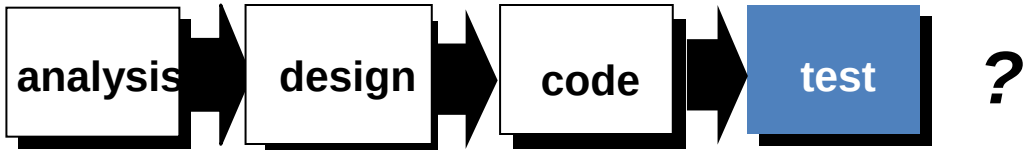
Creational	Structural	Behavioral
Factory	Adapter	Command
Singleton	Composite	Iterator
	Decorator	Observer
		Strategy

Mapping Designs to Code

Test-Driven Development, Refactoring

Objectives

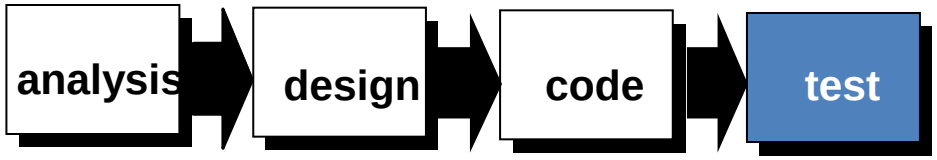
- To discuss when testing takes place in the life cycle



  - Test-driven development advocates early testing!
- To cover the strategies and tools associated with object oriented testing
  - Analysis and Design Testing
  - Class Tests
  - Integration Tests
  - Validation Tests
  - System Tests
- To discuss test plans and execution for projects

Testing Summary

- Testing affects all stages of software engineering cycle
- One strategy is a bottom-up approach – class, integration, validation and system level testing
- XP advocates test-driven development: plan tests before you write any code, then test any changes
- Other techniques:
  - white box (look into technical internal details)
  - black box (view the external behavior)
  - debugging (systematic cause elimination approach is best)



Activity Diagrams

Introduction(I)

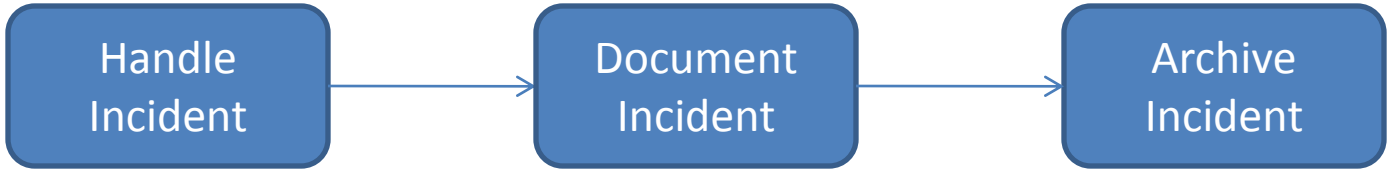
- Activity diagrams describe procedural logic, business process, and workflow.
- Activity diagrams focus on the action sequence of execution and the conditions that trigger or guard those actions.
- Activity diagrams focus on the activity's internal actions, not on the external interfaces

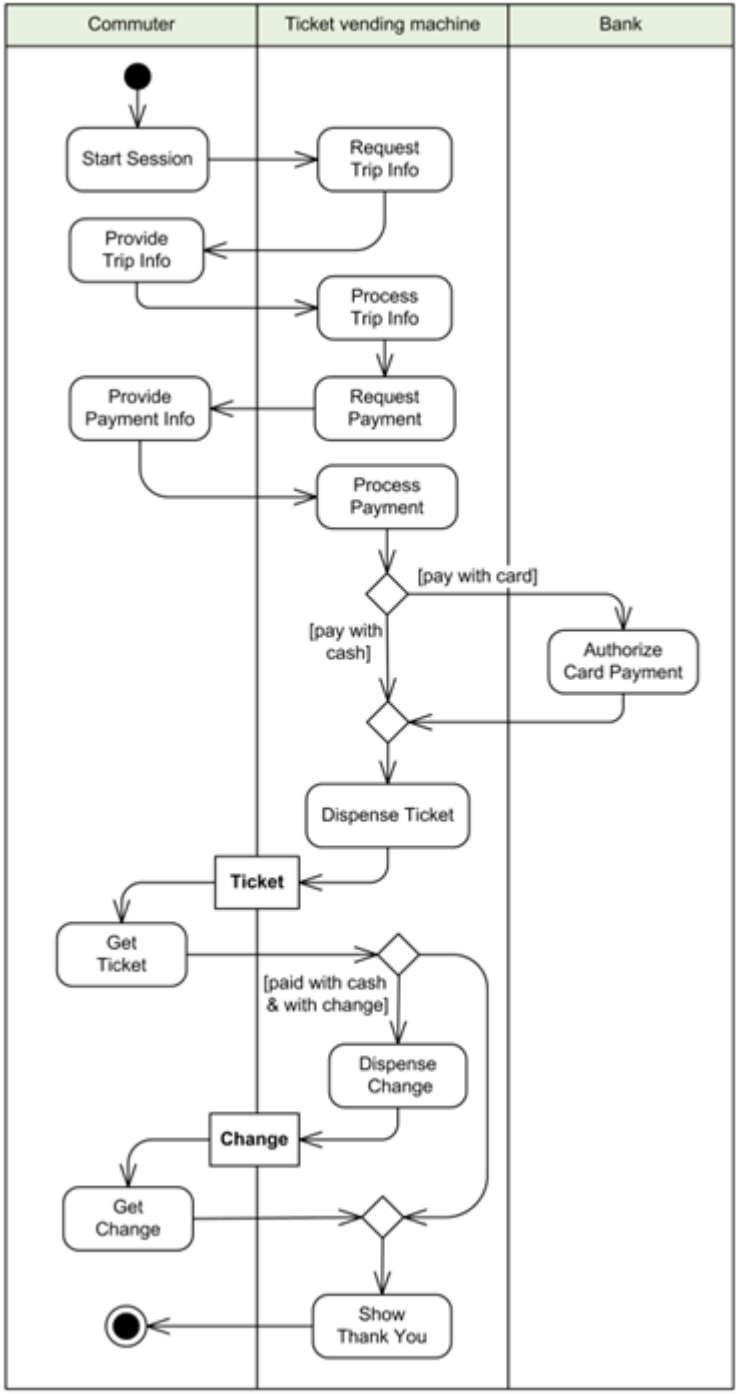
Introduction(II)

- Activity diagrams have similarities to flowcharts
  - But flowcharts notation does not support parallel behavior.
  - Business managers may prefer activity diagrams over flowcharts, because they are more understandable for non-technical people.
- An activity diagram is a special case of state chart diagram in which states are actions.

Introduction(III)

- An activity diagram shows flow control within a system.




Commuter
Ticket vending machine
Bank
Start Session
Request Trip Info
Provide Trip Info
Process Trip Info
Provide Payment Info
Request Payment
Process Payment
[pay with card]
[pay with cash]
Authorize Card Payment
Dispense Ticket
Ticket
Get Ticket
[paid with cash & with change]
Dispense Change
Change
Get Change
Show Thank You

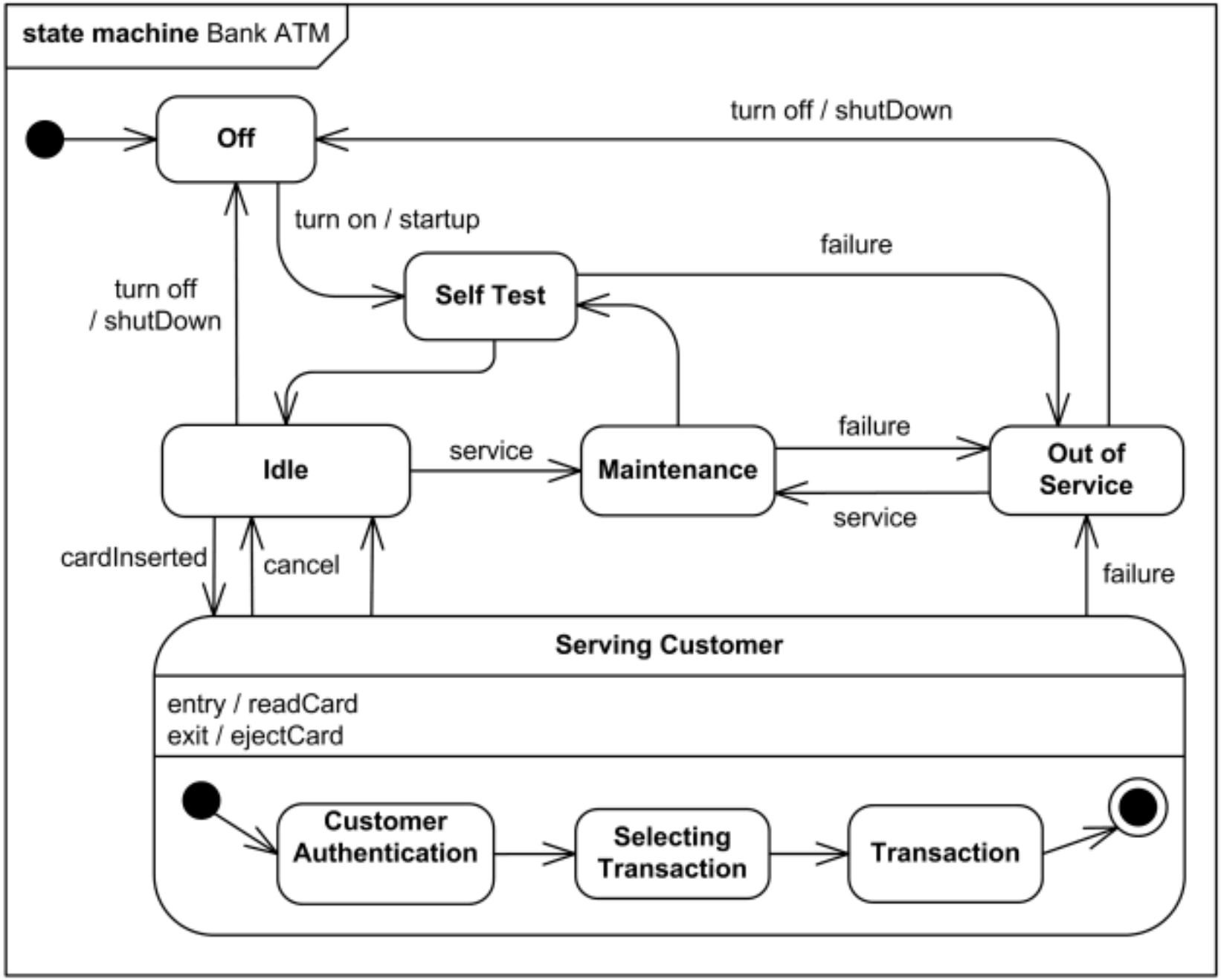
State Chart Diagrams

Introduction to State Diagrams

- Interaction diagrams are very good for communication
  - between clients and designers
  - between designers and other designers
  - between designers and implementors
- But they are not part of the **constructive core**
  - Constructive core means that part from which an actual implementation could be generated
- The constructive core of the UML is
  - class diagrams (for the static part)
  - state diagrams (for the dynamic part)
- State diagrams can describe very complex dynamic behavior

Elements of State Diagrams

- The basic elements of state diagrams are
  - *states* – the state in which the object finds itself at any moment
  - *transitions* – take the object from one state to another
  - *events* – cause the transition from one state to another
  - *actions* – take place as a result of a transition
- In the UML, many other extensions have been added, such as:
  - generalization/Inheritance among events (i.e. a mouse click and keyboard input as subclasses of user interface events)
  - hierarchical diagrams for managing complexity (from StateCharts)
  - guards – conditions on transitions
- State Diagrams should only be used for objects with significant dynamic behavior


state machine Bank ATM
Off
turn off / shutDown
turn on / startup
failure
turn off
/ shutDown
Self Test
Idle
service
Maintenance
failure
Out of
Service
service
cardInserted
cancel
failure
Serving Customer
entry / readCard
exit / ejectCard
Customer
Authentication
Selecting
Transaction
Transaction

Thank You