DESIGN PATTERNS APPLIED

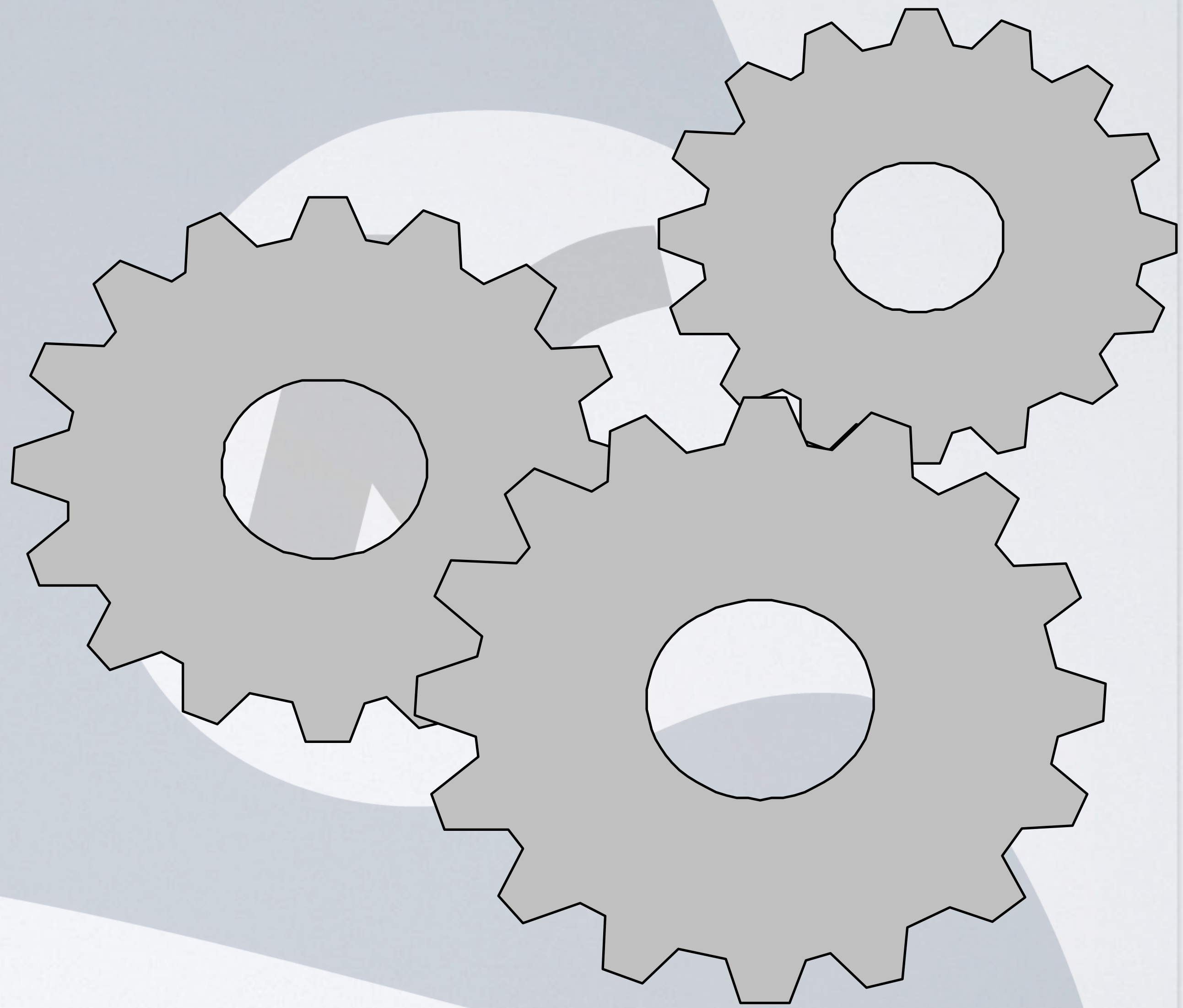
Presented By:
W. David Pitt
Keyhole Software
dpitt@keyholesoftware.com

AGENDA

- Object technology
- Elements of Reuse
  - Frameworks
  - Components
- Types of Reuse
- Reuse Anti-Patterns
- Measuring Reuse
- UML Primer
- Design Pattern Discussion and Exercises

OBJECT TECHNOLOGY NEW WAY OF THINKING

- Pros
  - Manage Complexity
  - Reuse
  - Shorter Development Time



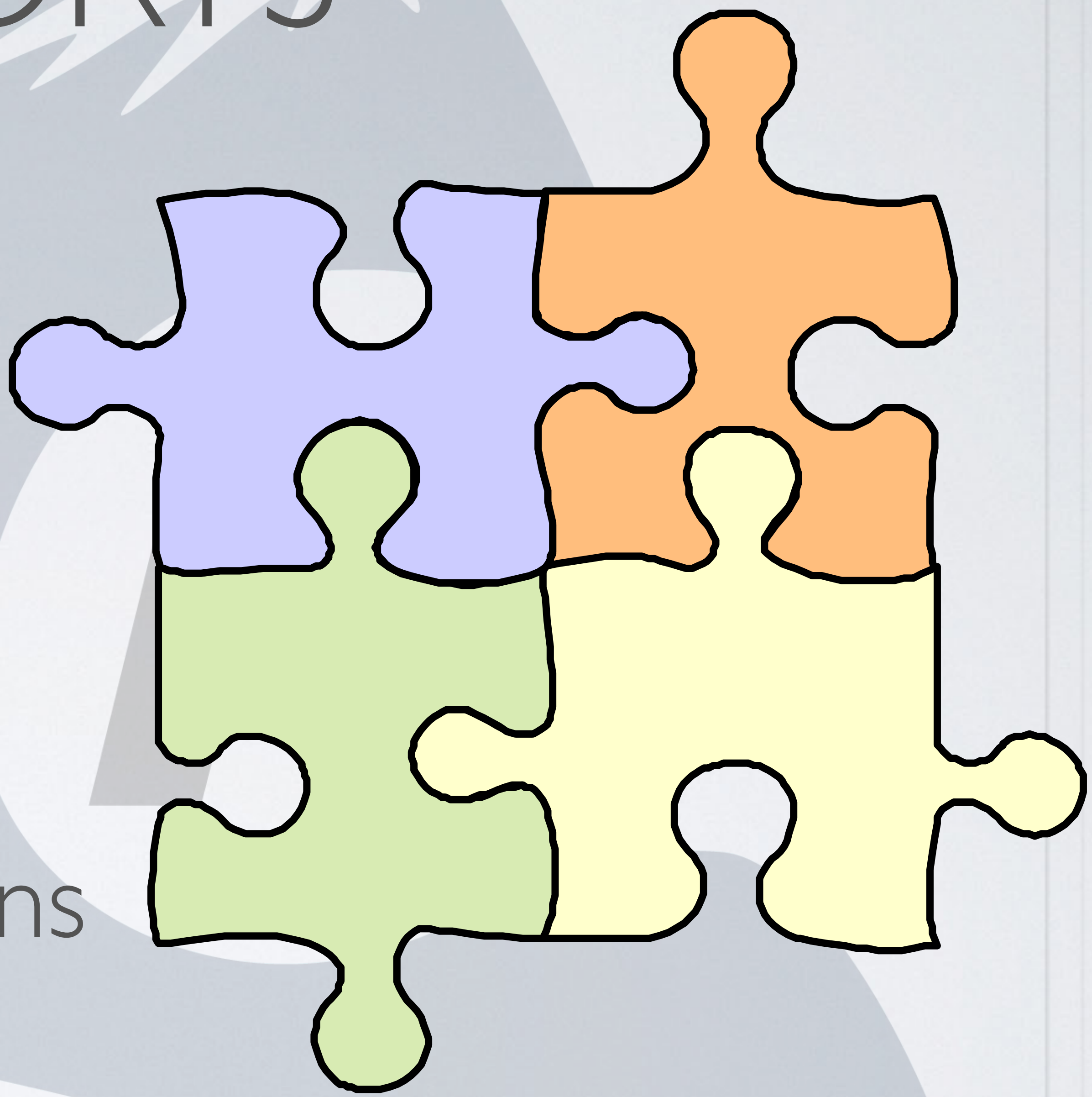
- Cons
  - Learning Curve
  - Room to abuse
  - Efficiency (?)

PROCEDURAL VS. OBJECT ORIENTED

```
// ===
// PROCEDURAL debit and print Account info
// ===
final int TYPE = 0;
final int ID = 1;
final int AMOUNT = 2;

// two dimensional object array that
// holds account information.
Object account[][] = new Object[1][3];

// put account info in array
account[0][TYPE] = "Savings";
account[0][ID] = "1000A";
account[0][AMOUNT] = new Double(10000.00);

double amount = ((Double) account[0][AMOUNT]).doubleValue();
// credit account amount
account[0][AMOUNT] = new Double(amount + 2000.00);

// print account info
System.out.println("Account "+account[0][ID]+" type ="+account[0][TYPE]+" balance = "+account[0][AMOUNT]);

//===
//=== Object Oriented approach, assume a Class Account definition
//===

Account account = new Account("1000A","Savings",10000.00);
account.credit(2000.00);
account.print();
```

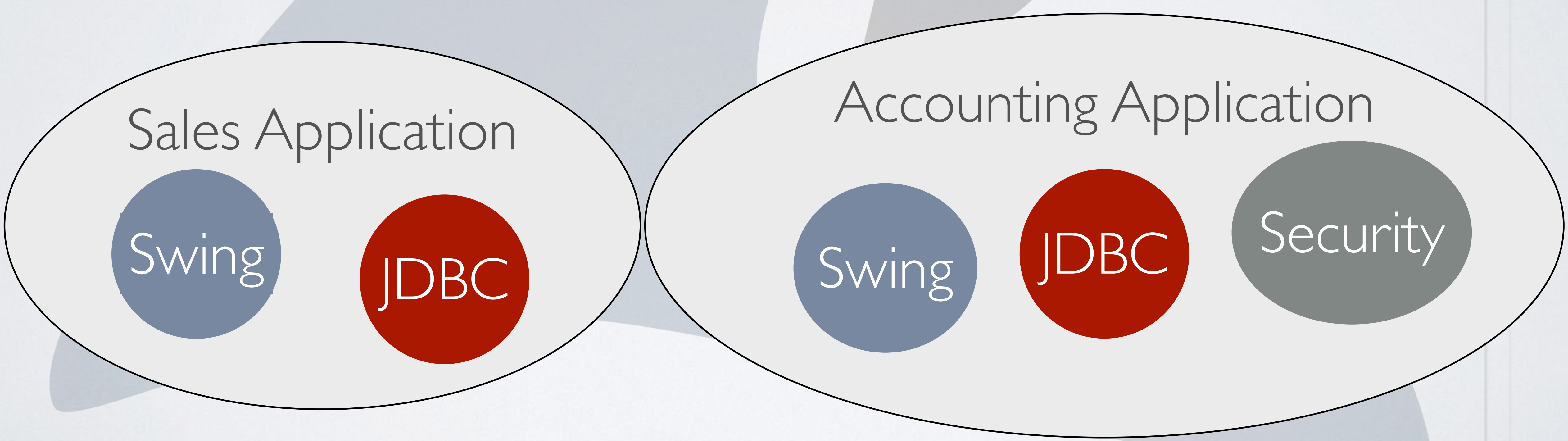
PRODUCTS OF DESIGN EFFORTS

- Frameworks
- Components
- Domain Specific Solutions (Prototypes)



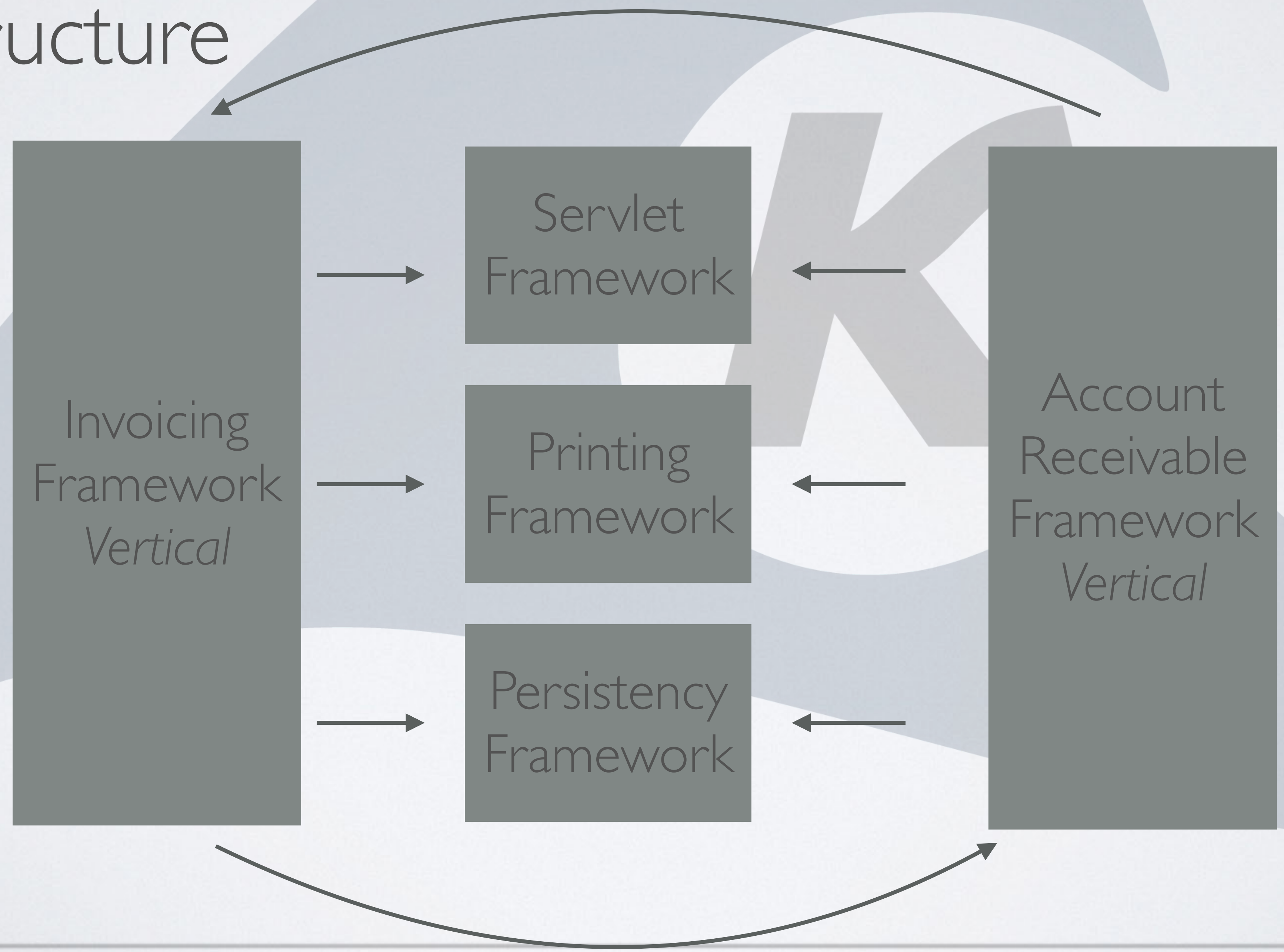
FRAMEWORKS

- Set of cooperation classes making up a reusable design
- Not domain specific, but context specific



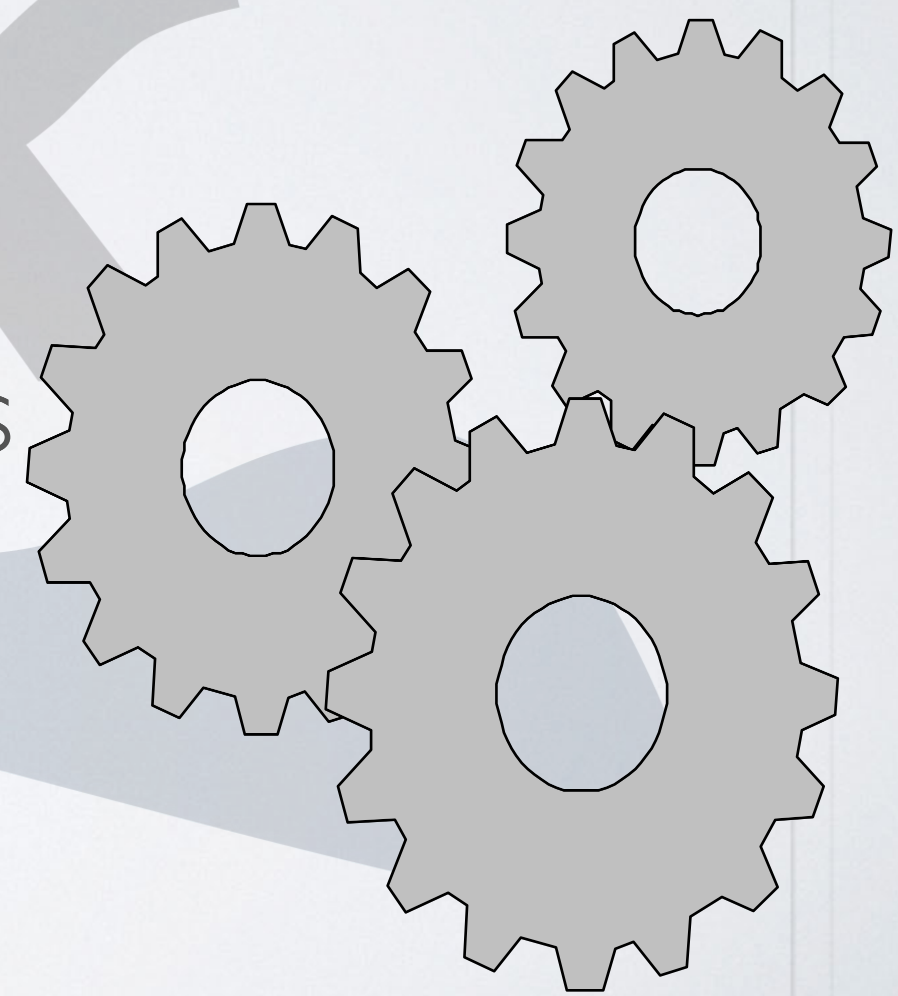
VERTICAL/HORIZONTAL FRAMEWORKS

- Vertical frameworks require more architectural infrastructure



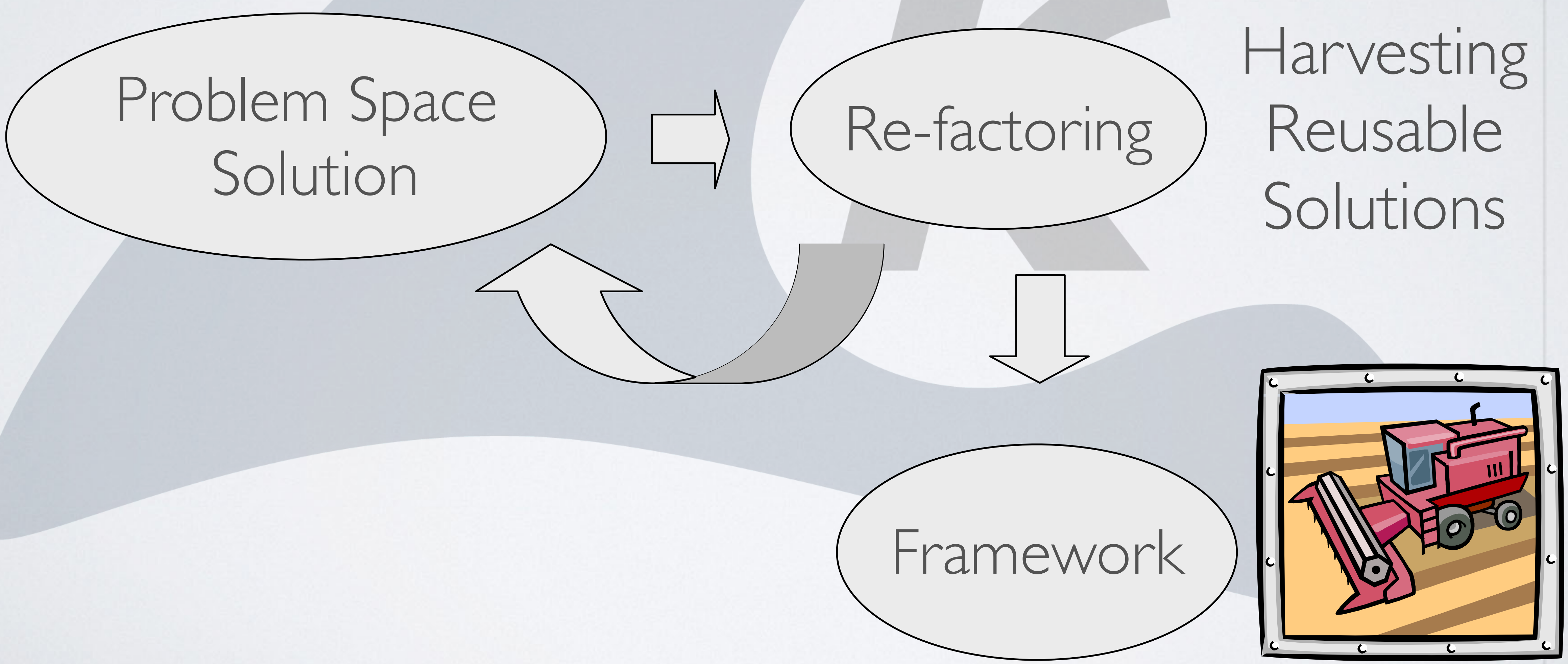
COMPONENTS

- Reusable solutions within the confines of component framework/architecture
  - ACTIVEX
  - Java Beans
- More granular than frameworks



DOMAIN PROBLEM SOLUTIONS

- Seeds future frameworks



EVERYTHING STARTS WITH DESIGN

- Generalize when possible
- Consistency
- Keep it simple
- Establish application architecture
- Accommodate re-factoring efforts in project plans

REUSE – THE HOLY GRAIL

- Misconception
  - Just because your development language is object oriented, doesn’t mean you will experience reusability
- Reality
  - Requires wholesale cultural commitment
  - Established architecture
  - Small reuse economies can be obtained, but hard to measure

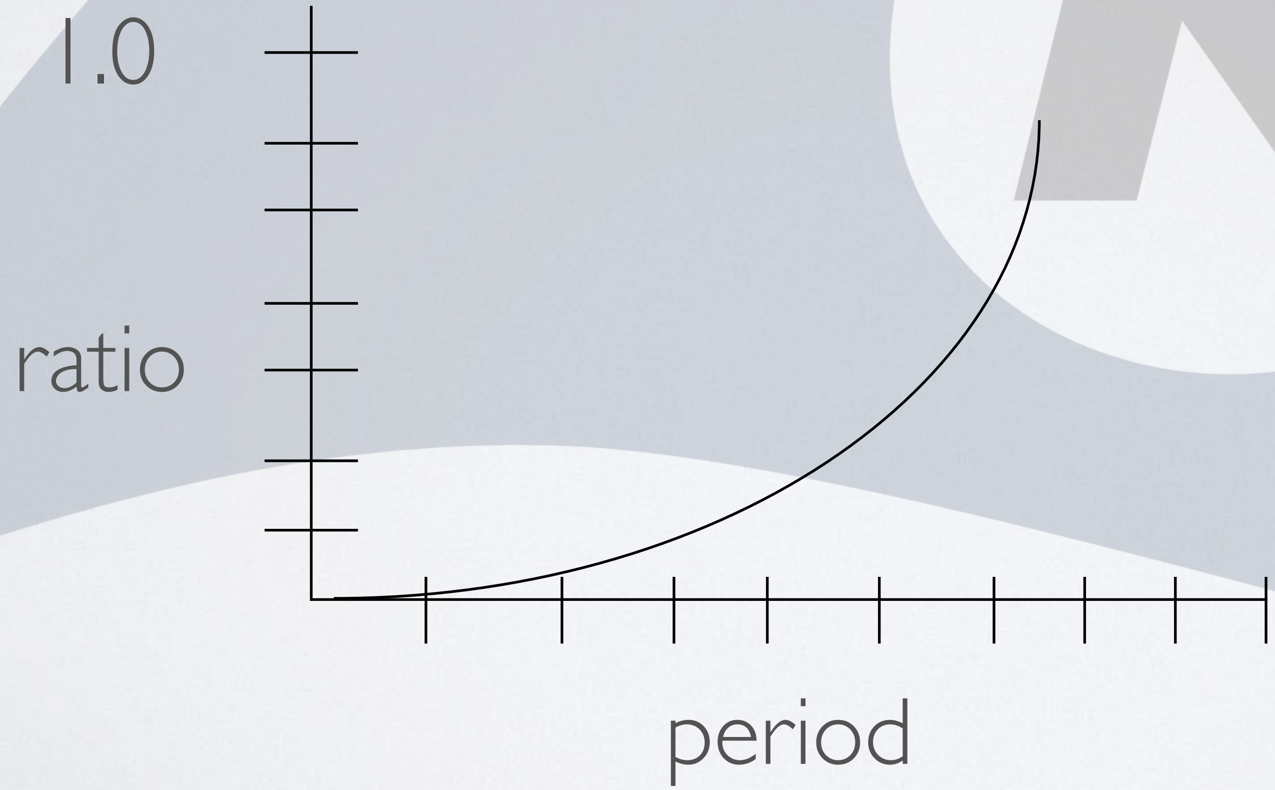
CLASSIFYING REUSE

- Ad Hoc – Cut and paste, almost done by default
- Latent – Framework usage
- Project – Development groups honor a consistent architecture and care share framework elements
- Systemic – Most to all projects utilize consistent architecture and practices. Shorter development cycles are noticed
- Cultural – Quantum leap in productivity as all levels of an organization participate in utilizing and progressing reuse. Formal measurement and repository facilities exist

MEASURING REUSE

- Measuring helps to quantify reuse

REUSE RATIO = Number of element reuses / total number of reusable elements



MEASURING PRODUCTIVITY

- Productivity should increase as reuse takes hold, productivity index can be plotted against reuse ration

ESLOC = effective source lines of code
Time = design through user acceptence
effort = staff months
B = ESLOC related skills factor (0.16 – 0.39)

PI = log 1.272 [ESLOC / (time 4/3 * (effort / B) 1/3] – 26.6

*The equation is based upon productivity formulas discovered by Lawrence H. Putnam in the 50s. The equation shown considers development effort, time, and project size factors in producing a productivity value that can be used to measure the affect that factors such as resources, schedules, tooling, and reuse affects productivity.

ELEMENTS TO WORK WITH

- Class
- Object
- Inheritance
- Polymorphism
- Association
- Delegation/Forwarding
- Overloading/Overriding

CAPTURING AND COMMUNICATING DESIGNS

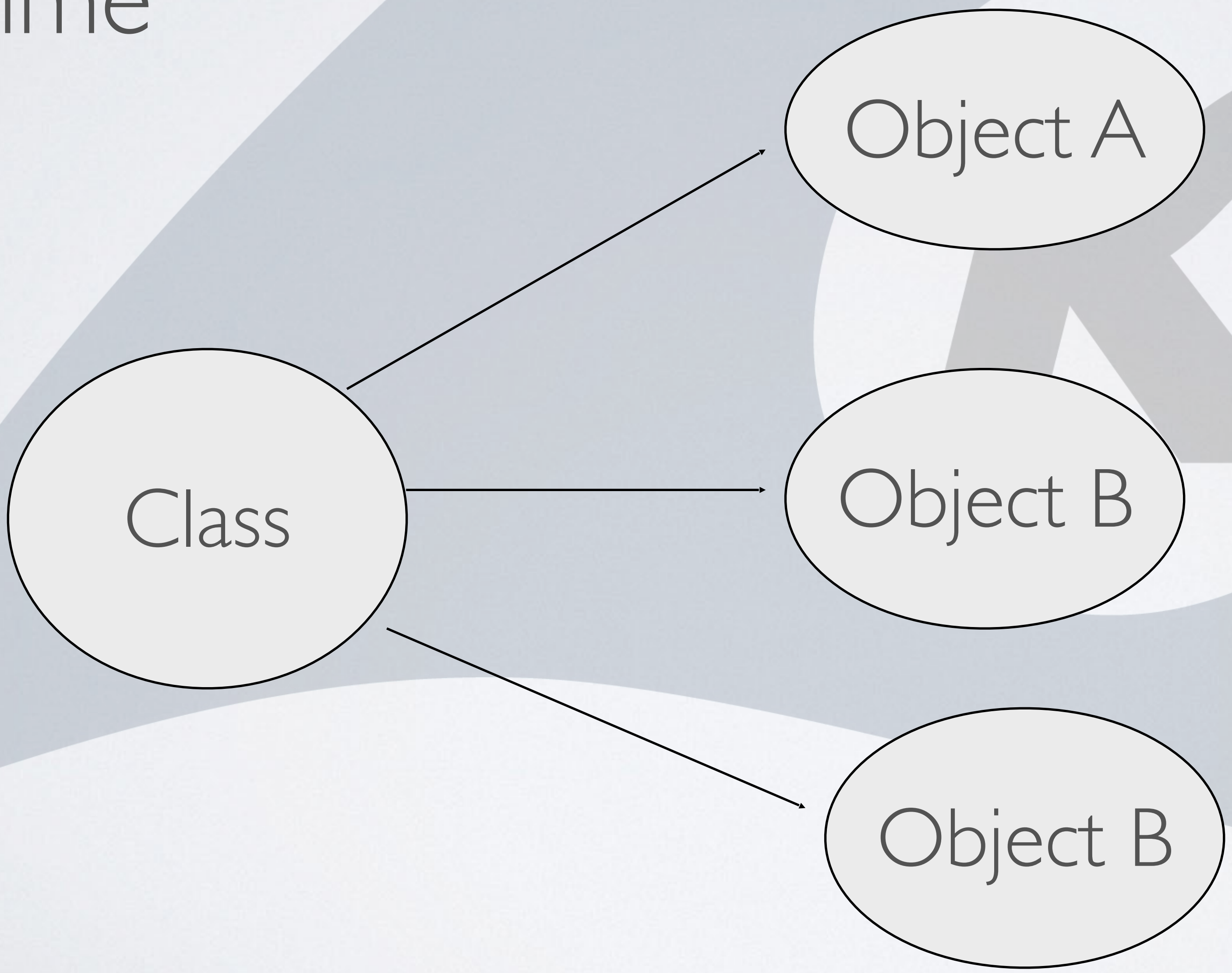
- UML (Unified Modeling Language)
- Standard supported by the OMG (Object Management Group)
- Unification of early modeling notations
  - Rumbaugh (OMT)
  - Booch
  - Ivar Jacobsen

DESIGN ELEMENT - CLASSES

- When in doubt, define a class
  - You can't add a method or attribute to primitive types or platform classes such as Strings
- Worried about performance?
  - Current JVM implementations are optimized for lots of objects, with many small methods

DESIGN ELEMENT - OBJECTS

- Classes are blueprints and become objects at runtime



DESIGN ELEMENT - INHERITANCE

- Facets of Inheritance
  - Subclassing (Implementation inheritance)
  - Subtyping (Interface inheritance)
    - Interfaces
    - Abstract implementation
    - Substitutability (form of Polymorphism)

DESIGN ELEMENT – INHERITANCE

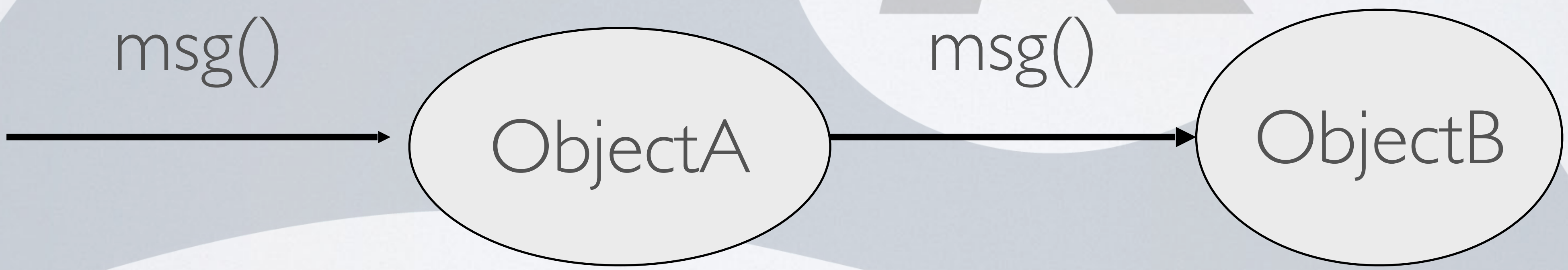
- Inheritance – Double edge sword
  - Obvious benefits, but commonly overused
  - Can result in brittle implementations
- White box vs. Black Box designs
  - Inheritance is used to create white box designs designs are configured through inheritance
  - Black box designs utilize object composition, delegation and forwarding, for more configurable encapsulated designs.

DESIGN ELEMENT - POLYMORPHISM

- Polymorphic types
  - Interfaces (emulates multiple inheritance, Configurable Implementations)
  - Abstract Classes (Configurable Algorithms)
- Polymorphic behavior
  - Method overloading (parameter polymorphism)
  - Method overriding

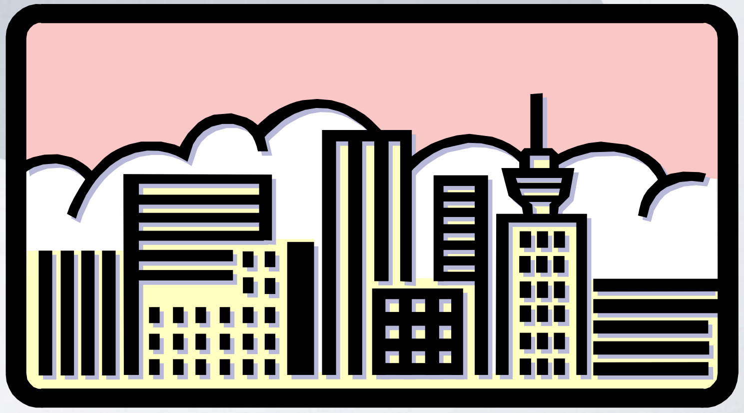
DESIGN ELEMENT - DELEGATION/FORWARDING

- Collaboration of classes
- Key to “Dynamic” configurable implementations



DESIGN PATTERNS – REUSABLE DESIGNS

- First Identified by…
  - Design Patterns (Elements of Reusable Object-Oriented Software
- Applies architecture pattern concepts pioneered by architect Christopher Alexander
  - Instead of walls and doors, objects and interfaces are used to express designs
- Describes Catalog of Designs



WHAT IS A DESIGN PATTERN?

- As quoted by Christopher Alexander….

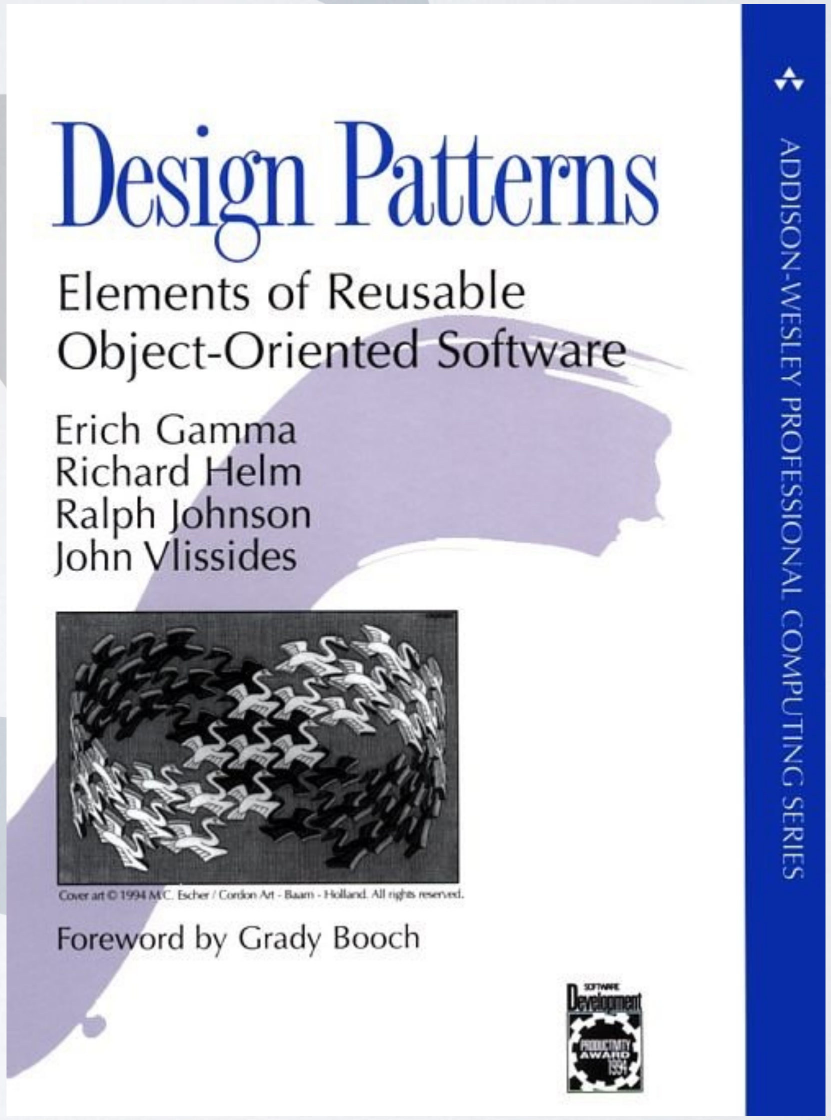
  “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to the problem, in such a way that you use this solution a million times over, without ever doing it the same way twice”

ARE THEY FRAMEWORKS?

- No…
  - More abstract
  - Smaller elements
  - Less specialized

GOF BOOK CONTENT

- Describes catalog of design Patterns



GOF PATTERN CATEGORIES

- Purpose (What the pattern does)
  - Creational
  - Structural
  - Behavioral
- Purpose by Scope (Pattern applies to classes or Objects) … Most patterns are object in scope
  - Class (Relationships fixed at compile time)
  - Object(Relationships that can be changed and configured at runtime)

CREATIONAL PATTERNS

- Makes systems independent of how objects are created, represented, and composed
  - Class Scope
    - Use inheritance
  - Object Scope
    - Delegates to other another object
- Encapsulate object creation and configuration
  - Flexibility in what, who, how, and when…
- They are closely related

STRUCTURAL PATTERNS

- Composing classes to form larger structures
  - Class Scope
    - Utilize inheritance to to compose interfaces and implementations
  - Object Scope
    - Compose objects to realize new functionality (dynamic,runtime configurations)

BEHAVIORAL PATTERNS

- Concerned with implementing algorithms and control flow between objects
  - Class Scope
    - Inheritance to distribute behavior between classes
  - Object Scope
    - Uses composition to delegate behavior to peer or cooperating objects
- Manage complex control flow with interconnecting objects

WHAT IS A DESIGN PATTERN?

- Patterns has four essential elements:
  - Name
  - Problem
  - Solution
  - Consequences

DESCRIBING DESIGN PATTERNS

- UML notation alone is not enough, textual descriptions accompanying graphical design is required
  - Name/Classification
  - Intent
  - Also Known As
  - Motivation
  - Applicability
  - Structure
  - Participants

WHY DESIGN PATTERNS?

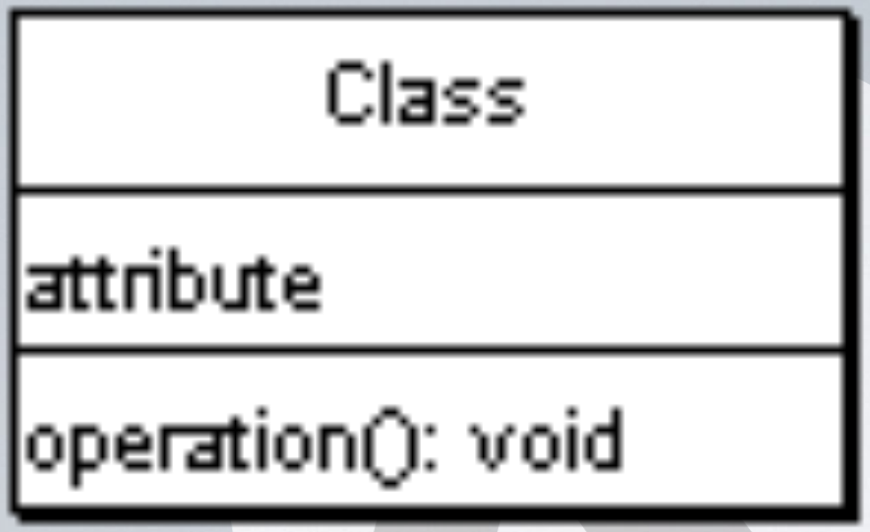
- Studying them helps promote OO way of thinking
- Collaboration and communication saves time
- Ready-made design solutions

UML PRIMER

- Patterns are described with text and a modeling notation.
- UML Notation is the arguable standard.

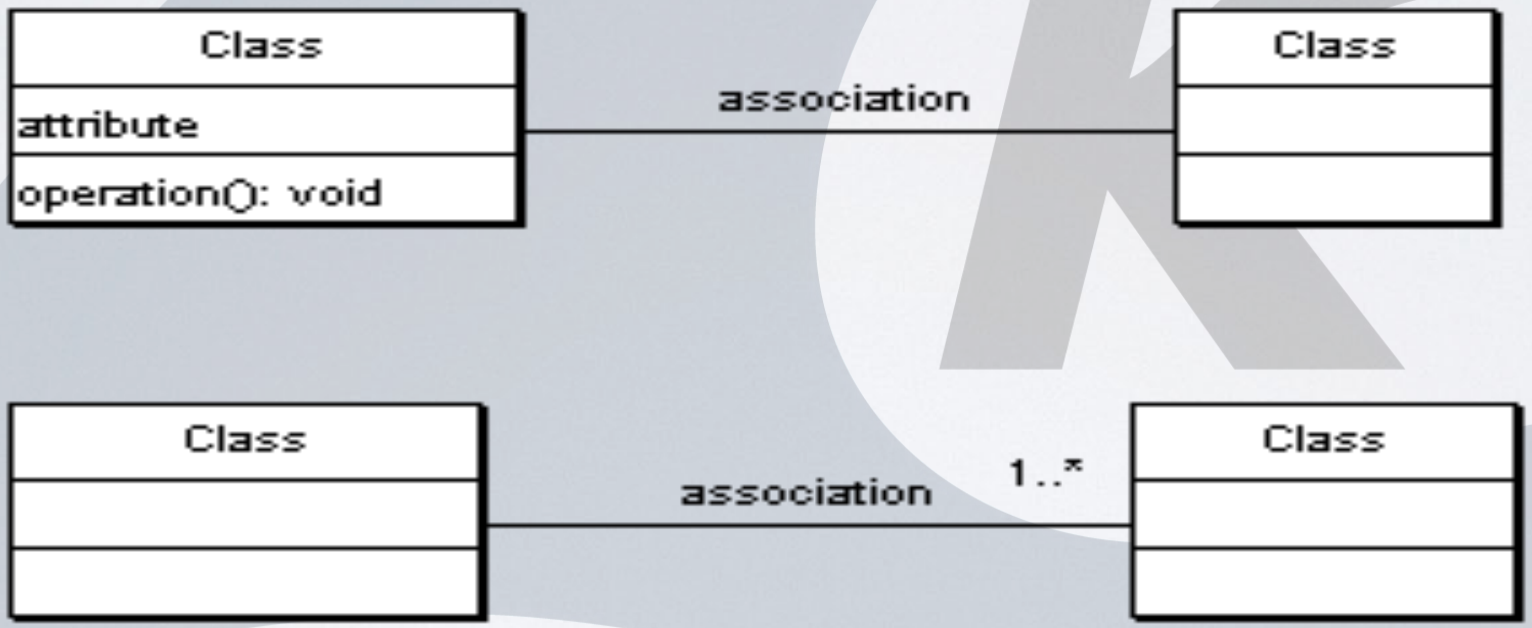
CLASSES (OBJECTS)

- Models Name
- Attributes
- Operations



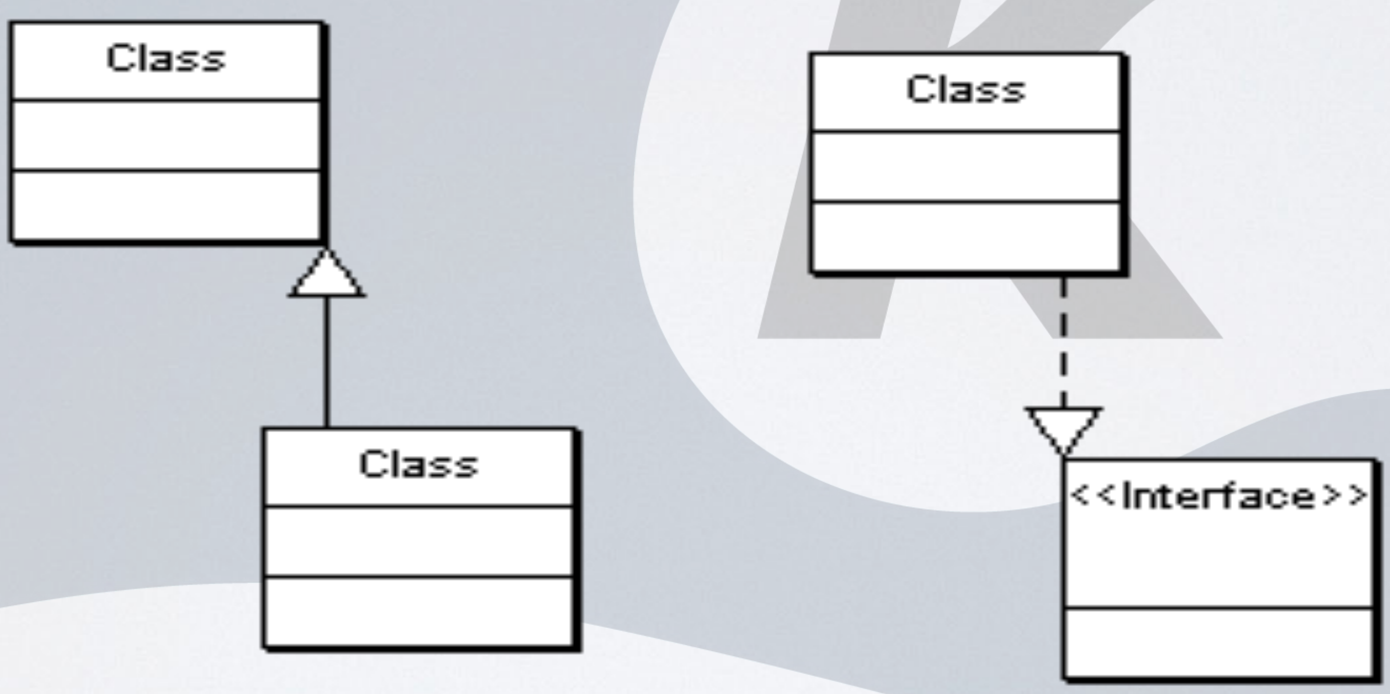
ASSOCIATIONS

- Class relationships



GENERALIZATION

- Inheritance and Interface Implementation



ADVANCED MODELING - STEREOTYPES

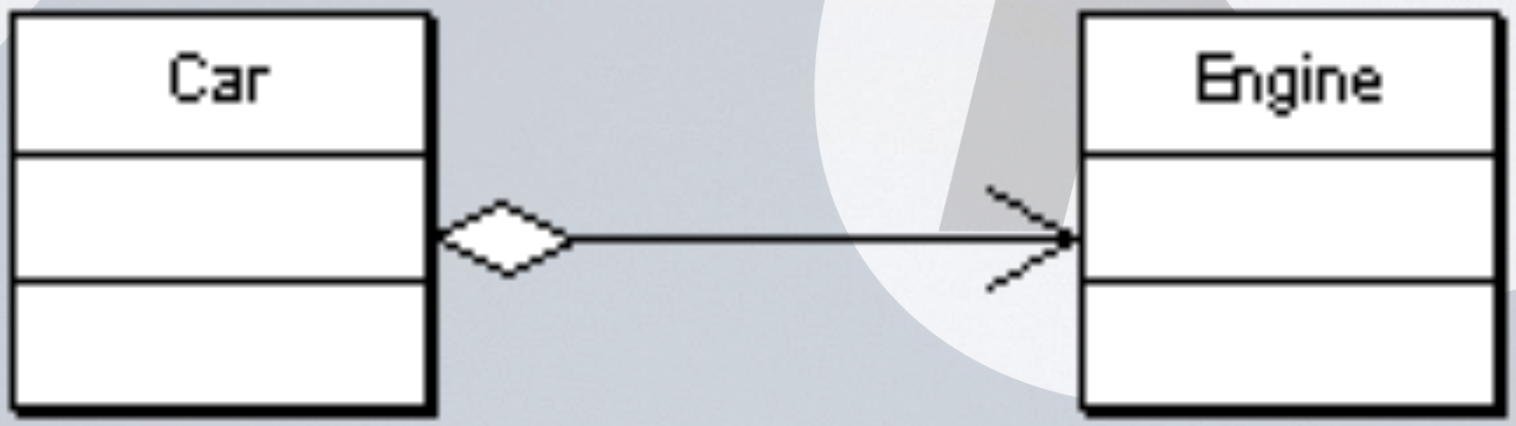
- Stereotypes
  - Way to extend UML elements
- Aggregation
  - Part of Relationship
- Composition
  - Stronger form of aggregation
  - Part of one whole
  - Typically lives or dies with the whole

ADVANCED MODELING STEREOTYPES

- Way to extend modeling elements

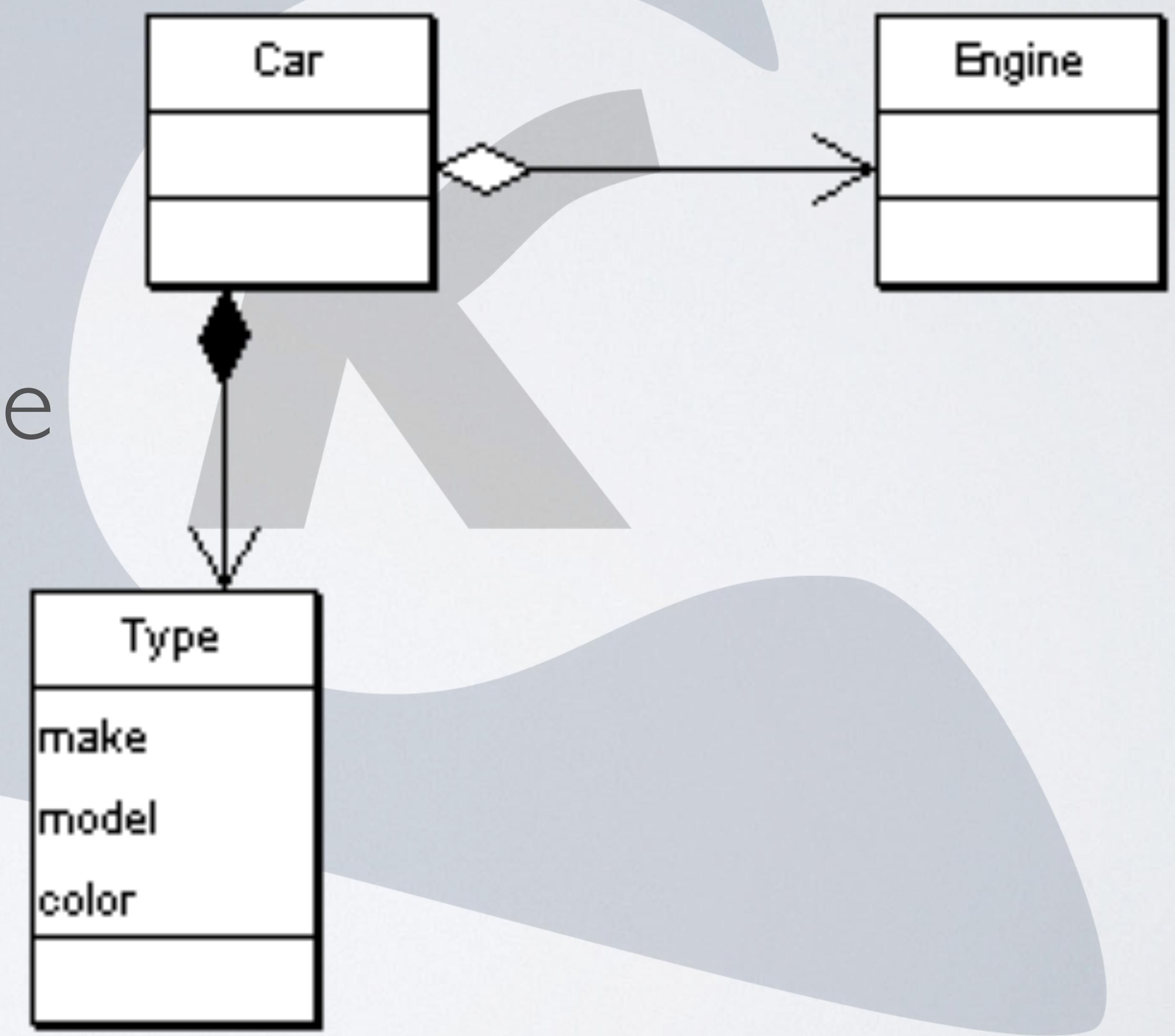
ADVANCED MODELING AGGREGATION

- Part of relationship



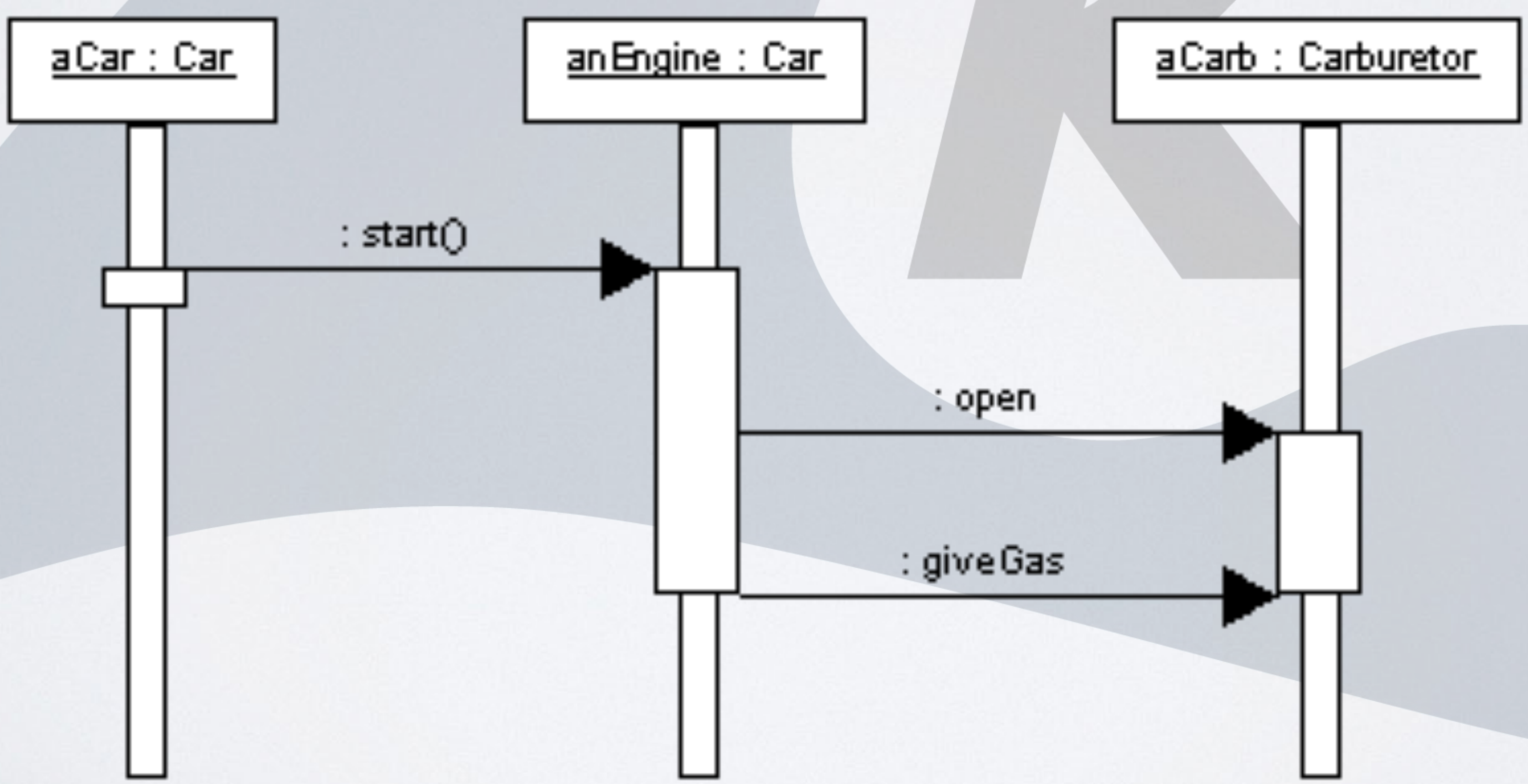
ADVANCED MODELING COMPOSITION

- Stronger form of Aggregation
- Part of one whole
- Typically dies with whole



DYNAMIC MODELING

- Sequence and Collaboration diagrams explore dynamic behavior



MODELING

- Start with conceptual model (UML light)
  - Identify
    - Classes (Nouns sometimes Verbs)
    - Associations
    - Attributes
    - Re-visit to identify generalizations and apply design patterns

CASE STUDY

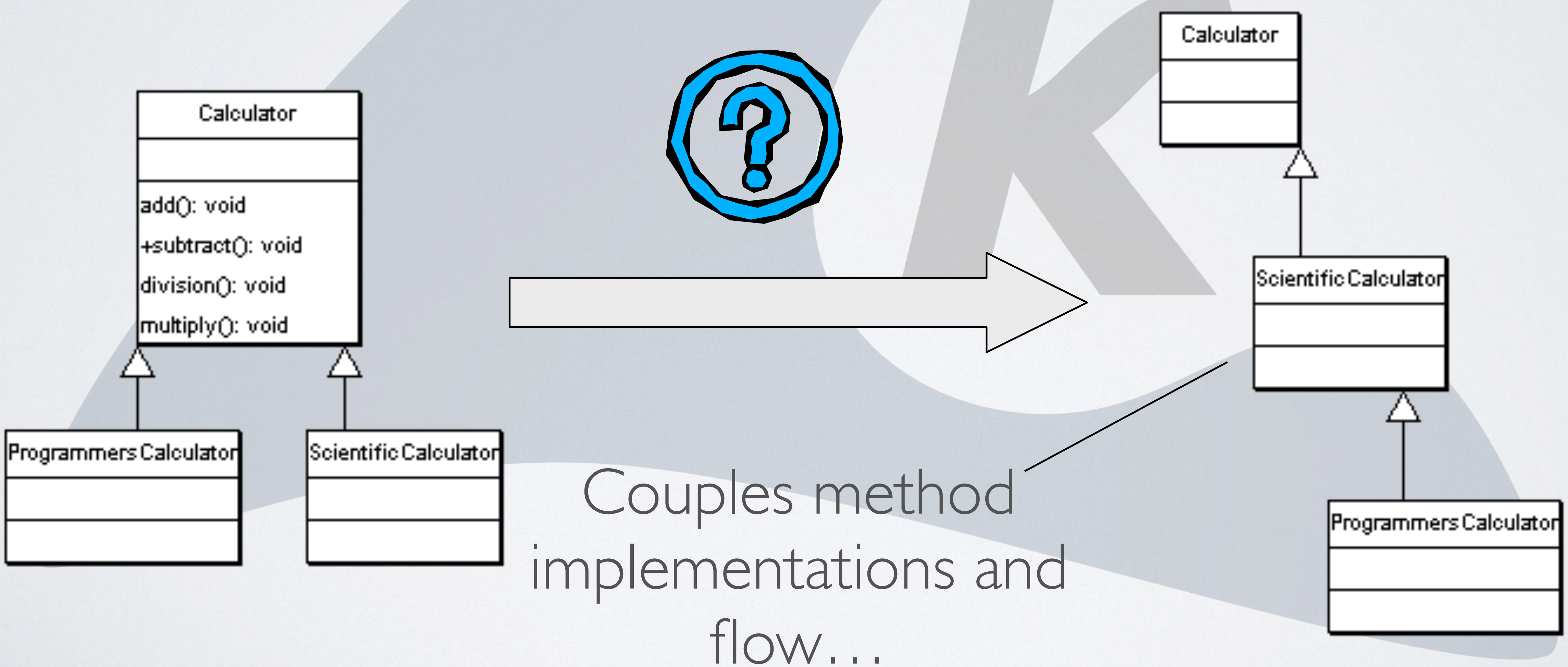
- Calculator design will be evolved using GOF design Patterns

EXERCISE #1 – CALCULATOR USING INHERITANCE

- This lab will describe a calculator design utilizing inheritance to produce different types of calculators.

INHERITANCE VERSE COMPOSITION

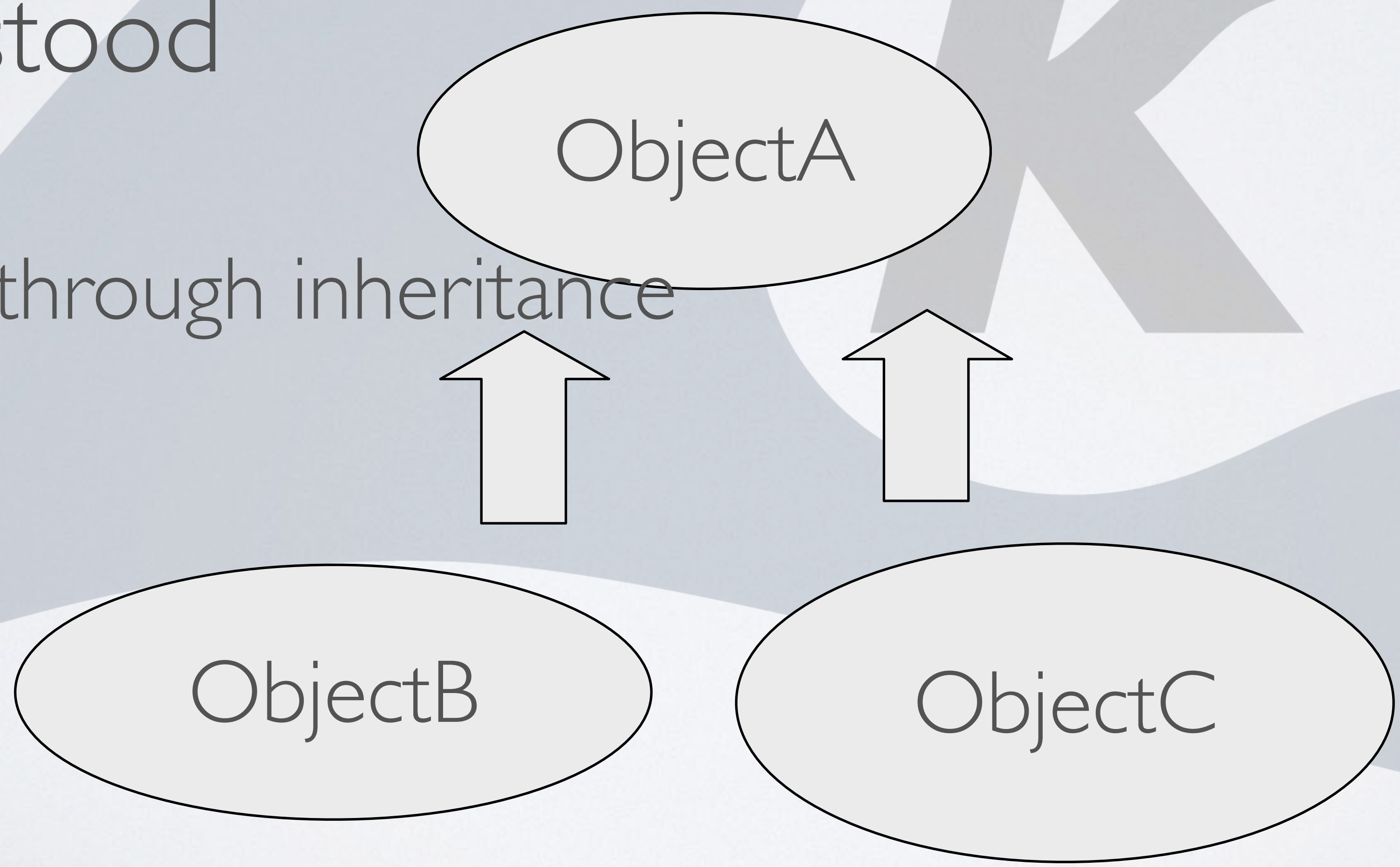
- Inheritance-based designs can be brittle
  - What happens when we need a scientific programmer's calculator?



INHERITANCE - WHITE BOX

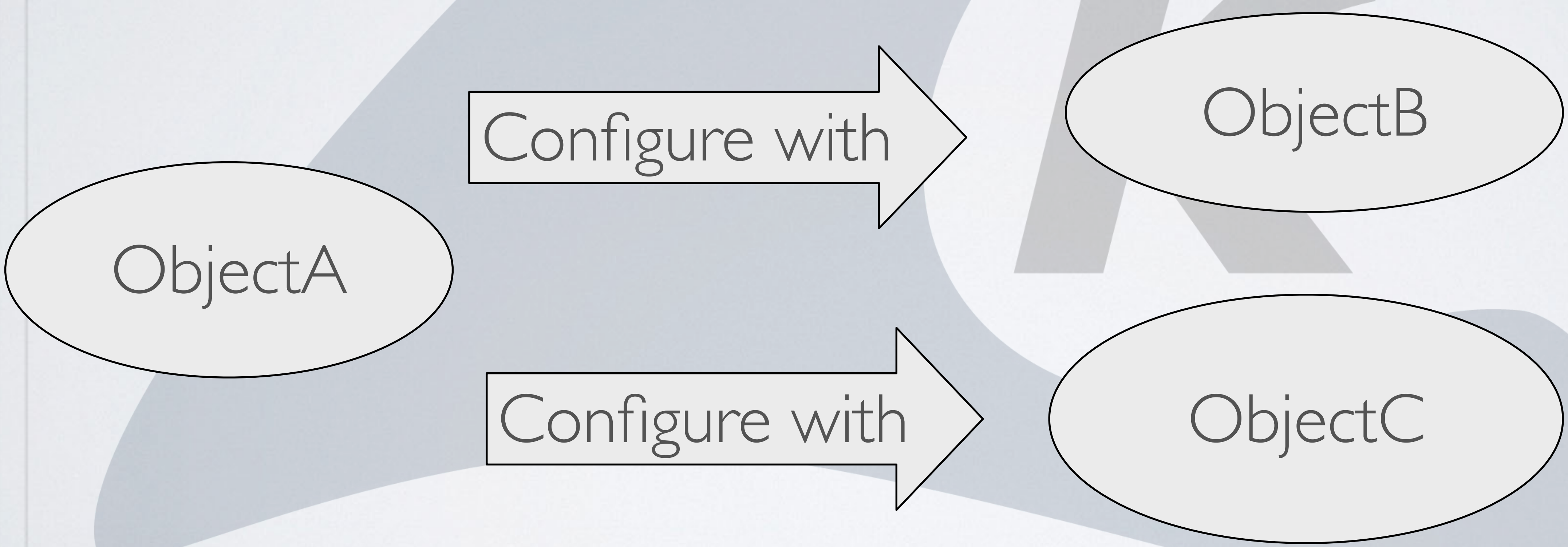
- Inheritance-based designs – Considered as white box since the inheritance structure must be understood

Overrides through inheritance



COMPOSITION – BLACK BOX

- Composition-based designs are considered Black Box as configuration occurs through association, delegation, and forwarding.



STRATEGY PATTERN
PURPOSE: BEHAVIORAL SCOPE: OBJECT

- Intent – Make algorithms interchangeable. Clients using the Strategy can utilize varying combinations of algorithms.
- Also Known As – Policy

STRATEGY

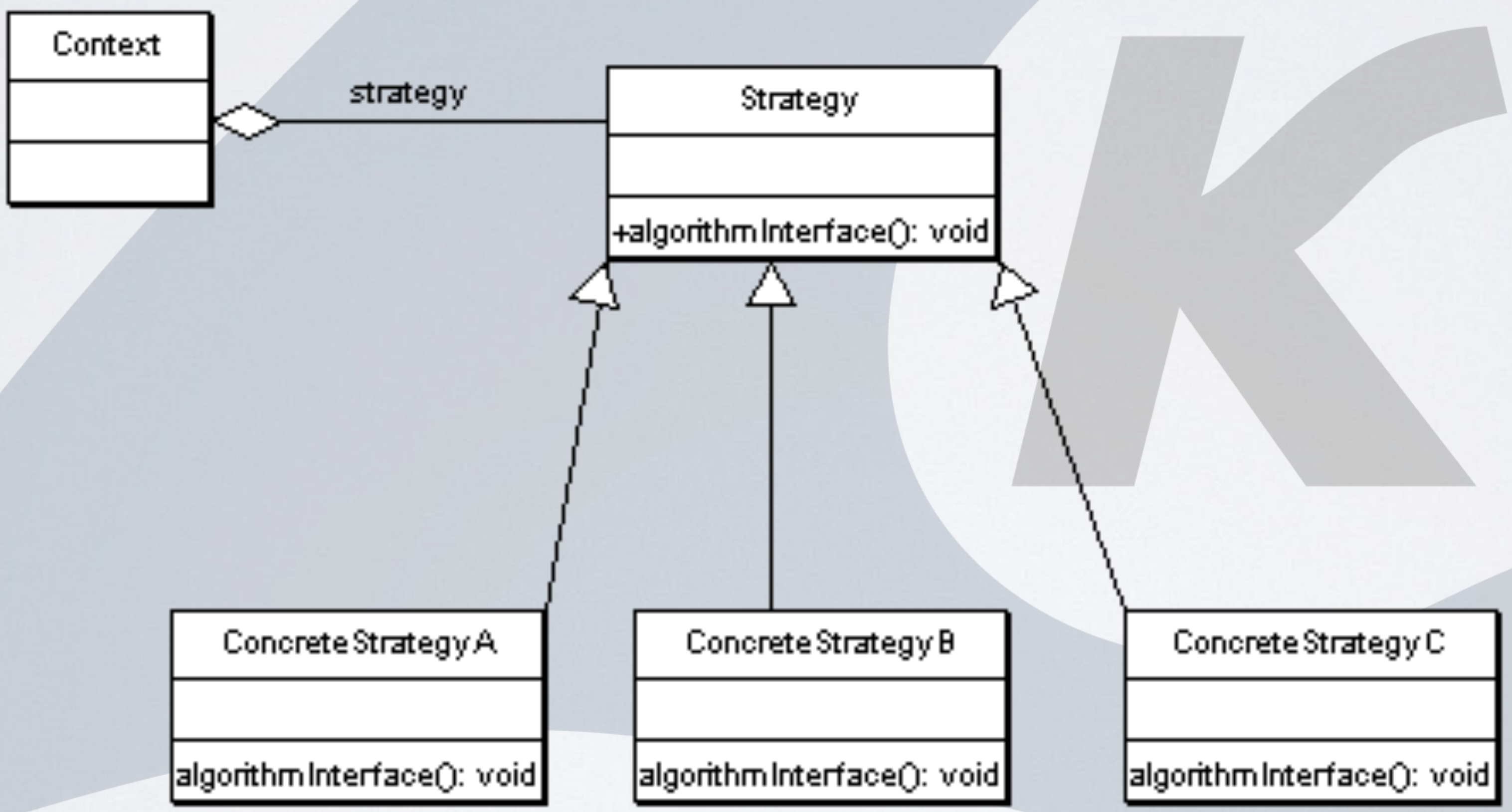
- Motivation
  - Create calculators that have an arbitrary configuration of operations
    - Inheritance requires large number of calculator types
    - Difficult to reconcile arithmetic operations in the hierarchy

STRATEGY

- Applicability – Use when
  - Related classes differ in behavior
  - Variations of an algorithm is required
  - Hide complex data structures and behavior
  - A class defines multiple behaviors that result in conditional logic to implement

STRATEGY

- Structure



STRATEGY

- Participants
  - Strategy
    - Algorithm supported by all strategy algorithms (subclasses)
  - ConcreteStrategy
    - Implements strategy interface algorithm
  - Context
    - Is associated with a ConcreteStrategy Object(s)
    - May define an access interface accessed from a ConcreteStrategy

STRATEGY

- Collaborations
  - Strategy and Context interact to implement and carry out a specific algorithm.
  - Context can pass itself as an argument that can then be called back upon from the strategy. (sometimes referred to as double dispatching)
  - Context forwards requests to a currently configured strategy(s). A family of strategy algorithms can usually be chosen.

STRATEGY

- Consequences (Benefits and Drawbacks)
  - Families of related algorithms
  - Alternative to sub classing
  - Strategies eliminate conditional statements
  - Choice of implementations
  - Clients must be aware of different Strategies
  - Communication overhead between Strategy and Context
  - Increased number of objects

EXERCISE #2 - STRATEGY

- Apply Strategy pattern to Calculator

EXERCISE #3 – CALCULATOR GUI

- This exercise will explore the flexibility of a composition based design by an exploiting graphical user interface design

ABSTRACT FACTORY
PURPOSE: CREATIONAL SCOPE: OBJECT

- Intent – Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Also Known As - Kit

ABSTRACT FACTORY

- Motivation – Calculator operations are defined and installed in logical groupings before they can be utilized. Users have to ensure that required operations have been installed.

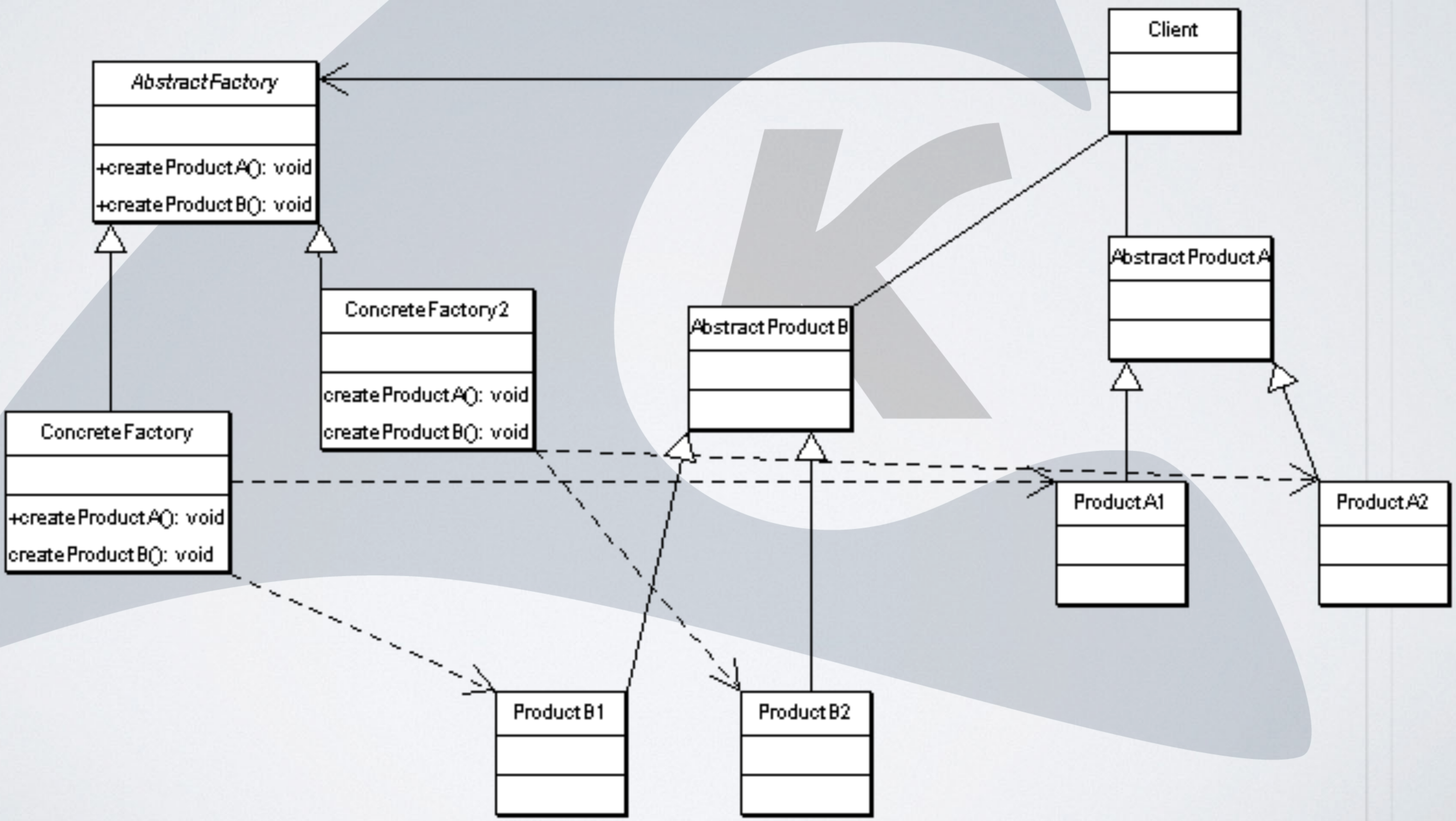
- The AbstractFactory pattern helps produce Calculator instances with a logical grouping of operations.

ABSTRACT FACTORY

- Applicability – Use When
  - A system should be independent of how its products are created, composed, and represented
  - A system should be configured with one of multiple families of products.
  - A family of related product objects is designed to be used together, and you need to enforce this constraint.
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

ABSTRACT FACTORY

- Structure



ABSTRACT FACTORY

- Participants
  - AbstractFactory – Interface for creating product objects
  - ConcreteFactory – product object implementation
  - AbstractProduct – Product interface
  - ConcreteProduct – Product implementation
  - Client – References abstract factory and abstract product types

ABSTRACT FACTORY

- Collaborations
  - Client is given access to a single instance of a ConcreteFactory that is used to obtain concrete product instances
  - AbstractFactory defers creation of product objects to ConcreteFactory subclasses.

ABSTRACT FACTORY

- Consequences (Benefits and Liabilities)
  - It isolates concrete classes
  - It makes exchanging product families easy
  - Promotes consistency among products
  - Supporting new kinds of products is difficult

EXERCISE #3 – ABSTRACT FACTORY

- This exercise applies the abstract factory to create different types of calculator instances

PROTOTYPE
PURPOSE: CREATIONAL SCOPE: OBJECT

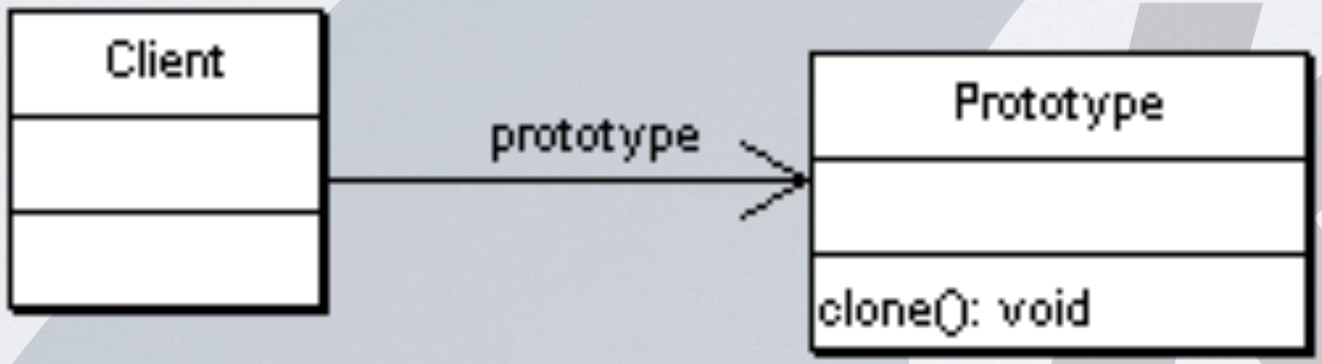
- Intent
  - Create objects using a prototypical instance
- Motivation
  - Create custom calculator types from copies (prototypes) of existing calculators

PROTOTYPE

- Applicability – Use when
  - System should be independent of how products are created, composed, and represented
  - Classes to instantiate are specified at runtime with dynamic loading
  - Avoid class hierarchies of classes that have parallel product hierarchies
  - Classes require a small amount of state to be initialized when instances are created. Cloning instances may be more convenient

PROTOTYPE

- Structure



PROTOTYPE

- Consequences
  - Many of the same consequences as AbstractFactory
  - Adding and removing products at runtime (easier to data drive)
  - Specifying new objects by varying values
  - Specifying new objects by varying structure
  - Reduced Subclassing

EXERCISE #4 – PROTOTYPE PATTERN

- Apply the prototype design pattern to create various calculator types

STATE PATTERN
PURPOSE: BEHAVIORAL SCOPE: OBJECT

- Intent – Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Also Known As – Objects for States

STATE PATTERN

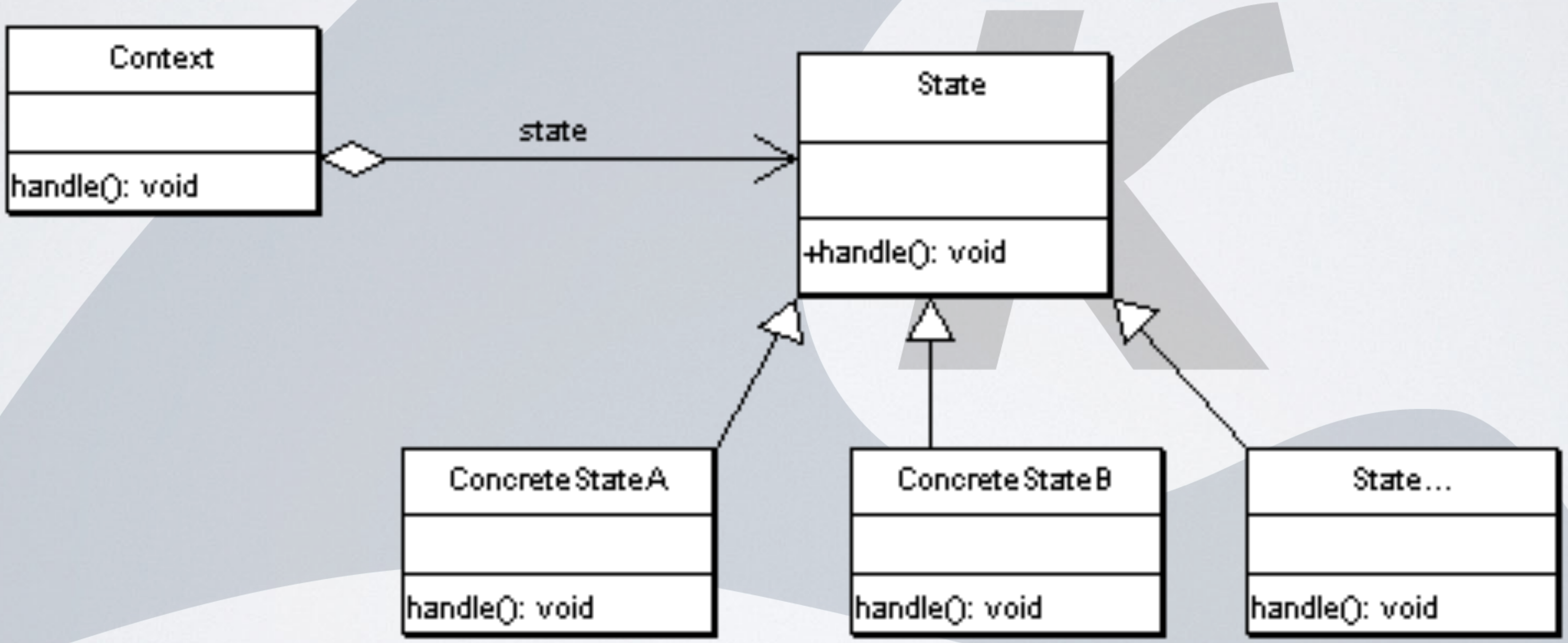
- Motivation
  - The calculator needs to vary output results based upon the type of operation. The state pattern allows an open ended number of display types to be configured

STATE PATTERN

- Applicability – Use the state pattern when
  - An objects behavior depends on its state, and I must change its behavior at run-time depending on that state
  - Operations have large conditional expressions that depend upon object state

STATE PATTERN

- Structure



STATE PATTERN

- Participants
  - Context
    - Defines usage interface
    - References an instance of a concrete state class
  - State
    - Defines interface for state behavior as applied to the context
    - ConcreteState subclasses will honor interface
  - ConcreteState subclass
    - Implements behavior accessing and manipulating state of the context

STATE PATTERN

- Collaborations
  - Context delegates state requests to the current ConcreteState object
  - Context may pass itself to the current state
  - Clients interface with the Context, they do not have to deal or have knowledge of specific state objects
  - State transition can be performed by either context of state objects

STATE PATTERN

- Consequences
  - Localizes state specific behavior and partitions for different states
  - Explicit state transitions (Class hierarchy has more meaning that simple primitive state values)
  - State objects can be shared (if they have no state)

EXERCISE #5 – STATE PATTERN

- Apply the State Pattern to the calculator so that operation specific output can be displayed

SINGLETON
PURPOSE: CREATIONAL SCOPE: OBJECT

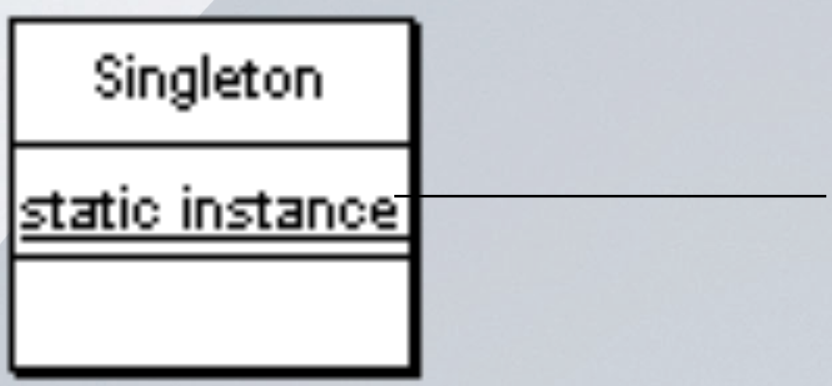
- Intent – Ensure a single instance of a class, and provide global access to it.
- Also Known As -
- Motivation – Calculator operation types are created for each type of calculator instance. Since operations do not have state, attributes, then they can be implemented as singletons. Efficiency is gained through a lower instance creation count.

SINGLETON

- Applicability – Use when
  - Only one instance of a class is required and it must be obtained in a global fashion

SINGLETON

- Structure



Returns single instance

SINGLETON

- Participants
  - Static method returns single instance
  - Instance method is defined to gain access through static method
- Collaborations
  - Clients access singleton operations through Singleton static accessing method

SINGLETON

- Consequences
  - Controlled access to sole instance
  - Reduced name space
  - Permits refinement of operations and representations
  - Permits variable number of instances
  - More flexible than class/static implementations

EXERCISE #6 - SINGLETON

- Apply singleton pattern to calculator pattern

SINGLETON VERSES STATIC IMPLEMENTATIONS

- Static implementations
  - Pros
    - Low memory impact
    - Faster access
    - Convenient
  - Cons
    - Can’t implement interfaces
    - Doesn't have notion of self

SINGLETON VERSES STATIC IMPLEMENTATIONS

- Singleton implementations
  - Pros
    - Can implement interfaces
    - Has notion of self, can be passed as argument
  - Cons
    - Implementation required
    - Instance impact
    - Extra message send required for access to instance

DECORATOR
PURPOSE: STRUCTURAL SCOPE: OBJECT

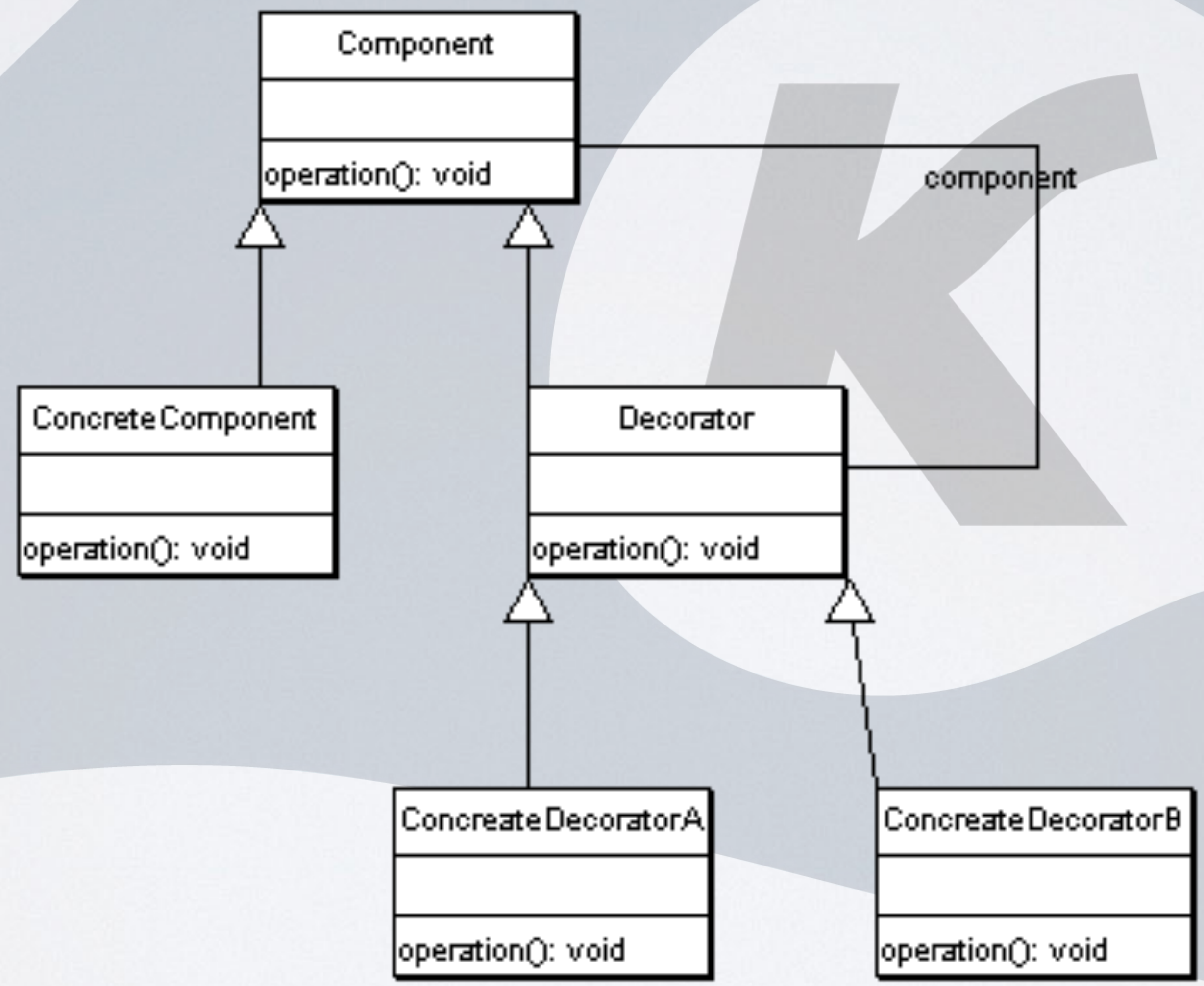
- Intent – Attach additional functionality to an object dynamically, as an alternative to sub-classing
- Also Known As – Wrapper
- Motivation – The calculator outputs results to the console. Output results need to be configurable at runtime.

DECORATOR

- Applicability – Use When
  - Adding responsibility to individual objects dynamically and transparently
  - Withdrawing responsibilities
  - Large number of subclasses are required to implement functionality

DECORATOR

- Structure



DECORATOR

- Participants
  - Component - Object that has operations added or decorated to
  - ConcreteComponent – Concrete object that has base operations added to
  - Decorator – References component object and defines conforming interface
  - ConcreteDecorator – Adds responsibility to Components

DECORATOR

- Consequences – Benefits or liabilities
  - More flexibility that static inheritance
  - Avoids feature-laden classes high up in the hierarchy
  - Doesn’t rely upon object identity. Decorated objects do not have the same identity as the component they decorate
  - Lots of Little Objects (Harder to debug and learn)

EXERCISE #7 - DECORATOR

- In this exercise you will re-factor the Calculator printing functionality into a more abstract design, and apply the Decorator pattern to provide formatting functionality on a dynamic basis.

MEMENTO
PURPOSE: BEHAVIORAL SCOPE: OBJECT

- Intent – Capturing and undoing object state without violating encapsulation
- Also Known As – Token
- Motivation – Provide memory and recall functionality for calculator results.

MEMENTO

- Applicability – Use When
  - Snapshot of all or partial object state must be saved so that it can be restored later
  - A direct implementation requires all private state to be made public, violating encapsulation

MEMENTO

- Structure

MEMENTO

- Participants
  - Memento – Holds encapsulated state, produced by originator
  - Originator – Creates memento with reference to state that can be restored later
  - Caretaker – Undo mechanism

MEMENTO

- Collaborations
  - Caretakers request a memento from an originator
  - Mementos are passive. Only the originator will retrieve or assign state

MEMENTO

- Consequences
  - Preserving encapsulation boundaries
  - It simplifies Originator
  - Mementos can be expensive
  - Defining narrow and wide interfaces (Java utilize inner classes)
  - Hidden cost in caring for mementos

EXERCISE #8 - MEMENTO

- Apply result memory and recall to Calculator implementation

ADAPTER
PURPOSE: STRUCTURAL SCOPE: CLASS,OBJECT

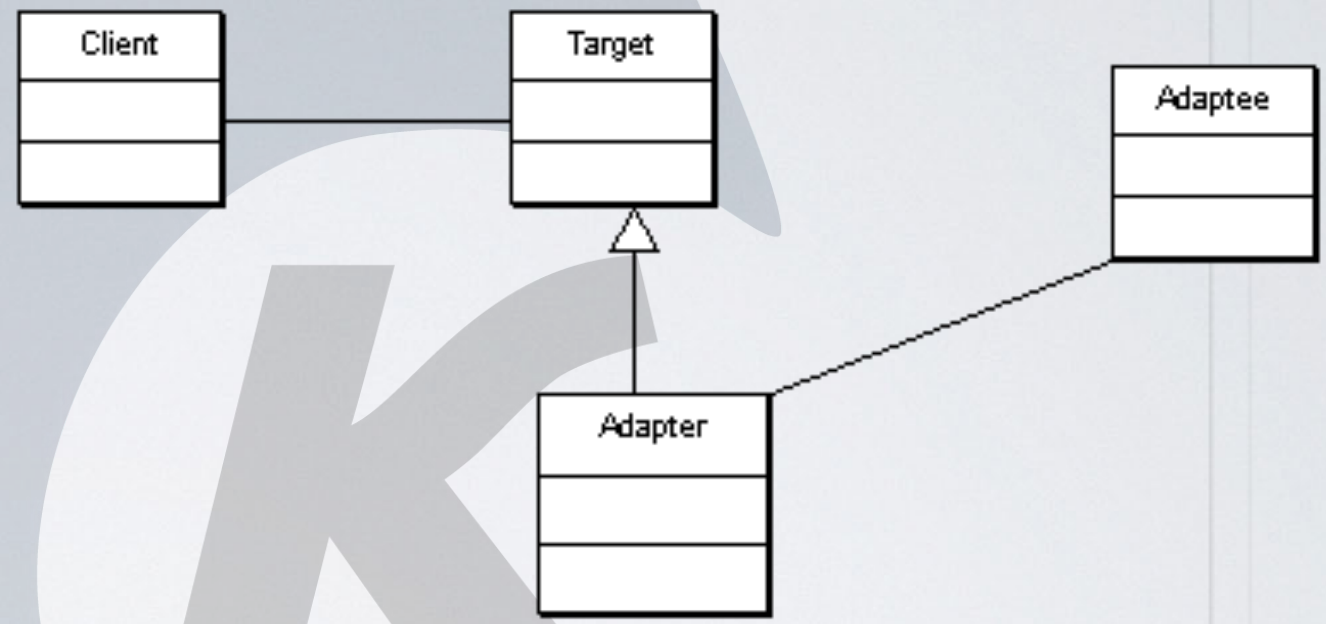
- Intent – Convert, adapt, the interface of a class to another interface
- Also Known As – Wrapper

ADAPTER

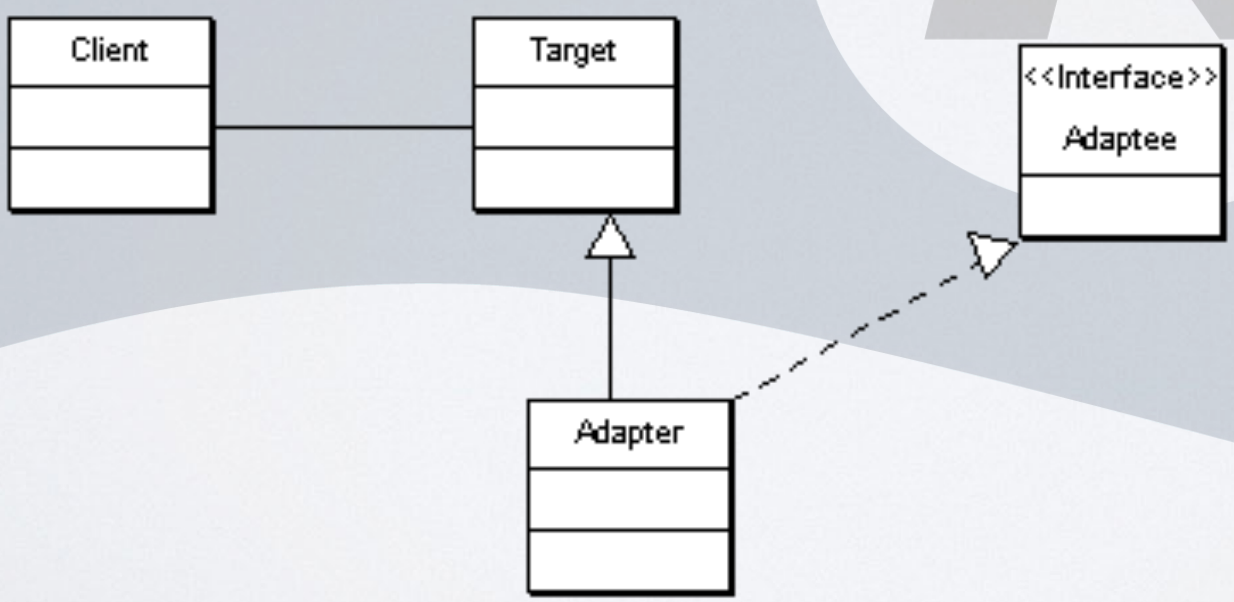
- Applicability (use when)
  - You want to use an existing class, but the interface does not match the one you need
  - You want to create a reusable design that may need to be compatible with unforeseen classes
  - You need to utilize several subclasses but it's impractical to modify all of their interfaces.

ADAPTER

- Structure
  - Class (Java Interface)
  - Object
    - UML is shown below



Object



Class

ADAPTER

- Participants
  - Target – interface accessed by Client
  - Client – collaborates with objects conforming to Target interface
  - Adaptee – Defines interface that target needs adapting to
  - Adapter – Adapts interface of Target to Adaptee interface

ADAPTER

- Collaborations
  - Clients invoke adapter operations that forward to the Adaptee operation

ADAPTER

- Consequences (Object Adapter)
  - Class Adapter
    - Commits Target to concrete Adapter interface
    - Target and Adapter are same instance, additional reference to Adaptee is not required
  - Object Adapter
    - Single Adapter implementations work with many Adaptees
    - Hard to override Adaptee behavior. Requires subclass of Adaptee and reference made by Adapter (Adaptee interface needs to be fully evolved)

EXERCISE #9 – OBJECT ADAPTER

- This exercise will apply the Object Adapter design pattern to the Calculator implementation

INTERFACES OR ABSTRACT CLASSES?

- Both support abstract designs
  - Abstract class designs provide configurable algorithms
    - Method overriding and overloading used to configure implementation
    - Can lead to lots of subclasses, care needs to be taken if method sequence changes.
  - Interfaces provide configurable implementations
    - Supports independent implementations (JDBC, EJB,etc)
    - Interface changes has big impact

OBSERVER
PURPOSE: BEHAVIORAL SCOPE: OBJECT

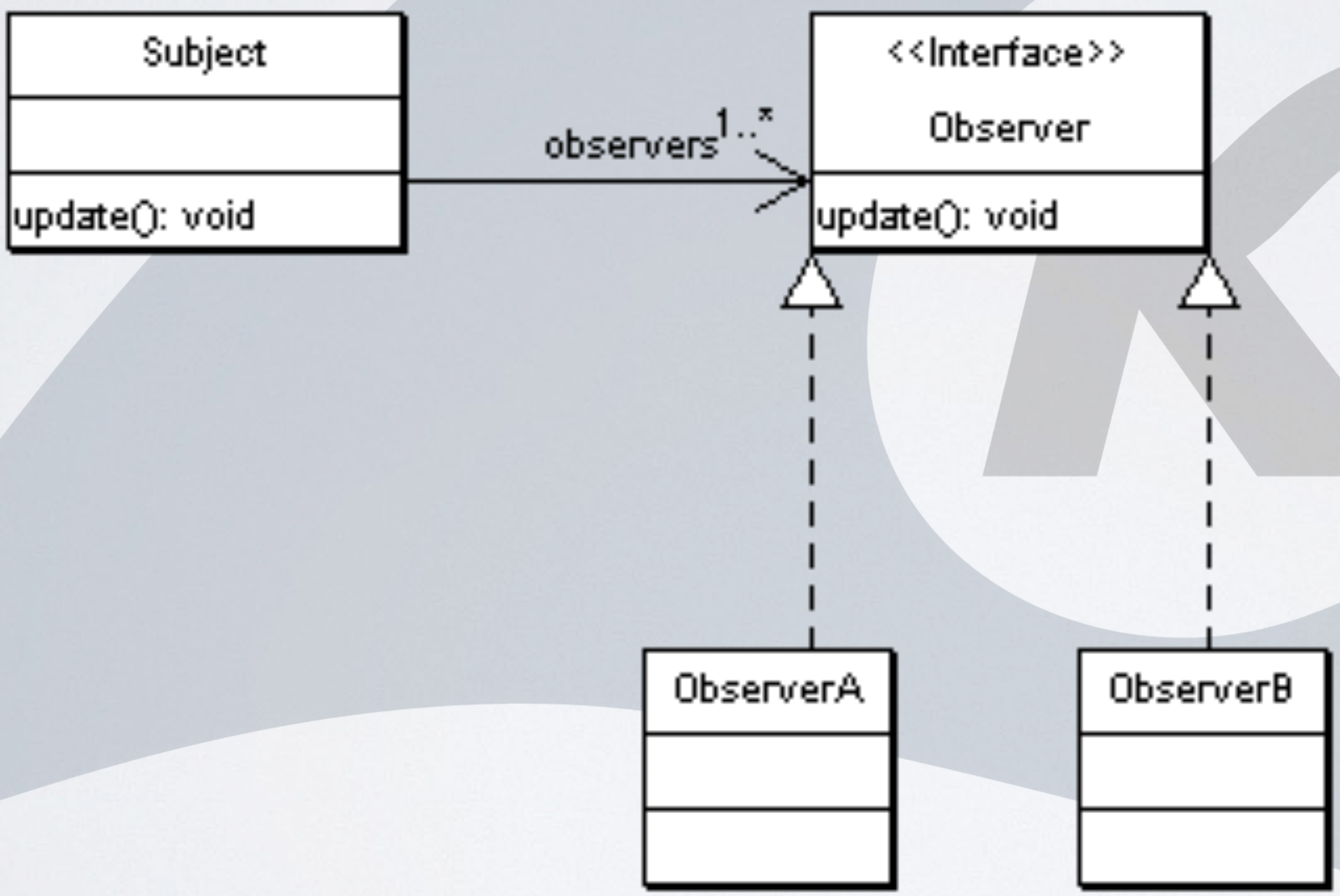
- Intent – Define a one-to-many relationship between objects that are notified and updated automatically
- Also Known As – Dependents, Publish-Subscribe
- Motivation – Whenever calculator operations are performed, notify interested objects of the activity

OBSERVER

- Applicability - (Use When)
  - When change to one object requires an unknown number of objects to be informed. (added dynamically)
  - A loosely-coupled implementation is required

OBSERVER

- Structure



OBSERVER

- Participants
  - Subject – is aware of observers
  - Observer – defines an update mechanism contract
  - ConcreteObserver – honors Observer contract

OBSERVER

- Collaborations
    - Subject notifies Observer instances of action, event change
    - Observers can query subject instance to perform Observer specific behavior
    - Subjects can pass themselves as arguments in the Observer change operation

OBSERVER

- Consequences
  - Abstract coupling between Subject and Observer
  - Support for broadcast communication (Event notification mechanism)
  - Unexpected Updates (Event storm, if lots of Observers exist)

EXERCISE #10 - OBSERVER

- Apply event mechanism to Calculator implementation using the Observer pattern

COMPOSITE
PURPOSE: BEHAVIORAL SCOPE: OBJECT

- Intent – Compose objects into tree structure to represent part-whole hierarchies.

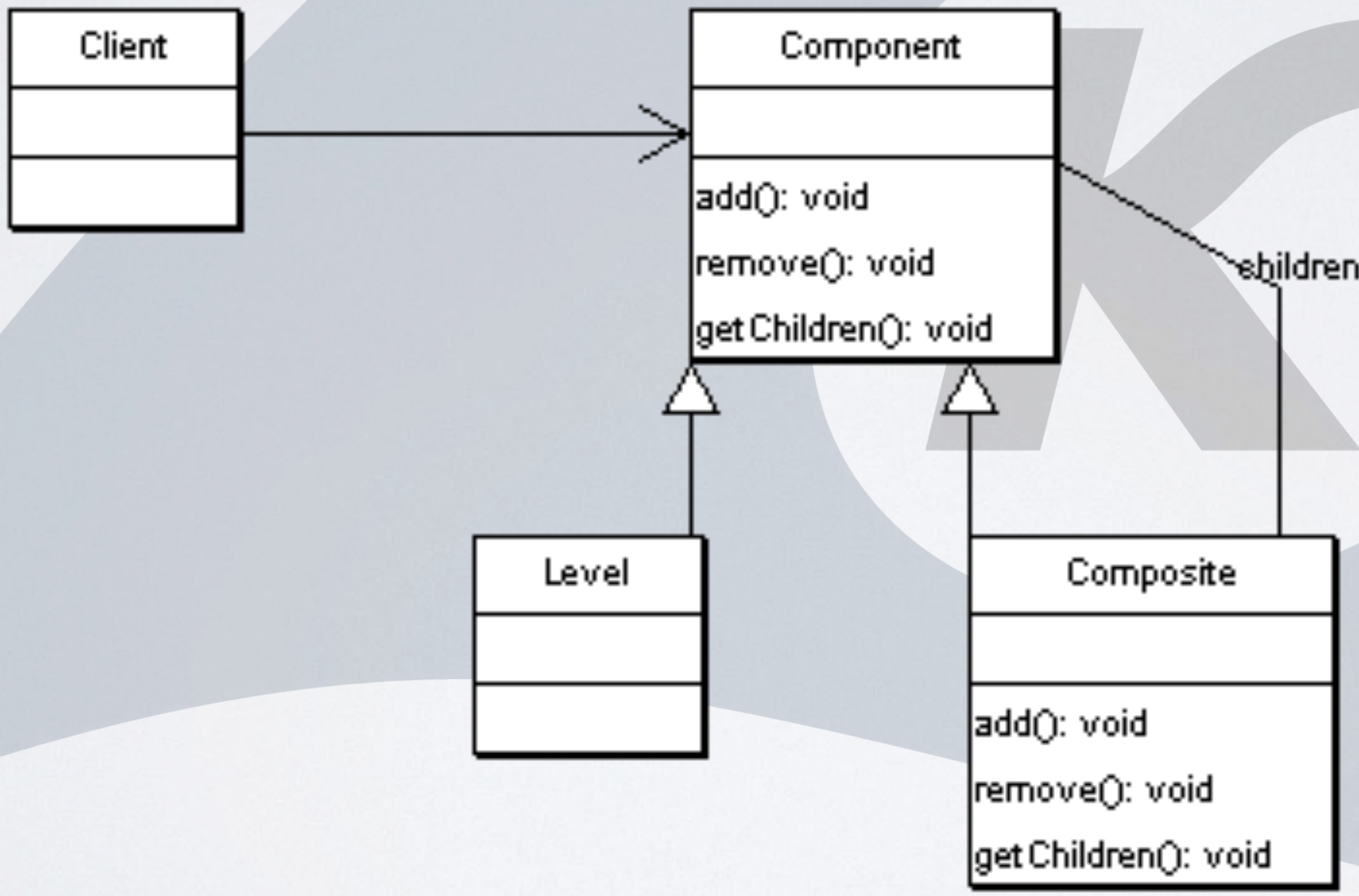
- Motivation – Operation instances represent a single operation. The composite pattern can be utilized to model an equation operation that defines a grouping of operations that are treated as a single Operation instance.

COMPOSITE

- Applicability (Use When)
  - You want to represent part-whole hierarchies of objects
  - Clients should deal with object groups and individual objects in a uniform manner

COMPOSITE

- Structure



COMPOSITE

- Participants
  - Component - Declares interface for all objects in the composition hierarchy
  - Leaf – Component that does not have children
  - Composite – implements behavior for components that have children
  - Client - Manipulates components using the uniform Component interface

COMPOSITE

- Collaborations
  - Clients interact with Component objects with methods defined in the Component hierarchy. Components that are leafs perform the request. Components that are not leafs will forward requests to their children components.

COMPOSITE

- Consequences
  - Hierarchies of classes can be composed to defined simple and complex structures recursively
  - Simplifies and abstracts client interaction with Components
  - Makes it easier to add new kinds of Components
  - Design could be to generalized, harder to restrict the types of components defined in the hierarchy

FLYWEIGHT PURPOSE: STRUCTURAL SCOPE: OBJECT

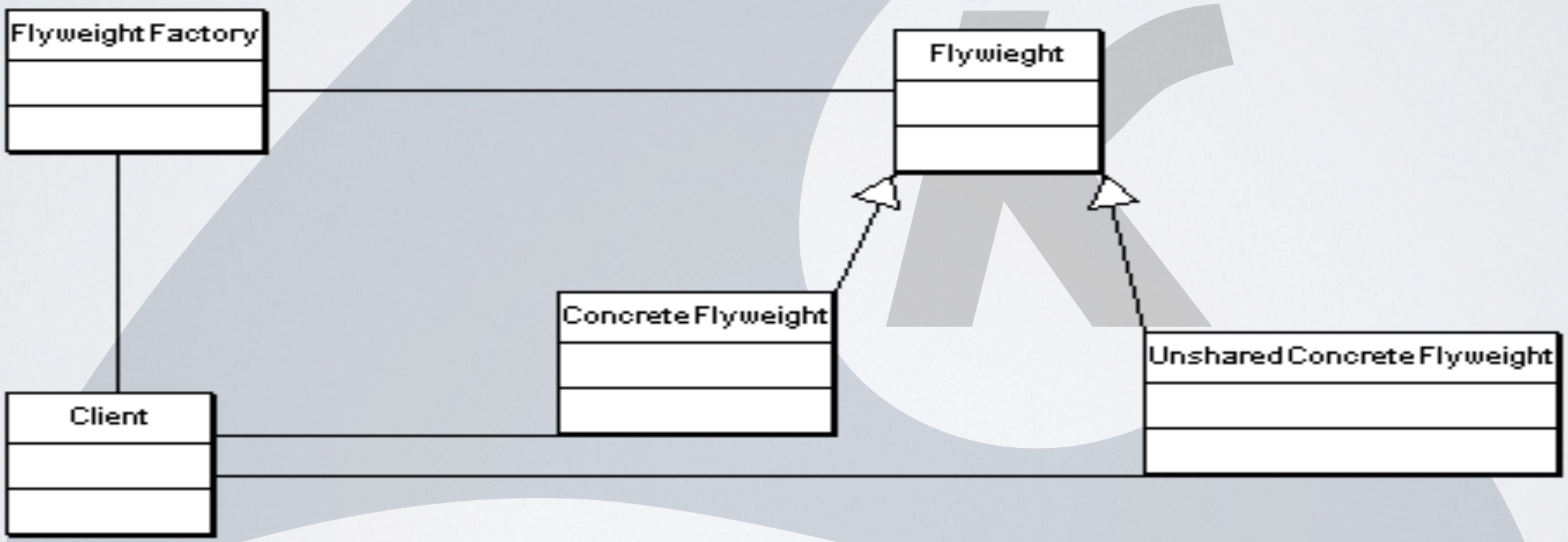
- Intent – Use sharing to support large numbers of fine grained objects efficiently
- Motivation - Operation objects can be shared among calculator instances.

FLYWEIGHT

- Applicability (Use When, All the following are true)
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity

FLYWEIGHT

- Structure



FLYWEIGHT

- Participants
  - Flyweight – declares and interface that flyweights can receive and act on extrinsic state.
  - ConcreteFlyweight – Implements the flyweight interface and adds storage for intrinsic state. (State must be sharable)
  - UnsharedConcreateFlywieght – Implements the flyweight interface, but instances are not shared.
  - FlyweightFactory – creates and manages flyweight objects
  - Client – maintains references to flyweight(s). Computes or stores the extrinsic state of flyweight(s)

FLYWEIGHT

- Collaborations
  - State that a flyweight needs to function is characterized as intrinsic or extrinsic.
  - Clients don't instantiate ConcreteFlyweights directly. Clients must obtain instances from the FlyweightFactory object.

FLYWEIGHT

- Consequences
  - Runtime costs are associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.

PATTERNS

- Not just for Object Design
  - Anti-Patterns
  - Analysis Patterns
  - Testing
  - ….?
- In the primitive form, patterns define a name, description, and solution

BIBLIOGRAPHY

- Design Patterns (Elements of Reusable Object-Oriented Software), Addison-Wesley
- Component Software (Beyond Object-Oriented Programming), Addison-Wesley
- Framing Software Reuse (Lessons From The Real World), Prentice Hall