PA103 - Object-oriented Methods for Design of Information Systems

Design Patterns

© Radek Ošlejšek
Fakulta informatiky MU
oslejsek@fi.muni.cz

Many levels of software patterns

- Small
  - Coding patterns, e.g. SmallTalk Best Practice Patterns
  - Code refactoring, e.g. Parameters reduction, ...
- Middle
  - Design patterns, e.g. Adapter, Facade, ...
  - Analysis and business patterns, e.g. Accounting, Measurement, ...
- Big
  - Architectural patterns, e.g. SOA, peer-to-peer, layers, ...

History

- 1977-1979: Christopher Alexander (an architect) defined the *pattern* term for building homes, buildings and towns.
- 1987: Kent Back and Ward Cunningham have applied Alexander's ideas to the development of SmallTalk GUI
- 1889-1991: James Coplien: Advanced C++ Idioms book.
- **1995: Gamma, Helm, Johnson, Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software**
- 1997: Martin Fowler: Analysis patterns
- Since that: many conferences and patterns for various domains and levels of abstraction, e.g. Data Modeling Patterns (David Hay), Java Modeling in Color with UML (Peter Coad), Java J2EE patterns, anti-patterns, real-time design patterns, ...

History – Alexander's Pattern Language

- What defines *quality* of buildings?
  - Freedom, lifetime, comfort, harmony
- Pattern: solution of the problem in given context
  - Entrance passage
  - Gradient of privacy
  - Lights in both sides of a room
- Combined patterns => pattern language
  - 2534 patterns, from coarse-grained to fine-grained

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

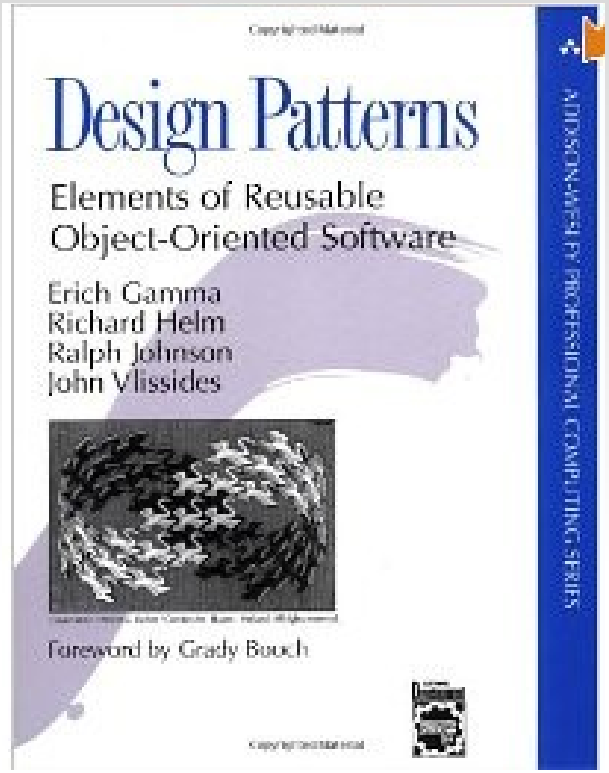
The Timeless Way of Building (1997)

A Pattern Language (1977)

GoF Design Patterns

The Gang-of-Four: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Design Patterns: Elements of Reusable Object Oriented Software (1995).



Why are we, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, called this? Who knows. Somehow the name just stuck. Hopefully like, the original Gang of Four, we have started a small cultural change with "Design Patterns..." And hopefully unlike the original Gang of Four we will not meet such an untimely end for our ("counter-revolutionary"?) ideas.

Vlissidies pronunciation (dialogue from the internet discussion with his long-time colleague):

Q: And I'd like to make sure I pronounce his name correctly.
A: The weird part is I've known John for years and I'm not even sure :-)
The pronunciation I generally use is VLIH-Suh-dees. I've heard others use VLIH-SEE-DEES...

Design Patterns

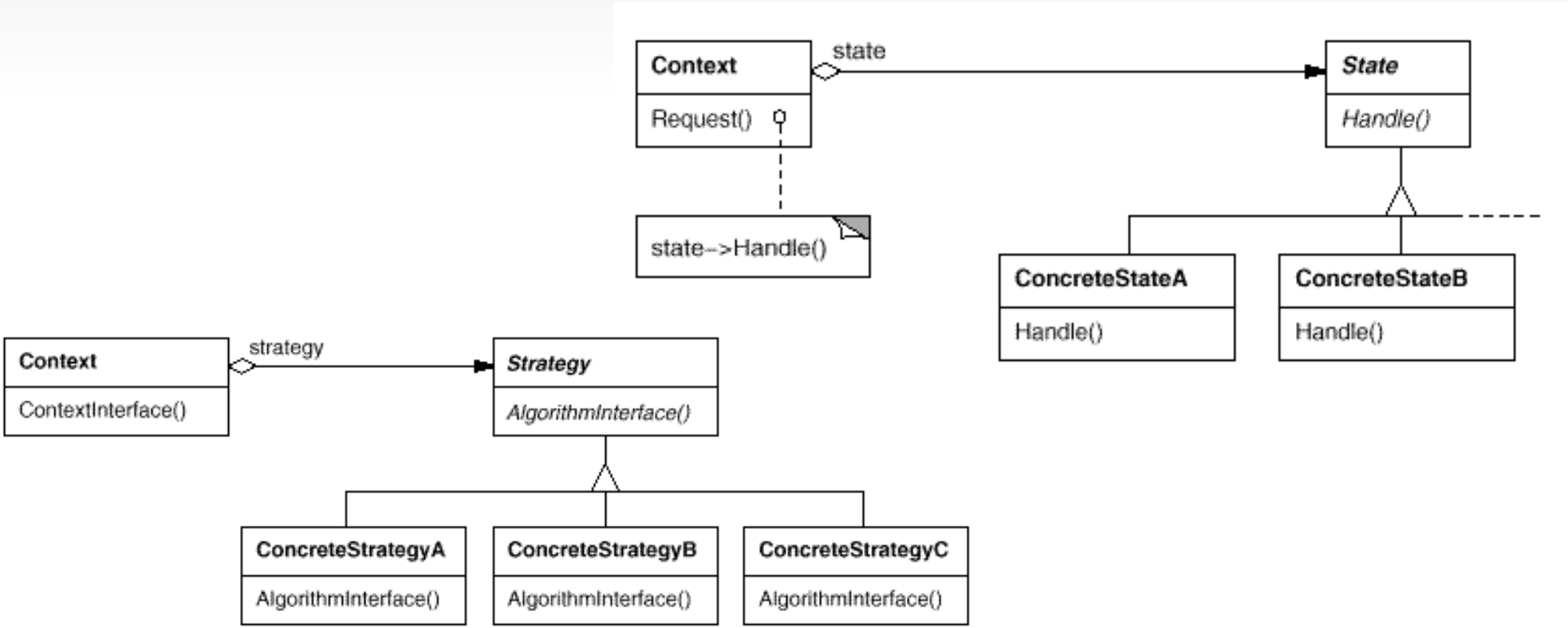
- Design pattern is a general reusable solution to a commonly occuring problems in software design.
  - Description or template of how to solve a problem that can be solved in many different situations.
  - Patterns facilitate reuse of successful software architectures and designs.
  - Patterns capture the static and dynamic structure and collaboration among key participants in software design.
  - Pattern is not a finished design. It's rather a metamodel which have to be instantiated. Moreover, it is necessary to
    - consider compromises and consequences
    - make design and implementation decisions
    - implement them and combine them with other patterns
  - Common vocabulary, e.g. “Here we use the *Observer* pattern.”
    - enhanced communication between/within development teams
    - faster design
    - culture

Patterns vs. Design Confidence

- General inexperience with OO design
  - Is my model correct?
- Patterns improves design confidence
  - You can alway lay the blame on “Gang of Four”
  - Keeps free space for creativity
- Problem: many people know patterns
  - Partially, without the deep understanding,
  - Patterns are better and better as you use them longer and longer

Common Problems

- Wrong premise: The best is to apply as many patterns as possible to my model
- Applying pattern to solve wrong problem
- Cost and sources required to re-design software by means of design patterns
- Everything can be solved by the last learned pattern.
- Patterns have often very similar visual structure. Designer therefore have to fully understand all aspects of patterns.



Essential Elements of Design Patterns

- Pattern name
  - Increases design vocabulary, enhances communication.
- Problem
  - Describes when to apply the pattern.
  - Explains the problem and its context.
  - Often illustrated on example.
- Solution
  - Describes the elements that make up the design, relationships, responsibilities and collaborations.
  - Does not describe specific concrete implementation.
- Consequences
  - Results and trade-offs of applying pattern.
  - Critical for evaluating design alternatives, understanding coasts, understanding benefits of applying pattern.
  - Includes the impacts of a pattern on a system's flexibility, extensibility and portability.

Design Pattern Description

- **Name and Classification:** Essence of pattern
- **Intent:** What it does, its rationale, its context
- **AKA:** Other well-known names
- **Motivation:** Scenario illustrates a design problem
- **Applicability:** Situations where pattern can be applied
- **Structure:** Class and interaction diagrams
- **Participations:** Objects/classes and their responsibilities
- **Collaborations:** How participants collaborate
- **Consequences:** Trade-offs and results
- **Implementation:** Pitfalls, hints, techniques, etc.
- **Sample Code**
- **Known Uses:** Examples of pattern in real systems
- **Related Patterns:** Closely related patterns

Basic Classification of GoF Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
  - Hides specifics of the creation process
  - May want to delay specifying a class name explicitly when instantiating an object
- **Structural patterns:**
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects, e.g. hierarchies.
  - Use inheritance to compose protocols or code
- **Behavioral patterns:**
  - Deal with dynamic interactions among societies of classes and objects
  - Distribute responsibility

Creational Patterns

- **Abstract Factory**
  - Factory for building related objects
- **Builder**
  - Factory for building complex objects incrementally
- **Factory Method**
  - Method in a derived class creates associates
- **Prototype**
  - Factory for instantiating new objects by cloning them from a prototype
- **Singleton**
  - Factory for singular (sole) instance in the system

Structural Patterns

- **Adapter**
  - Translator of „server“ interface to our „client“ code
- **Bridge**
  - Abstraction for binding one of many implementations
- **Composite**
  - Structure for building recursive aggregations
- **Decorator**
  - Extends an object transparently
- **Facade**
  - Simplifies and aggregates the interface for a complex subsystem
- **Flayweight**
  - Many fine-grained objects shared efficiently
- **Proxy**
  - One object approximates another, e.g. due to efficiency or memory requirements

Behavioral Patterns

- **Chain of Responsibility**
  - Request delegated to the responsible service provider
- **Command**
  - Request is first-class object
- **Iterator**
  - Aggregate elements are accessed sequentially
- **Interpreter**
  - Language interpreter for a small grammar
- **Mediator**
  - Coordinates interactions between its associates
- **Memento**
  - Stores and recovers object state

Behavioral Patterns (cont.)

- **Observer**
  - Dependents update automatically when subject changes
- **State**
  - Object whose behavior depends on its state
- **Strategy**
  - Abstraction for selecting one of many algorithms
- **Template Method**
  - Algorithms with some steps supplied by a derived class
- **Visitor**
  - Operations applied to elements of a heterogeneous object structure

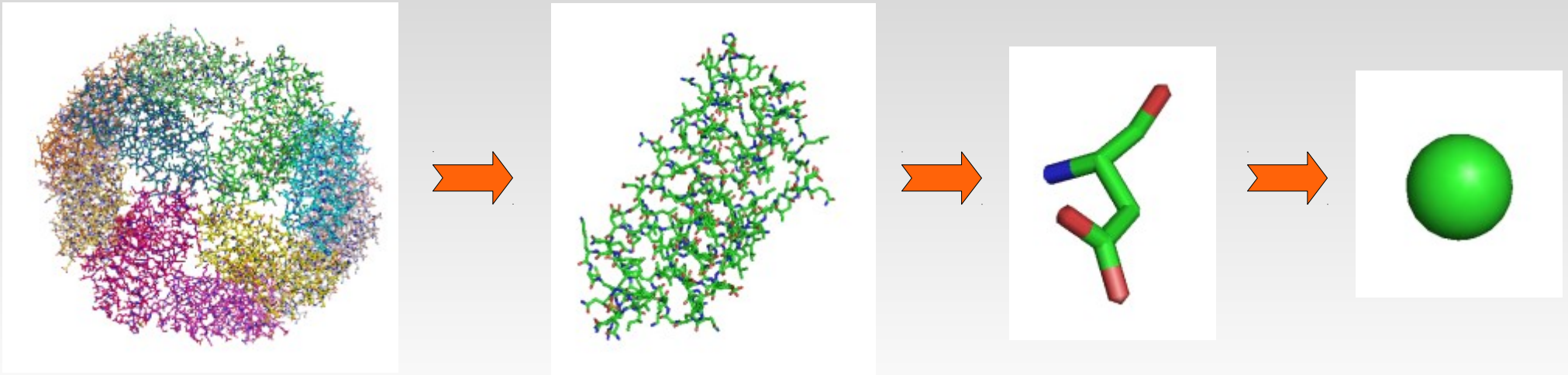
Organizing the Catalog

- Scope is the domain over which a pattern applies
  - Class scope: Relations between base classes and their subclasses (static semantics).
  - Object scope: Relationships between peer objects. Reuse of collection of objects is better achieved through variations of their composition, rather through sub-classing.
- Some patterns apply to both scopes

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Case Study: Chemical Structures

Initial Model

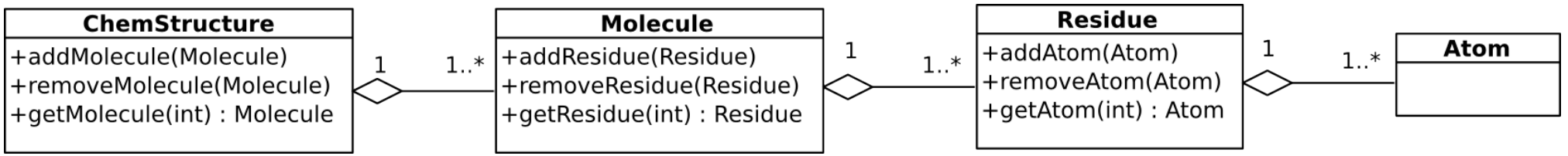


Chemical Structure

Molecule

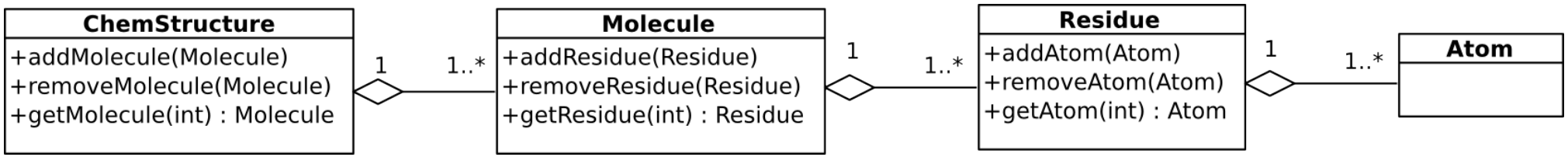
Residue

Atom

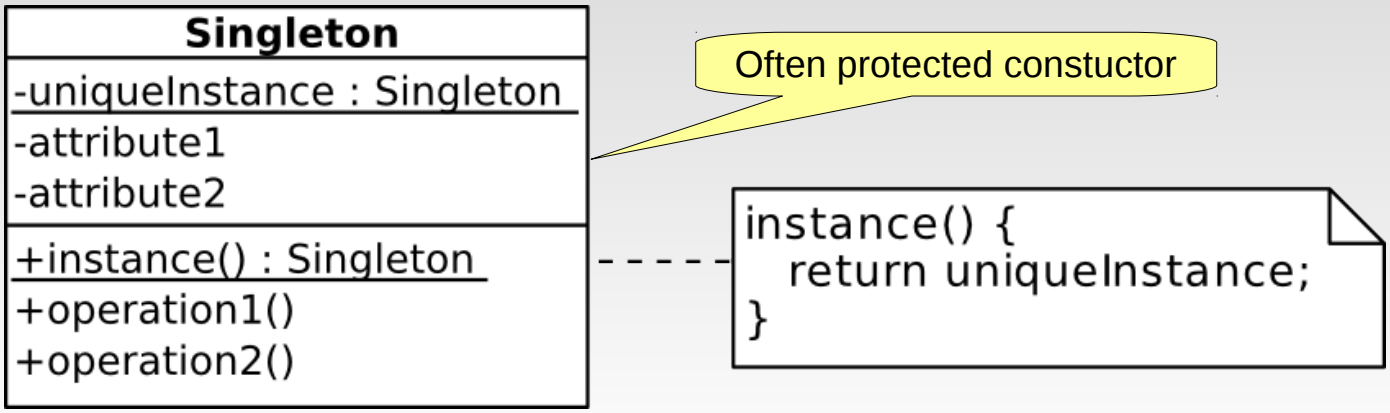


Initial Model (cont.)

- Every object in running OO system must be referenced
  - Every class must be associated with another class, OR
  - There exist only a few instances (often just one instance) in the system and these instances have well-known access points (well-known addresses).
- Problem of our model:
  - Atoms are references from their residuum, residua from their molecule, etc. But what about (multiple) chemical structures? Who gives me pointer to concrete chemical structure?



Singleton Pattern




```
public class Singleton {

    private static Singleton uniqueInstance;

    public static Singleton instance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    public void operation1() { … }

    public void operation2() { … }
};
```

Singleton invocation and use:

```
// First call. New instance is created in memory
Singleton.instance().operatio1();
…
// Next call. Invocation of the same or another method
// is performed on existing sole instance
Singleton.instance().operatio2();
```

Singleton Properties

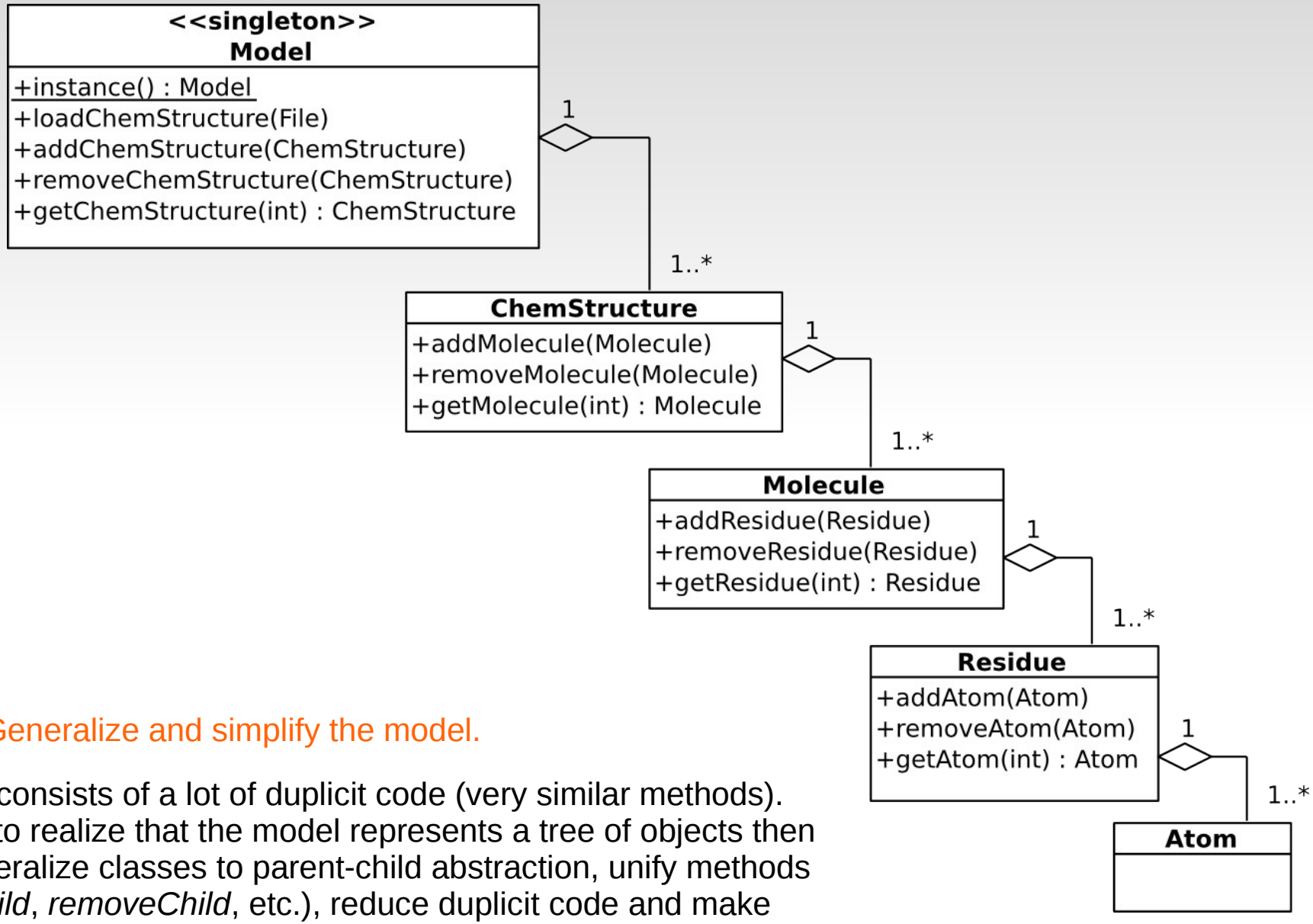
Applicability: Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

Consequences:

- Controlled access to sole instance.
- Reduced name space
  - Avoids polluting the name space with global variables that store sole instances.
- Permits refinement of operations and representation
  - The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class.
  - You can configure the application with an instance of the class you need at run-time.
- Permits a variable number of instances.
- More flexible than class operations
  - Another way is to use class operations (i.e. static methods). But it usually makes it hard to change a design to allow more than one instance of a class. Moreover, static functions are never virtual, so subclasses can't override them polymorphically.

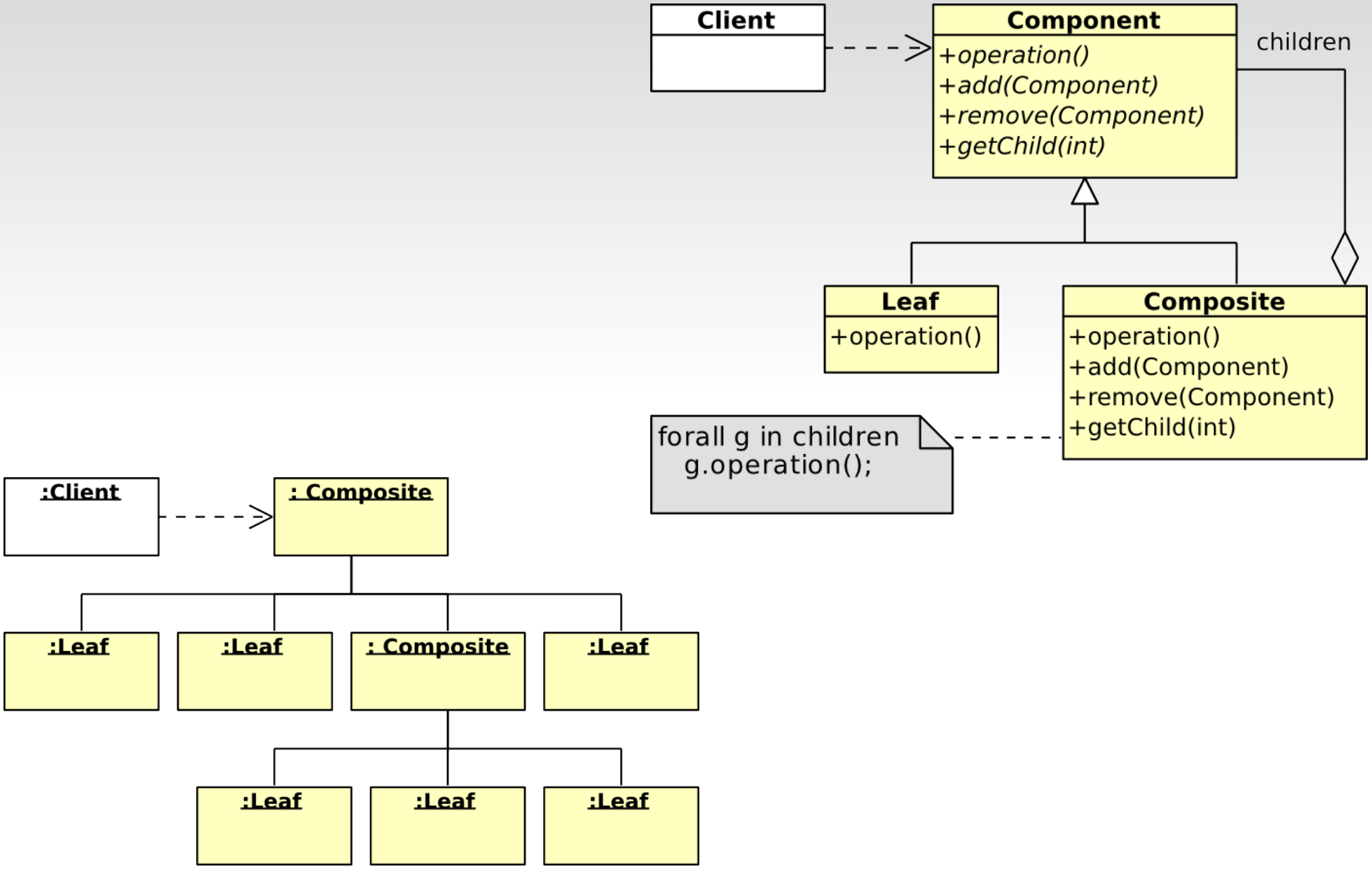
Singleton – Case Study



Next step: Generalize and simplify the model.

This model consists of a lot of duplicit code (very similar methods). If we come to realize that the model represents a tree of objects then we can generalize classes to parent-child abstraction, unify methods (e.g. *addChild*, *removeChild*, etc.), reduce duplicit code and make the model easily extensible and manageable.

Composite Pattern



Composite Properties

Applicability: Use the Composite pattern when

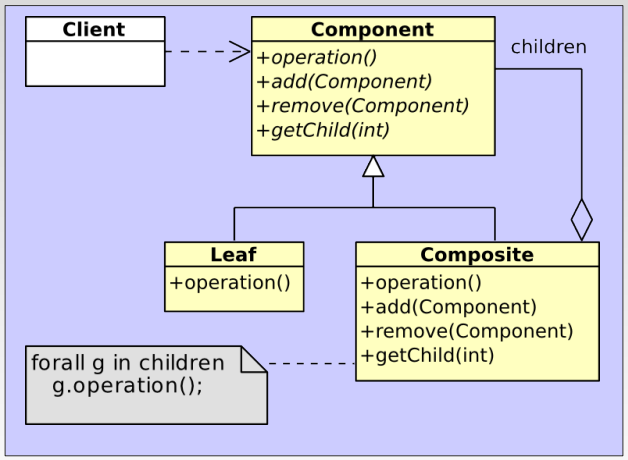
- you want to represent *part-whole* hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Consequences:

- Defines class hierarchies consisting of primitive objects and composite objects.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly.
- Makes it easier to add new kinds of components.
- Can make your design overly general.
    - The disadvantage is that it makes it harder to restrict the components of a composite.

Composite – Case Study

Q: How to hide impelemenation of children list (now it's indexed array) and how to support various iterations through children and/or various impelementations of children list in subclasses?



<<composite-component>>
ModelElement
+*add(ModelElement)*
+*remove(ModelElement)*
+*get(int) : ModelElement*
+*setColor(Color)*

subElements
1..*

superElement

<<composite-leaf>>
Atom
-color : Color
+setColor(Color)

<<composite-composite>>
GroupElement
+add(ModelElement)
+remove(ModelElement)
+get(int) : ModelElement
+setColor(Color)

setColor() is applied to all children

<<singleton>>
<<composite-composite>>
Model
+instance() : Model
+loadChemStructure(File)

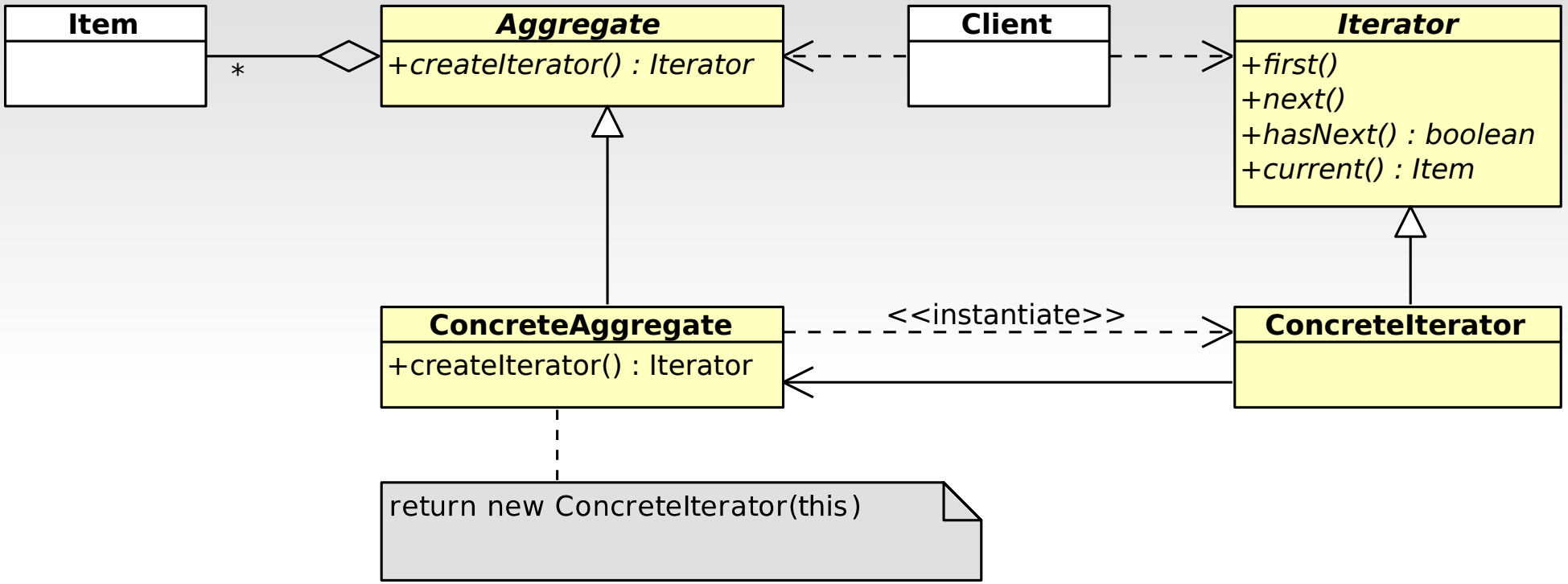
<<composite-composite>>
Molecule

<<composite-composite>>
ChemStructure

<<composite-composite>>
Residue

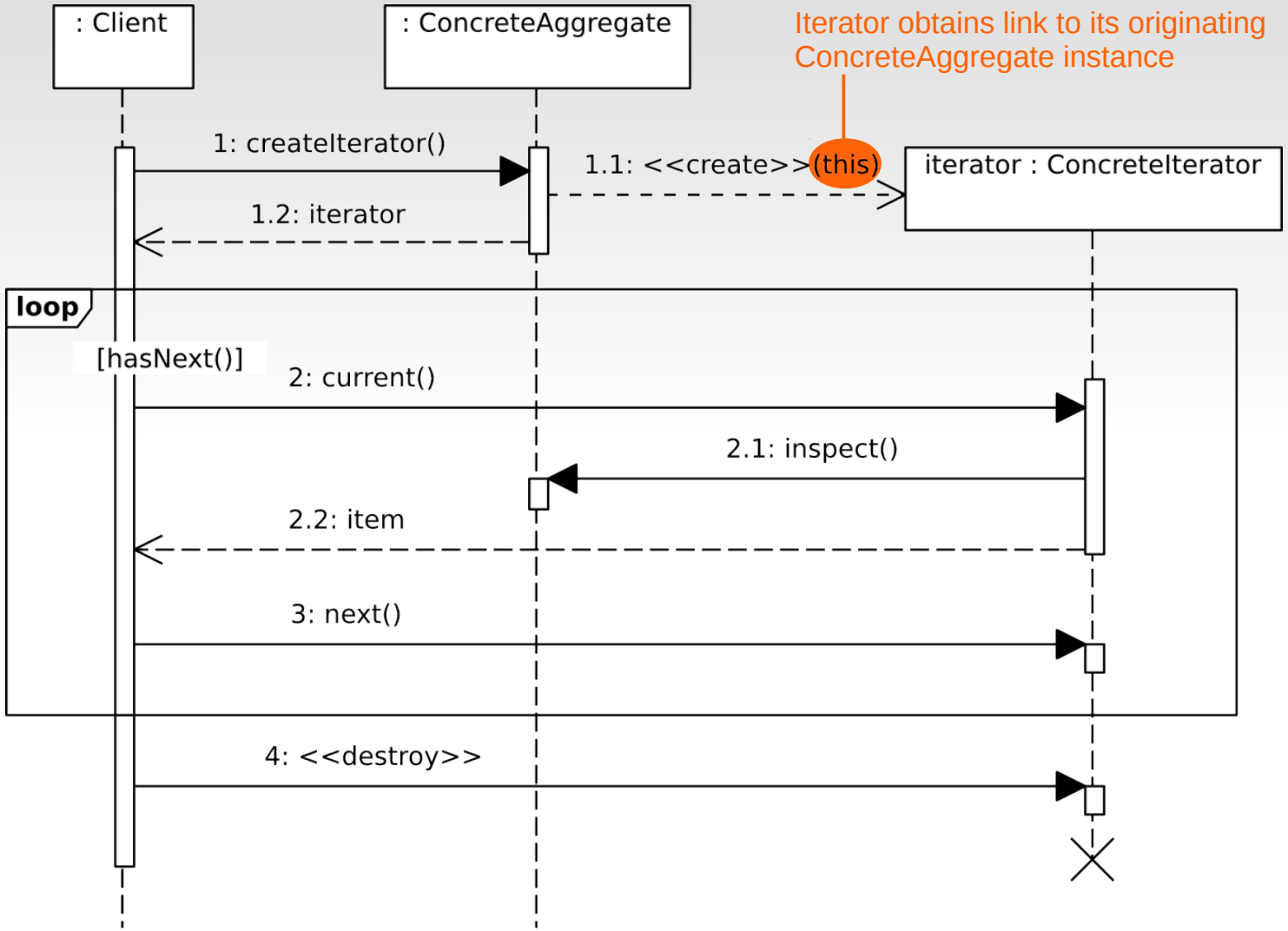
Patterns are often combined

Iterator Pattern



See *java.util.Collection* and *java.util.Iterator*, for instance.
Thanks to the Iterator Pattern, Java can support *"foreach"* cycle.

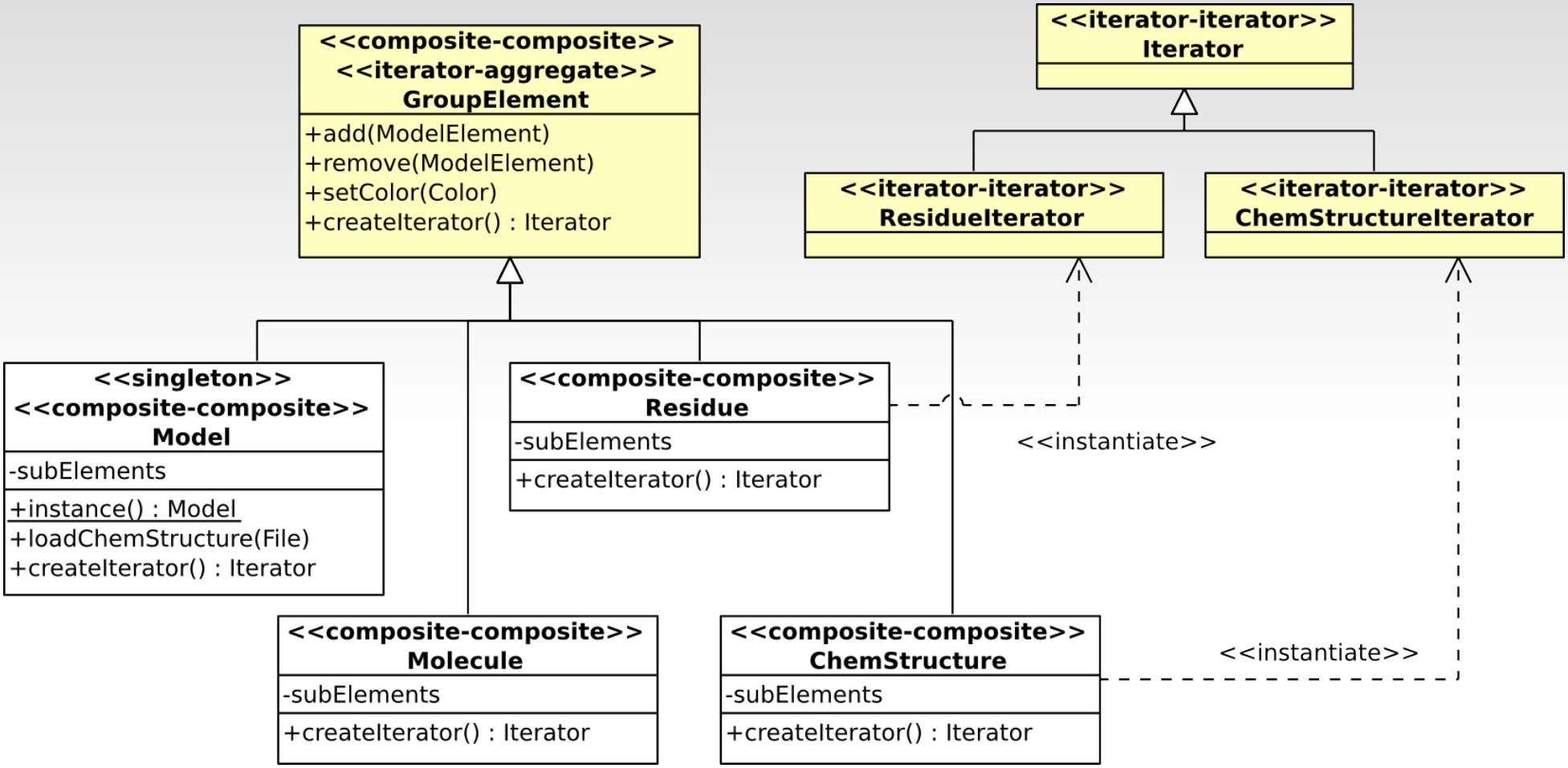
Iterator Pattern (cont.)



Iterator Properties

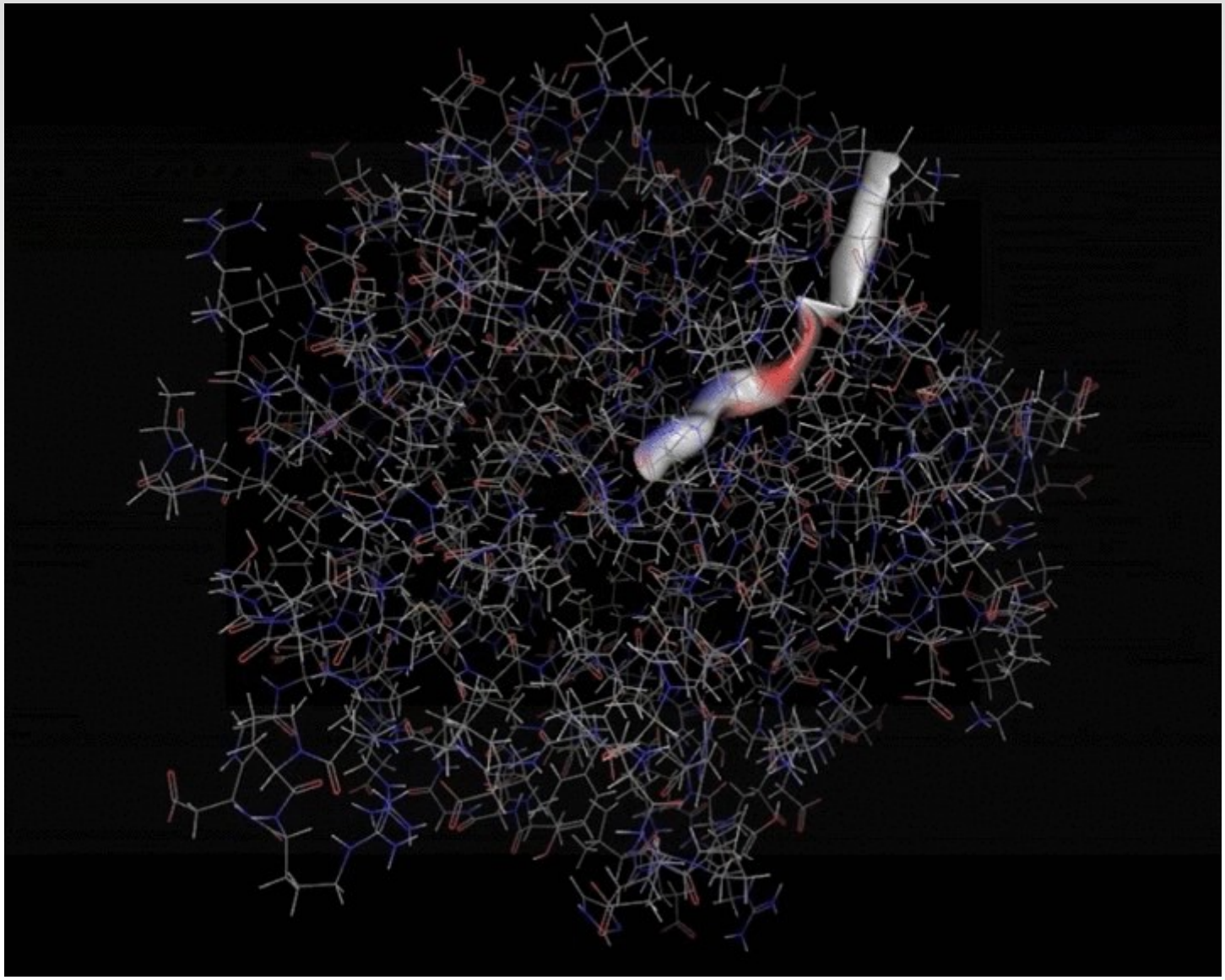
- **Applicability:** Use the Iterator pattern
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).
- **Consequences:**
  - It supports variations in the traversal of an aggregate.
  - Iterators simplify the Aggregate interface.
  - More than one traversal can be pending on an aggregate.

Iterator – Case Study

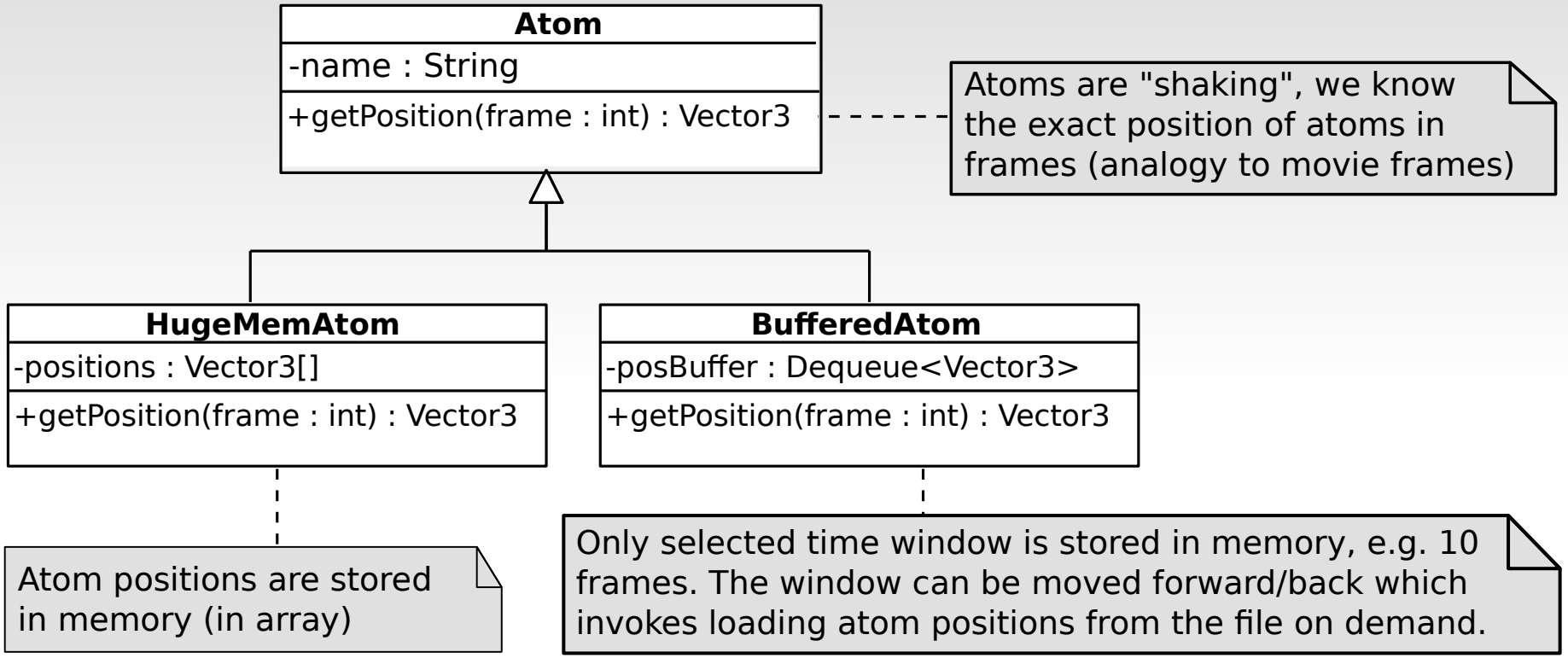


Note 1: The *subElements* attribute has been moved from *GroupElement* to individual subclasses.
Note 2: It is not necessary to have a special iterator for each composite class. On the contrary, one iterator can be shared by multiple composists.

Case Study Extension: „Shaking“ Molecule

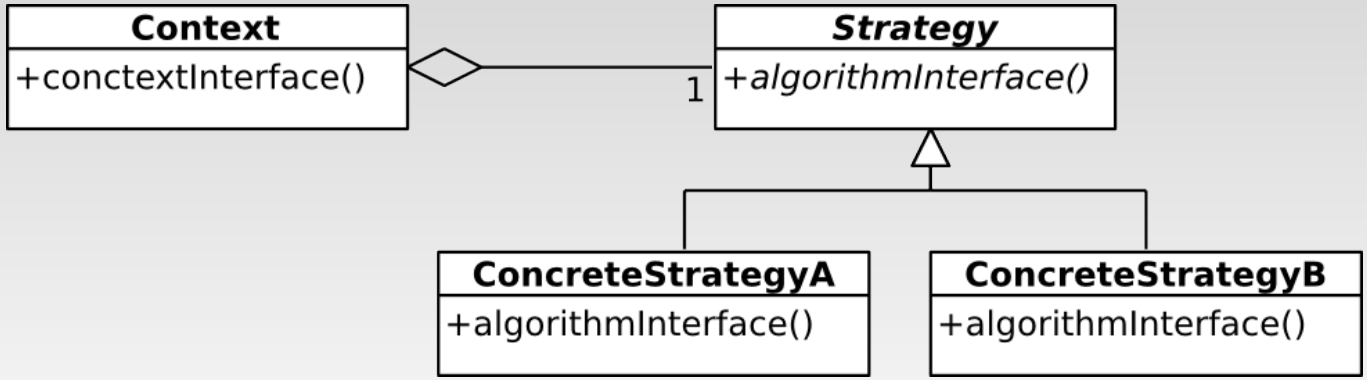


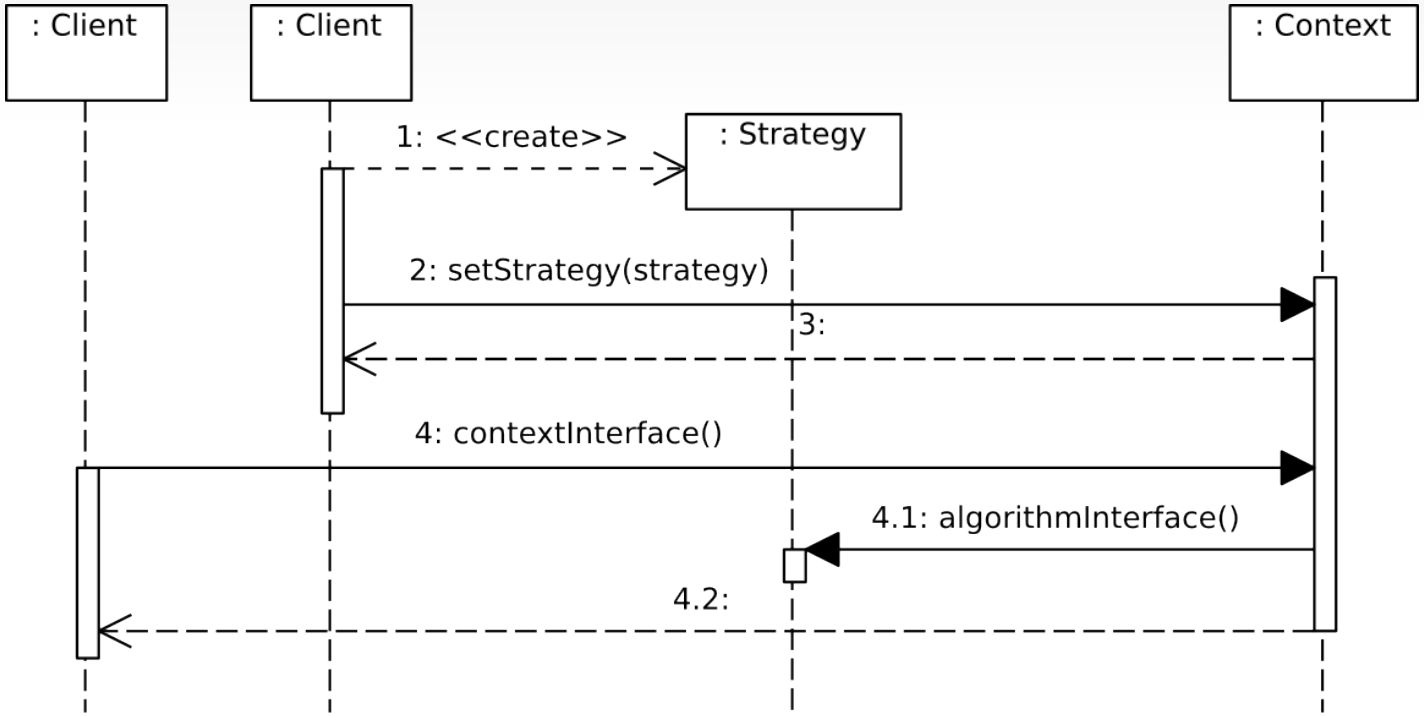
Case Study Extension: „Shaking“ Molecule



There can be many subclasses with common interface but different behavior.
Subclasses can be therefore understood as different **strategies** of the computation of atom position.

Strategy Pattern



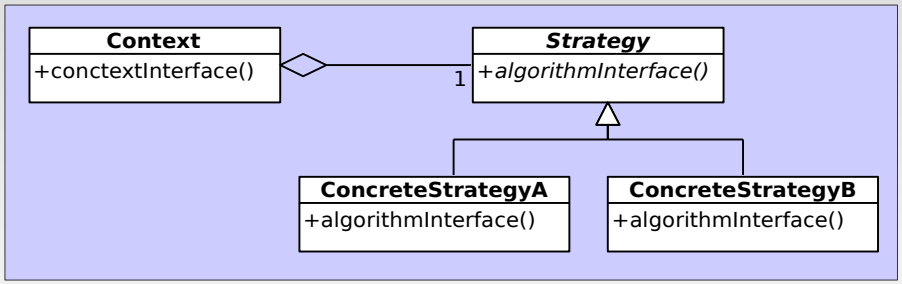


Example: Context = window of text editor, Strategy = line-breaking algorithm

Strategy Properties

- **Applicability:** Use the Strategy pattern when
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm.
  - An algorithm uses data that clients shouldn't know about.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
- **Consequences:**
  - Hierarchies of Strategy classes define families of related algorithms (useful for reuse).
  - An alternative to subclassing.
  - Strategies eliminate conditional statements (switch-case).
  - Different implementations of the same behavior.
  - Client must understand how Strategies differ before it can select the appropriate one.
  - Communication overhead between Strategy and Context.
  - Increased number of objects.

Strategy – Case Study



Every atom can have its own strategy

<<strategy-context>>
Atom
-name : String
+getPosition(frame : int) : Vector3

1

<<strategy-strategy>>
AtomPositionStrategy
+getPosition(frame : int) : Vector3

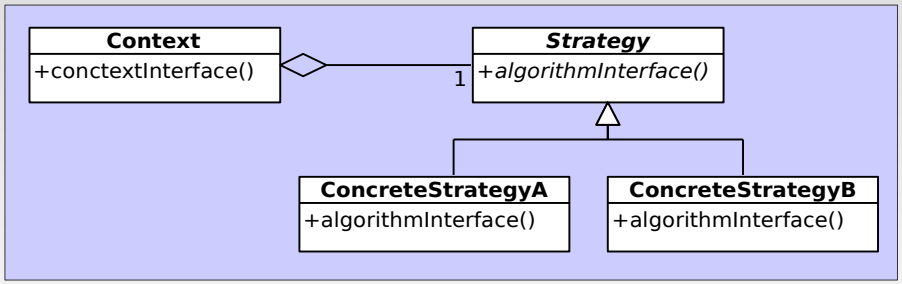
<<strategy-strategy>>
AllInMemStrategy
-positions : Vector3[]
+getPosition(frame : int) : Vector3

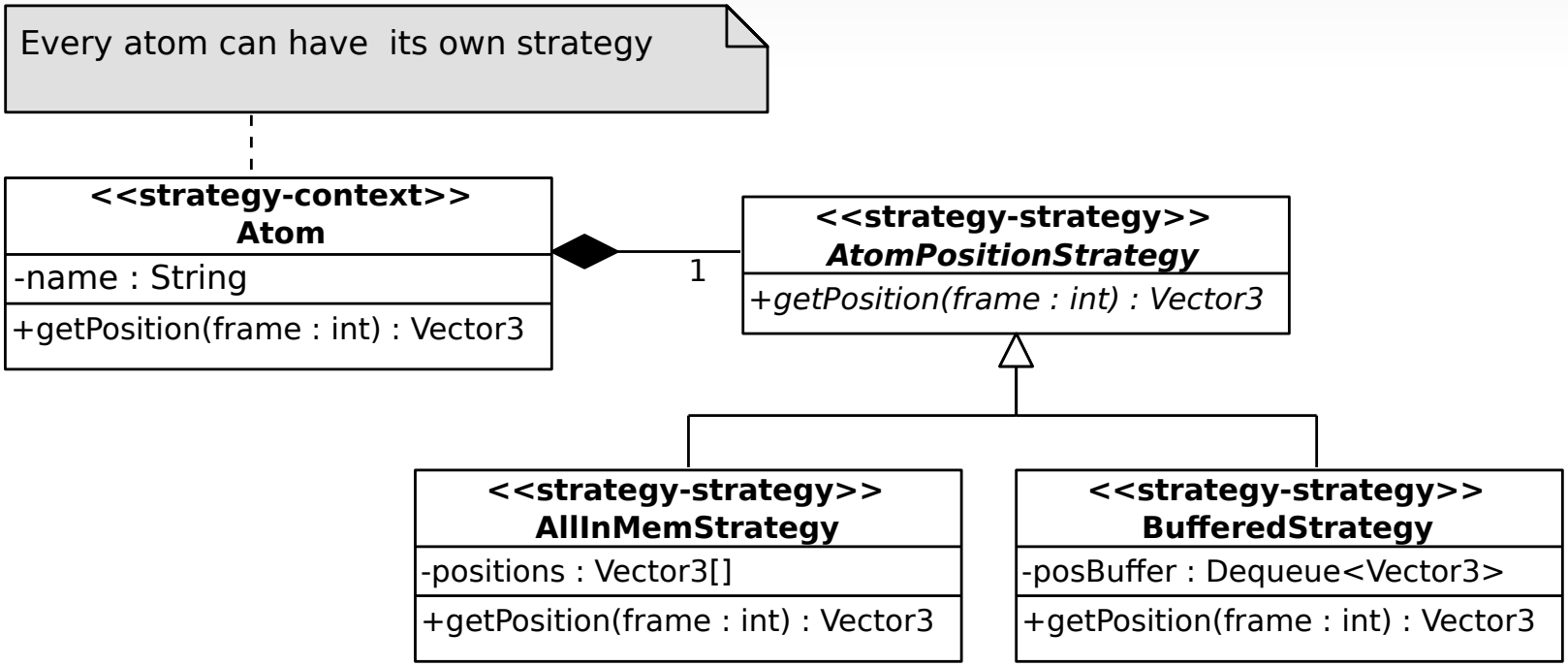
<<strategy-strategy>>
BufferedStrategy
-posBuffer : Dequeue<Vector3>
+getPosition(frame : int) : Vector3

Strategy in Java

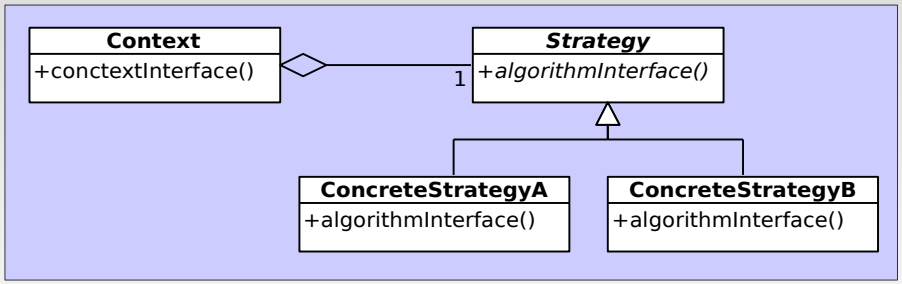
- **Question:** Which well-known part of Java Core PI uses Strategy?
- **Hint:** Focus on sorted collections.
- **Answer:** Comparator defines strategy for sorting objects in SortedSet, for instance.

Strategy – Case Study





Strategy – Case Study



Every atom can have its own strategy

<<strategy-context>>
Atom
-name : String
+getPosition(frame : int) : Vector3

1

<<strategy-strategy>>
AtomPositionStrategy
+getPosition(frame : int) : Vector3

<<strategy-strategy>>
AllInMemStrategy
-positions : Vector3[]
+getPosition(frame : int) : Vector3

<<strategy-strategy>>
BufferedStrategy
-posBuffer : Dequeue<Vector3>
+getPosition(frame : int) : Vector3

Questions?