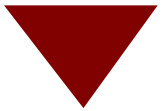
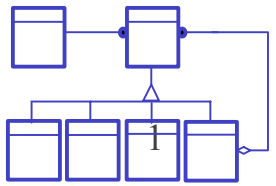


Developing Business Frameworks

Object Expo - NYC
March 1998

Scott H. Koehler

Koehler Consulting, Inc.
Holliston, MA (508)
429-1589 Tel.
email: info@koehlerconsult.com



Our Background

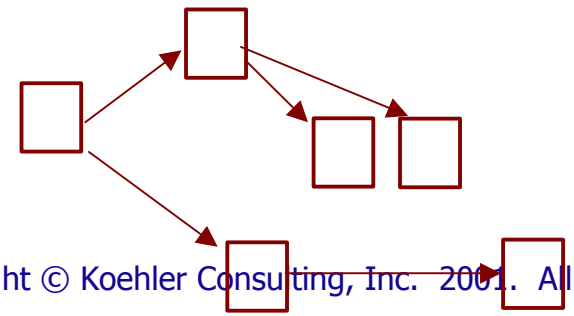
- Vertical industry business object applications since mid-80s
  - insurance and finance
  - C++ and Smalltalk
- Designed and assisted marketing an insurance framework (selling system) for a major insurance company
- Participated in object harvesting project sponsored by an industry consortium

Our Background (cont.)

- Built and market a life insurance tax compliance business object framework
  - (www "*KCI Tax Server*")
- Credit card payment processing system for insurance policies
- Defined contribution service engine
  - 401(k) transactions (loans, withdrawals, deferrals, etc.)

Business Frameworks

- A group of classes that work together to provide capability for a particular type of business application
- Collaborating business objects
- "White box" reuse
- Can contain multiple "components"
- Specialized for a particular vertical industry
  - E.g.insurance, banking, health care, etc.



Other Related Definitions

- Business object
  - an object that corresponds to an entity familiar in the business domain
  - e.g. Stock, Bond, InsurancePolicy
- Class library
  - a collection of classes that represent entities of the business used for creating business objects
- Component
  - encapsulated software services that can be used to build systems, which are packaged for reusability
  - "Black box" reuse

Framework Characteristics

- Software application comprised of cooperating business objects that deliver a particular business benefit
- Generally strong interrelationships among the business objects
- Multiple business objects combine to provide higher grain components
- Business objects are rarely "generic"
  - company specific rules and methods

Why Business Frameworks?

- Method of building object oriented business applications
- Extensible business applications
- Reusable components
- Development time

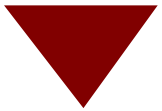
Buy versus Build

- Evaluation considerations:
  - Conventional thinking...
  - Time-to-market
  - Acquisition cost versus development cost
    - success rate of new projects
  - Competitive advantage
  - How close a fit
  - Vertical frameworks are not "generic"
    - Quality of the offering -- is it OO?
    - Architecture and design

Buy versus Build (cont.)

If you buy....

- What you might get:
  - Intellectual property
  - Existing design models
  - Existing source code
  - A foundation to extend
    - gets past the clean sheet of paper
    - infrastructure
  - A defined set of requirements
  - Tested code



Building from Scratch

- Define the requirements
- Define goals & objectives
- Conduct OO Analysis
- Determine the architecture
  - define design principles
- Create object models / diagrams
- Perform OO Design
- Construct the framework(s)
- Iterate

Requirements

- Functional specifications
  - English descriptions of the required processing
  - Try to avoid implementation level specifications
  - May require archaeology of current systems

Requirements (cont.)

- Use cases
  - a description of a sequence of actions performed by a system when interacting with other systems or users
  - various cases for how a system is used
  - consists of:
    - name
    - actors
    - pre-conditions, post-conditions
    - script
  - Effective for interactive systems

Sample Use Case

Execute a stock trade

Actors:

Phone Rep, VRU

Pre-conditions:

Authenticated customer

SSN entered

Post-conditions:

Completed transaction

Script

Customer requests trade

Phone rep selects "trade" transaction

System requests verification information

Phone rep confirms identify of customer

Customer identifies security and amount

……

Goal and Objectives

- *"This framework will ultimately provide the single source of all business rules and information for the XXX portion of the business....*

Object Oriented Analysis

- Focuses on "what" the system should do
- Starts with business description of system requirements
- Identify object classes (nouns)
- Identify attributes
- Identify services / operations
- Explore class generalizations and specializations
- Identify relationships between objects
- Construct models

CRC Card Example

Class: Account
Superclass: None
Description: This class represents a customer's account with his/her bank. This class interfaces with the bank database and also updates bank accounts.

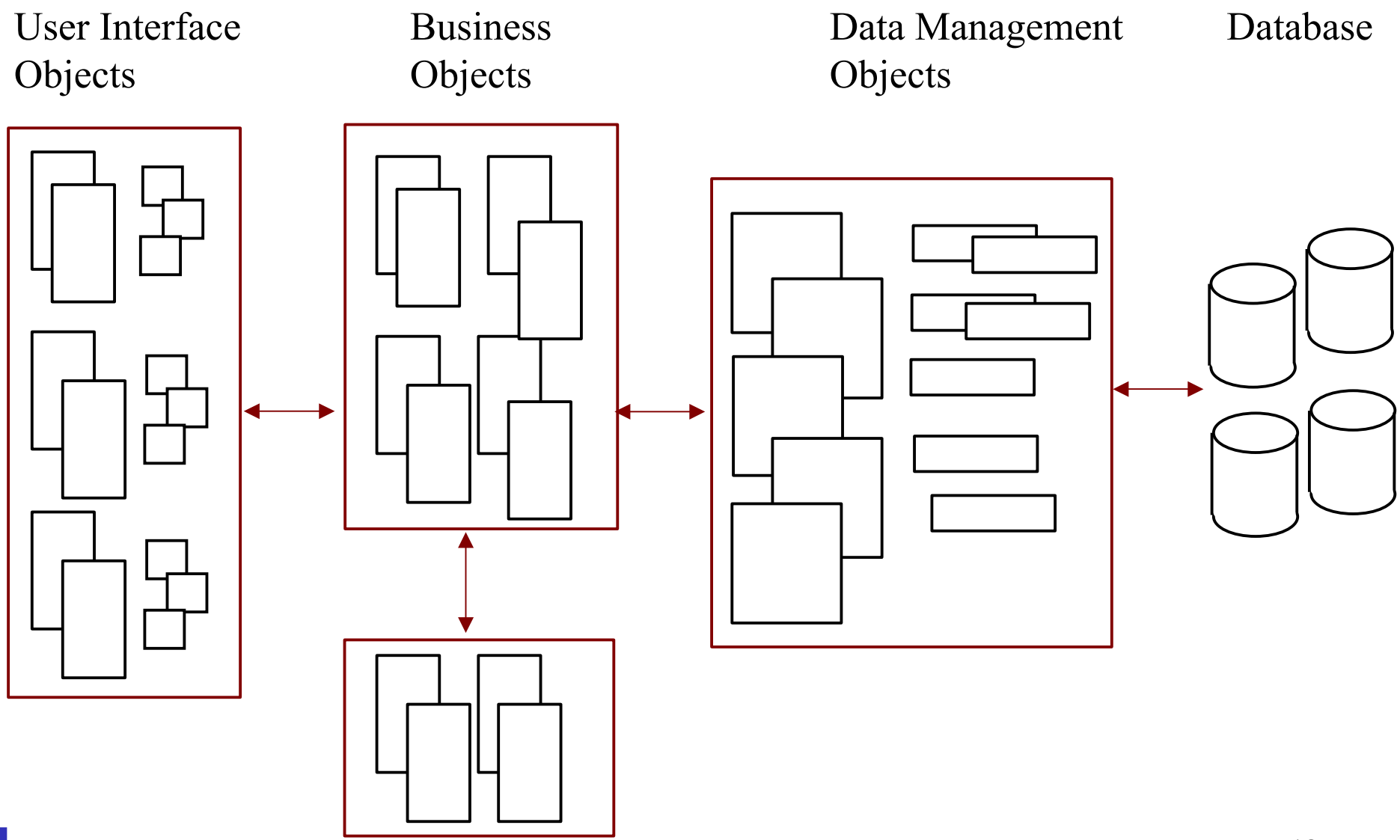
Responsibilities	Collaborators
deposit (AMT) accepts a floating point number and updates account with deposit amount.	deposit transaction
withdraw (AMT) accepts a floating point number and removes amount from account, if available.	withdrawal transaction

Attributes: balance - returns floating point number.

Business Object Architecture

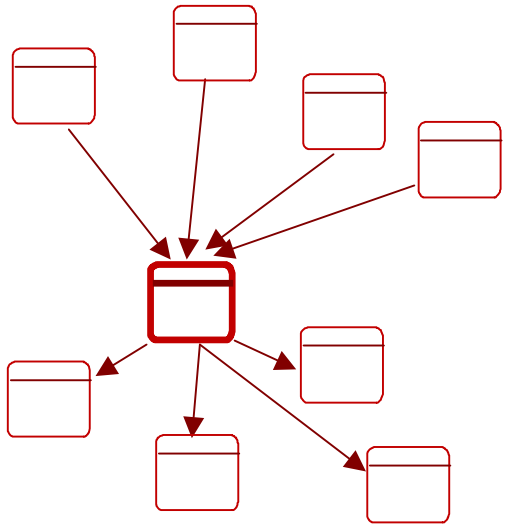
- Architecture breaks systems into subsystems
- Subsystems / components:
  - Enhance reusability
  - Reduce complexity
  - Provide a division of labor
- Business object frameworks enable a layered architecture
- Business objects form reusable components

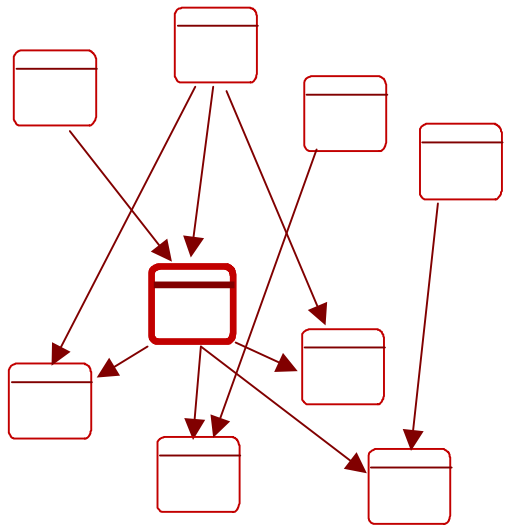
Layered Architecture



Architectural Considerations

- Interfaces;
  - Clean
  - Generic
  - Limited entry point with minimal coupling
- Encapsulated knowledge
- Platform independence
- Scalability

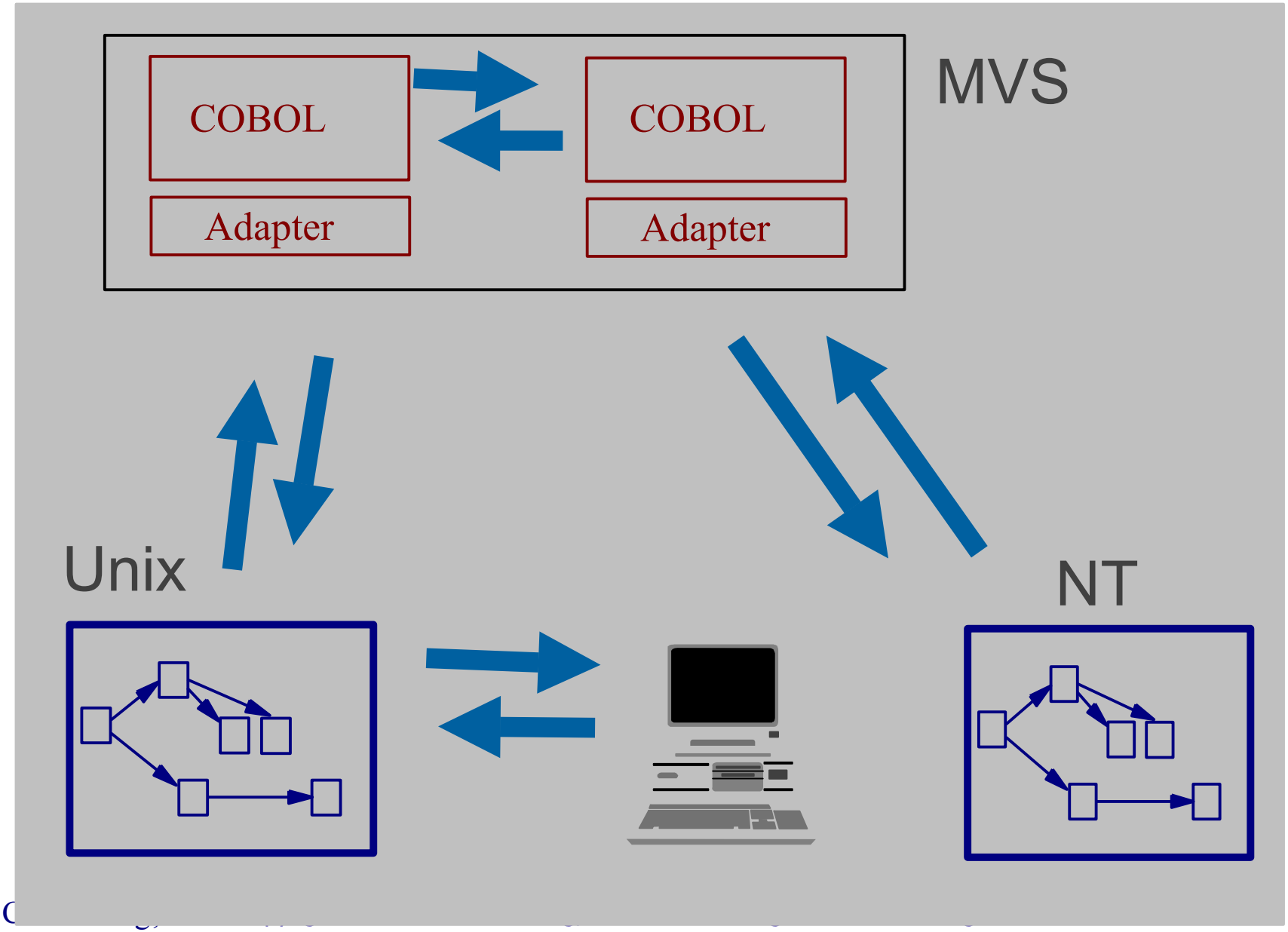




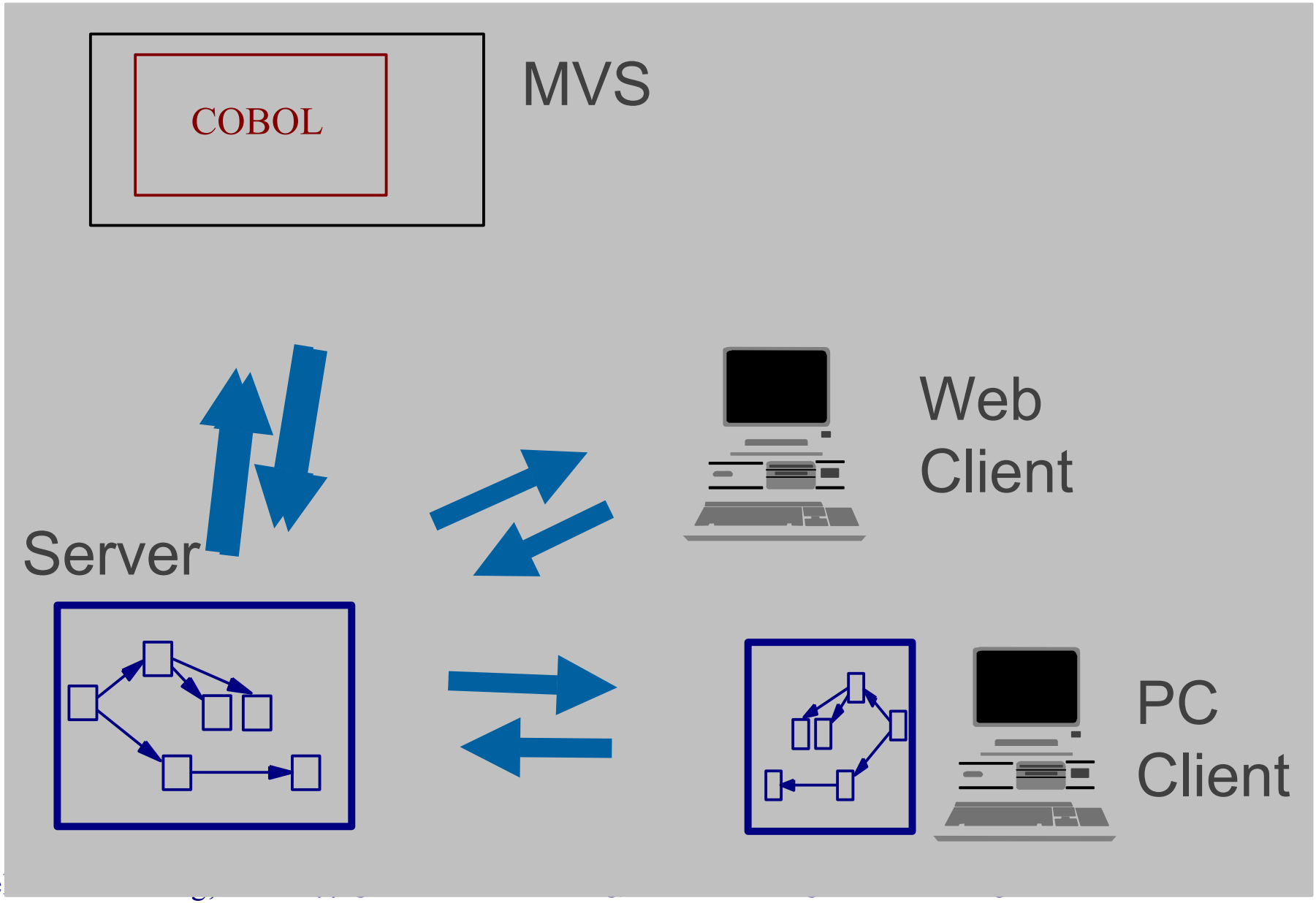
Architectural Considerations (cont.)

- Consider need to service multiple "clients"
- Develop use cases (usage scenarios) to understand how the framework/component will integrate with other subsystems

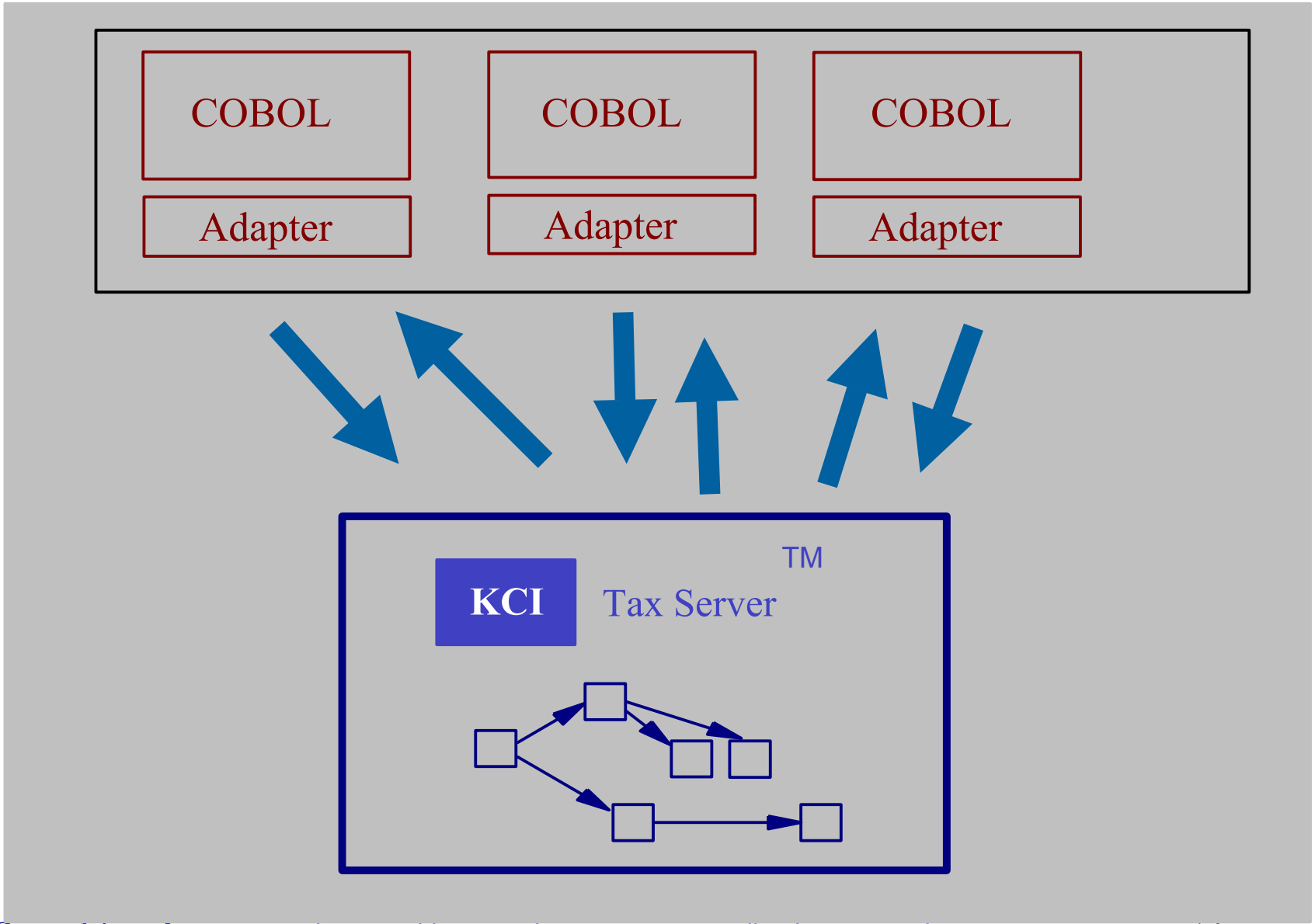
Sample Architecture #1



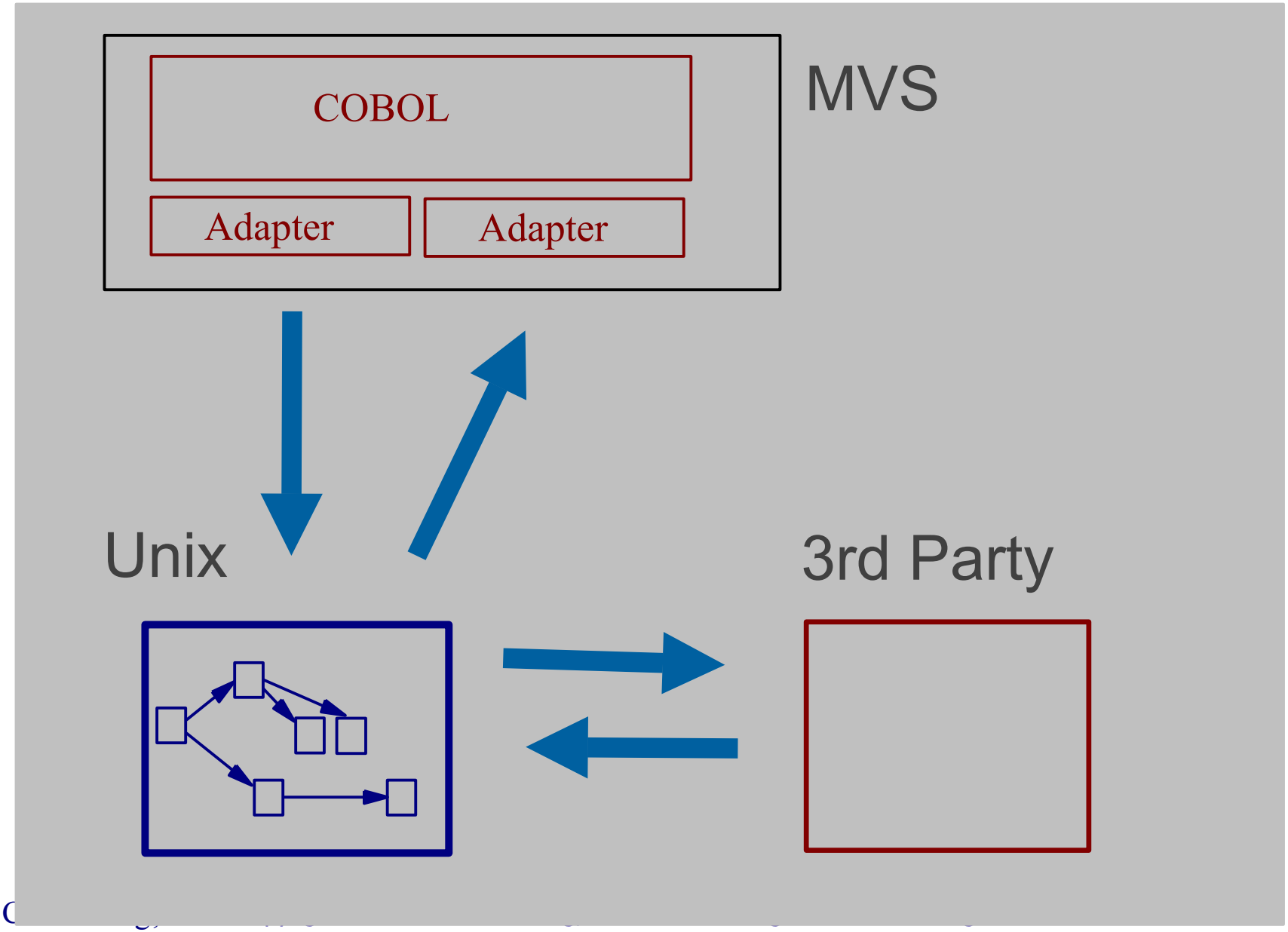
Sample Architecture #2



Sample Architecture #3



Sample Architecture #4



Design Principles

- A written set of stated design objectives to use as a reference during the development phase
- Addresses current and future uses of the framework
- Affects the representation and placement of knowledge
- Influences design decisions

Design Principle

- *"The tax subsystem will contain the knowledge of the tax law with other interfacing subsystems exhibiting little or knowledge of the tax law."*

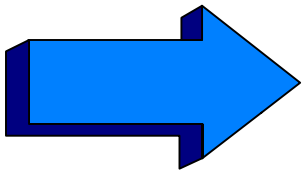
reference KCI, *KCI Tax Server Architecture*

Business Object Models

- Define relationships between business objects
- Defines how business objects interact
- Provide a view of the business
- Provide a design for constructing software

What's a Business Object?

- An object is a software entity comprised of process and data that models "real world things"

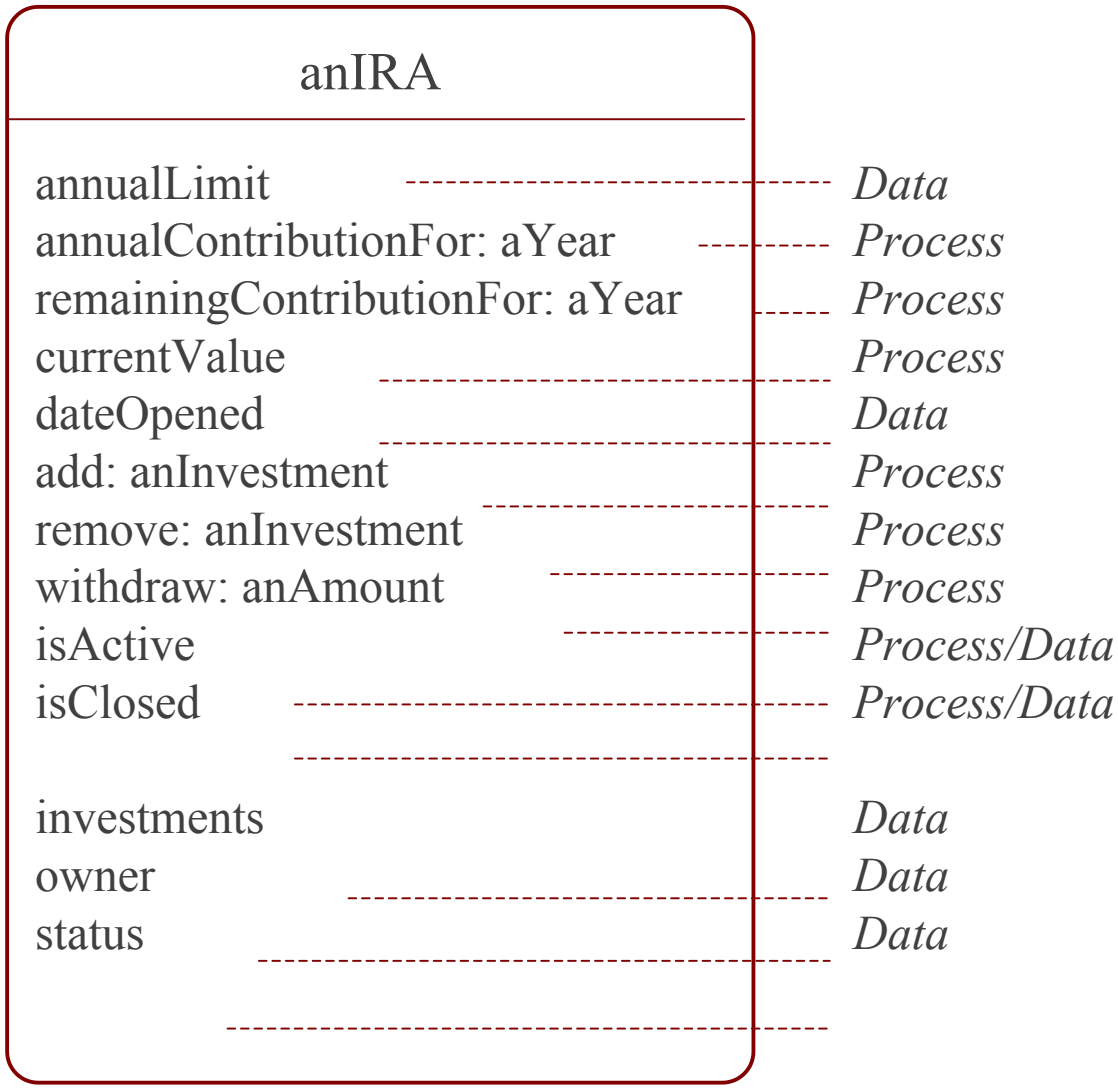


a Business Object is an object that models a fundamental entity of the business

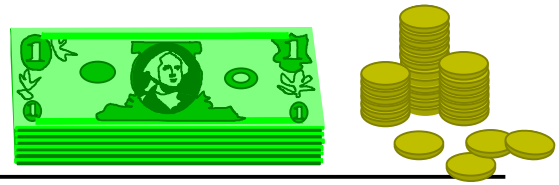
Why Business Objects?

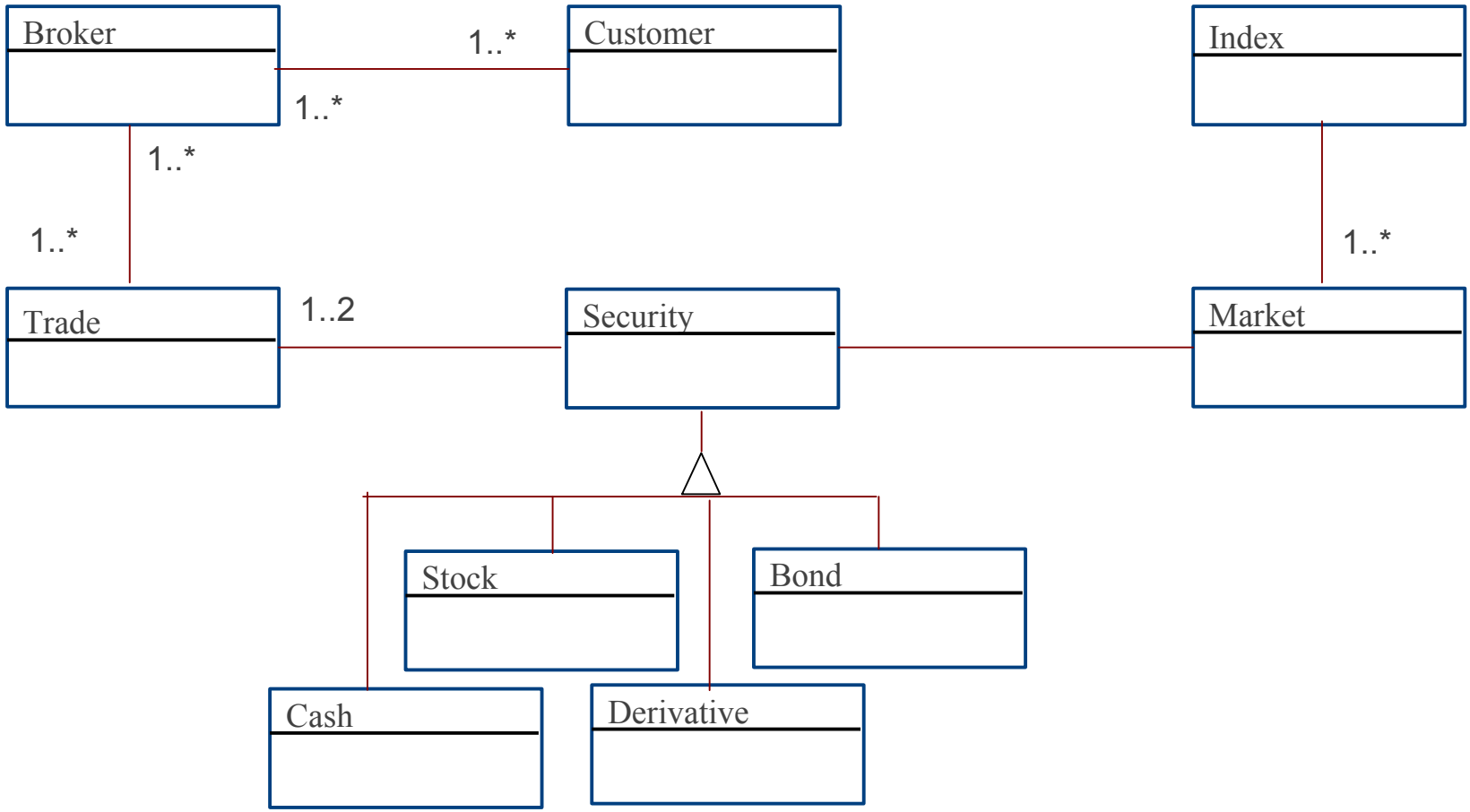
- Mimics its real world counterpart
- Provides a reusable component
- Couples process with related data
- Isolates business rules in logical place
- Models complexity
- Extensible when domain changes
- Maps to business person's view

A Sample Business Object

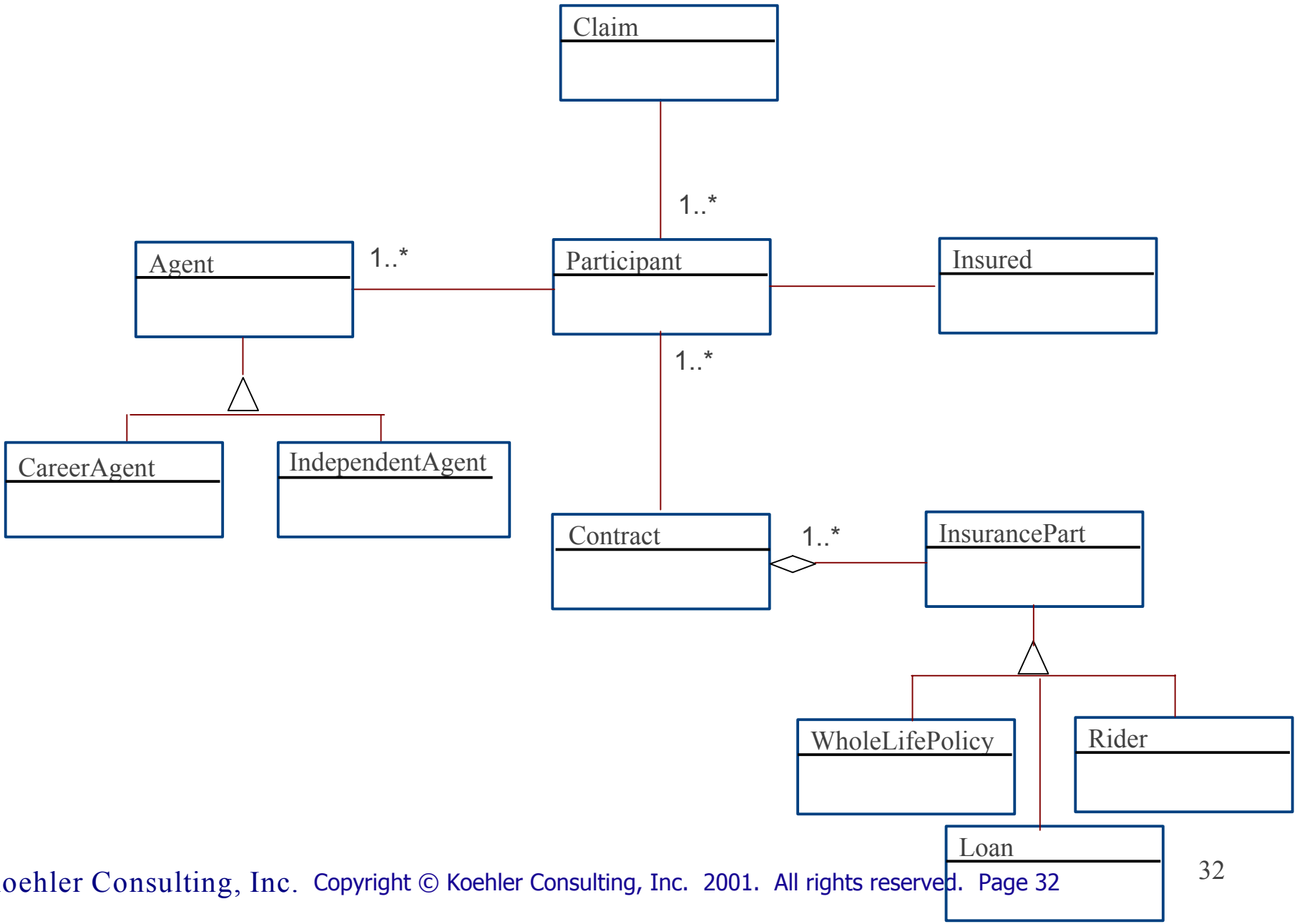


Finance Industry

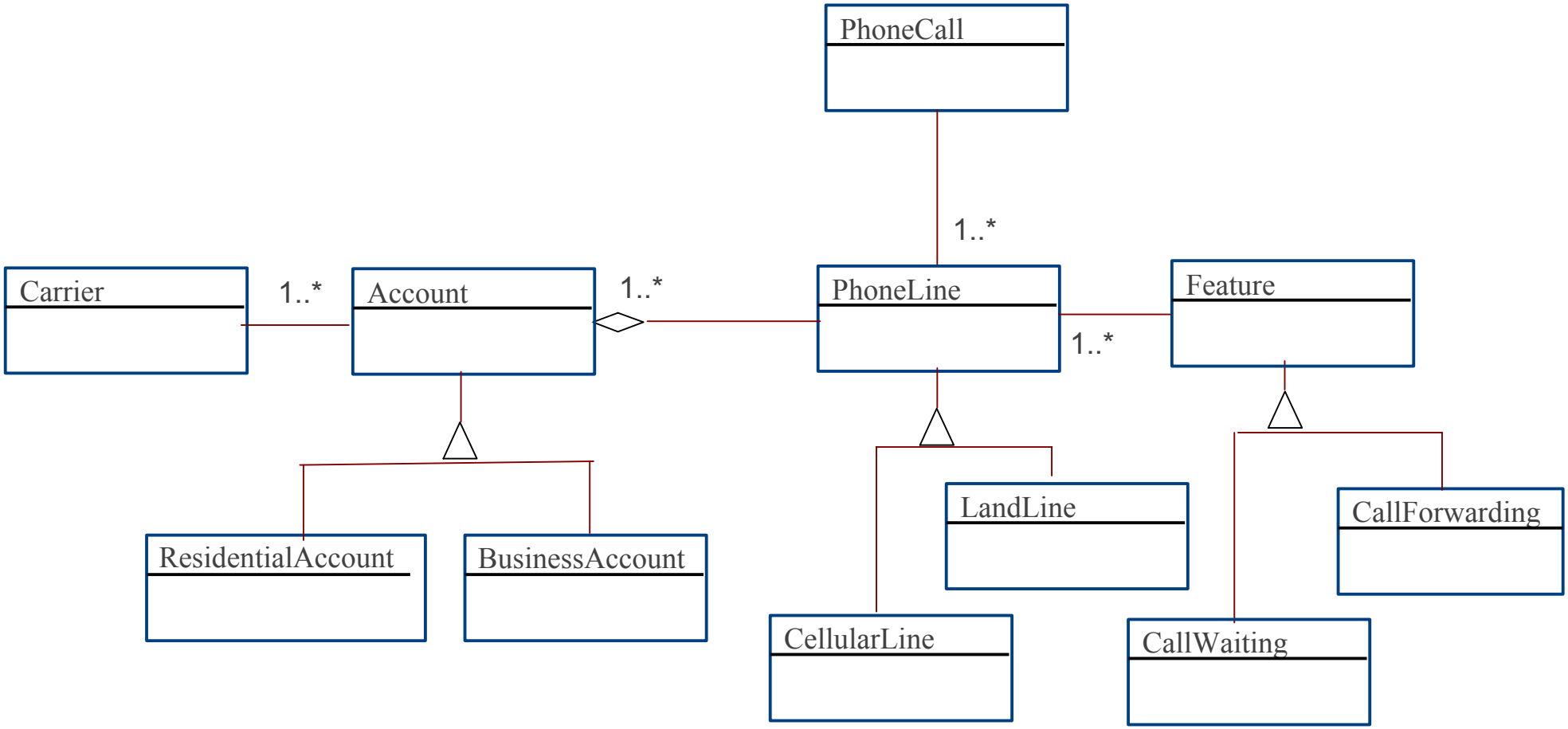




Insurance Industry

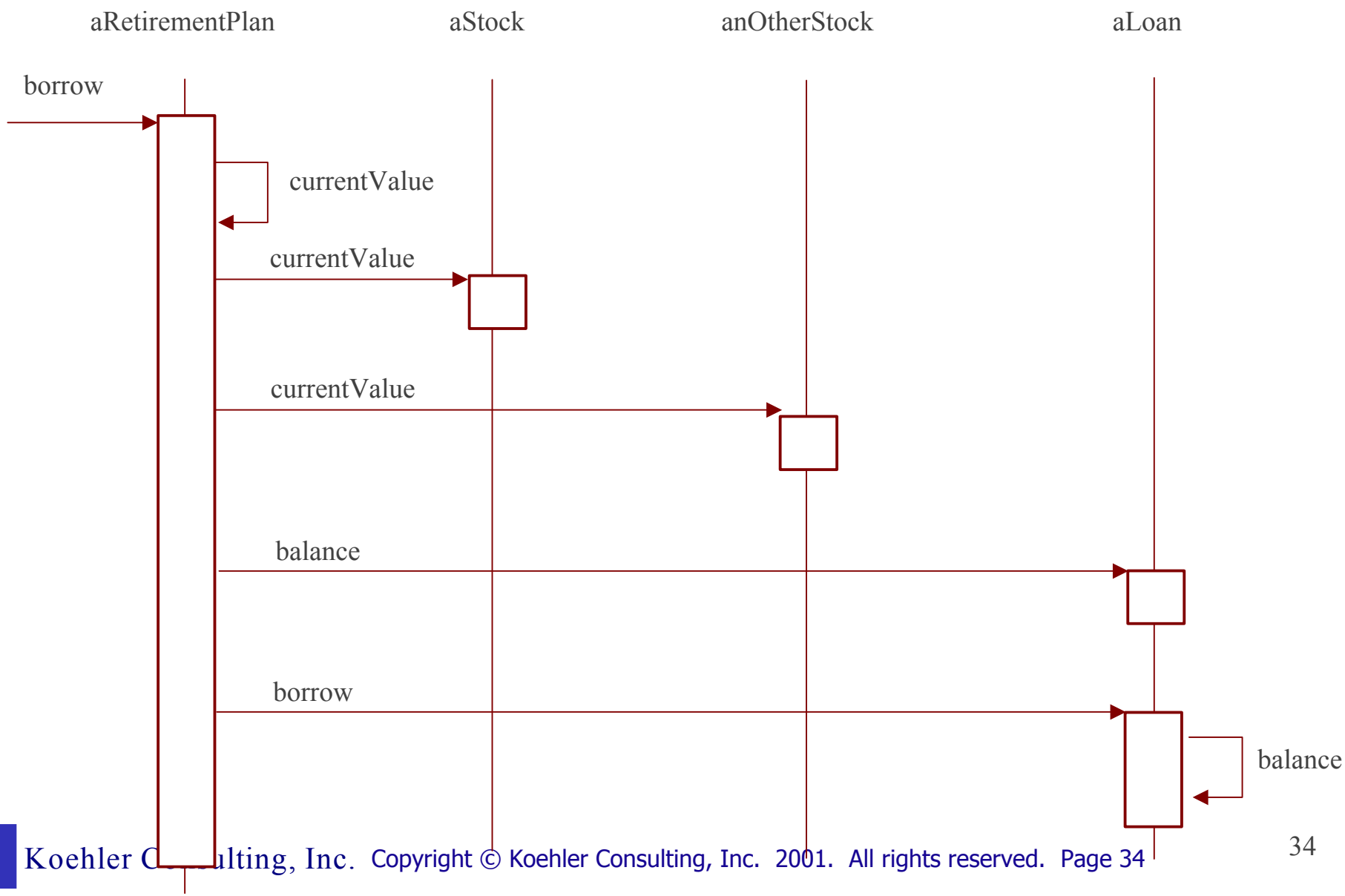


Telecommunications Industry



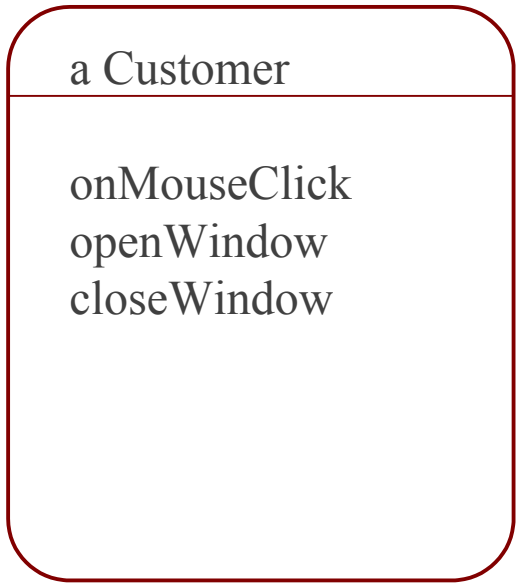
Business Object Interactions

Object Interaction Diagram



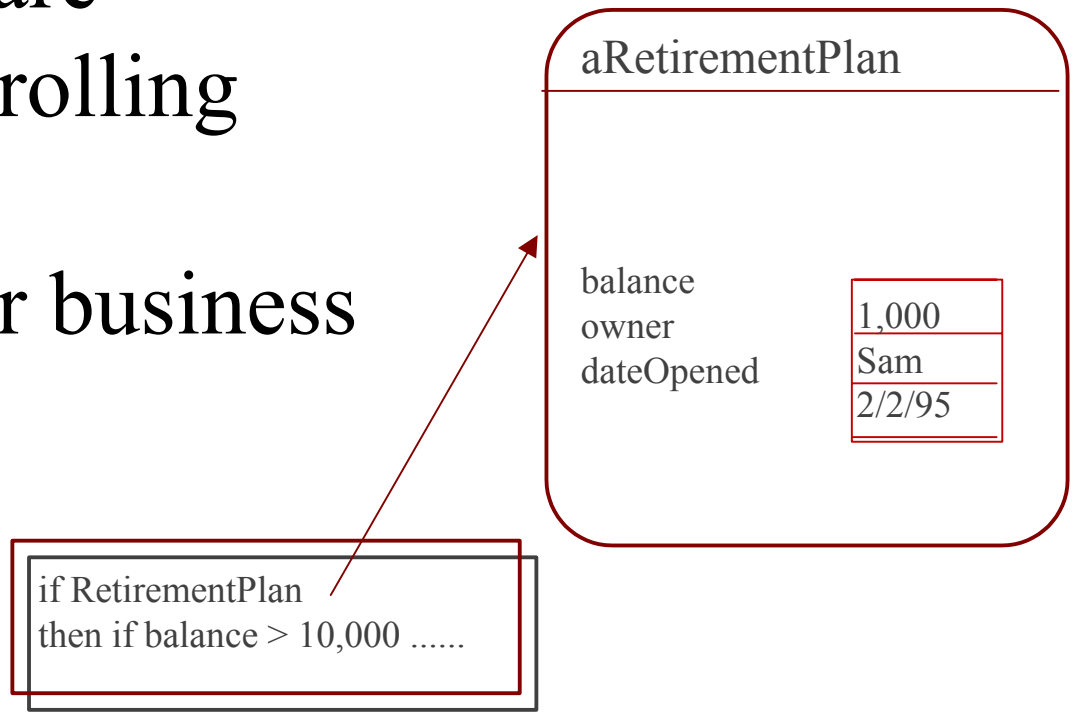
Beware of Business Objects Like ...

- Window objects



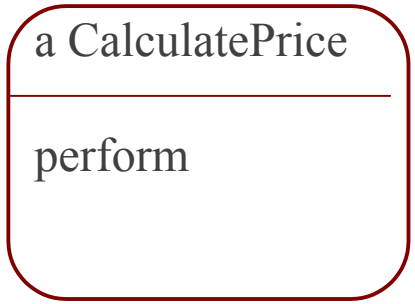
Beware of Business Objects Like ....

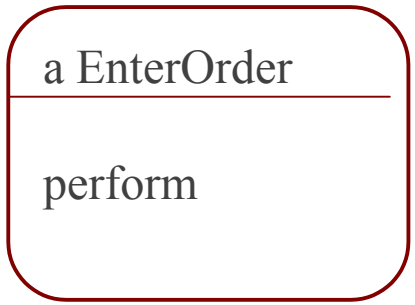
- Passive objects that are manipulated by controlling processes
- Objects missing their business rules

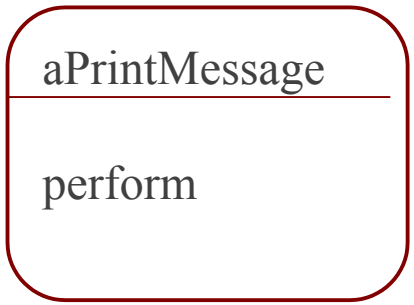


Beware of Business Objects Like .....

- Procedural objects

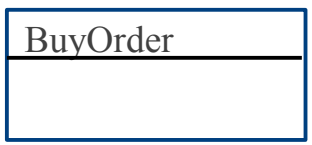






Process Objects

- An alternative to direct messaging between domain business objects
- Objectifies a business processes
- Can track the state of the process
- Facilitates reengineering of the process, at a later time



Process Objects (cont.)

- Process objects are appropriate models for business processes or tasks that:
  - create and destroy objects
  - are made up of sub-tasks
  - have special cases
  - involve business rules
  - take place over period of time -- can track state

reference: David Taylor "*Business Engineering with Object Technology* -- 1995 John Wiley & Sons

Process Objects (cont.)

- Be careful not to overdo / misapply
- Be wary of distributing knowledge outside of domain object

Process Object Sample

Mutual Fund Exchange Process:

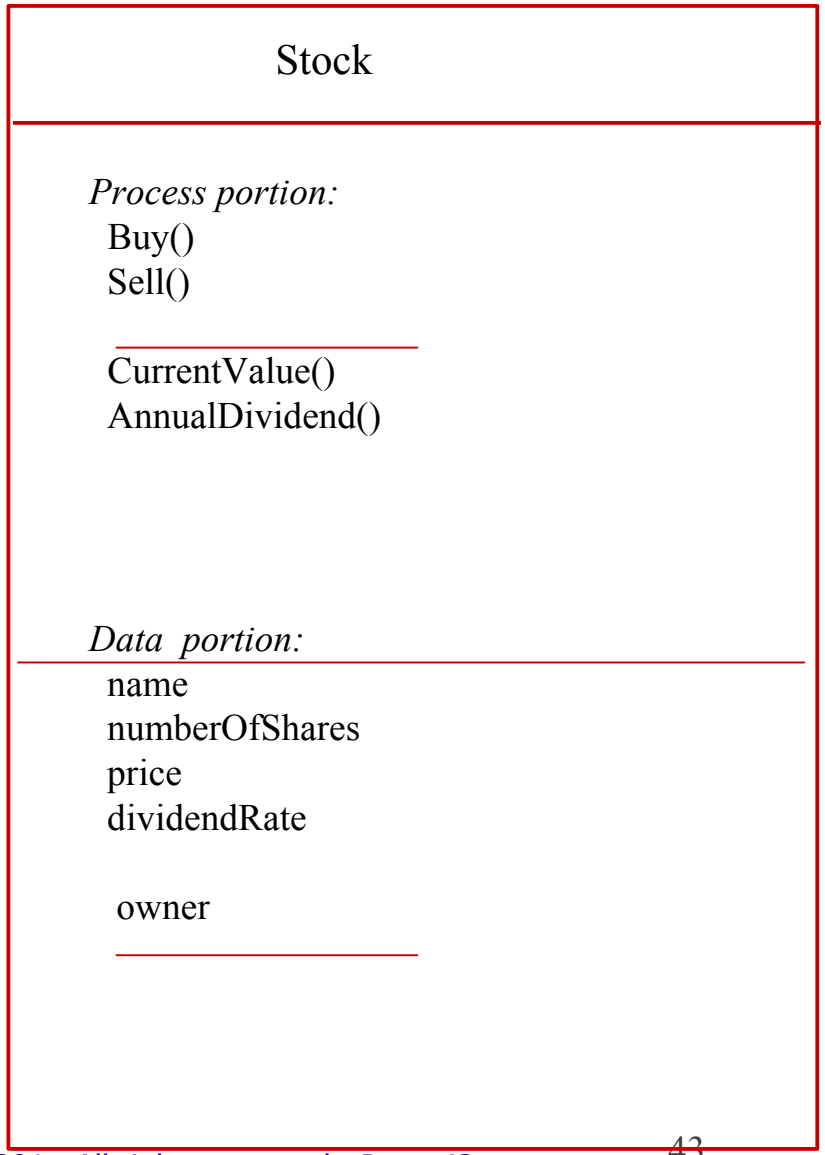
- Account number?
- Verify address, please....
- Select fund to exchange from
- Select fund to exchange to
- Amount?
- Confirm
- Execute
- Please note confirm number....

OT Principles for Framework Design

- Classes
- Instances
- Messages / State
- Association / Composition
- Inheritance
- Polymorphism
- Encapsulation

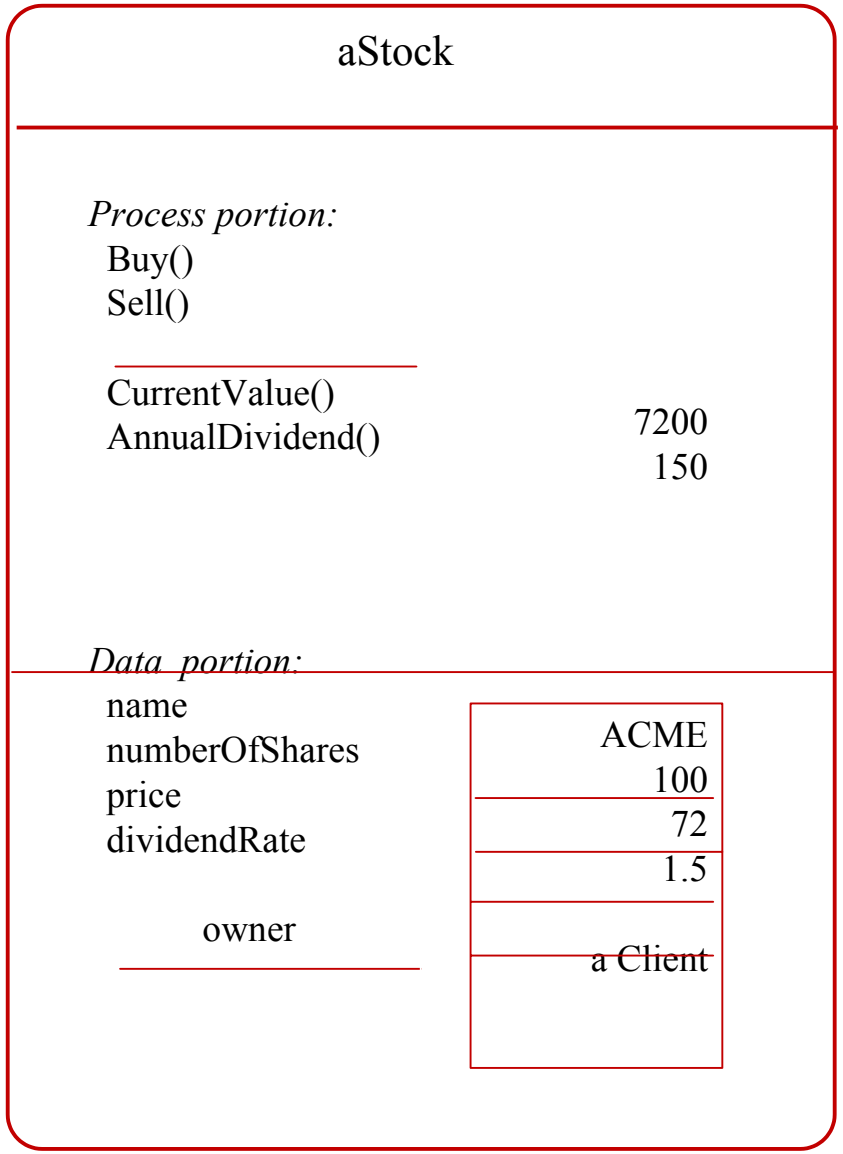
Class

- A class offers a description for an object -- a template
- Defines a model



Instance

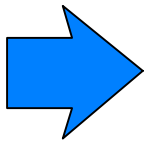
- An object is an instance of a class
- Has state
- Responds to messages

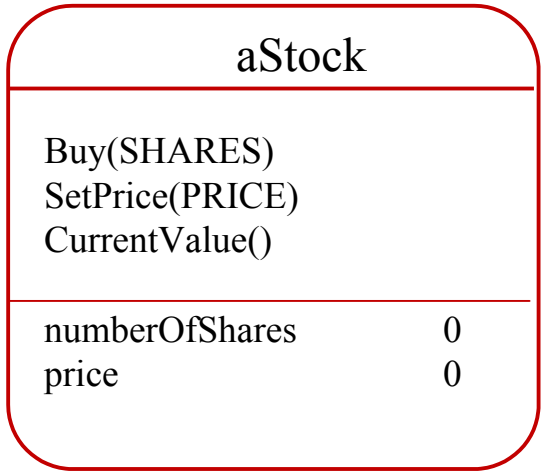


Messaging / State

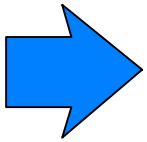
- Objects respond to messages
- Objects change state

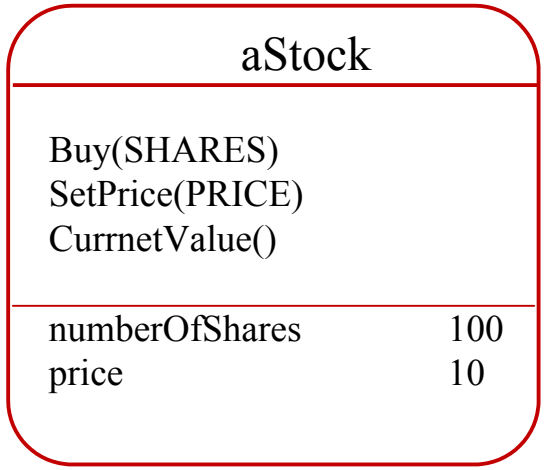
STOCK = new Stock



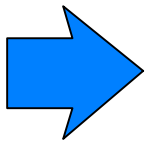


STOCK->Buy(100)
STOCK->SetPrice(10)





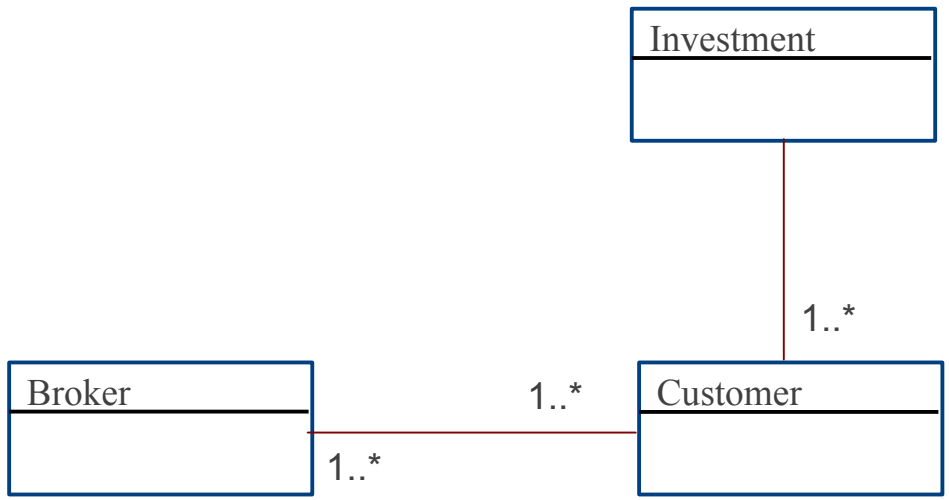
CURRVAL = STOCK->CurrentValue()
print CURRVAL



1000

Association

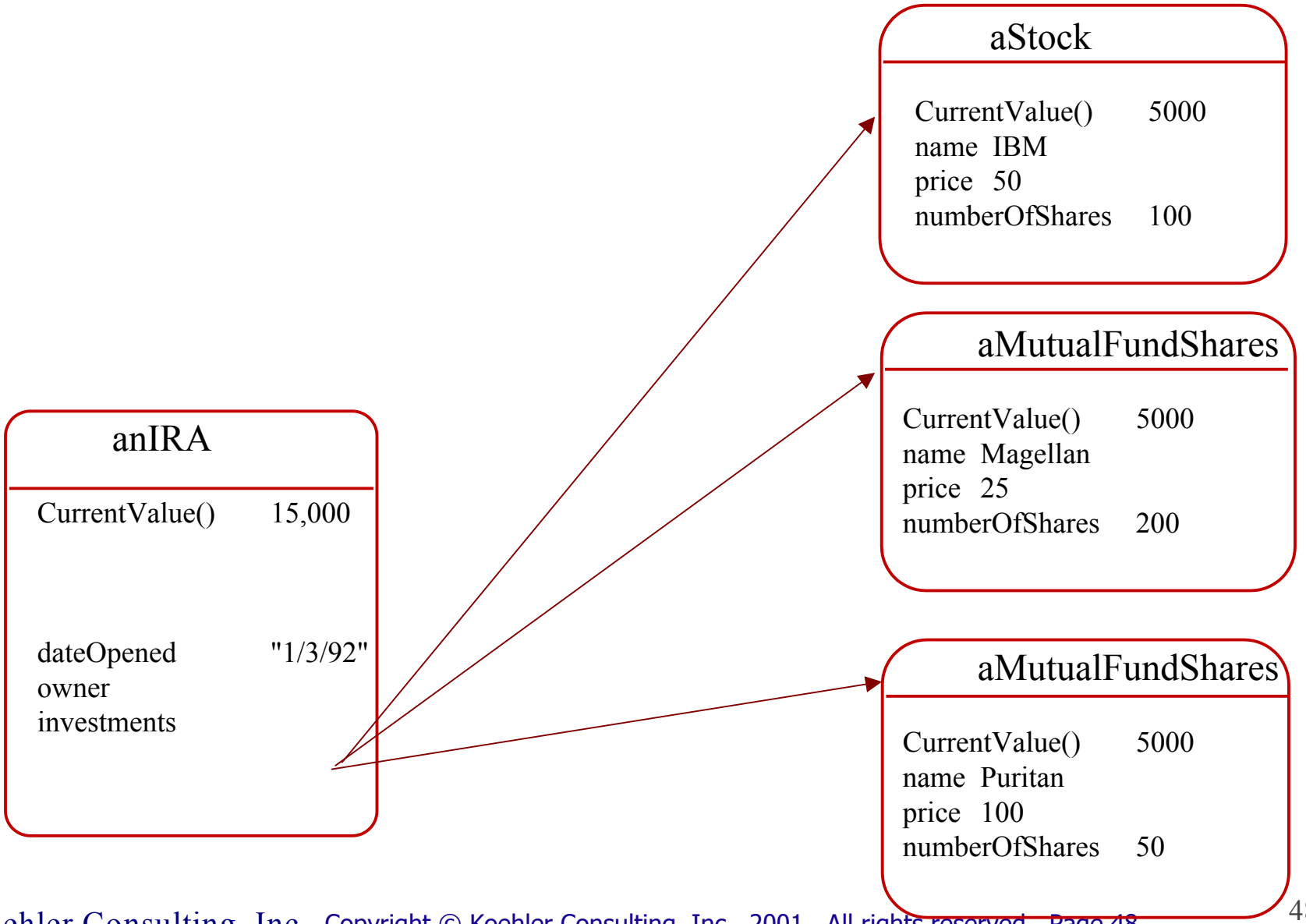
- Models relationships between objects
- Specifies a cardinality / bi-directional

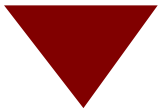


Composition

- a.k.a. Aggregation
- Part-whole relationships
- Objects embedded within other objects
- Much tighter coupling, one object "owns" another
- Facilitates modeling complexity

Composition (cont.)





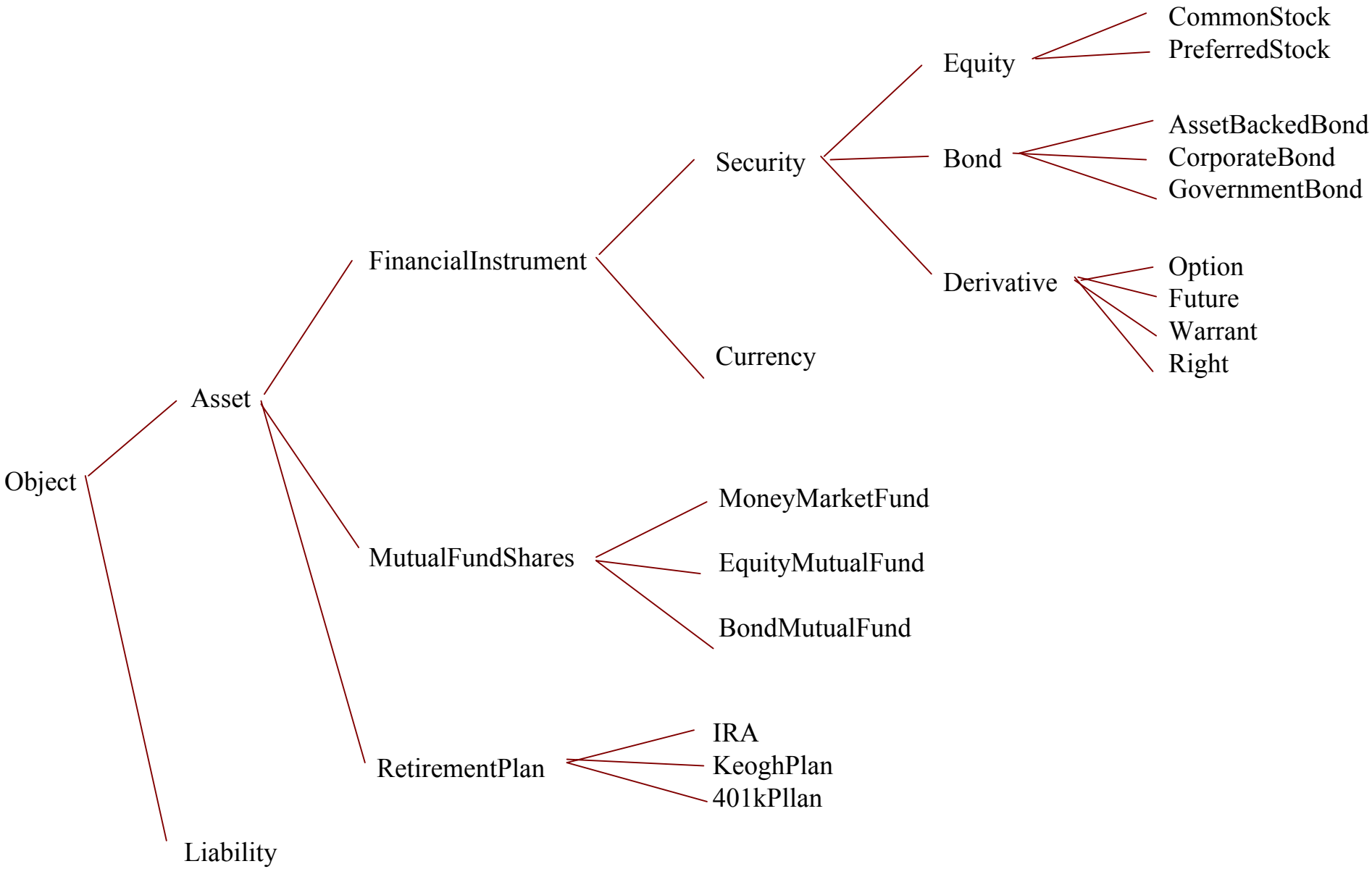
Inheritance

- Provides method of handling shared behavior
- Many business objects are "sort-of" like something else
- Subclasses provide the specialization of the generalized behavior defined in superclasses.
- Facilitates extensibility

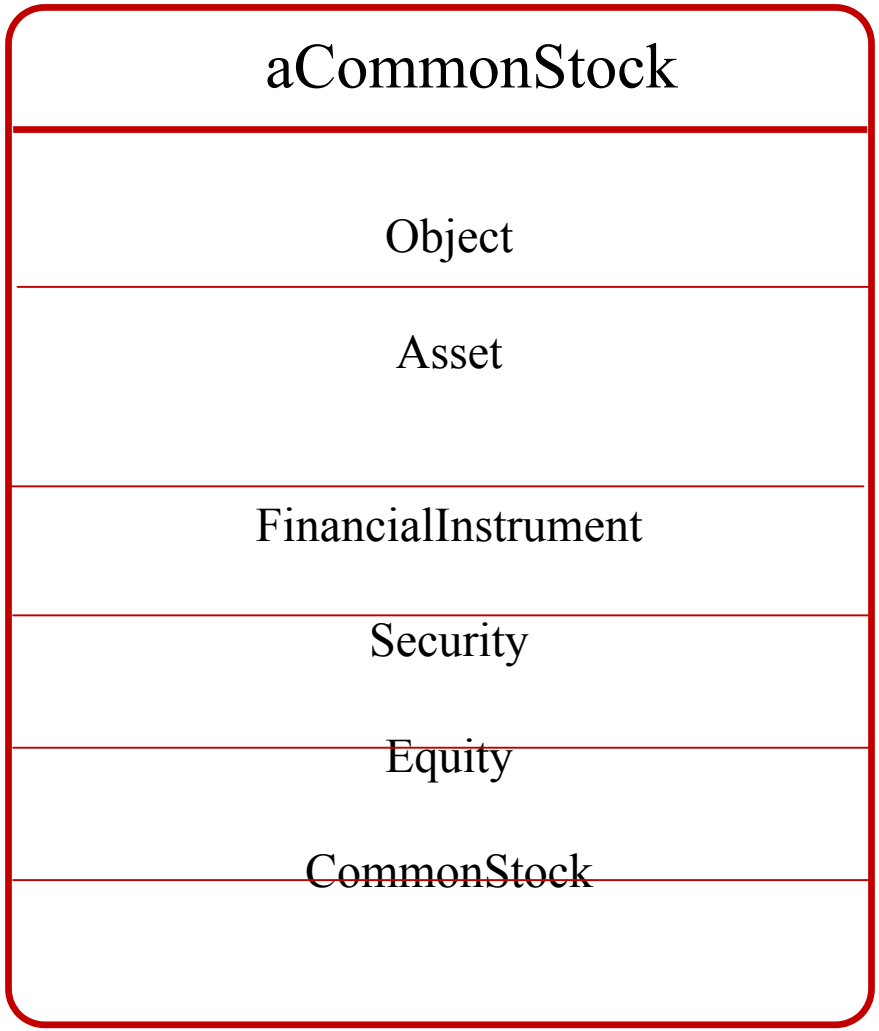
Attribute / Behavior Matrix

	CommonStock	MutualFundShares	CorporateBond	Option	Currency
Attributes:					
Current Value	units * price * factor	units * price * factor	units * price * factor	units * price * factor	units * price * factor
Units	NumberOfShares	NumberOfShares	specified	specified	specified
NumberOfShares	specified	specified	N/A	N/A	N/A
Factor	1.0	1.0	calculated	1.0	1.0
Price Today	specified	Calculated	specified	specified	1
Price Previous	specified	Calculated	specified	specified	1
PriceChange	PriceToday - PricePrevious	PriceToday - PricePrevious	PriceToday - PricePrevious	PriceToday - PricePrevious	PriceToday - PricePrevious
Coupon Rate	N/A	N/A	specified	N/A	N/A

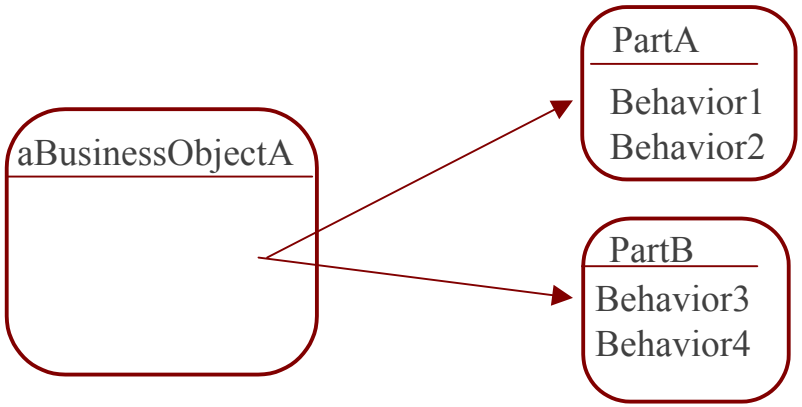
Class Hierarchy



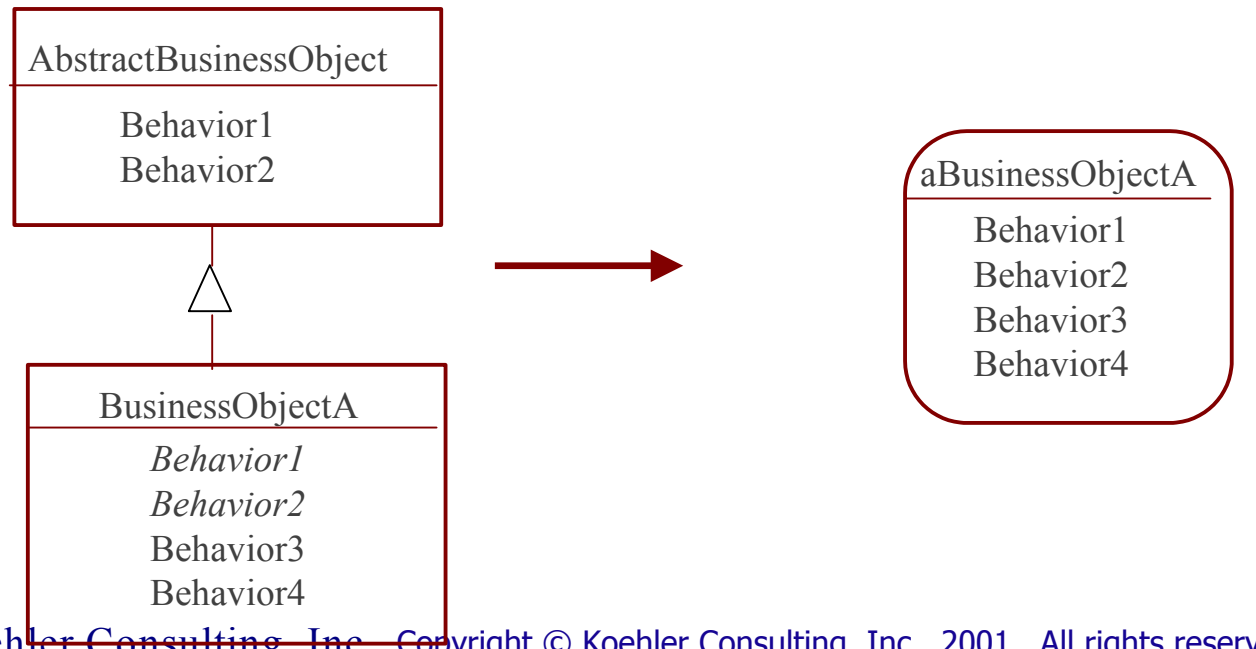
Object Anatomy



Inheritance Versus Composition



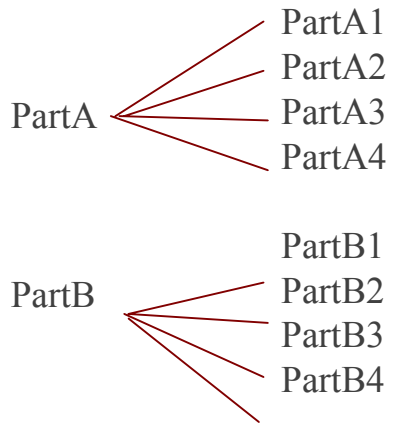
Composition

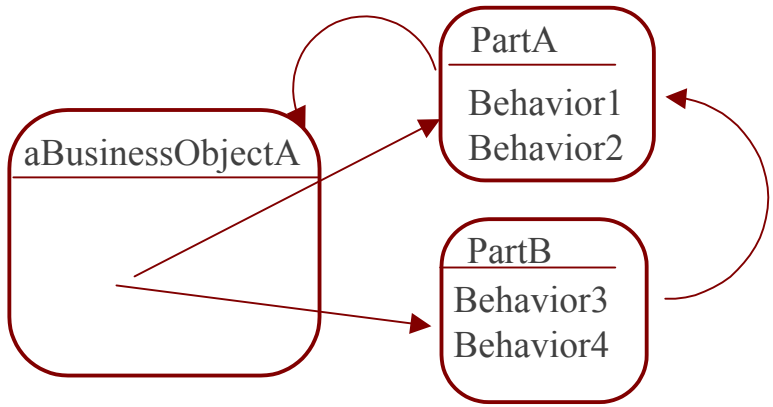


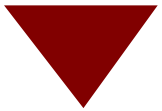
Inheritance

Inheritance Versus Composition (cont.)

- Business objects often cannot be broken down into well-defined parts
- Composition approach can often lead to many little hierarchies
- Parts are often not very independent
- Deep hierarchies are often rigid -- avoid subclassing for minor differences





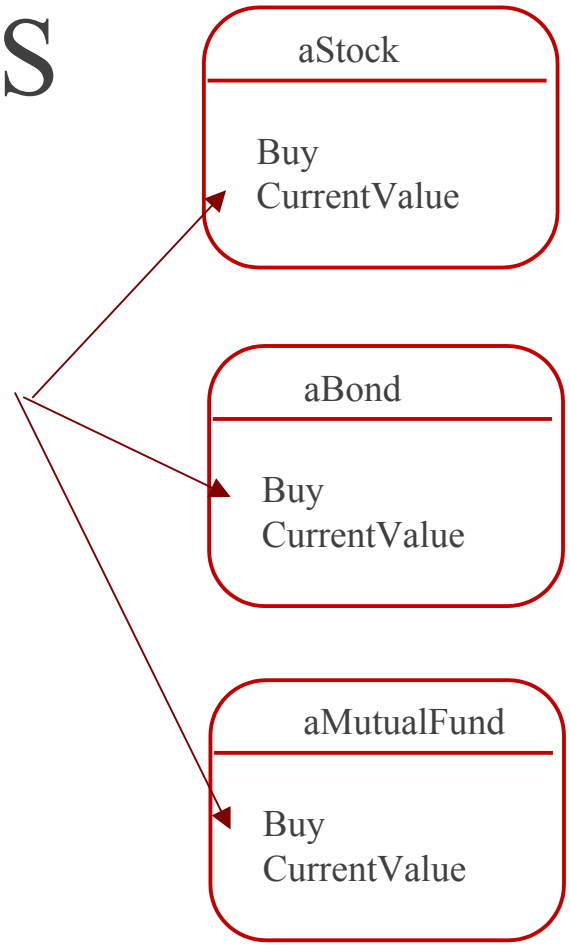


Polymorphism

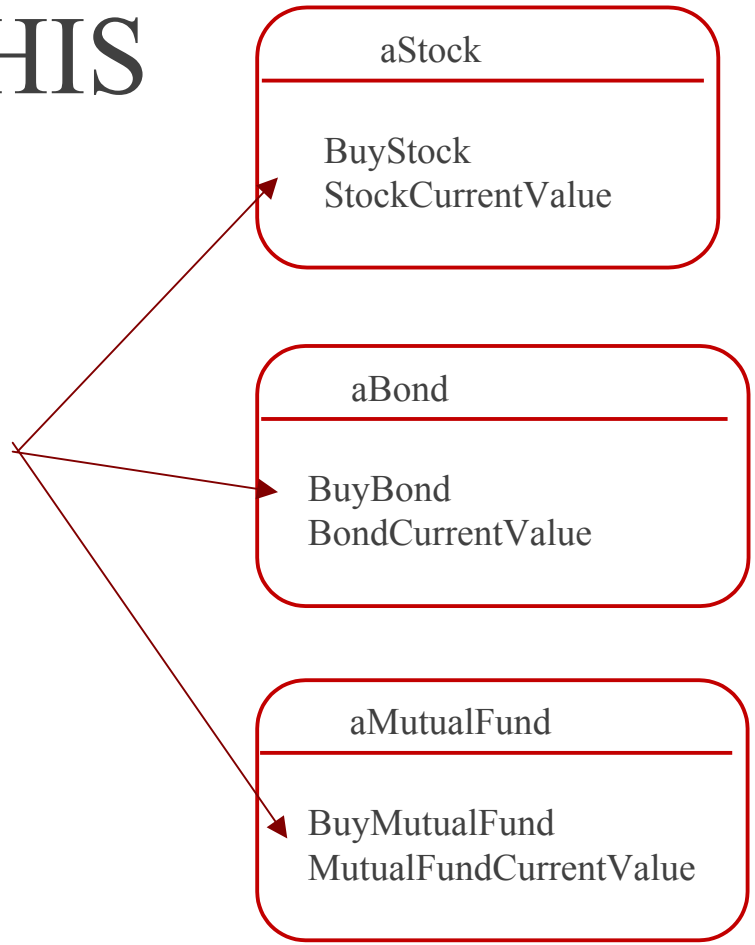
- Multiple business objects present a common interface for the same type of operation
- Replaces if / case statements present in procedural implementations
- Allows main application to be generic

Polymorphism (cont.)

WANT THIS

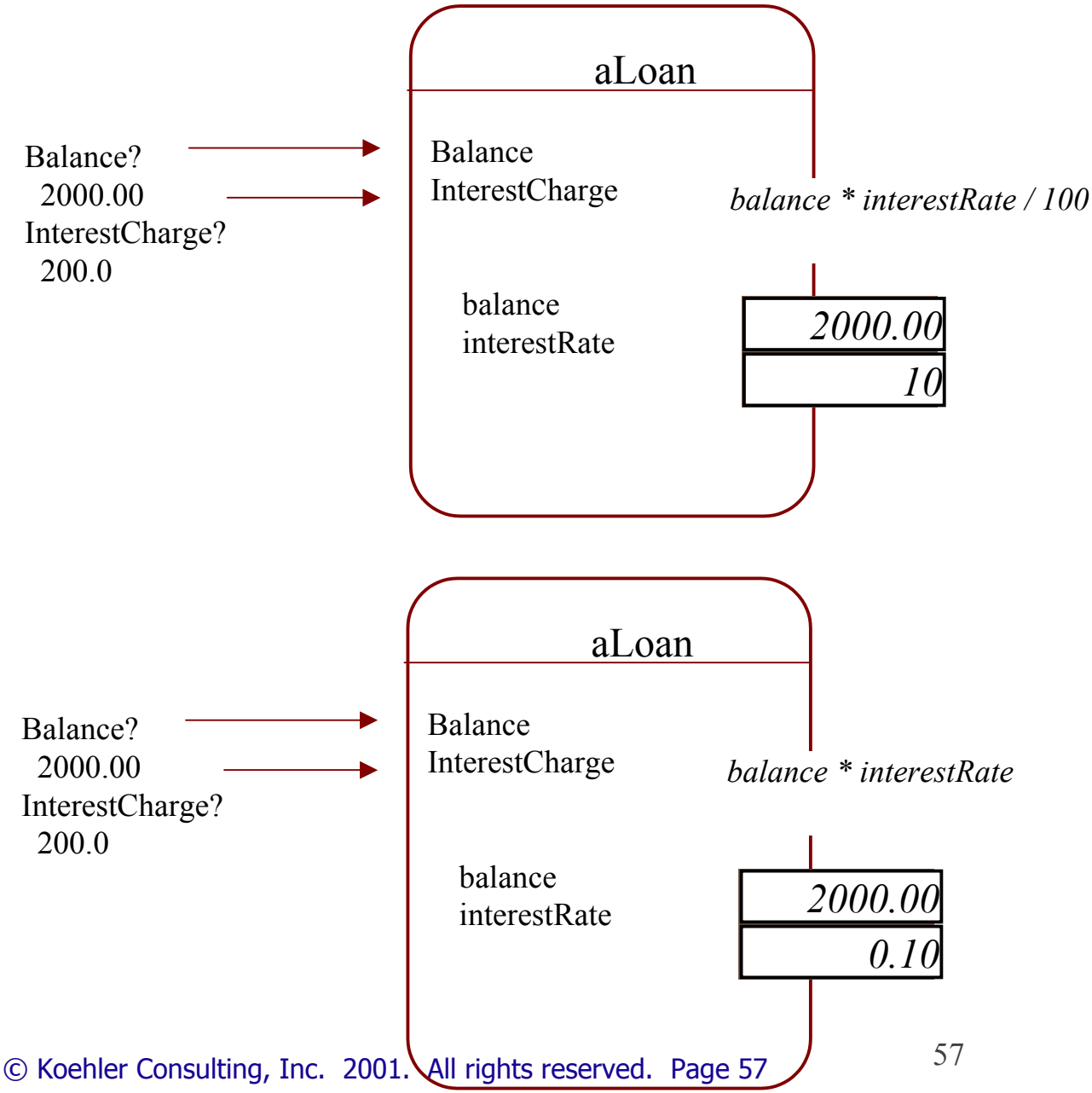


NOT THIS



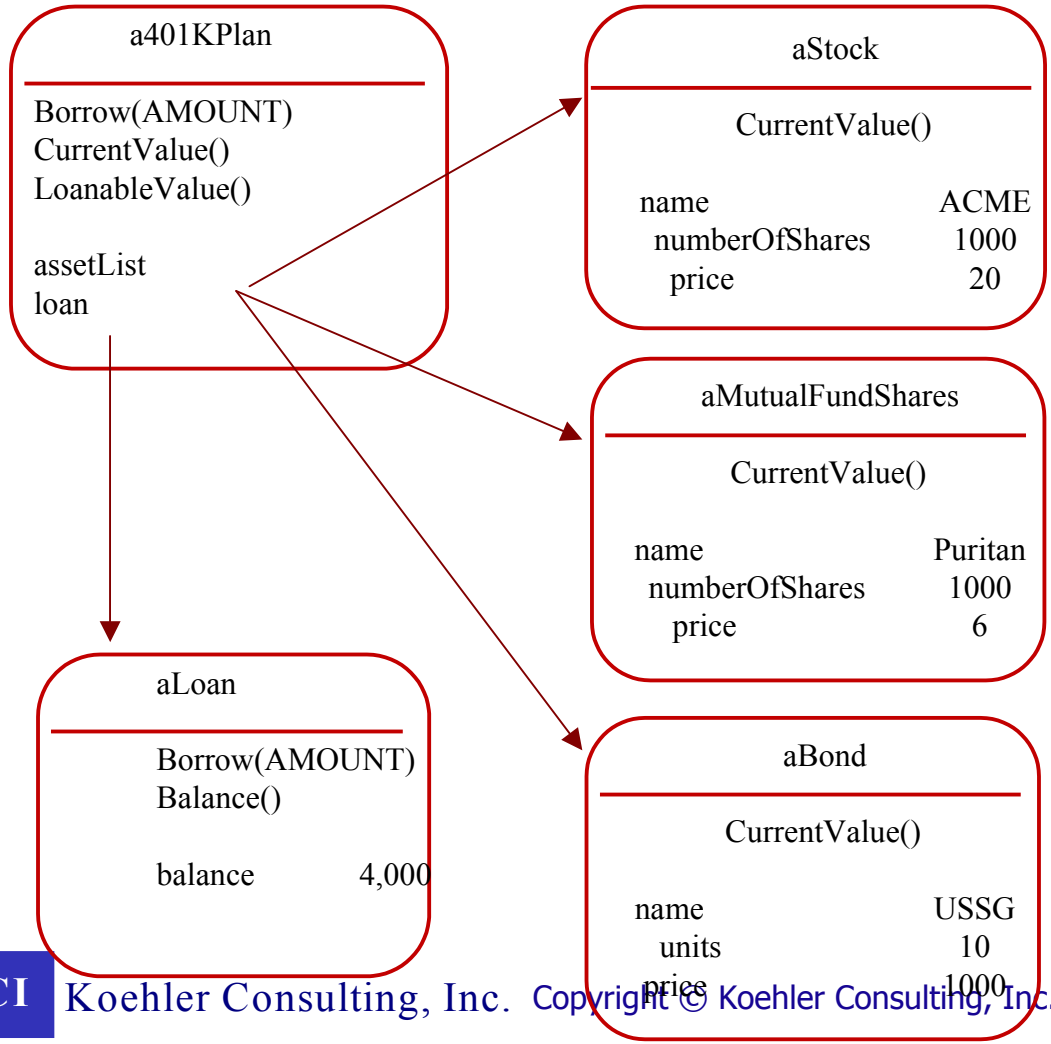
Encapsulation

- Encapsulation hides business object implementation details from users of the object



Sample Application

Facts: 401kPlan invested in three investments
32,000 net current value in the plan
4,000 existing loan balance
Scenario: client wishes to borrow an additional 15,000.

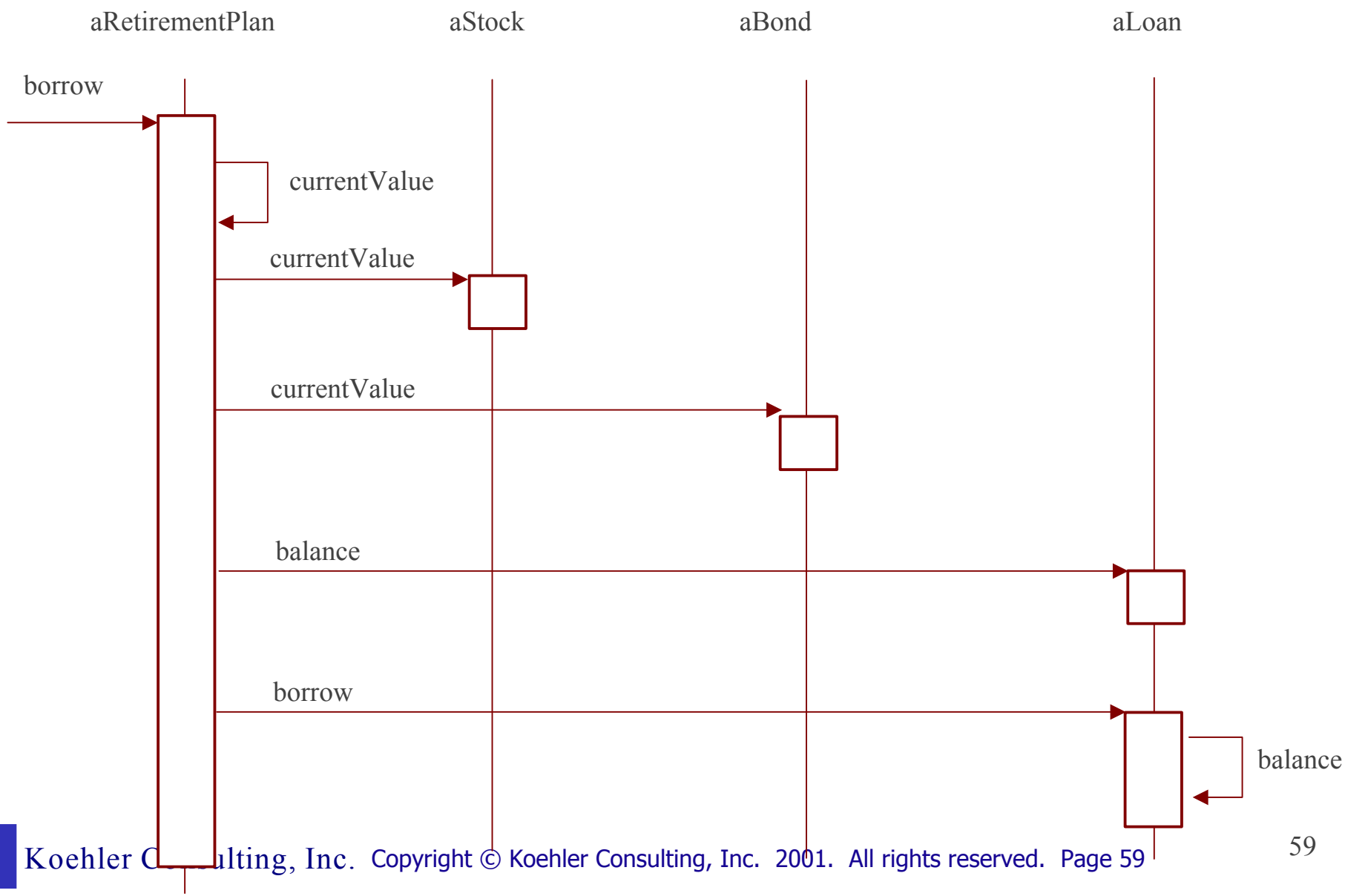


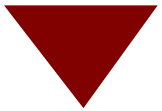
Pseudo-code

```
401KPlan::Borrow(AMOUNT)
        REALAMT = min(AMOUNT, LoanableValue())
        LOANAMT = LOAN->Borrow(REALAMT)
        return LOANAMT

401KPlan::CurrentValue()
        TOTVAL = 0
        iterate on assetList (ASSET)
          TOTVAL = TOTVAL + ASSET->CurrentValue()
        end iteration
        return TOTVAL

401KPlan::LoanableValue()
        TOTVAL = CurrentValue()
        // Can only borrow 50% of value
        NETVAL =(TOTVAL * .50) - LOAN->Balance()
        return NETVAL

Asset::CurrentValue()
        return price * numberOfShares

Bond::CurrentValue()
        return price * units

Loan::Borrow(AMOUNT)
        balance = balance + AMOUNT
        return AMOUNT
```


Sample Application (cont.)

Object Interaction Diagram

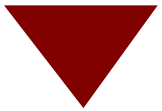




Design Patterns

- Design Patterns
  - Reusable design techniques
  - Recurring techniques found in well design OO software
  - Popularized by recent book
  - Provides a name and description of the technique
- Gamma, et al describe 23 patterns

reference Gamma, Helm, Johnson, and Vlissides. 1995. *Design Patterns: Elements of Resuable Object-Oriented Software.* Addison-Wesley Publishing Company.

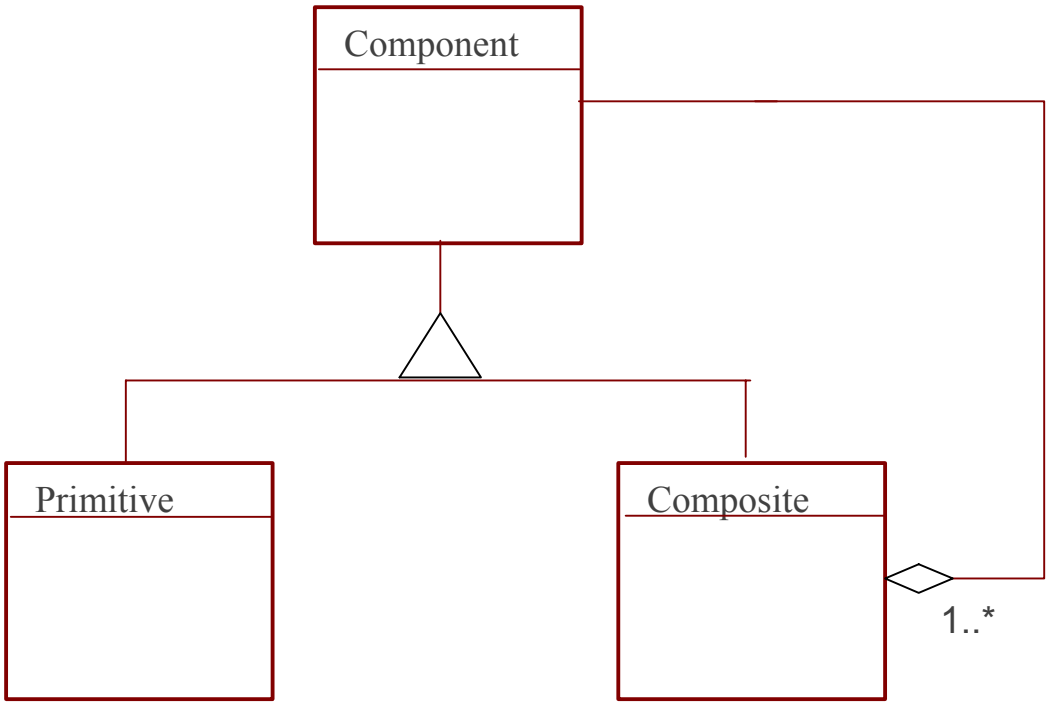


Design Patterns (cont.)

- Consists of
  - Name -- enhances design vocabulary
  - Intent
  - Problem -- with a context
  - Solution --
    - Structure
    - Participants
    - Implementation
    - Sample code
    - Consequences -- tradeoffs

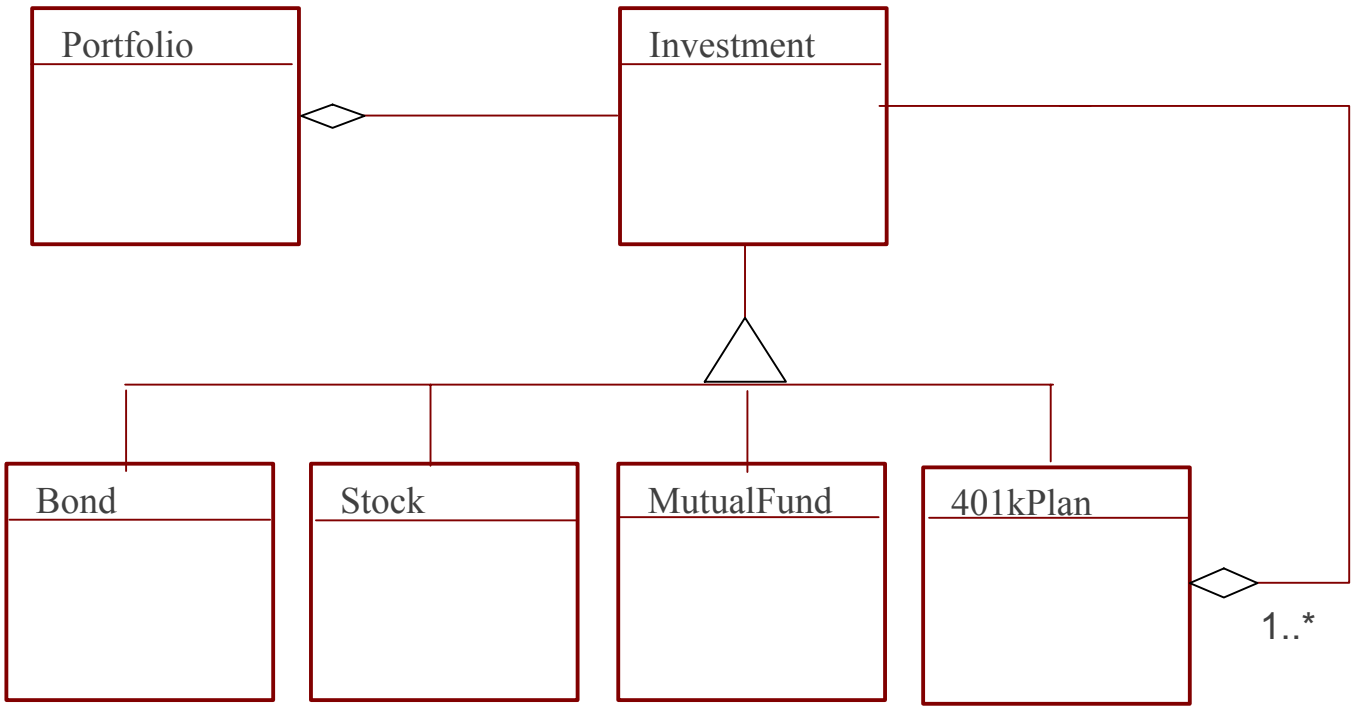
Design Patterns - Composite

Intent: The composite pattern allows client objects to treat individual objects and compositions of objects uniformly.

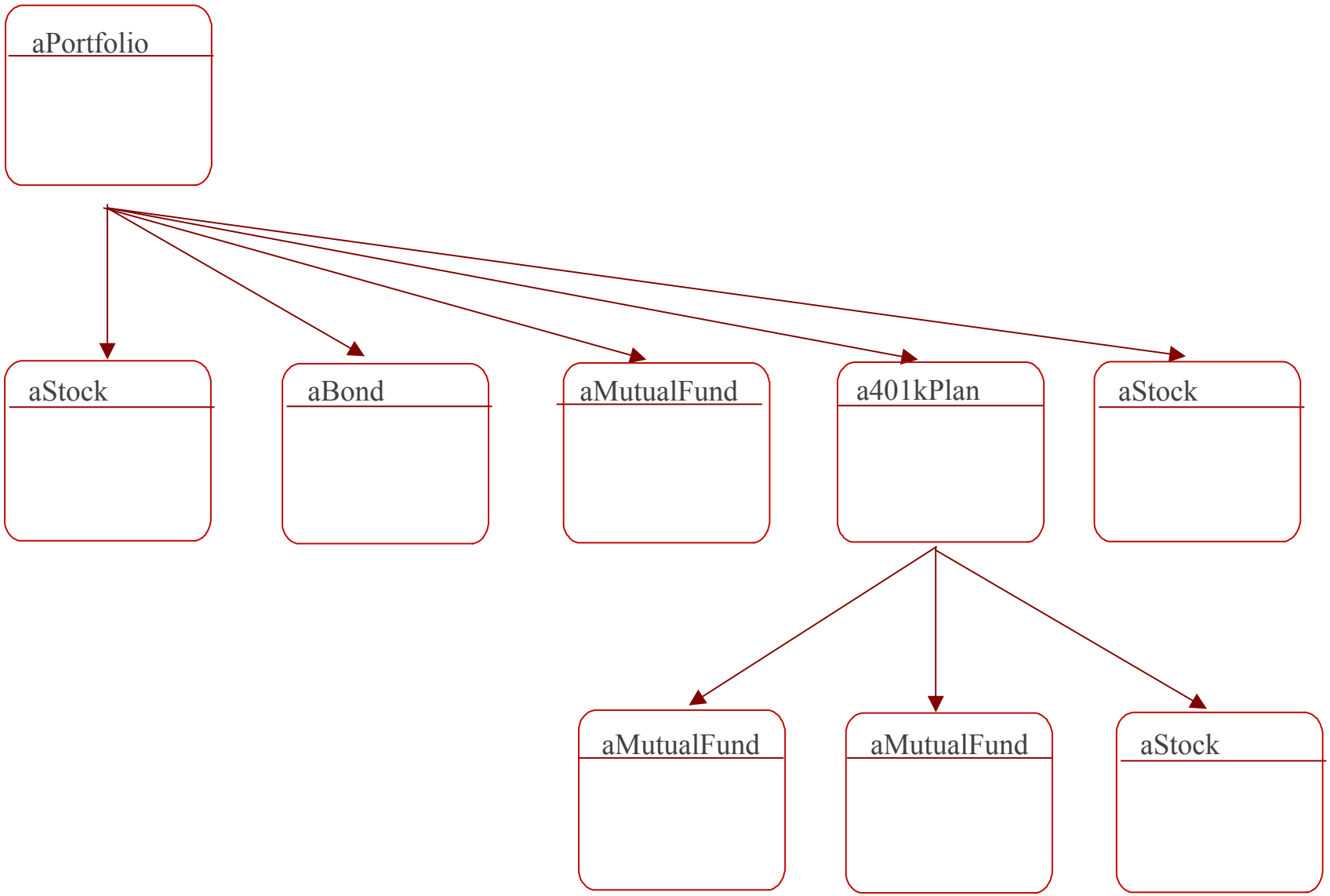


reference Gamma, Helm, Johnson, and Vlissides. 1995. *Design Patterns: Elements of Resuable Object-Oriented Software.* Addison-Wesley Publishing Company.

Design Patterns - Composite

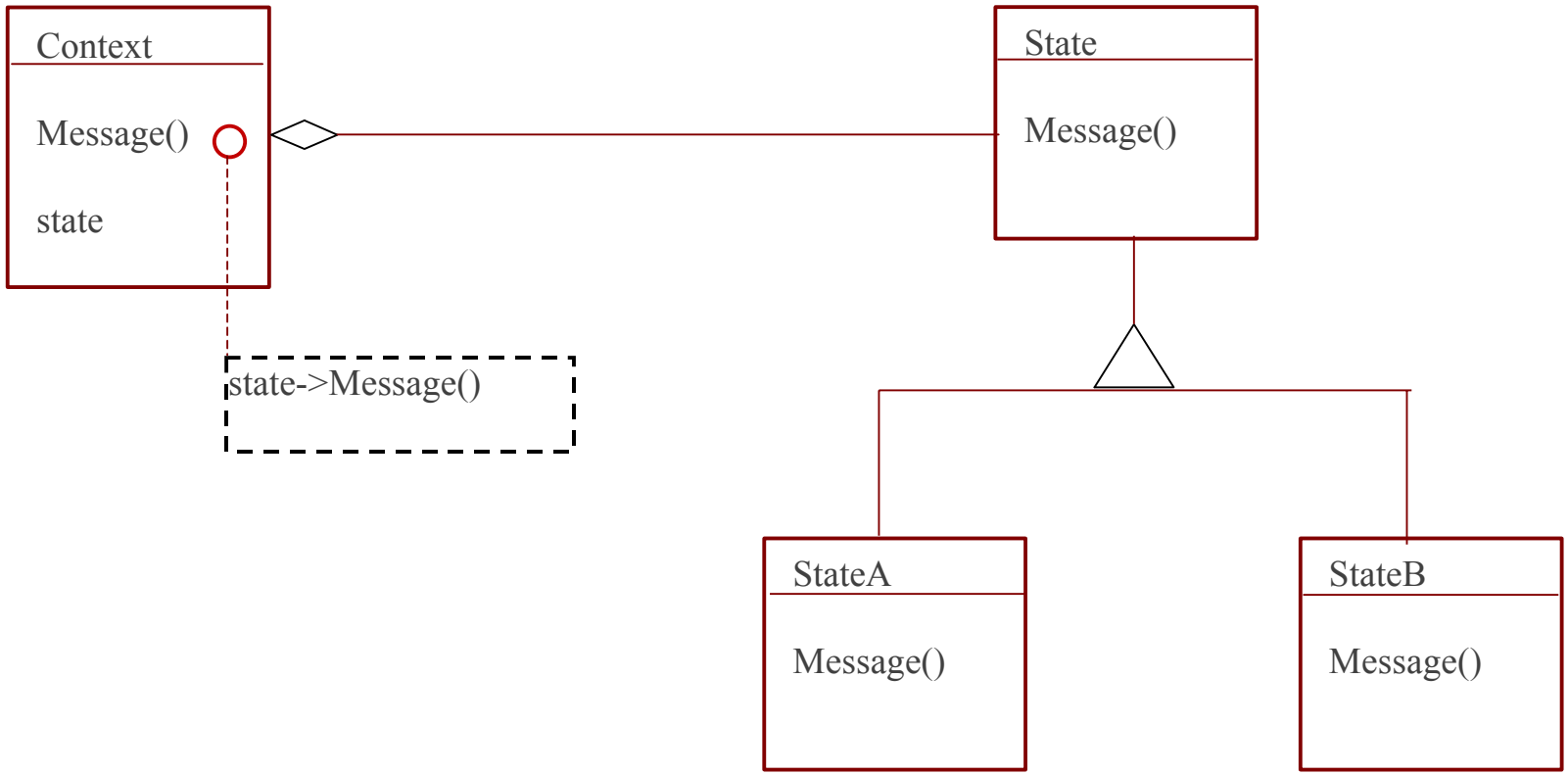


Design Patterns - Composite



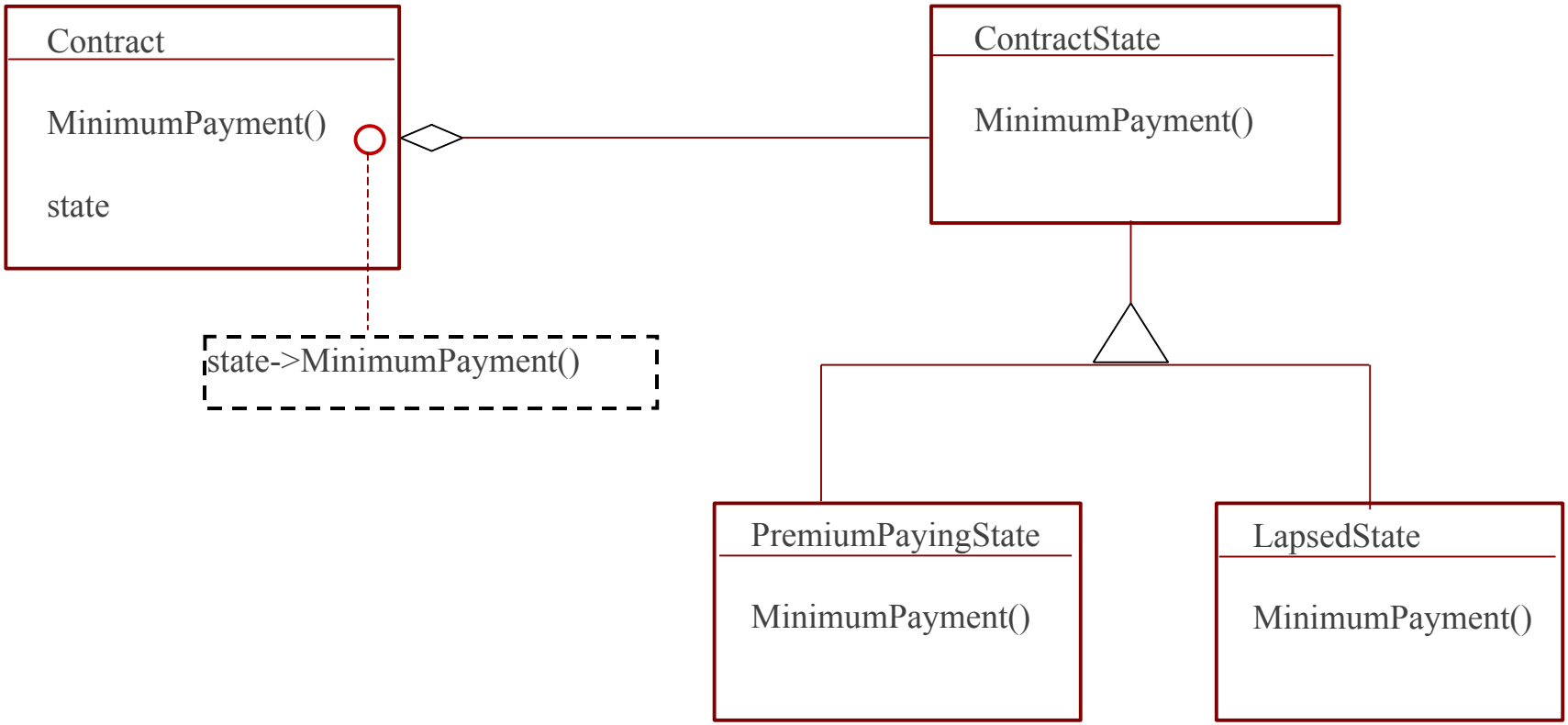
Design Patterns - State

Intent: Provide a method of allowing an object to change its behavior when its internal state changes, without changing the class of the object.

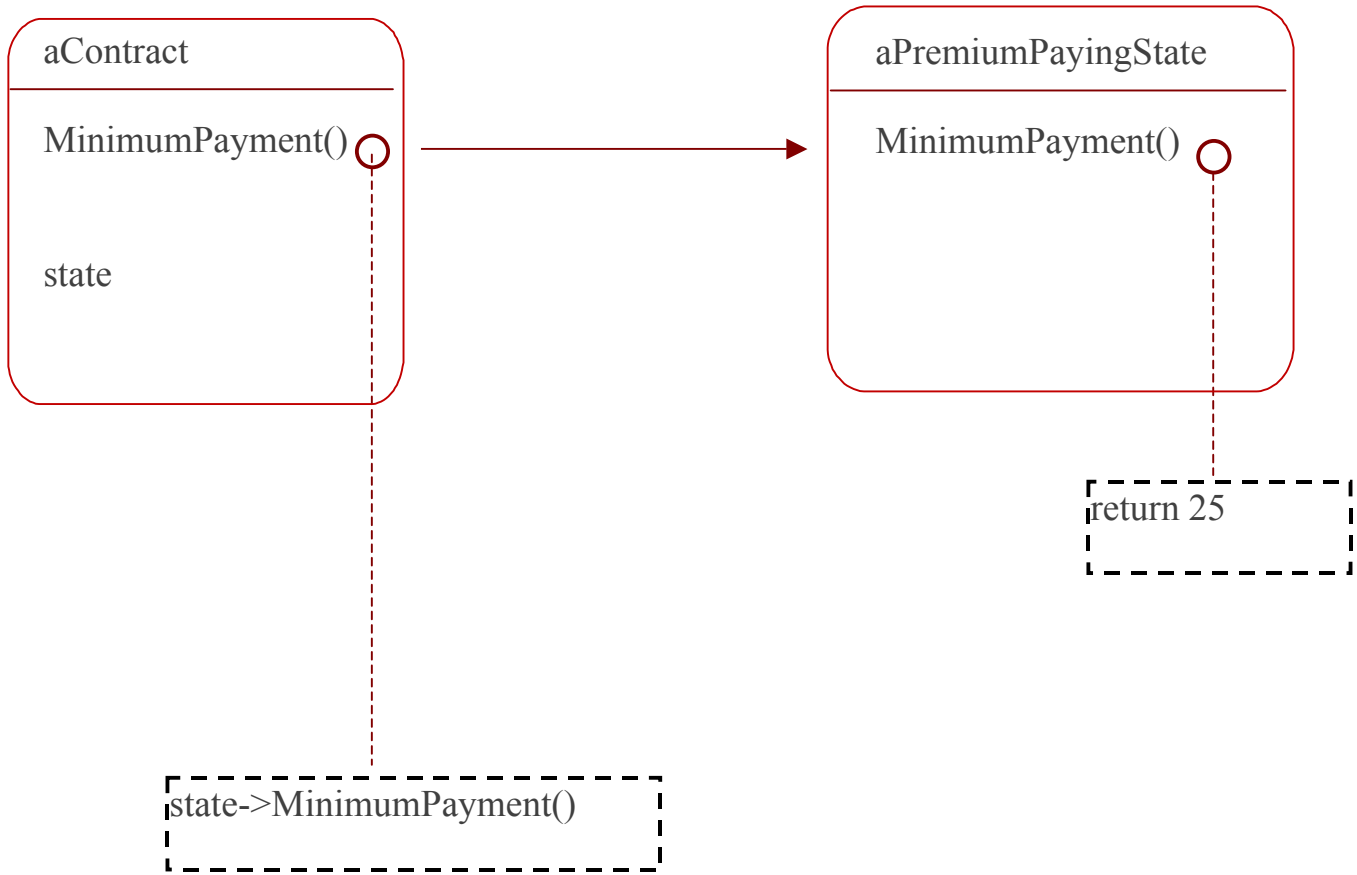


reference Gamma, Helm, Johnson, and Vlissides. 1995. *Design Patterns: Elements of Resuable Object-Oriented Software*. Addison-Wesley Publishing Company.

Design Patterns - State

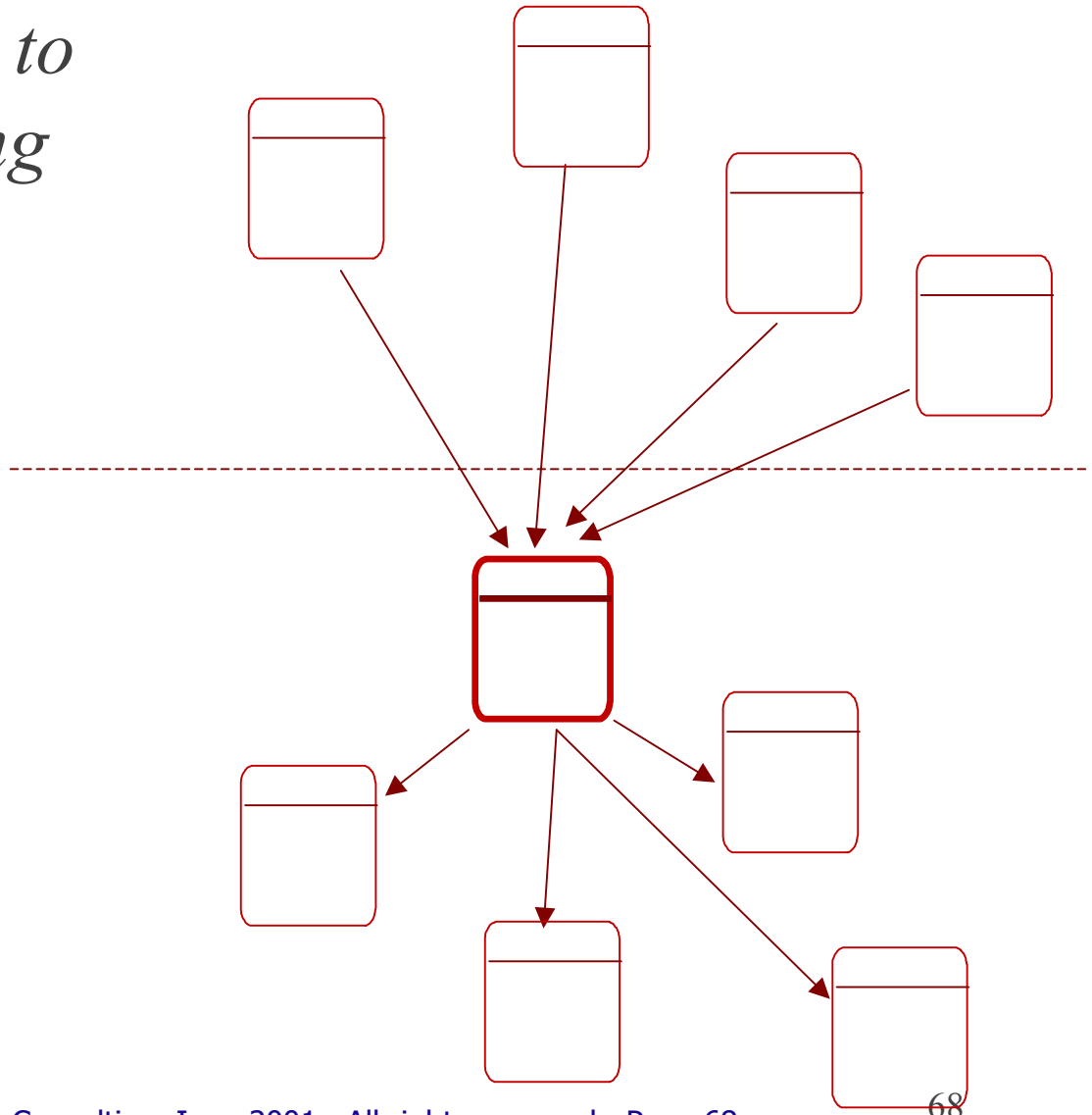


Design Patterns - State



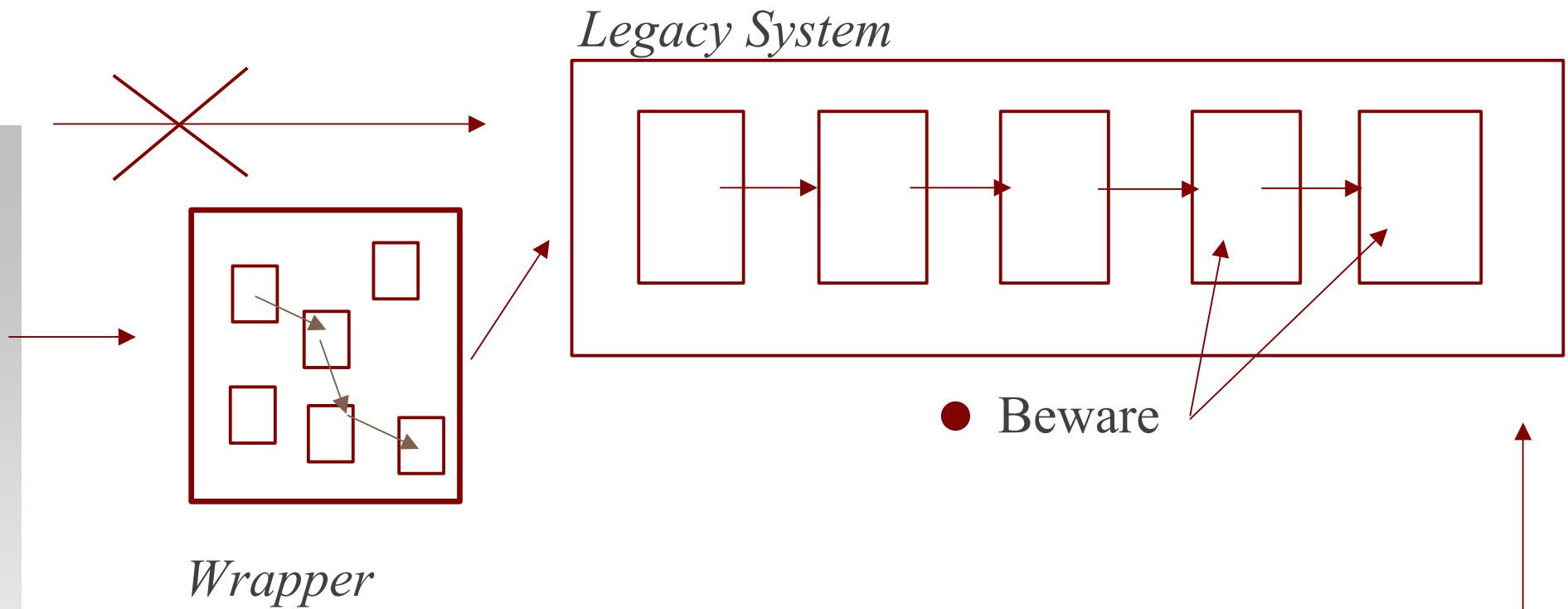
Design Patterns - Facade

Intent: Provide a central interface to a subsystem to minimize coupling among objects.



- Promotes subsystem independence

Framework Enabled Legacy Migration



- Beware
- Object front-end
- New development

Questions

