

Object Design

CSE 308: Software Engineering

Where We've Been

- [Analysis

- objective is to describe the purpose of a system

- [System design

- describes the system in terms of architecture

- we also select HW/SW and COTS components

Object Design

- [During object design, we identify additional objects and refine existing objects
- [Object design includes:
 - reuse
 - service specification
 - object model restructuring
 - object model optimization

System Continuity

- [Software systems are complex, subject to continuous change, and integrated under time pressure
 - this leads to interface and behavior mismatches
- [Solutions: precise specifications, reuse, and optimization

Plan of Attack

- [Overview of object design
- [Reuse concepts
- [Reuse activities
- [Managing reuse

An Overview of Object Design

Object Design Overview

- [Software development aims to fill the gap between a problem and an existing machine
 - Analysis identifies objects representing problem concepts
 - System design applies abstraction and looks for reuse
- [Object design fills in the gaps: identifying new solution objects, adjusting reused components, and specifying detail

Object Design Activities

— [Reuse

- class libraries are selected for data structures
- design patterns are selected for solving problems

— [Interface specification

- subsystem services are specified as class interfaces
- produces a subsystem API

Specification

Identifying missing attributes & operations

Specifying visibility

Specifying types & signatures

Specifying constraints

Specifying exceptions

Reuse

Identifying components

Adjusting components

Identifying patterns

Adjusting patterns

Object Design Activities

— [Restructuring

- manipulation of system to increase reuse, etc.
- graph transformation on subsets of a model

— [Optimization

- addresses system performance requirements
- changing algorithms, adding redundancy, etc.

Restructuring

Revisiting inheritance

Collapsing classes

Realizing associations

Optimization

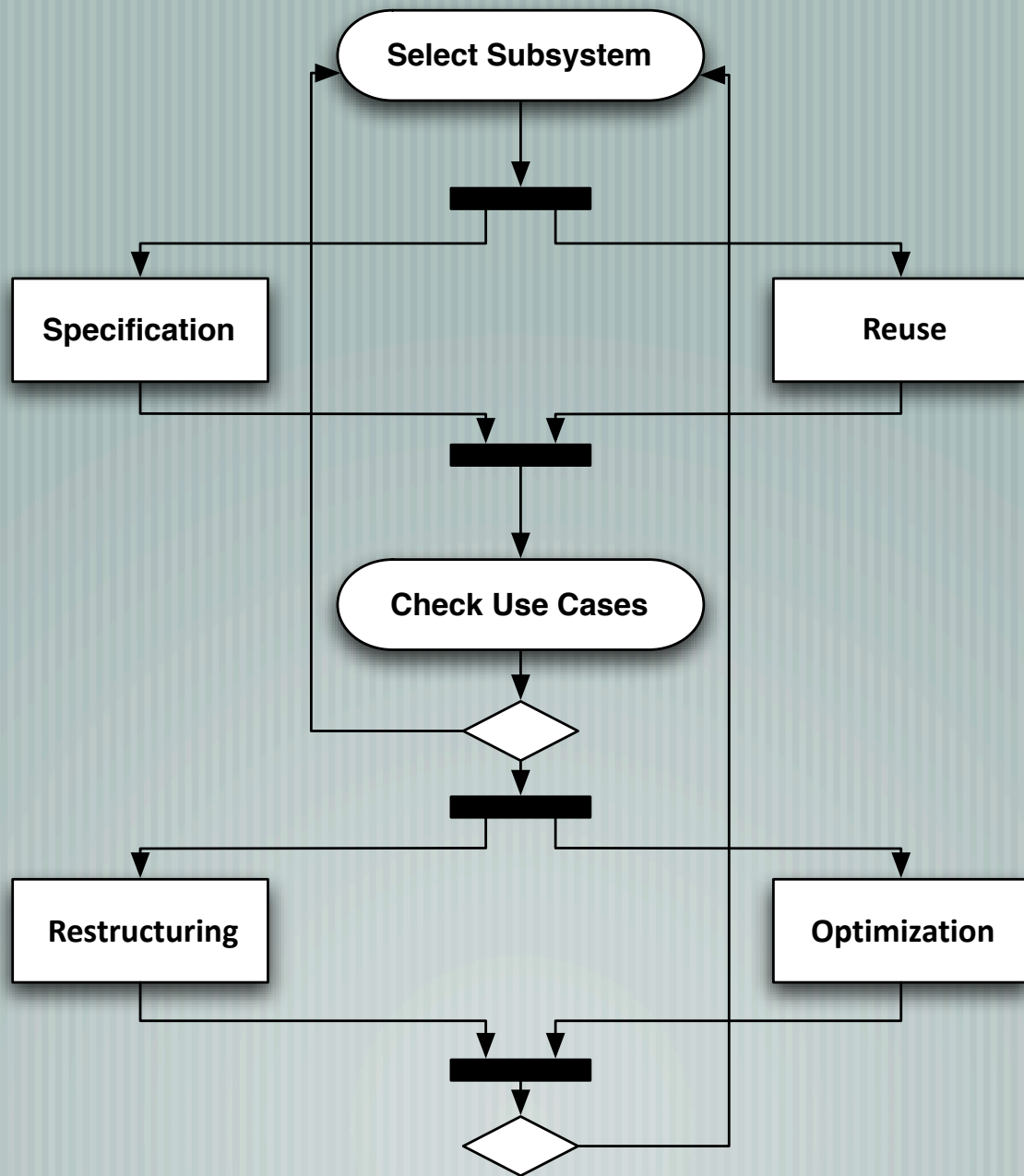
Optimizing access paths

Caching complex computations

Delaying complex computations

Ordering of Activities

- [Object design is not sequential
- [Interface specification and reuse generally occur first
- [Restructuring and optimization occur once model is stable
- [However, all activities occur in iterations



Reuse Concepts

Reuse Concepts

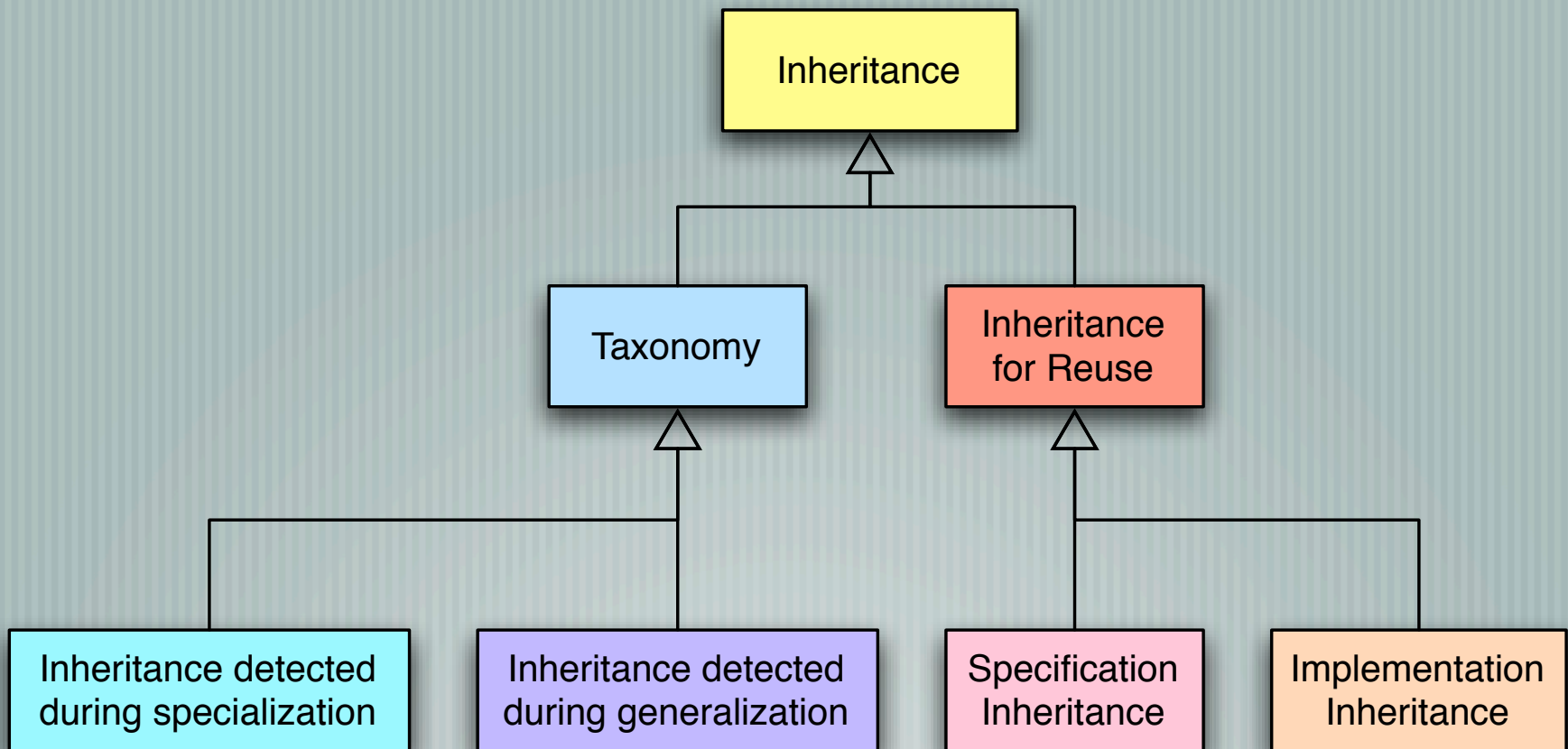
- [Application objects and solution objects
- [Specification inheritance and implementation inheritance
- [Delegation
- [The Liskov Substitution Principle
- [Delegation and inheritance in design patterns

Application/Solution Objects

- [Application objects (domain objects) represent relevant concepts from the application domain
- [Solution objects represent components that do not exist in the application domain
 - ex. data stores, UI objects, middleware
- [Object design refines both types of objects and identifies additional solution objects

Inheritance

- [Inheritance organizes objects into taxonomies
 - focus is to reduce redundancy and enhance extensibility
- [Implementation inheritance: sole purpose is to reuse code
 - subclassing an existing class
- [Specification inheritance: classifies concepts into type hierarchies (also called interface inheritance)
 - one object is as good as another for type purposes



Delegation

- [Alternative to implementation inheritance

- [A class delegates to another class if it implements an operation by resending a message to another class

- adds dependencies between classes

- [This is preferable to implementation inheritance

- more robust, does not interfere with existing code

The Liskov Substitution Principle

- [Formal definition for specification inheritance:
 - if a piece of client code uses the methods provided by a superclass, developers should be able to add new subclasses without having to change the client code
- [More formally, if a type S can be substituted for all T s, then S is a subtype of T
 - this leads to strict inheritance

Design Patterns

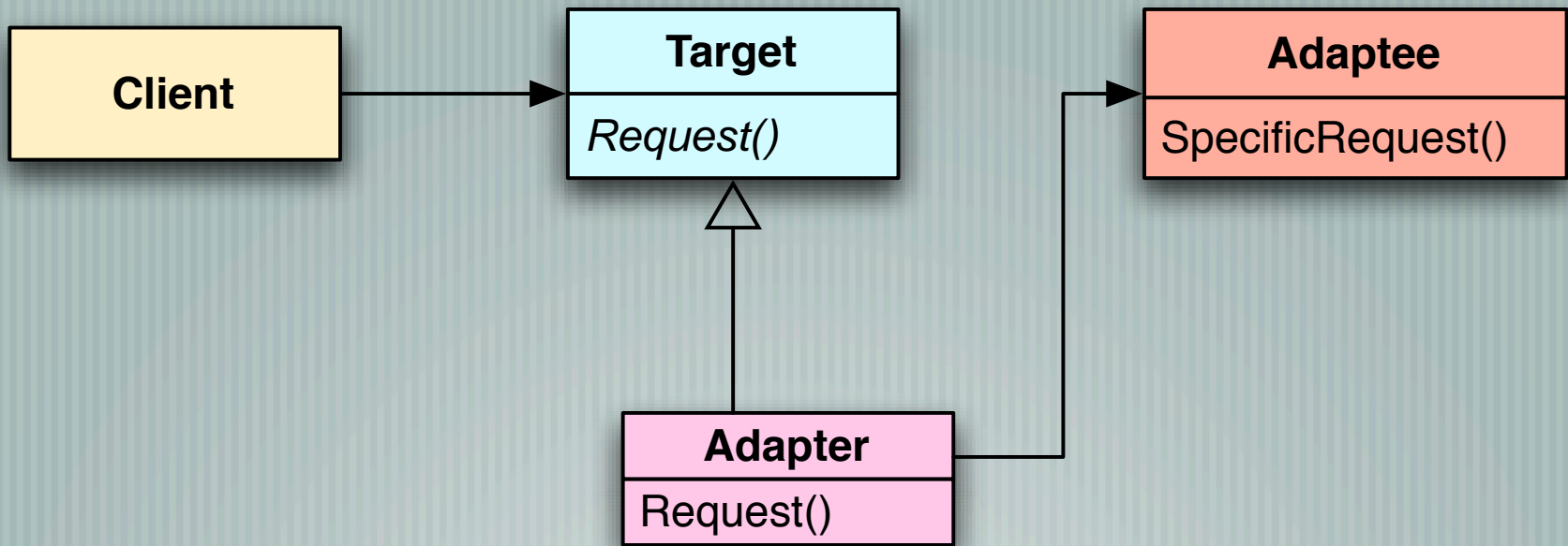
- [Design pattern: a template solution
- [Design pattern elements:
 - name
 - problem description
 - solution (set of classes and interfaces)
 - consequences (tradeoffs and alternatives)

Design Pattern Components

- [Client class: accesses the pattern
- [Pattern interface: part of the pattern visible to the client
- [Implementor class: provides low-level behavior
- [Extender class: specializes the implementor class
 - provides different implementation/extended behavior

The Adapter Pattern

- [Adapter is used to write a class that complies with an existing interface and reuses behavior from an existing class
- [The Adaptor implements each method declared in the interface in terms of requests to the existing class
- [This pattern uses both inheritance and delegation



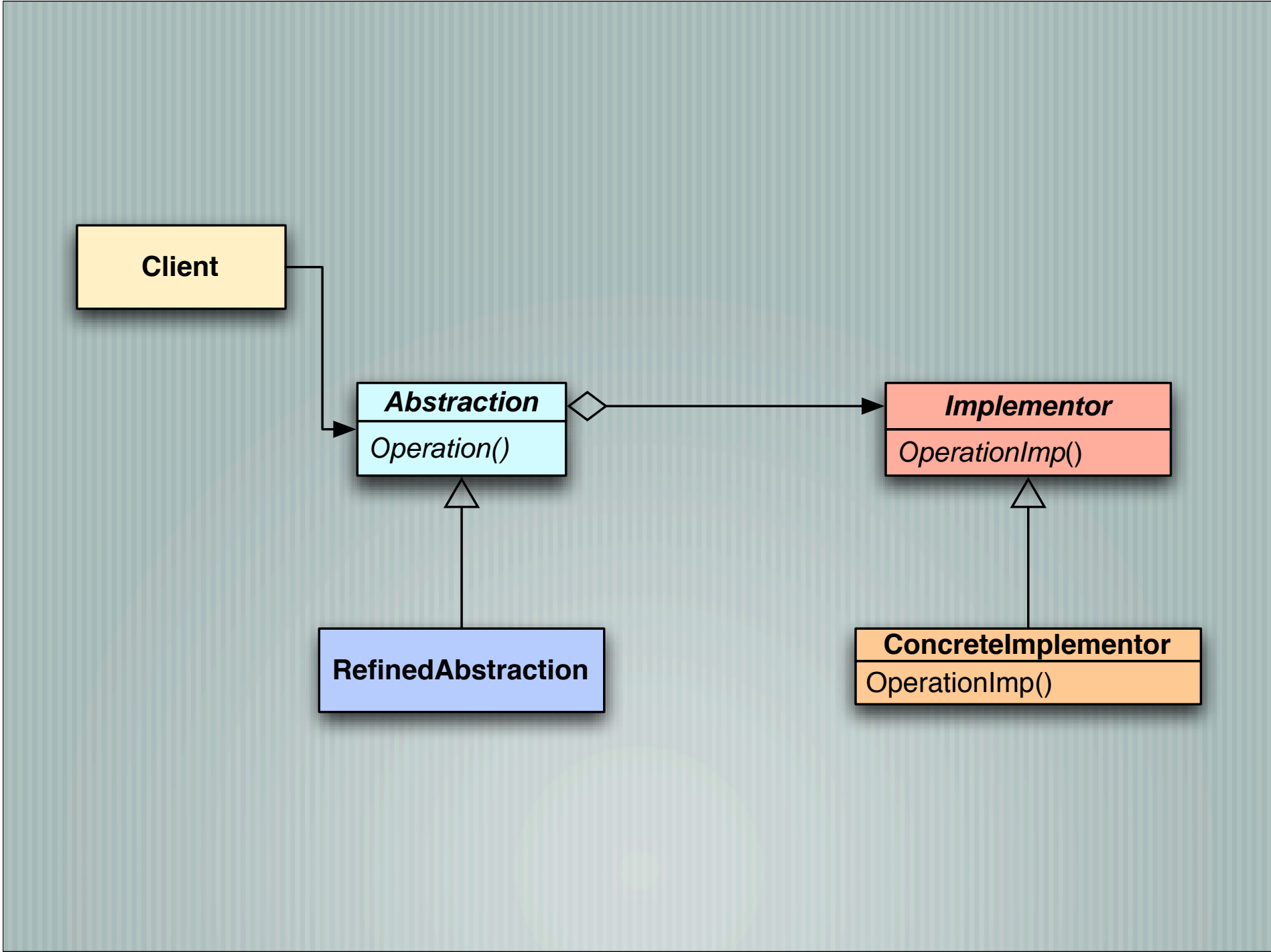
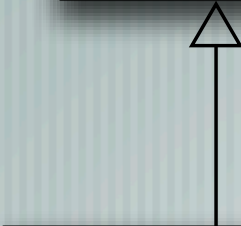
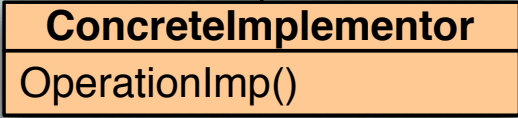
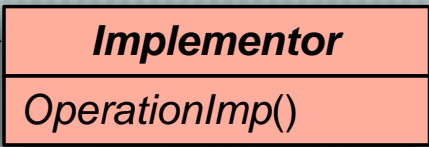
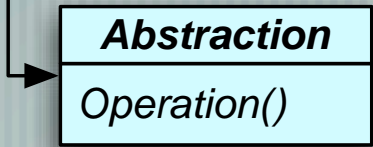
Reuse Activities

Selecting Patterns and Components

- [Conflict: stable system architecture vs. flexibility for change
- [Solution: design for change
- [Sources of change include new vendors/technologies, new implementations, new views, new complexity, and errors
- [Delegation, inheritance, and abstract classes help to decouple interface from implementation

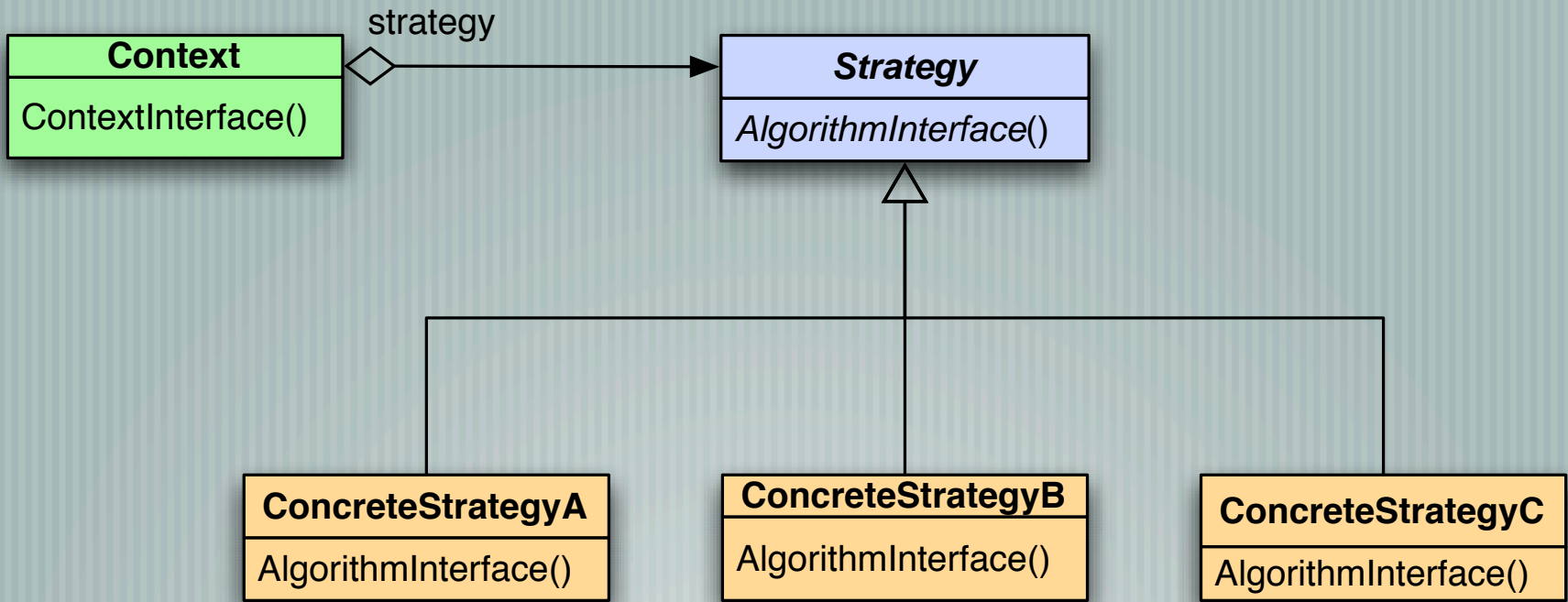
The Bridge Pattern

- [Incremental development means that subsystems are not completed at the same time, or may take multiple forms
- [Problem: dynamically substitute multiple realizations of the same interface
- [Solution: use of Abstraction class and Implementor interface
 - concrete Implementors can be substituted at run-time



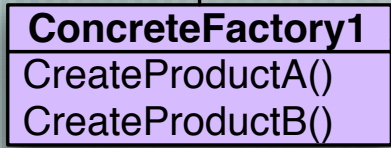
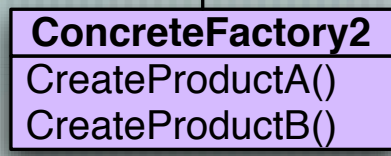
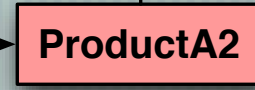
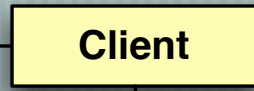
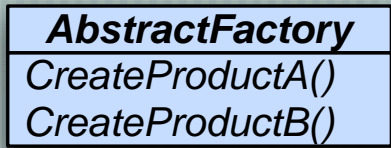
The Strategy Pattern

- [Problem: allow different algorithms to be selected at run-time
 - ex. use of different network protocols (including new ones)
- [Solution: encapsulate different interface implementations
 - multiple implementations may be used during run-time



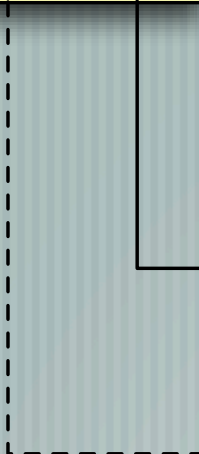
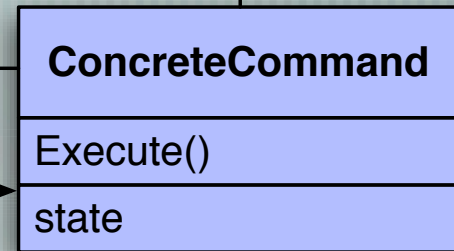
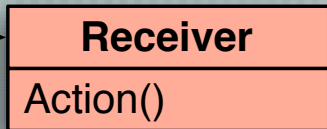
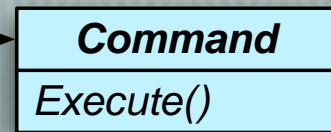
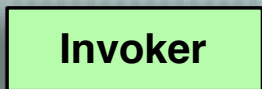
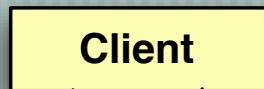
The Abstract Factory Pattern

- [Problem: mixing and matching products from different manufacturers (interoperability)
- [Solution: separate generic objects from concrete instances
 - factories provide methods for creation
 - clients can only access abstract interfaces



The Command Pattern

- [Decouple commands from their handling
 - this makes it easy to record, execute, or undo requests without knowing their contents
- [Often used in Model-View-Controller architectures



The Composite Pattern

— [Problem: provide a uniform way for clients to deal with single components and collections of components

— ex. Java AWT/Swing component hierarchy

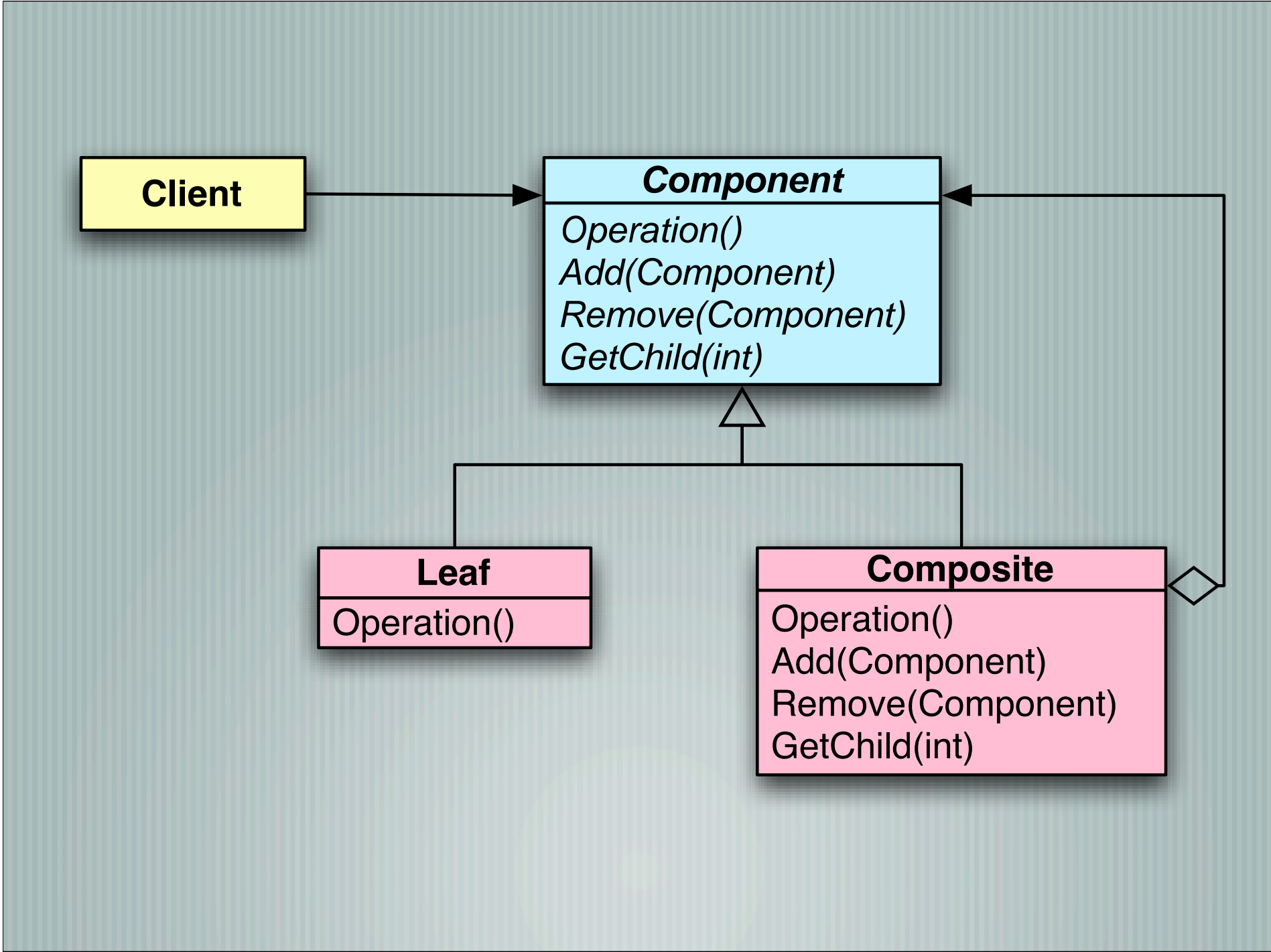
— [Solution: abstract component object can represent single elements or aggregated elements

Client

Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)

Leaf
Operation()

Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)



Application Frameworks

- [Application framework: a reusable partial application that can be specialized or extended for custom applications
- [Provides hook methods, which can be overridden to provide custom behavior
 - hook methods decouple interfaces and behaviors from application context
- [Classified by process position or extension technique

Process Position Classes

- [Infrastructure frameworks

- used internally within a software project

- [Middleware frameworks

- integrate existing distributed applications/components

- [Enterprise application frameworks

- application specific, focus on particular domains

Extension Technique Classes

- [Whitebox frameworks

- rely on inheritance and dynamic binding

- functionality is extended by subclassing/overriding

- [Blackbox frameworks

- define interfaces for pluggable components

- functionality is reused via interfaces/delegation

Patterns vs. Frameworks

- [Frameworks focus on reuse of concrete designs and algorithms in a particular language
- [Design patterns focus on reuse of abstract designs and cooperating classes
 - design patterns are building blocks for frameworks

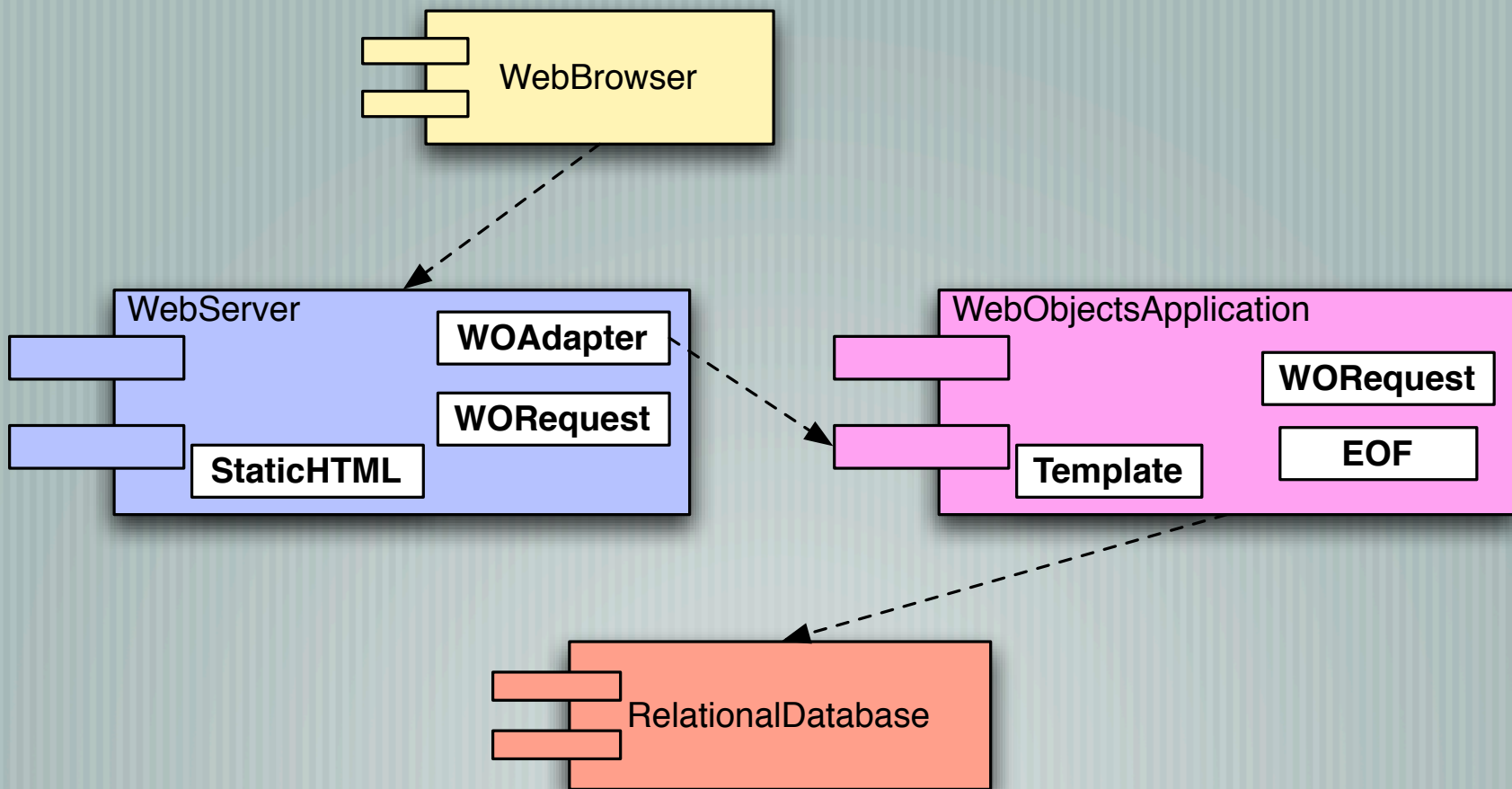
Libraries vs. Frameworks

- [Framework classes are designed for families of related applications
- [Class libraries are more generic, with smaller scope for reuse
- [Frameworks are active; they control the flow of control
- [Frameworks use class libraries to simplify their own development and perform basic tasks

Components vs. Frameworks

- [Components are self-contained classes
 - “black boxes” that can be plugged together
 - less tightly coupled than frameworks
- [Frameworks simplify the development of infrastructure and middleware, while components simplify end-user code

WebObjects Framework Example



Managing Reuse

Advantages of Reuse

- [Lower development effort
- [Lower risk
- [Widespread use of standard terms
- [Increased reliability

Reuse Challenges

- [Not Invented Here (NIH) syndrome

- developers often mistrust code they haven't written

- [Process support

- most SE methods are aimed at creation, not reuse

- [Training

- developers must be trained to employ reuse

Documenting Reuse

- [Reuse activities involve two types of documentation:
 - documentation of the solution being reused
 - should include references to use, example of use, alternative solutions, and trade-offs encountered
 - documentation of the system reusing the solution
 - should reference reused solutions and document which classes use which components

Assigning Responsibilities

— [Component expert

— usually a developer with third-party training

— [Pattern expert

— [Technical writer

— [Configuration manager

Where We're Going

- [Object Design: Interface Specification
- [Implementation: Mapping Models to Code
- [Testing

Interface Specification

Plan of Attack

- [Interface Specification Concepts
 - Introduction to the Object Constraint Language (OCL)
- [Interface Specification Activities
- [Managing Object Design

Review of Object Design

- [Analysis and system design bridge the gap between problem statement and (code) solution
- [Object design fills in the last missing pieces before implementation
 - This involves identifying new solution objects, preparing for reuse, and specifying subsystem interfaces

Object Design Activities

- [We identify and refine objects to realize subsystems
 - Objects and interfaces are still changing
- [We need a way for developers to communicate clearly and precisely about low-level details of the system

Specification Activities

- [Identification of missing attributes and operations
- [Specification of type signatures and visibility
- [Contract specifications
 - Specification of invariants
 - Specification of preconditions and postconditions

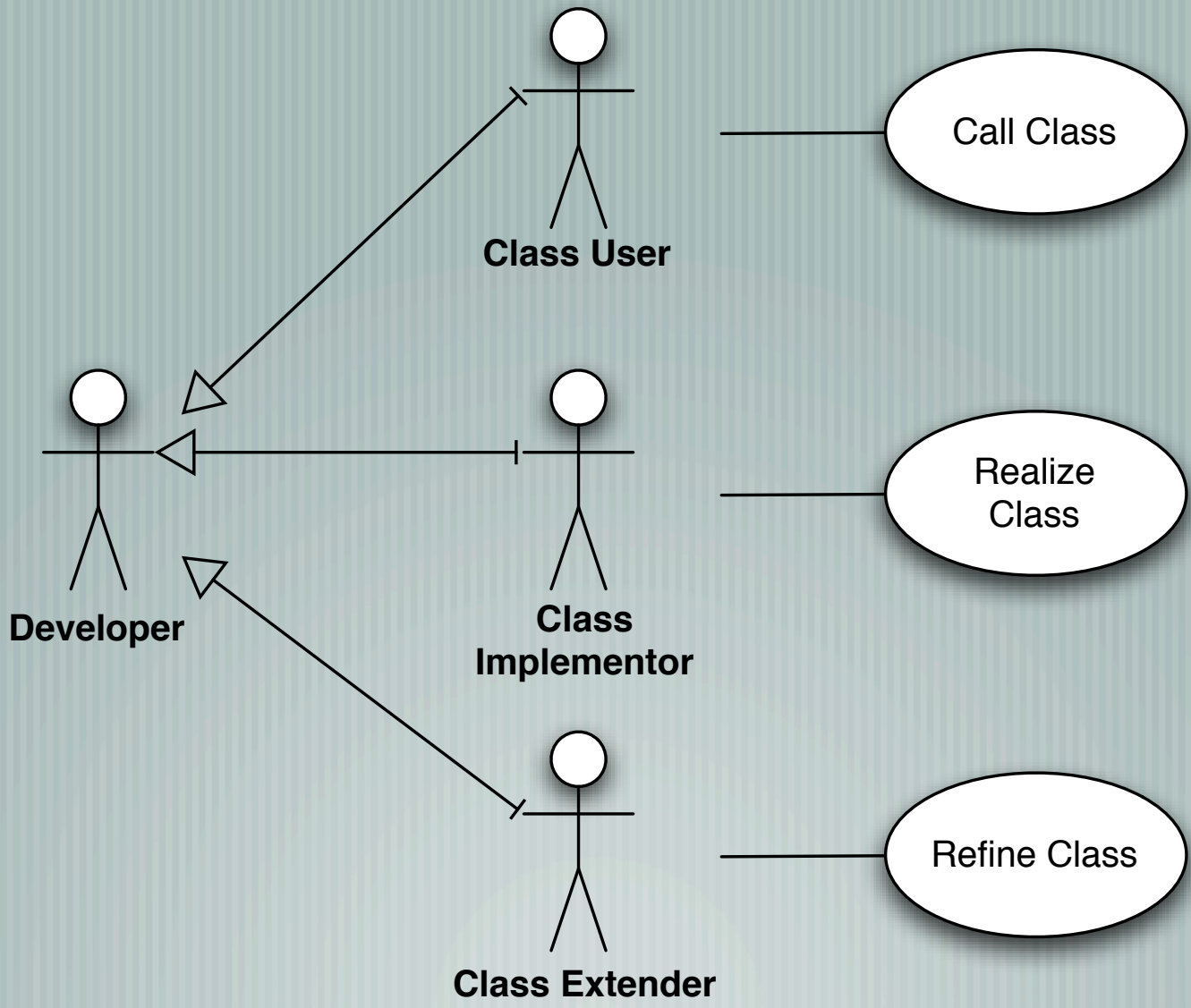
Object Design Goal

- [Up to this point, we have constructed a set of models
 - analysis object model, subsystem decomposition, HW/SW mapping, boundary use cases, design patterns, etc.
- [The goal of object design is to integrate all of this information into a coherent whole
- [Interface specification aims to reduce integration difficulty

Interface Specification Concepts

Implementor, Extender, User

- [Implementor: implements the class under consideration
 - designs internal data structures
- [User: invokes operations provided by the class
- [Extender: specializes the class under consideration



Types, Signatures, Visibility

- [Type: specifies range of values that an attribute can take on
- [Signature: tuple containing an operation's parameter types and the type that it returns
 - ex. `acceptPlayer(player):void`
- [Visibility: specifies whether an attribute or operation can be used by other classes

Levels of Visibility

- [Private: attribute can only be accessed by the defining class
 - UML symbol: -
- [Protected: can be accessed by defining class or its subclasses
 - UML symbol: #
- [Public: can be accessed by any class
 - UML symbol: +

Contracts

- [Contracts are constraints on a class
- [Enable users, implementors, and extenders to share assumptions about the class
- [Specify constraints that the user must meet
- [Specify constraints that are ensured by the implementor

Types of Constraints

- [**Invariant**

- predicate that is always true for all instances of a class

- [**Precondition**

- predicate that must be true before operation is invoked

- [**Postcondition**

- predicate that must be true after operation is invoked

Object Constraint Language

- [Allows formal specification of constraints on model elements
- [A constraint is represented as a boolean expression
 - depicted as a note attached to a UML element
 - can also use a textual representation:
context Tournament inv: self.getMaxNumPlayers() > 0

<<invariant>>
getNumPlayers() > 0

<<precondition>>
getNumPlayers() <
getNumPlayers()

<<precondition>>
!isPlayerAccepted(p)



<<precondition>>
isPlayerAccepted(p)

<<postcondition>>
isPlayerAccepted(p)

<<postcondition>>
!isPlayerAccepted(p)

OCL Text Expressions

- [Context: indicates attached entity to which expression applies
- [Second keyword is either inv, pre, or post
 - corresponds to UML stereotype for invariant or pre/postcondition
- [Third part is the actual OCL expression
 - operations can only be used if they have no side effects
 - OCL "self" is equal to Java/C++ "this"

OCL Preconditions

- [Context is an operation

- ex. `Tournament::acceptPlayer(p)`

- operation parameters can be used in the expression

- [An operation can have several preconditions

- all preconditions must be true before operation can be invoked

OCL Postconditions

- [Postconditions are written the same as preconditions
 - use keyword “post” instead of “pre”
- [Use @pre to refer to the pre-execution value of self or an attribute:
 - $\text{getNumPlayers()} = \text{@pre.getNumPlayers()} + 1$
- [An operation may have more than one postcondition

OCL Support in Java

— [iContract is a Java tool that uses Javadoc tags to represent OCL constraints

```
/** The maximum number of players is positive at all times  
 * @invariant maxNumPlayers > 0  
 */
```

OCL Collections

- [In general, OCL constraints involve an arbitrary number of classes and attributes
- [Three cases of navigation:
 - Local attribute: attribute is local to class of interest
 - Directly related class: involves single association
 - Indirectly related class: involves a series of associations

Collections in OCL

— [Used to deal with many-to-many associations

— [Set: used to navigate a single association

— [Sequence: used to navigate a single ordered association

— [Bag: multiset used to accumulate objects when accessing indirectly related objects

— [OCL uses dot notation for attributes and the \rightarrow operator for collections

Quantifiers: forAll and exists

- [forAll(variable | expression) is true if expression is true for all elements in the collection
 - Ex. matches->forAll(m:Match | m.start.after(t.start))
- [exists(variable | expressions) is true if expression is true for at least one element in the collection

Interface Specification Activities

Interface Specification

- [Interface specification includes the following activities:
 - Identifying missing attributes and operations
 - Specifying type signatures and visibility
 - Specifying preconditions and postconditions
 - Specifying invariants
 - Inheriting contracts

Missing Attributes/Operations

- [Analysis focuses on system functionality
 - we may have missed details independent of the application domain (ex. system misuse)
- [This type of auditing may reveal a need to revise sequence diagrams and/or add new methods/objects

Specifying Types

- [This step adds detail to our object model
 - ex. how will the system track time?
- [This step also maps system classes/attributes to built-in types
 - ex. will we use Java Strings to represent certain data?

Specifying Visibility

- [In this step, we separate operations into two categories:
 - class interface methods
 - utility methods (for internal use only)
- [We also decide whether each attribute should be externally accessible

Pre- and Postconditions

- [Contract: agreement between user and implementor
- [Preconditions describe the part of the contract that the user must respect
- [Postconditions describe what the implementor guarantees (if the class user fulfills his part of the contract)
- [Pre- and postconditions also specify class dependencies
 - ex. order of execution

Specifying Invariants

- [Invariants provide an overview of the essential properties of a class
- [They form a permanent contract that extends operation-specific contracts
- [Can be identified by extracting common properties

Heuristics for Constraints

- [Focus on the lifetime of a class
- [Identify special values for each attribute
- [Identify special cases for associations
- [Identify ordering among operations
- [Use helper methods to compute complex conditions
- [Avoid constraints that involve many association traversals

Contract Inheritance

- [Polymorphism means that a descendant class can be substituted for any class
- [A contract that holds for the superclass must also hold for the descendant
- [This is called contract inheritance

Inheriting Contracts

- [A method of a subclass is allowed to weaken the preconditions of the method it overrides
- [Methods must ensure the same or stricter postconditions as their ancestors
- [A subclass must respect all of its ancestors' invariants
 - it may strengthen those invariants

Managing Object Design

Management Challenges

- [Increased communication complexity

- number of participants increases at this stage

- decisions must be consistent with project goals

- [Consistency with prior decisions and documents

- developers may question and reevaluate prior decisions

- we must maintain a record of revised decisions

Documenting Object Design

— [Object Design Document

- describes object design trade-offs, subsystem interface guidelines, subsystem decomposition, and class interfaces
- Used to exchange information among teams
- audience includes system architects, implementors, and testers

Documentation Approaches

- [Self-contained ODD generated from model
 - generate documentation automatically from UML model
 - disadvantages:
 - redundancy with Requirements Analysis Document
 - ODD must be changed whenever the code changes

Documentation Approaches

- [ODD as extension of RAD

- object design is considered as the set of application objects, augmented with solution objects
- advantage: less redundancy, more consistency
- disadvantage: adds irrelevant information to RAD

Documentation Approaches

- [ODD embedded into source code
 - once ODD becomes stable, automatically generate class stubs via tool support (ex. Javadoc), then abandon ODD
 - forward engineering process
 - advantage: greater consistency with source code
 - ODD is regenerated when source code changes

ODD Outline

1. Introduction

1. Object design trade-offs
2. Interface documentation guidelines
3. Definitions, acronyms, abbreviations

2. Packages

3. Class interface

4. Glossary

Assigning Responsibilities

- [Core architect — develops coding guidelines and conventions
 - ensures consistency with prior decisions (SDD and RAD)
- [Architecture liaisons — document public interfaces
 - negotiate changes to public interfaces
- [Object designers — refine interface specification
- [Configuration manager
- [Technical writers

Contracts During RA

- [Constraints may be used earlier in the requirements analysis (RA) process
- [Tradeoffs that must be considered include:
 - communication among stakeholders
 - level of detail and rate of change
 - level of detail and elicitation effort
 - testing requirements

Next Time

- [Mapping models to code
 - Mapping concepts
 - Mapping activities
 - Mapping implementation