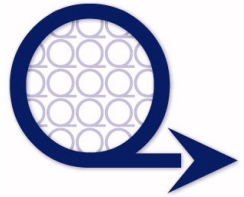


GOF Patterns in Java

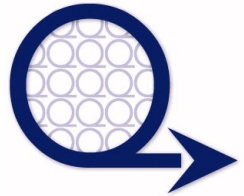
Kirk Knoernschild

Chief Technology Strategist

QWAN*tify*, Inc.

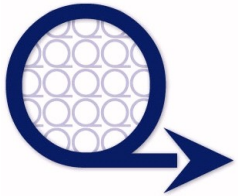
www.qwantify.com

kirk@qwantify.com



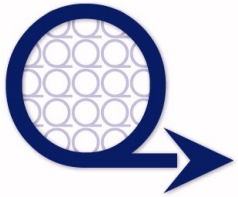
GOF Patterns in Java

- Pattern Review
- The Patterns
- Pattern Retrospective



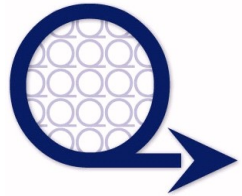
Patterns Defined

- Recurring solution to common problem tailored to context
- Patterns have at least the following:
  - Name, Problem, Solution, Consequences
- Patterns are to design as Algorithms are to code



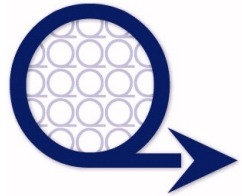
Pattern Review

- Must tailor to context
- Benefits
  - Proven design, communication
- Negative Effects
  - Hype, Proliferation, Overuse, Misapplication



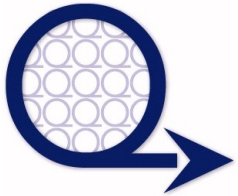
GOF Patterns

- 23 seminal patterns
- Creational (5) (Singleton, Builder)
  - Patterns for creating complex structures
- Structural (7) (Decorator)
  - Patterns for representing complex structures
- Behavioral (11) (Strategy, Command, Observer, Mediator)
  - Patterns for accommodating complex collaborations and algorithms



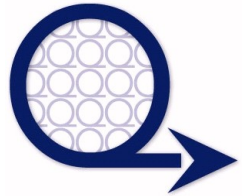
GOF Patterns in Java

- Pattern Review
  - The Patterns
- Pattern Retrospective



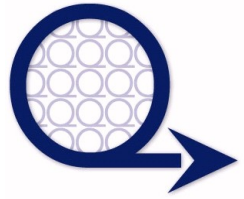
Command

- Intent: Encapsulate a request as an object allowing you to parameterize clients with different requests
- Our Problem: Lot of Data Access Objects (DAO), each with strikingly similar functionality

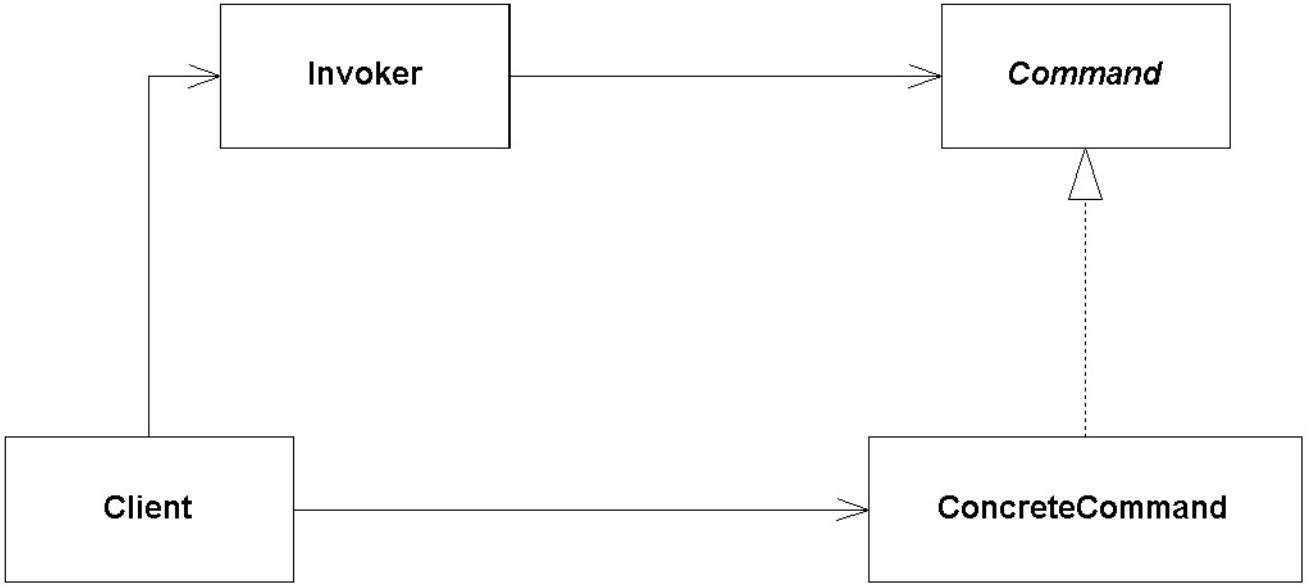


Possible Solutions

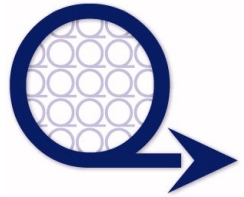
- Alternative: Inherit all DAO from a common base class
- Command: Parameterize a generic DAO with a SQL request
- Tradeoffs
  - Lots of SQL request classes
  - Easy to add new SQL request classes
  - Any class can be a Command if the Command is an interface



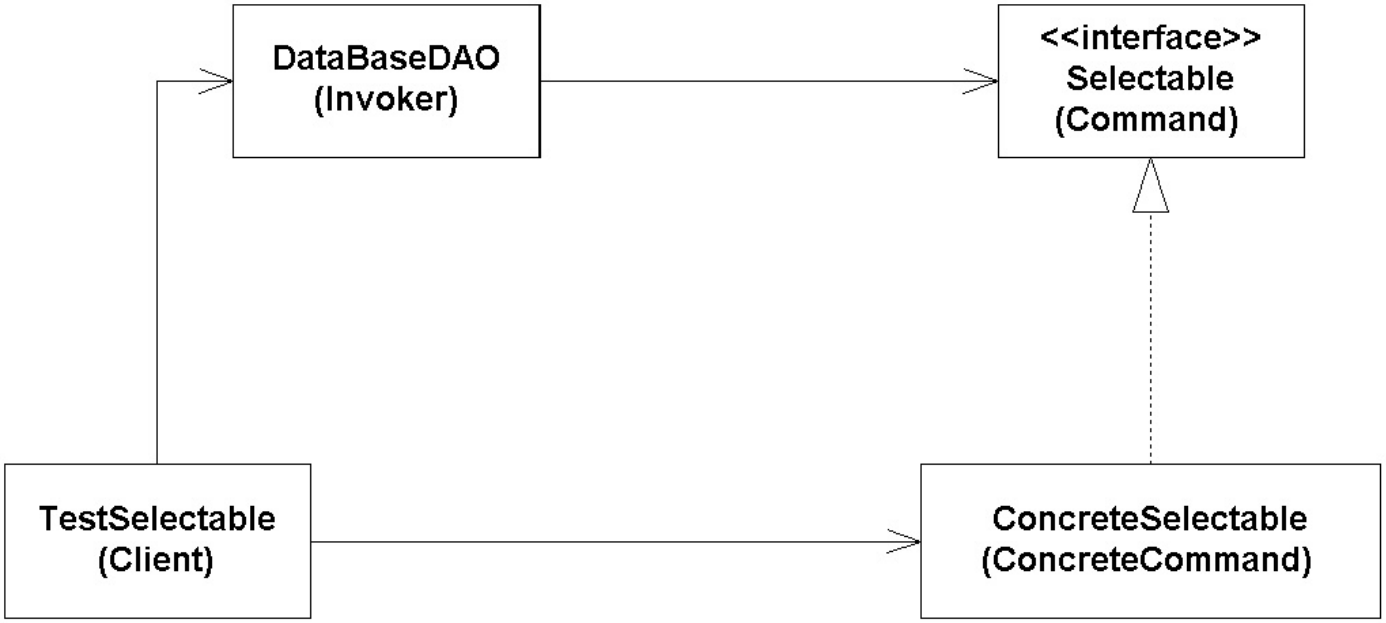
Command Structure

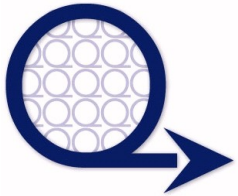


1. Client creates the ConcreteCommand
2. Invoker receives the Command
3. Invoker issues request by calling Command operation(s)



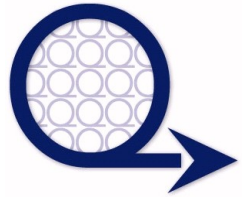
DAO Command Structure





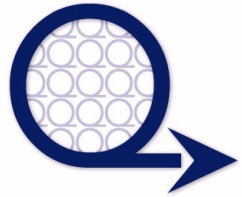
Singleton

- Intent: Ensure a class has only one instance, and provide a global access point
- Our Problem: DataBaseDAO is inherited from a common base class to support other types of datasources

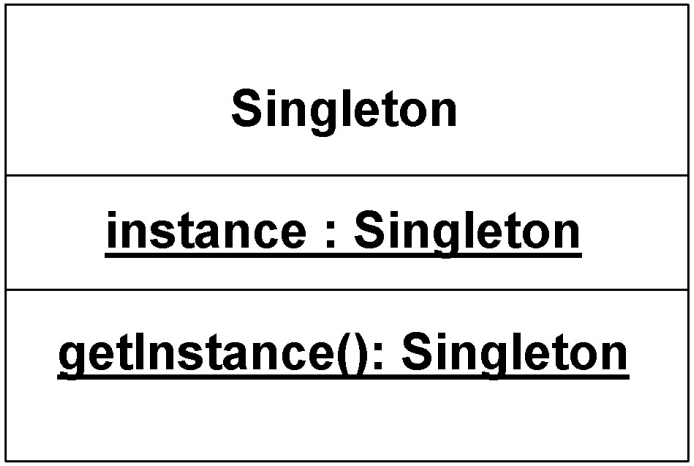


Possible Solutions

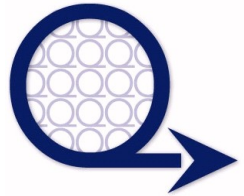
- Alternative: A utility class or static methods
- Singleton: DataBaseDAO with private constructor and static getInstance method
- Tradeoffs
  - Supports polymorphism and callbacks
  - Minimize object creation



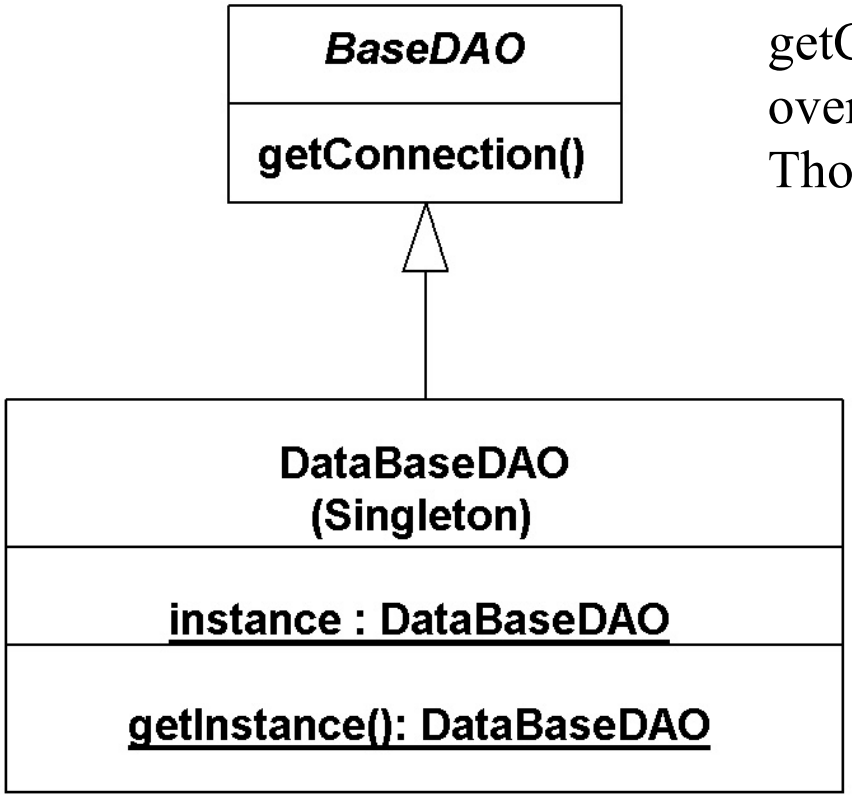
Singleton Structure



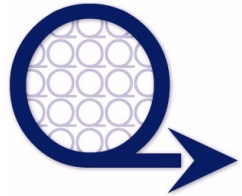
1. Static instance attribute of Singleton datatype
2. Static getInstance method that returns a reference to instance



DataBaseDAO Singleton

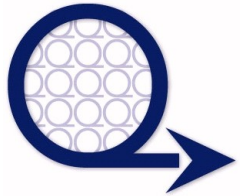


getConnection method can be overidden by other DAO types (ex. Those accessing a legacy system)



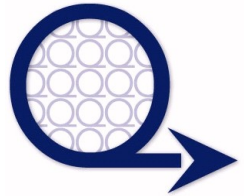
Decorator

- Intent: Add responsibilities to an object dynamically
- Our Problem: Need ability to log and execute SQL statement without bind variables

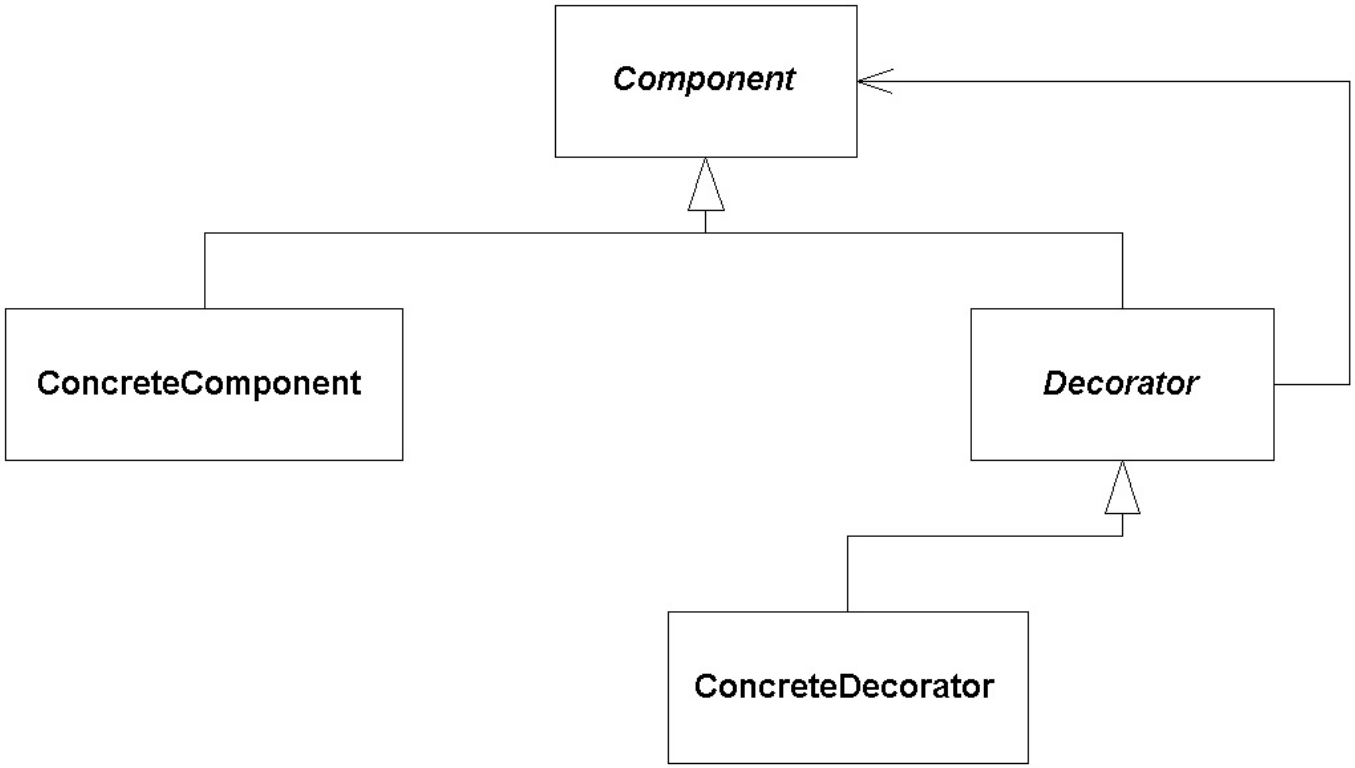


Possible Solutions

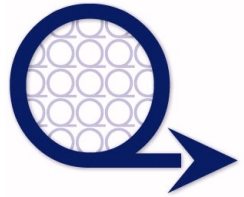
- Alternative: Utility class that accepts Selectable, parses it, and returns SQL string
- Decorator: Class implementing Selectable that accepts Selectable to constructor and returns appropriate SQL String
- Tradeoffs
  - Non-invasive way to enhance functionality
  - Additional classes with more complex learning curve (or maybe just a different way of thinking about utility classes)



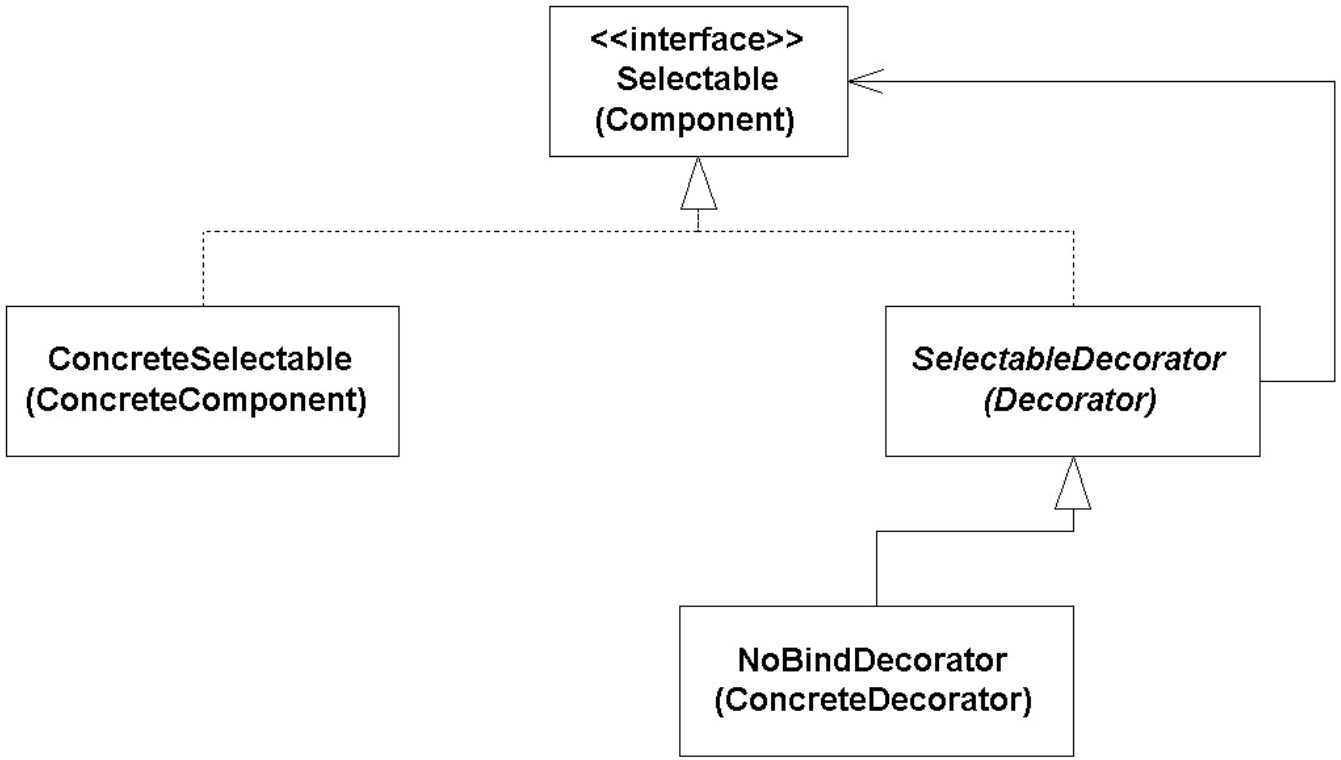
Decorator Structure

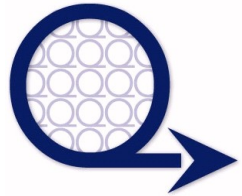


Decorator is configured with a Component
ConcreteDecorator provides custom behavior
Decorator invokes operations on Component



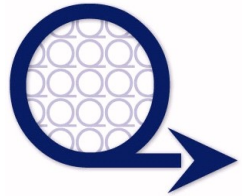
Selectable Decorator





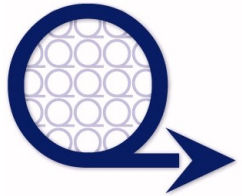
Strategy

- Intent: Define a family of algorithms, encapsulate each one, and allow them to vary independently
- Our Problem: Returning ResultSet to clients of DataBaseDAO is limited to a JDBC datasource

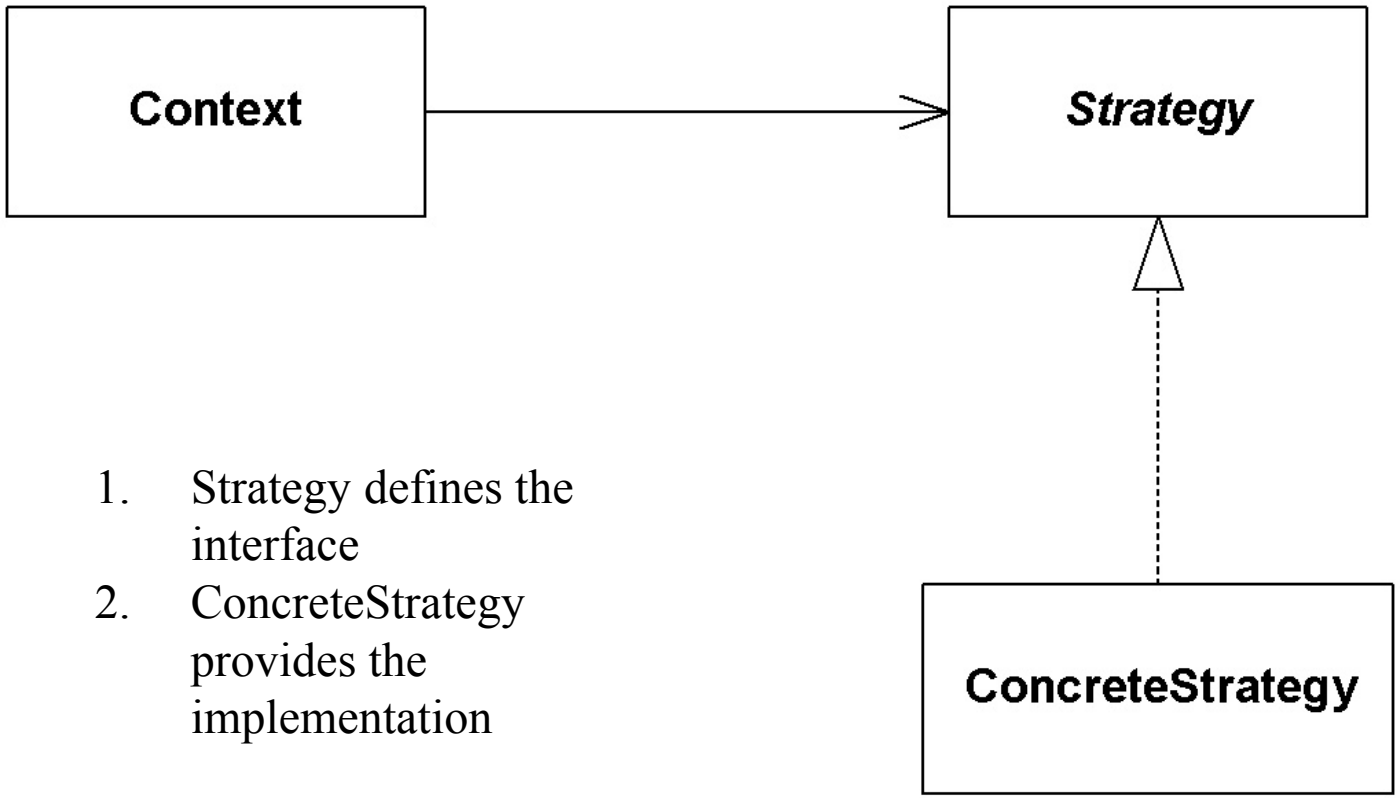


Possible Solutions

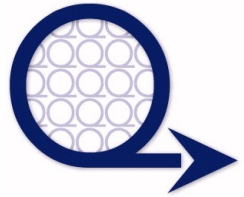
- Alternative: Pass back a bean or implement ResultSet for other datasources
- Strategy: Create a DataCursor that represents a TabularRecordSet
- Tradeoffs
  - No dependency on ResultSet and JDBC
  - More classes and increased complexity



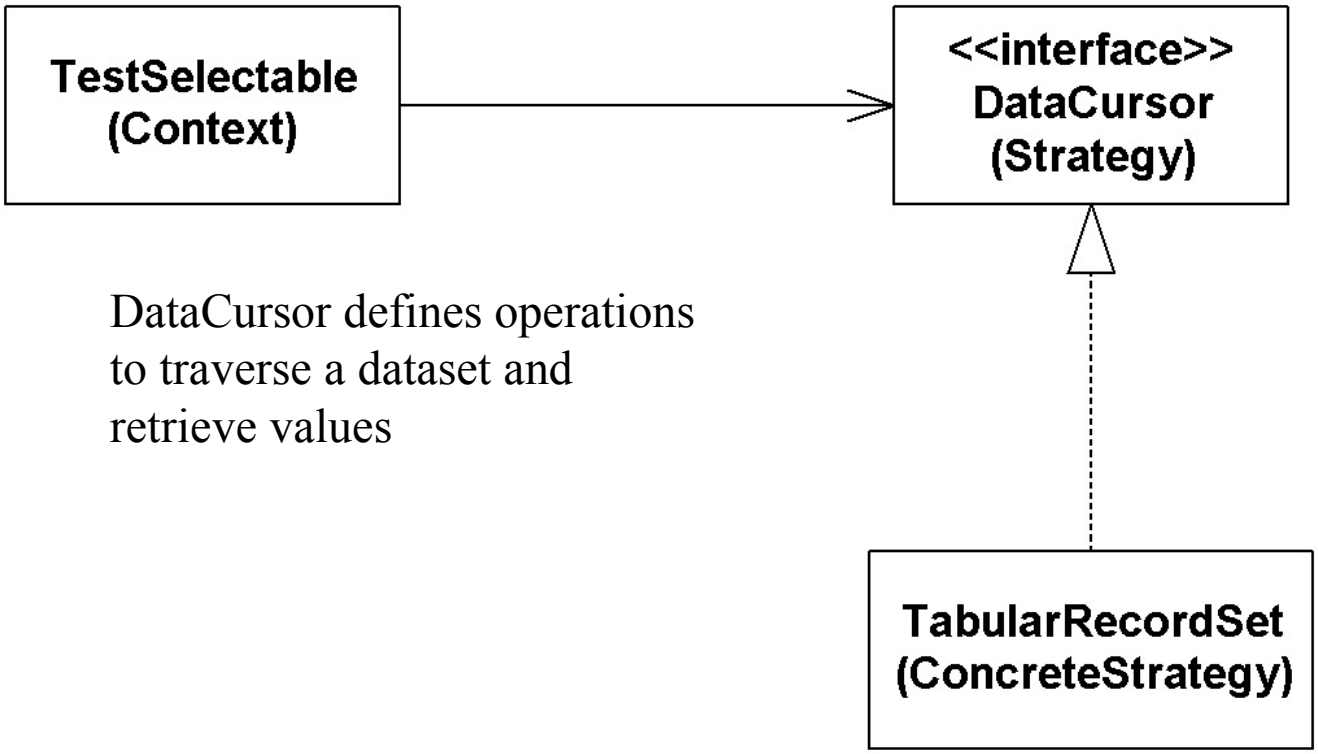
Strategy Structure



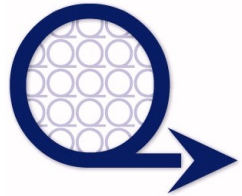
1. Strategy defines the interface
2. ConcreteStrategy provides the implementation



DataCursor Strategy

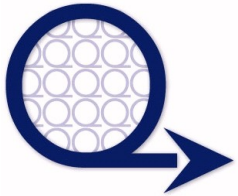


DataCursor defines operations to traverse a dataset and retrieve values



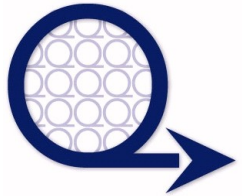
Mediator

- Intent: Define an object that encapsulates how a set of objects interact
- Our Problem: Queue updates and inserts so they are all part of the same Logical Unit of Work (LUW)

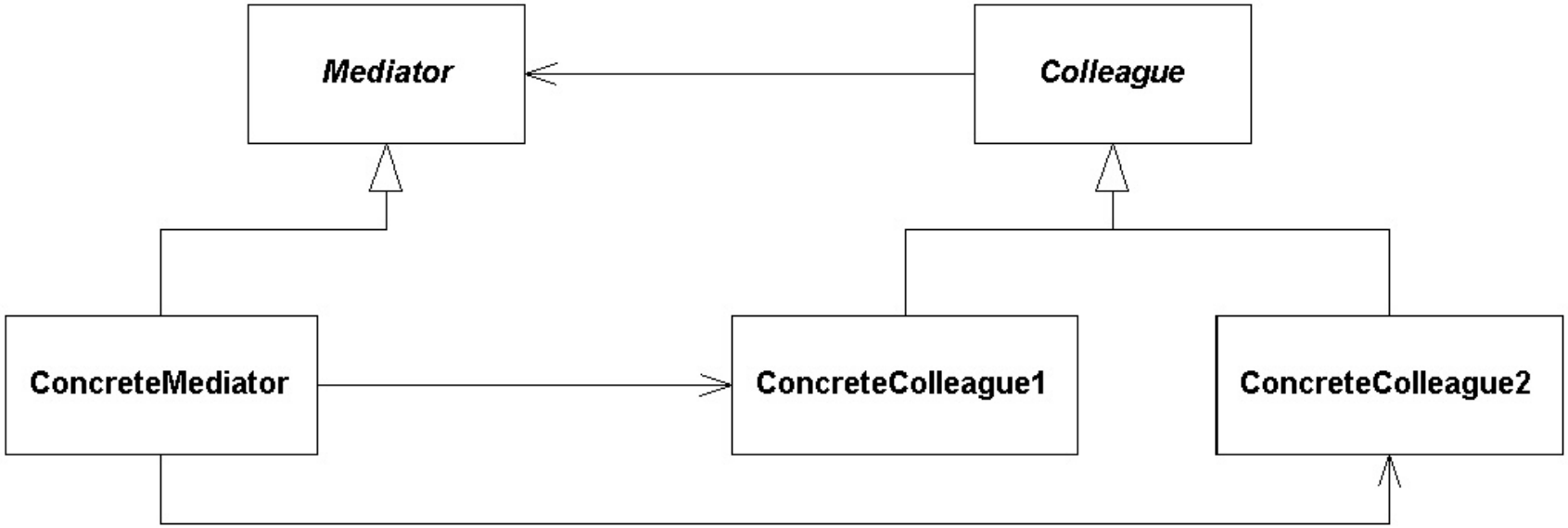


Possible Solutions

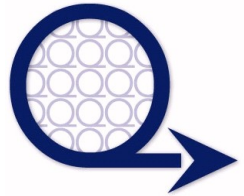
- Alternative: Code it each time or provide utility classes to offer some of the reusable functionality
- Mediator: Create a DAOMediator with which Updateable and Insertable instances are registered
- Tradeoffs
  - Simplifies transaction management
  - Centralizes code resulting in bloated mediators



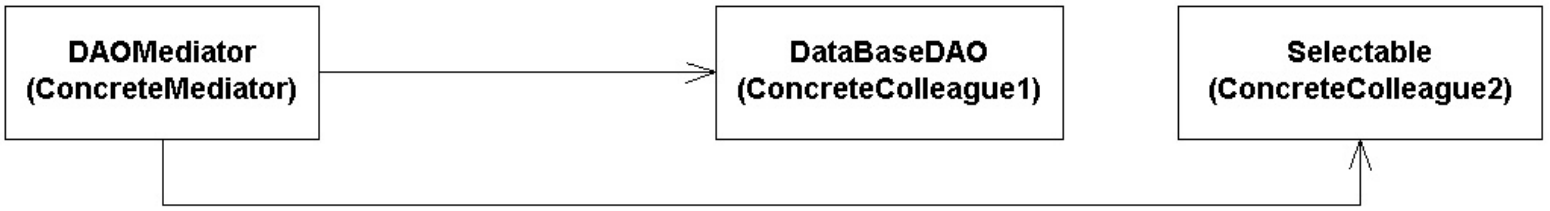
Mediator Structure

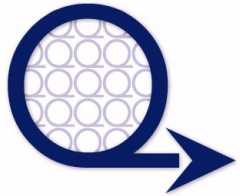


ConcreteMediator manages collaboration between Colleague instances
Colleague instances communicate with each other through Mediator



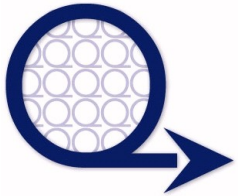
DataBaseDAO Mediator





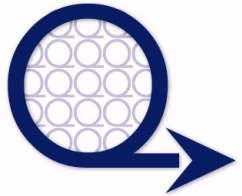
Observer

- Intent: Define relationship between objects so that when one object changes its state, all its dependents are notified and updated
- Our Problem: When using the Mediator for inserts, how do we manage foreign keys for child tables

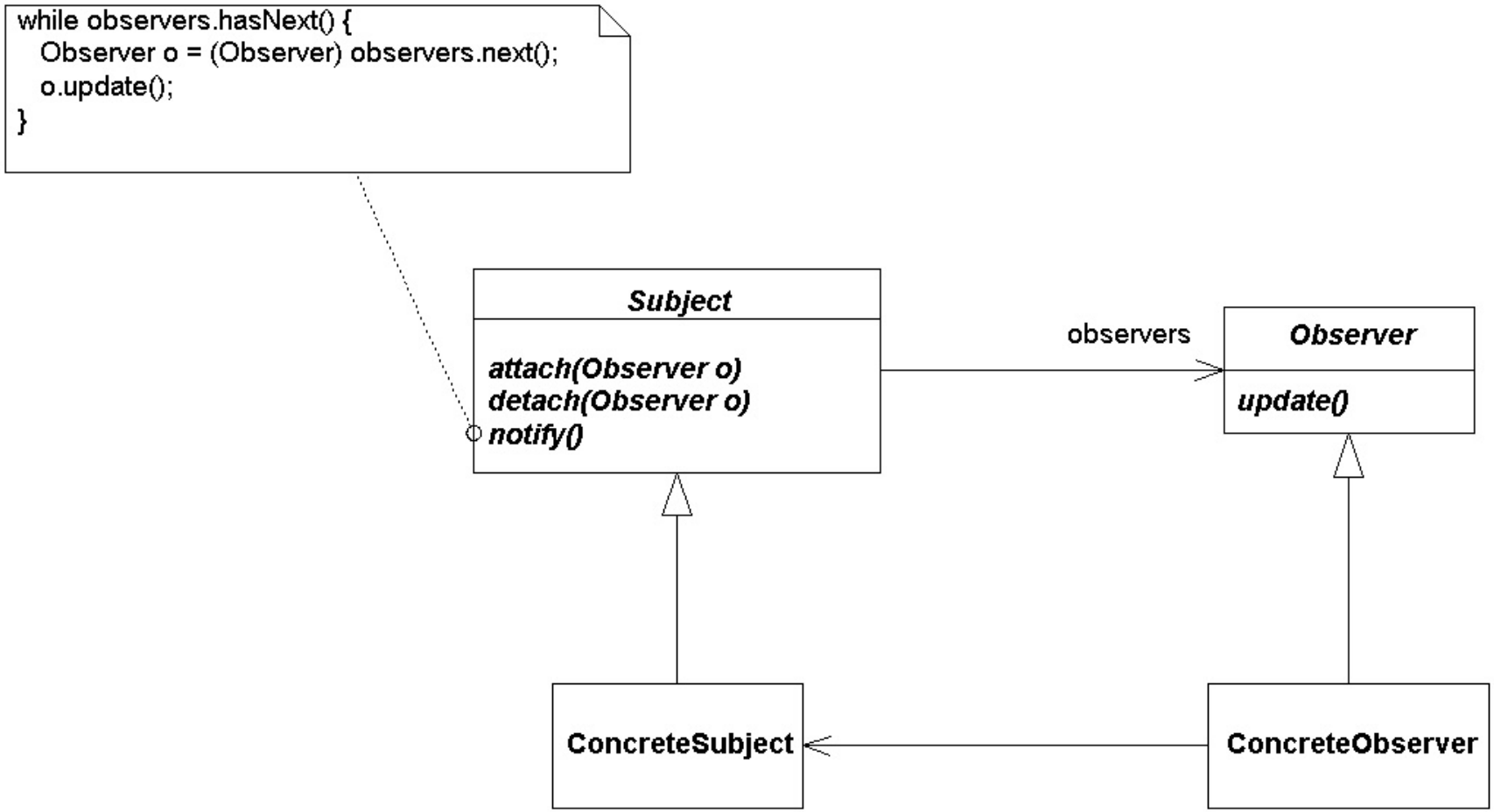


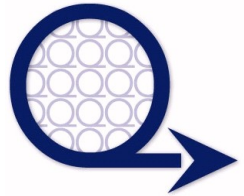
Possible Solutions

- Alternative: Manage keys using an Array
- Listener: Create a KeyListener so that Insertables can be notified of their necessary key values
- Tradeoffs
  - Consistent key management
  - Abstraction complexity

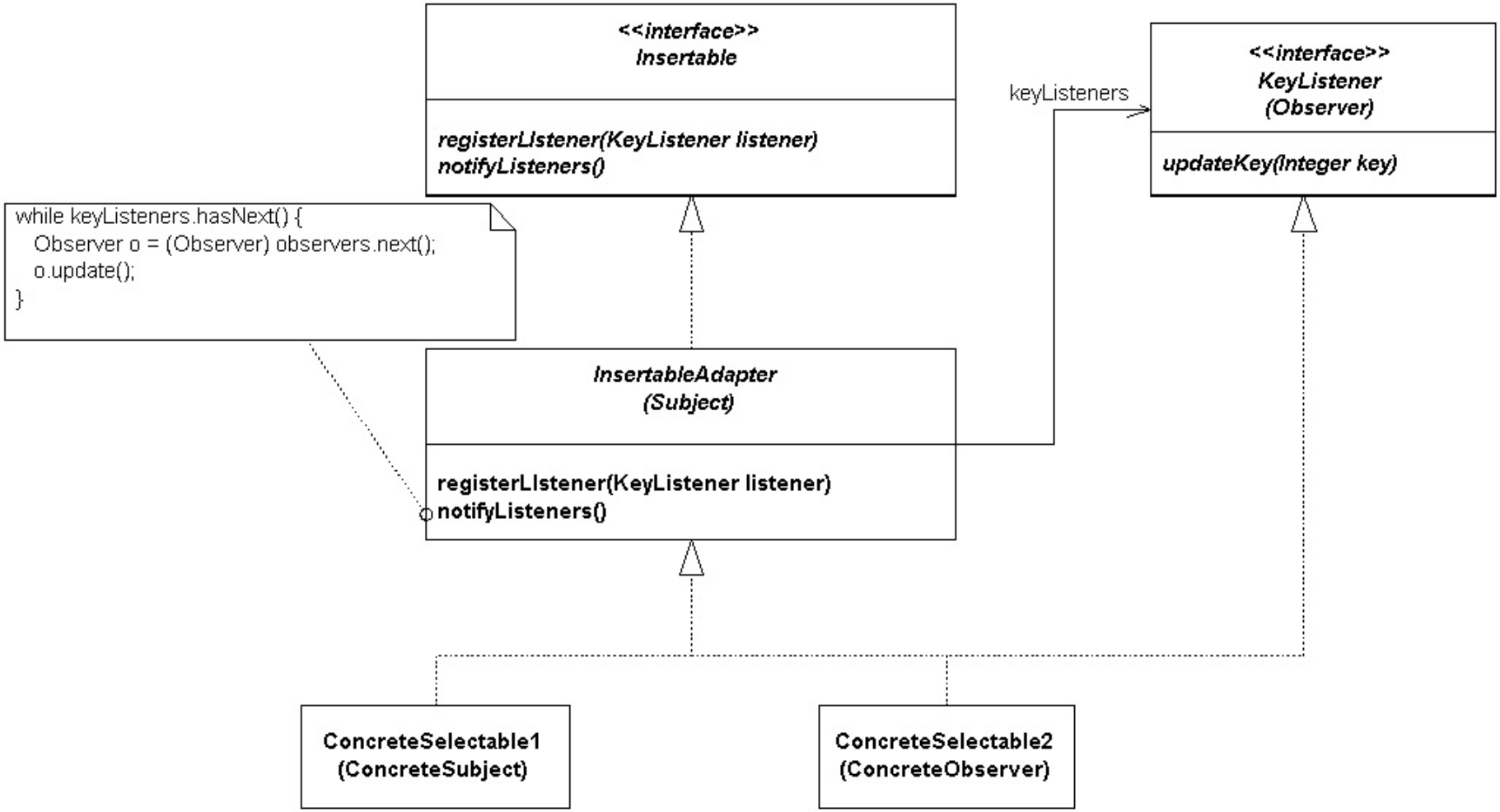


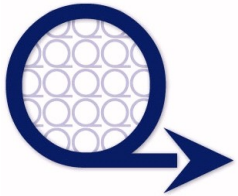
Observer Structure





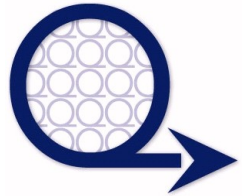
KeyListener





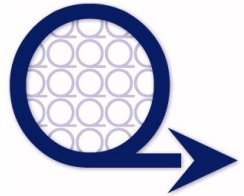
Builder

- Intent: Separate the construction of an object from its representation so that the same construction process can create different representations
- Our Problem: Business objects must be built differently (ie. Lazy load, fully initialized)

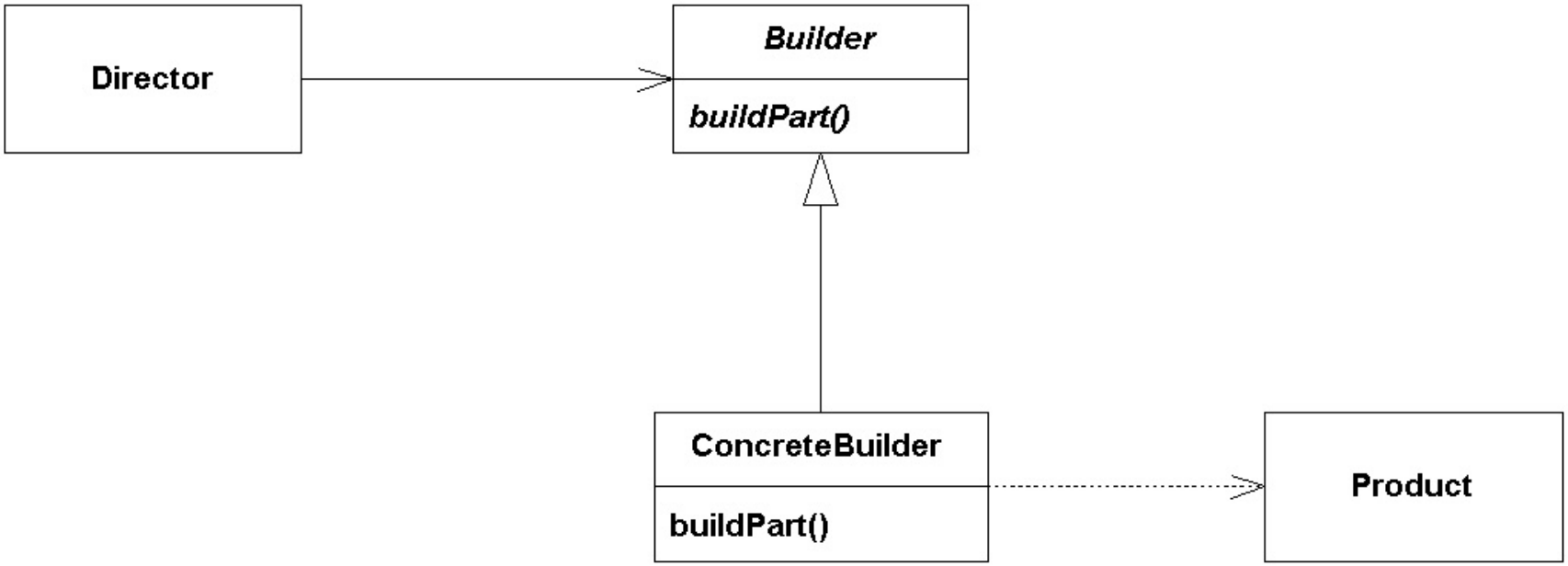


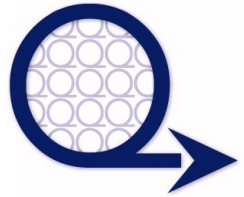
Possible Solutions

- Alternative: Retrieve the data and set the appropriate values on the business object
- Builder: Configure a business object with a builder that initializes the values
- Tradeoffs
  - Flexible way to build business objects using different and unknown constructions processes
  - Adding new business objects (Products) could prove very difficult as all builders may need to be modified

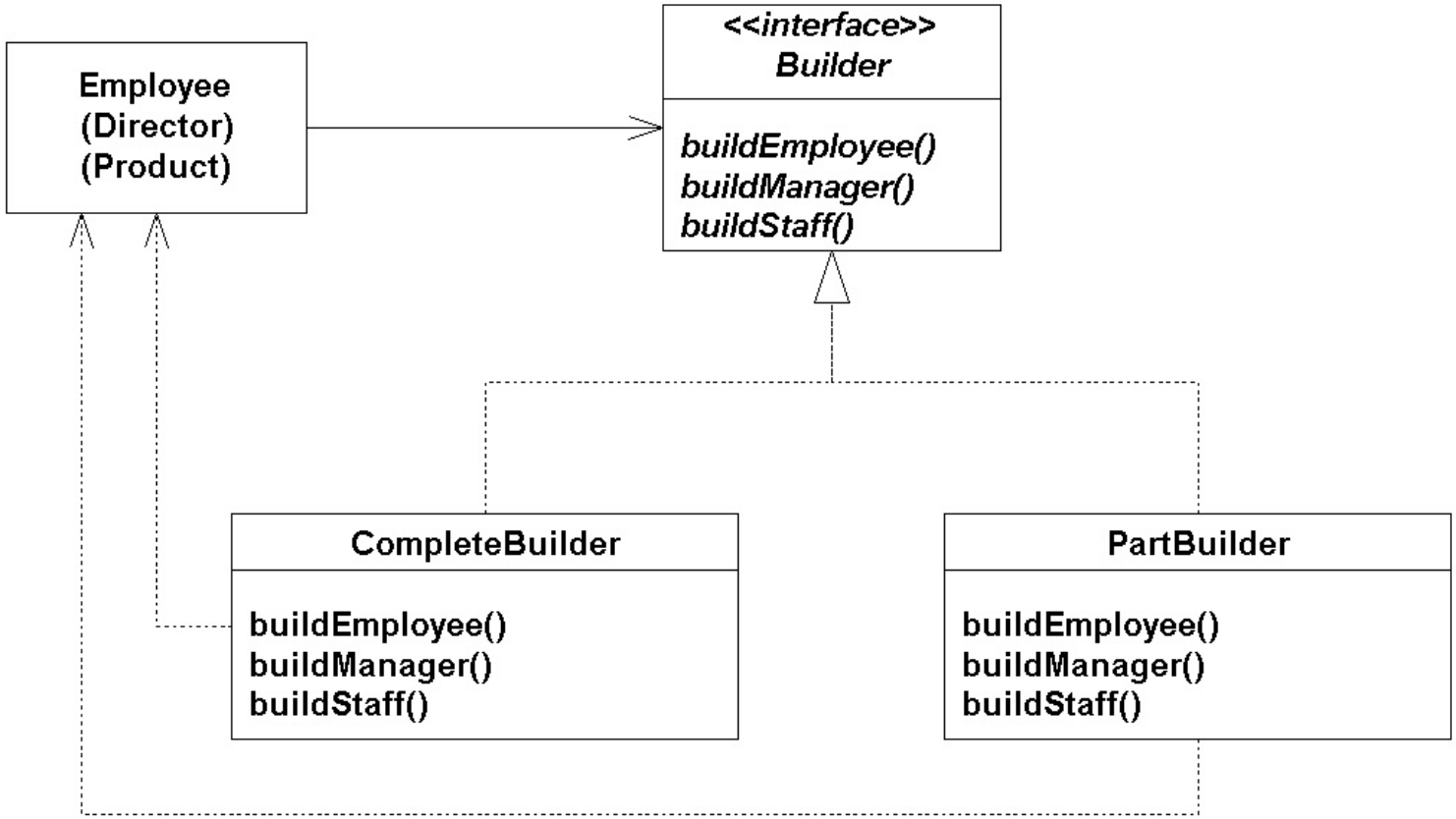


Builder Structure

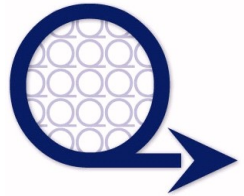




Business Objects

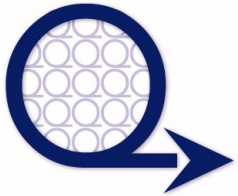


- buildEmployee is actually a Factory Method
- Originating Employee is Director created by Factory Method
- Manager and Staff are Employee instances created by builders



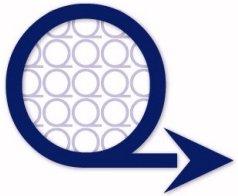
GOF Patterns in Java

- Pattern Review
- The Patterns
- Pattern Retrospective



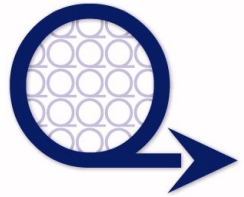
Applying Patterns

- Difficult to identify up-front need
- Need usually arises based on complex behavior or structure
- Knowing patterns help offer template solution
- Tailoring pattern to context based on need for flexibility

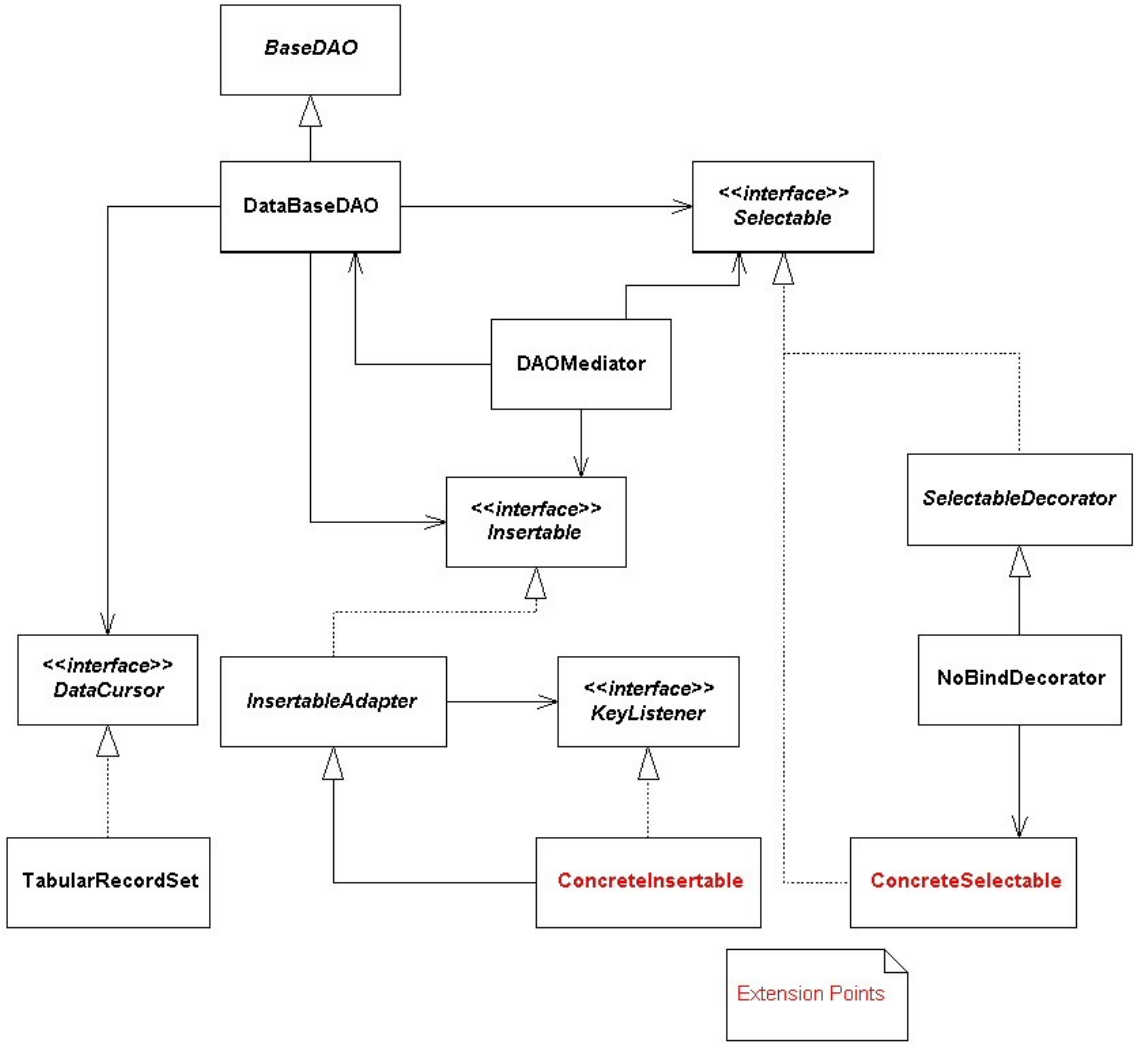


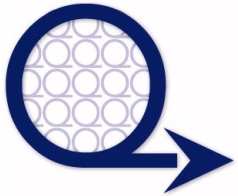
Compound Patterns

- Patterns rarely used individually or in a vacuum
- Single hierarchy/composition structure may consist of many patterns
  - Ex. Insertable is a Command, Adapter, Observer, Decorator



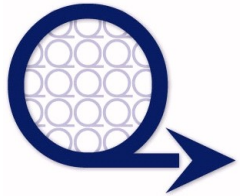
Overall Structure





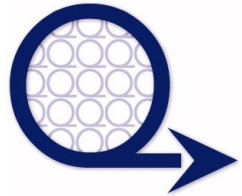
Common Traits

- Abstraction
- Hierarchy
- Coupling
- Cohesion



Gleaning Heuristics

- Capture rules common to many patterns
- Famously, "*favor object composition over class inheritance*"
- Examples of others…
  - Avoid implementation inheritance
  - Abstractly couple classes
  - and many, many more…



Additional Resources

- www.kirkk.com
  - JarAnalyzer download and general information on software development.
- www.qwantify.com
  - Whitepapers, articles, and blogs on a variety of technical topics.
- www.extensiblejava.com
  - Resource devoted exclusively to dependency management.

Please complete your session evaluation forms