Software Engineering

Lesson Design Pattern 04
Composite, Iterator
v1.0a

Uwe Gühl

Fall 2007/ 2008

Contents

- Composite
- Iterator

Composite

- Intent:
  - Compose objects into tree structures to represent part-whole hierarchies
  - A client could treat individual objects and compositions of individual objects in the same way
  - ... is a Structural Pattern

Composite

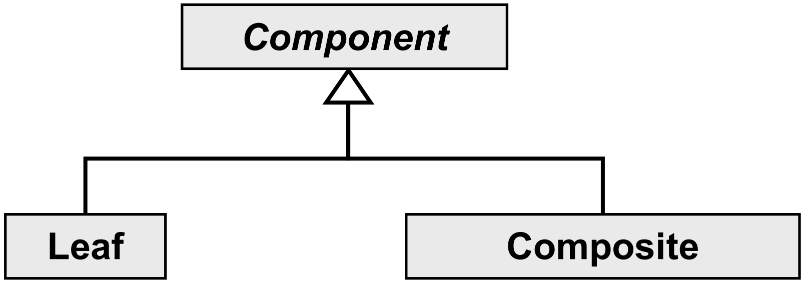
- Motivation
  - Graphical applications offer often the possibility to create complex widgets, larger components or diagrams out of simple components
  - Discussion of a simple approach
    - Primitive classes for basic graphical objects
    - Container classes to collect this primitive graphical objects
    - Difficulty: These classes have to be treated by clients always in a different way – an application has to differ between primitive and container objects, the code complexity is raising

Composite

- Ideas
  - If the objects and the composition of objects should be treated in the same way, they need something like a common interface
  - So clients could access them transparently
  - If a dynamical adding of objects and container objects should be possible, a recursive use has to be established

Composite

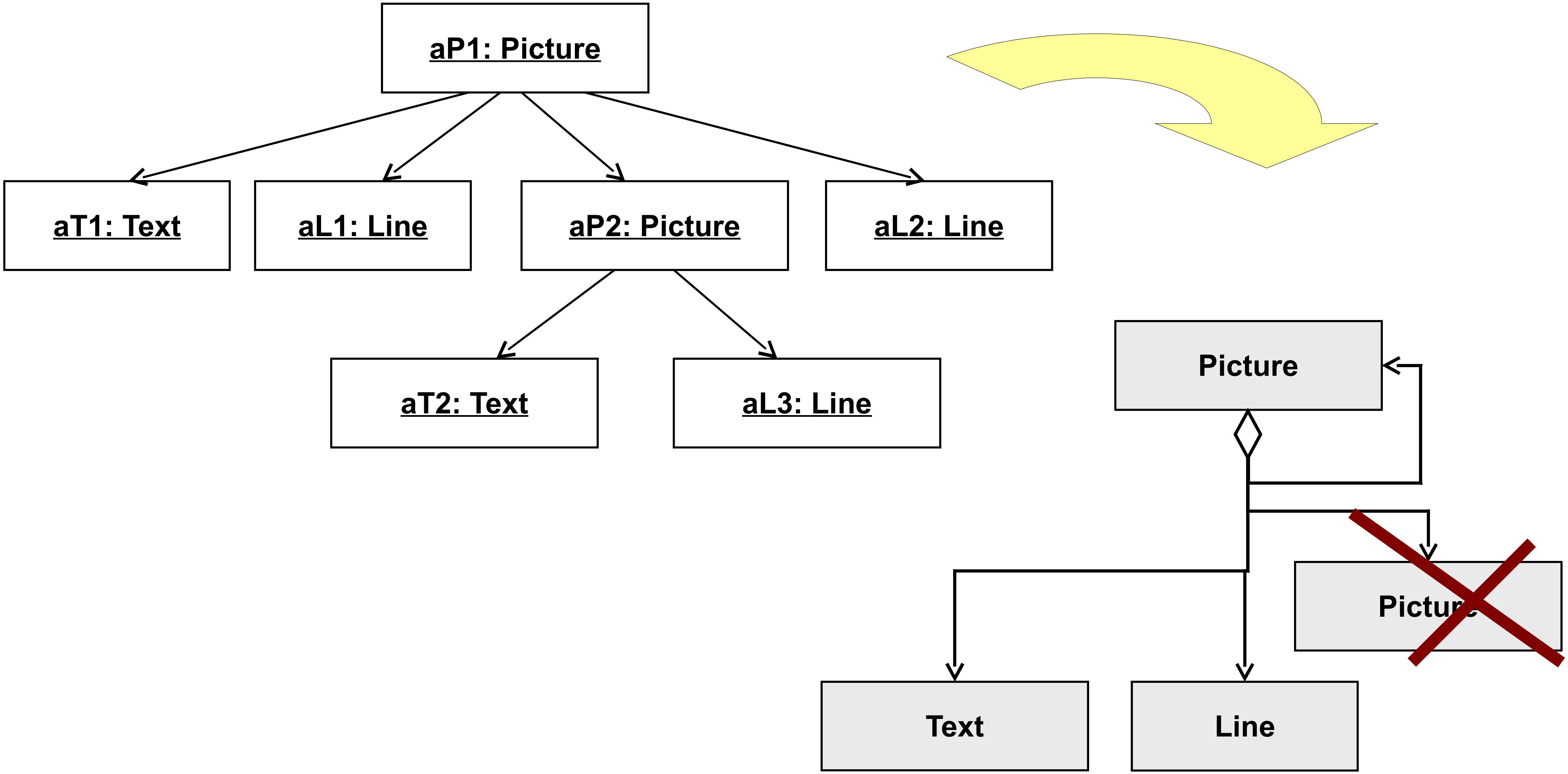
- Solution
  - Introducing of an abstract class representing both
    - primitive objects and
    - containers of primitive objects
  - Let's call this abstract class ***Component***.
    - The class of primitives is something like a **Leaf** and
    - The container class is our **Composite**.



  - ... and how to establish the recursive idea?

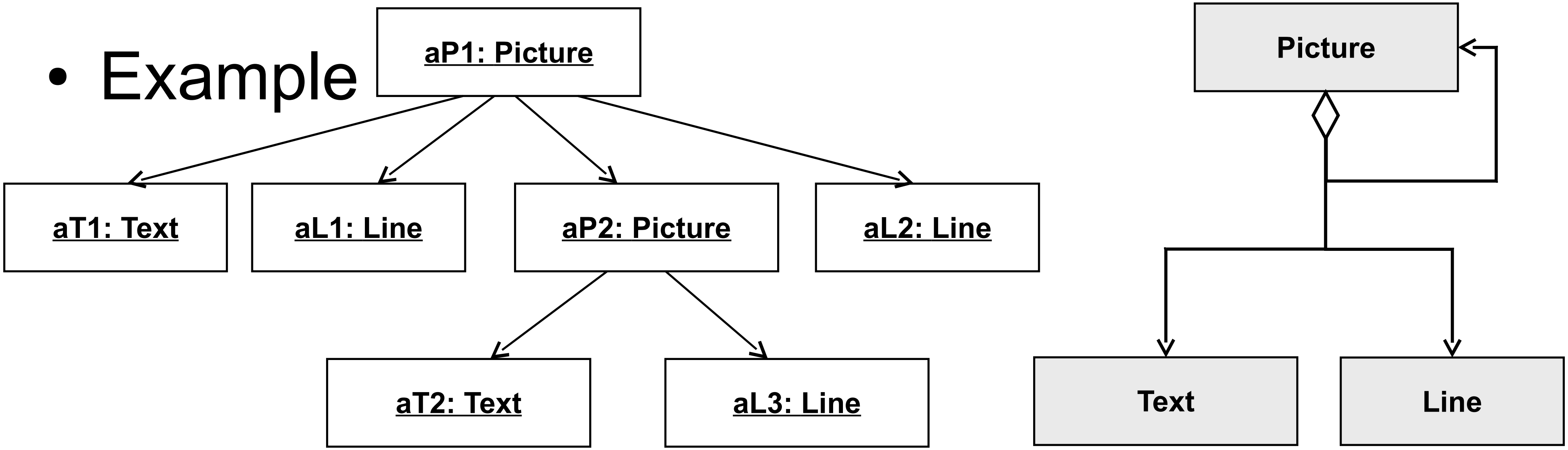
Composite

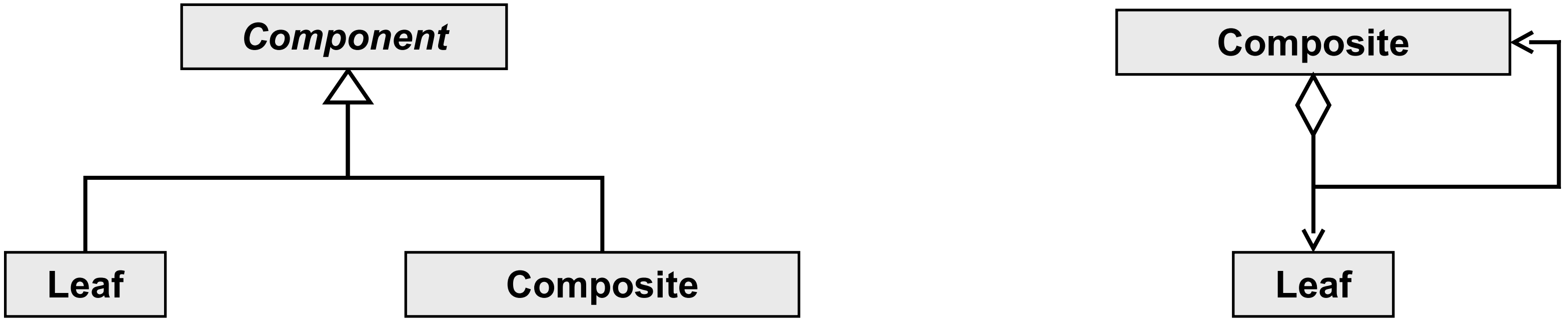
- Example



Composite

- Example

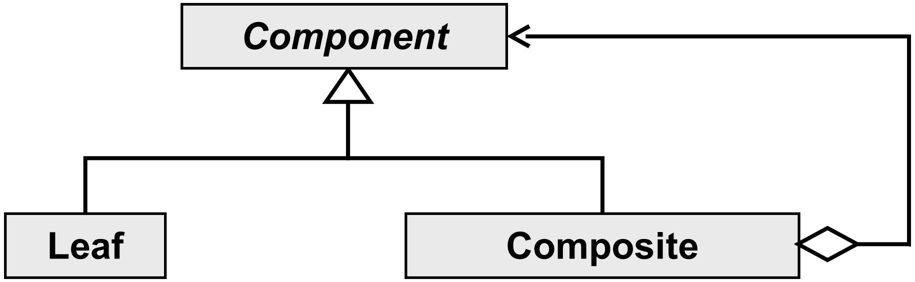




  - .. and how to put ideas together?

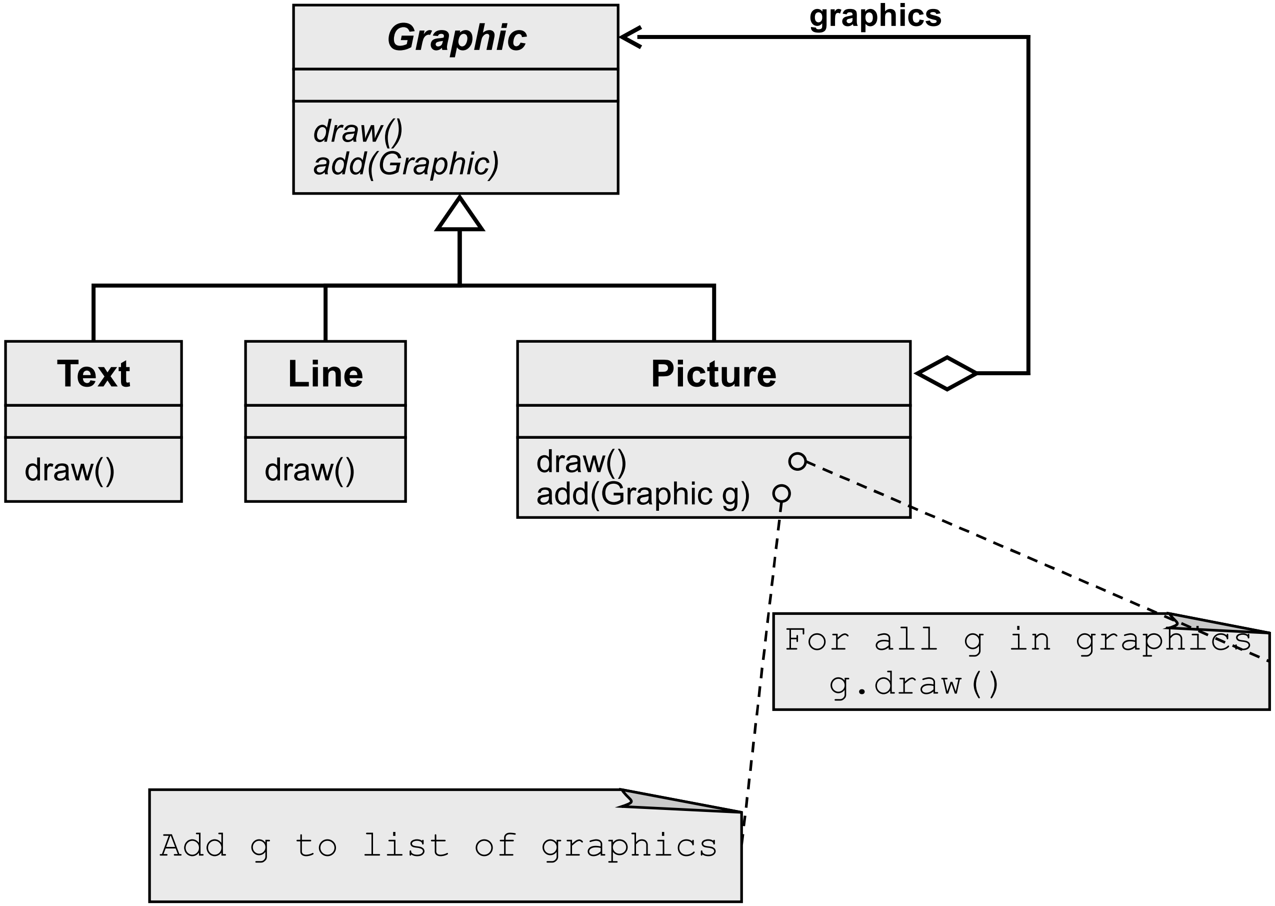
Composite

- Solution
  - The abstract class ***Component*** is needed for the abstract interface
  - The relation between ***Component*** and **Composite** makes recursion possible
    - The container class **Composite** could always contain either another **Composite** container or a **Leaf**
    - After a **Leaf** no further recursion is possible



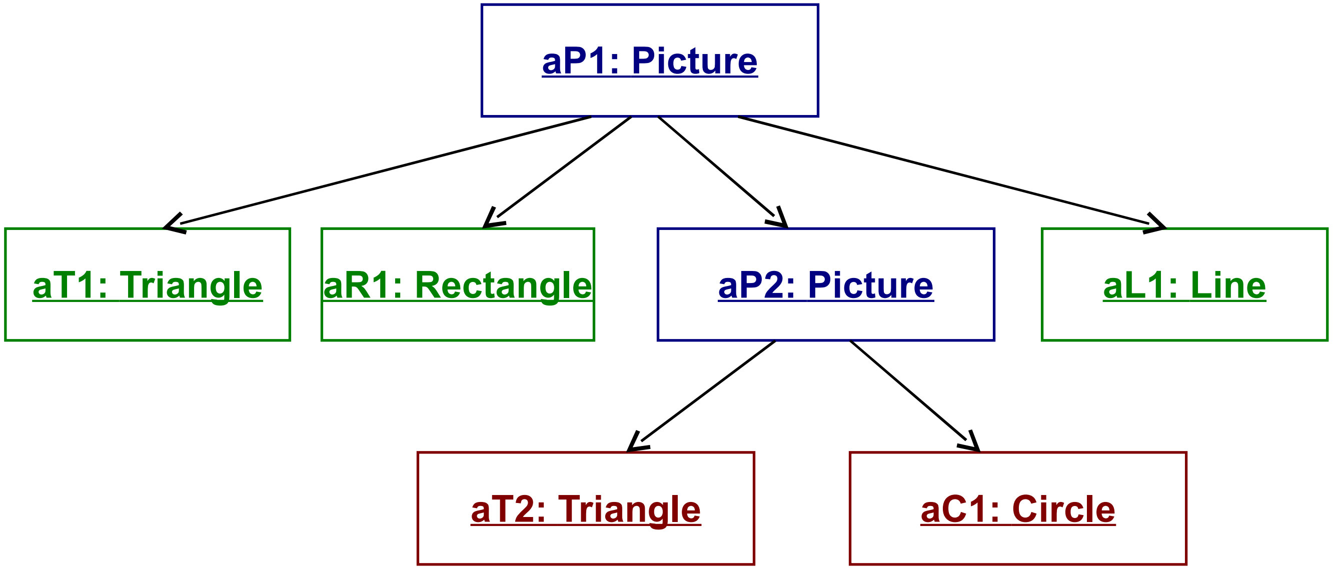
Composite

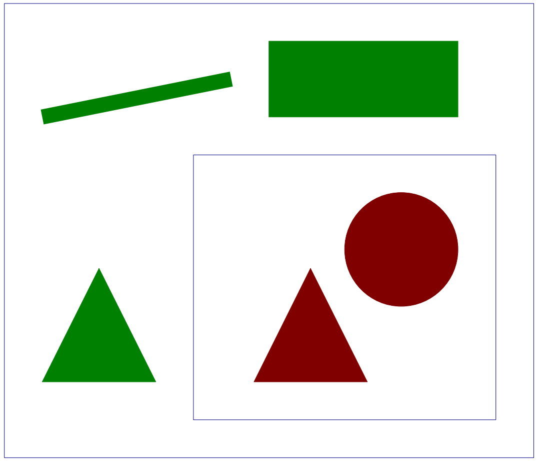
- Example



Composite

- Example
  - Typical composite object structure out of recursive combined graphical objects





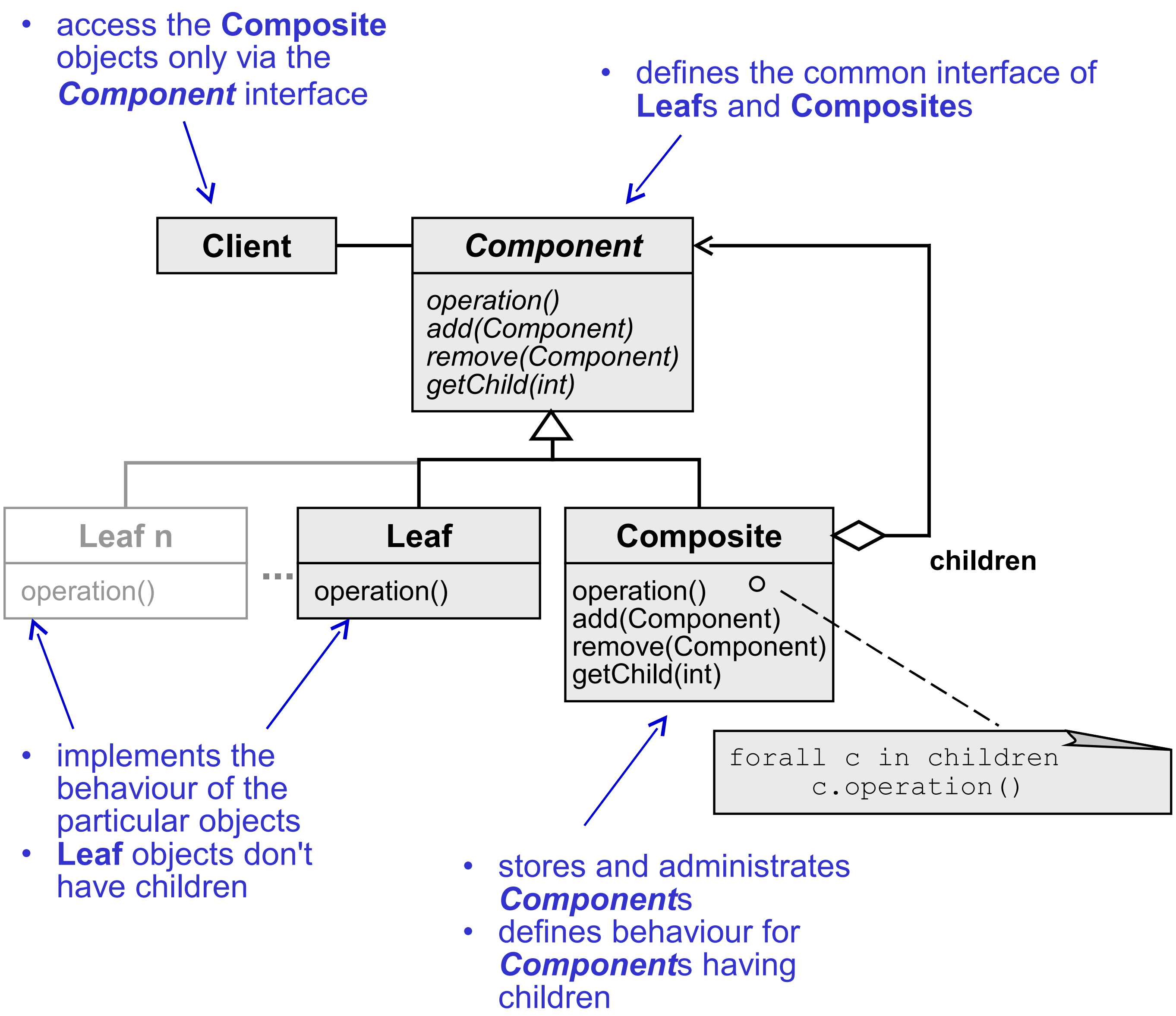
  - With this structural pattern groups of graphical figures could be created

Composite

- Consideration
  - To offer the same accessibility the Composite and the component need the same interface
  - So clients have transparent access
  - The Composite sends requests to the component and executes additional activities (e. g. drawing of a border)
  - Recursive use of multiple Composites allows dynamical adding of functionality

Composite

- Structure



```
forall c in children
    c.operation()
```

- access the **Composite** objects only via the ***Component*** interface

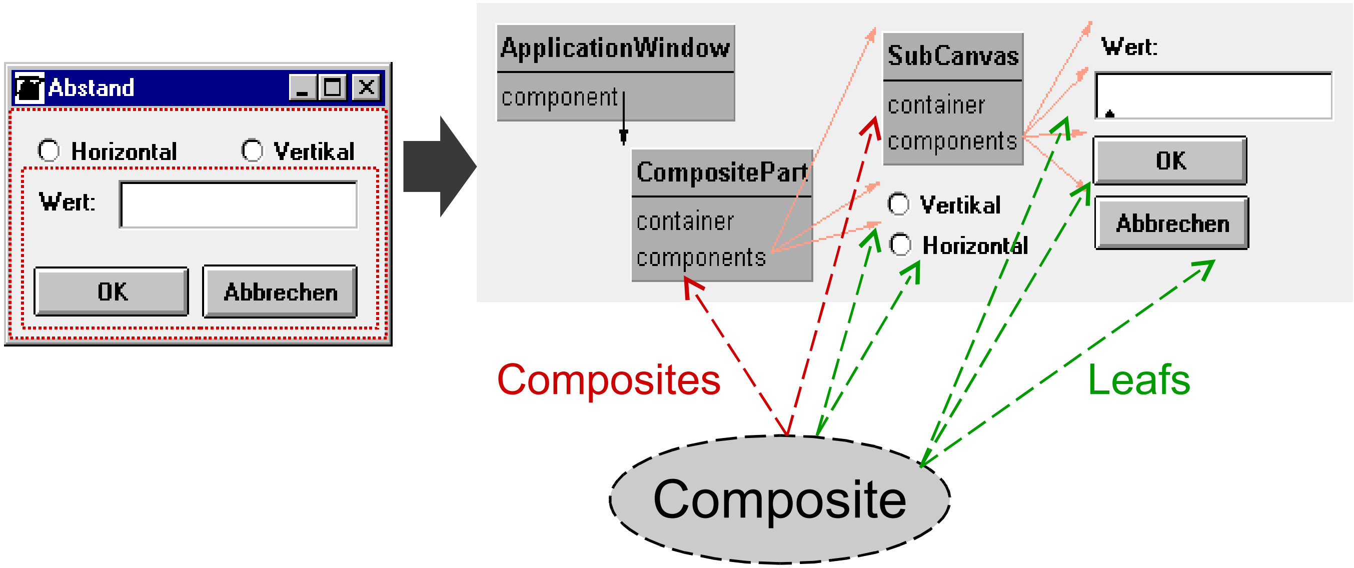
- defines the common interface of **Leaf**s and **Composite**s

- implements the behaviour of the particular objects
- **Leaf** objects don't have children

- stores and administrates ***Component***s
- defines behaviour for ***Component***s having children

Composite

- Example



Composite

- Collaboration
  - Clients use the ***Component*** class interface to interact with all the objects and object containers
    - If there is an interaction with a **Leaf**, the request is executed directly
    - If there is an interaction with a **Composite**, the **Composite**
      - forwards the request to its children
      - performs additional operations before or after forwarding – if defined

Composite

- Applicability
  Use the Composite Pattern
  - to represent part-whole hierarchies of objects
  - if clients should be able to handle
    - individual objects and
    - compositions of individual objects

    in the same way

Composite

- Consequences

+ defines class hierarchies with objects and composites

+ Simplifies the client – individual and composed objects could be treated similar

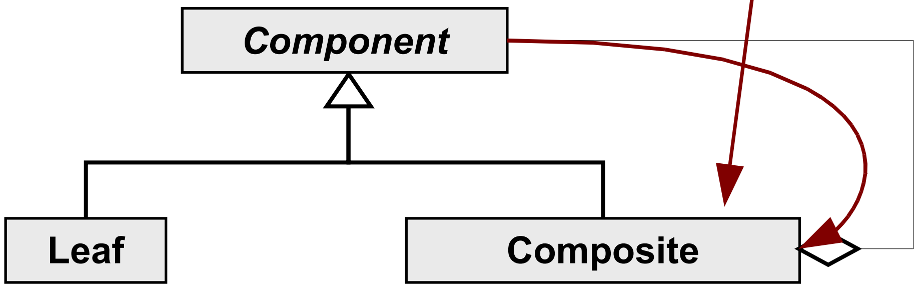
+ Makes it easy to add new components and objects as the client code has not to be changed

– The overly general design makes it harder to restrict the components of a composite, for example if a specific composite should have only defined components

  - Run time checks could be necessary

Composite

- Implementation
  - Explicit references to parent objects to simplify navigation
    - moving in the structure
    - deleting a component



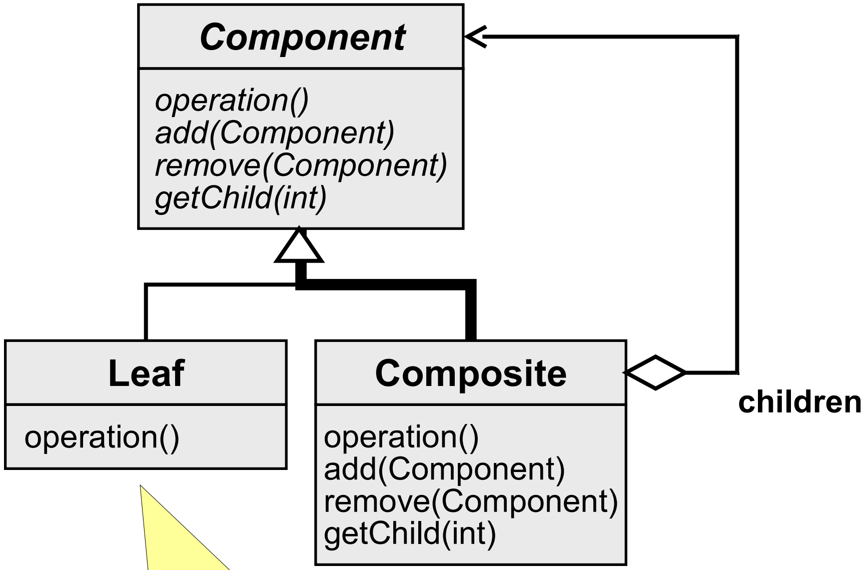
Composite

- Implementation
  - Maximizing the Component interface
    - Find the maximum number of operations which could be shared by Leaf and Composite
    - Component offers default implementations,
      Leaf and Composite subclasses overwrite
      ➔ conflict, if operations are supported, which don't make sense for sub classes, e. g. accessing children makes no sense for Leafs

Composite

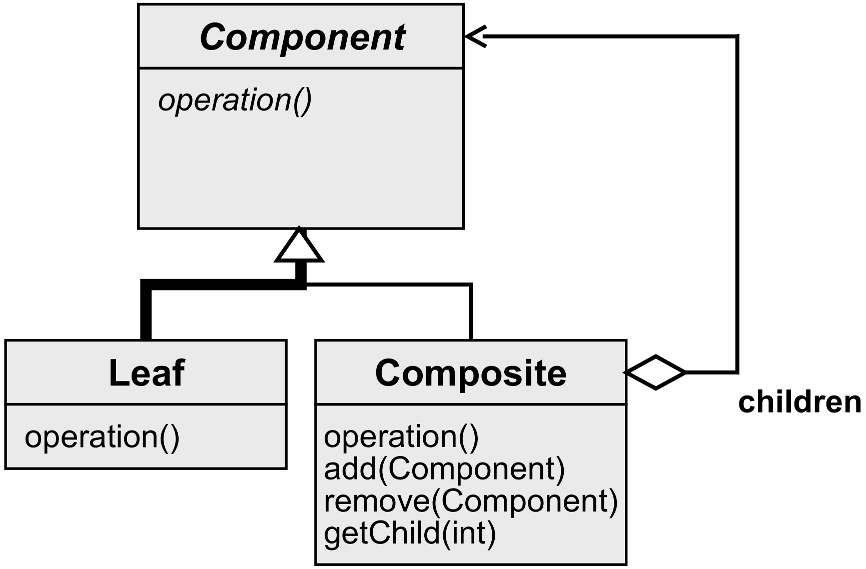
- Implementation
  - Declaring child management operations

+ Transparency



Leaf must handle add(), remove() ... exceptions?

+ Safety



Composite

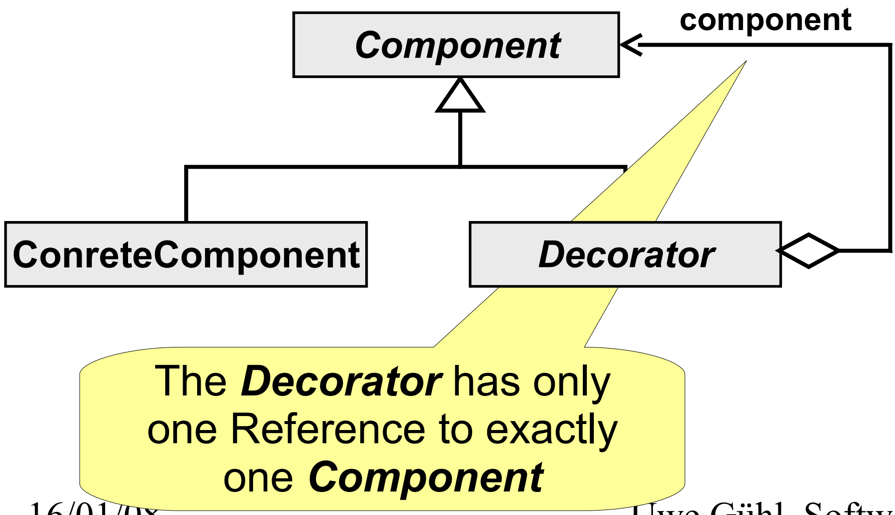
- Implementation
  - Caching to improve performance
    - Example: Picture class could cache the bounding box of its children
      --> If children are not visible drawing or search for children of children could be avoided
    - Components must know their parents to realize this idea
  - Clarification who should delete components
    - Idea: Composites are responsible for deleting children, if they get deleted

Composite

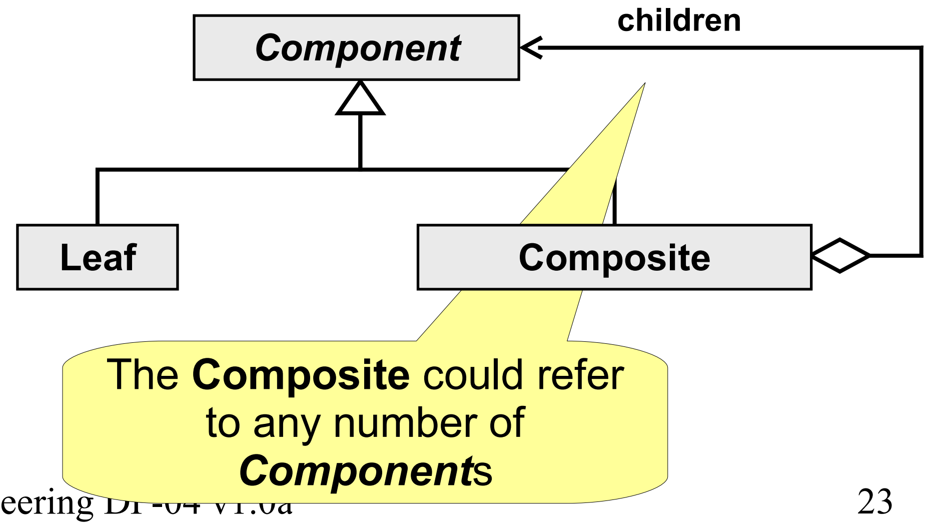
- Known Uses (see [GHJ+95])
  - Graphical frameworks like VisualWorks Smalltalk and HotDraw [Joh92]
  - Java Swing Classes and Java AWT package (Component, Container, Label, TextField, Panel, Frame, Dialog, ...)
  - Apache Jakarta Commons library, e. g. the class CLICommand for combined commands (Macros)
  - Credit system [CV02]

Composite

- Related Patterns
  - Decorator
    - Decorator and Composite could work together, then they have usually a common parent class.
    - Decorators support the Component interface with operations like Add, Delete, and GetChild
    - Main difference between Decorator and Composite:



The ***Decorator*** has only one Reference to exactly one ***Component***



The **Composite** could refer to any number of ***Component***s

Composite

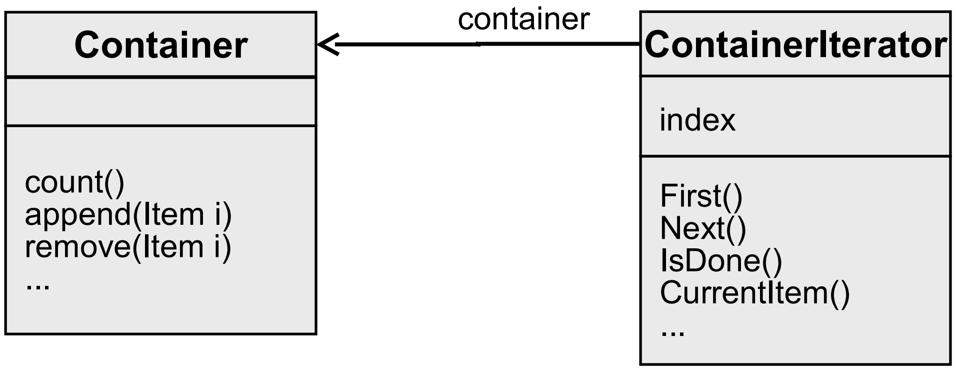
- Related Patterns
  - Chain of Responsibility
    The component-parent link is used for a Chain of Responsibility
  - Flyweight could be used to share components not referring to parents any more
  - Visitor localizes operations and behaviour instead of distribution across Composites and Leafs
  - Iterator could be used to traverse composites

Iterator

- Intent:
  - Provide a way to access the elements of an aggregate object (e. g. a collection) sequentially without exposing its underlying representation
  - ... also known as “Cursor”
  - ... is a Behavioral Pattern

Iterator

- Motivation
  - To access or to operate on elements of a complex data structure like a collection, a tree, or a hash table, one would not like to take care about internal implementation details
  - The Iterator should do all this stuff
    - An iterator object is responsible to access and to traversal a specific container
    - The iterator offers a corresponding interface

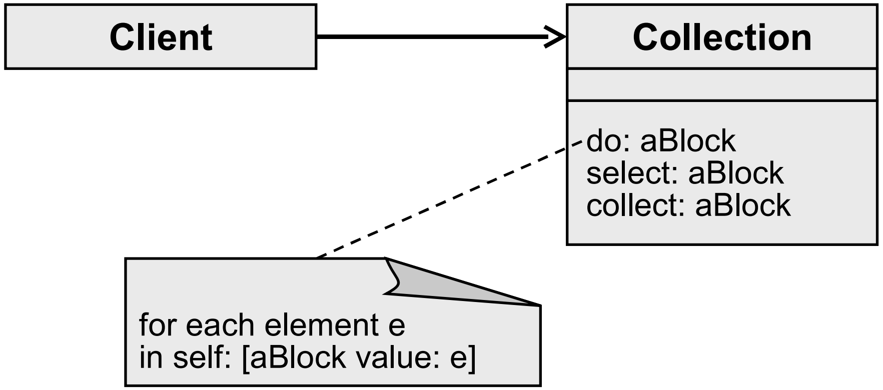


Iterator

- Solution
  - Encapsulation of the code to traverse an object structure; two possibilities
    - Internal Iterator: The data structure itself implements the needed functionality
    - External Iterator: The code to traverse the object structure gets released in an own object
      - Advantage: Storage of the current position possible

Iterator

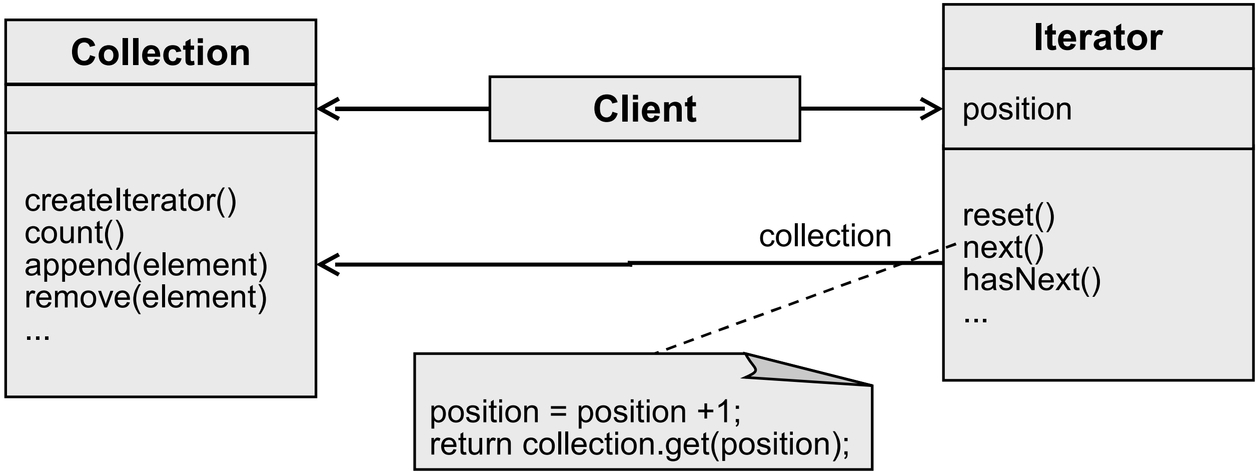
- Structure
  - Internal Iterator



Remark:
This Smalltalk example could not be implemented reasonable in C++ or Java

Iterator

- Structure
  - External Iterator

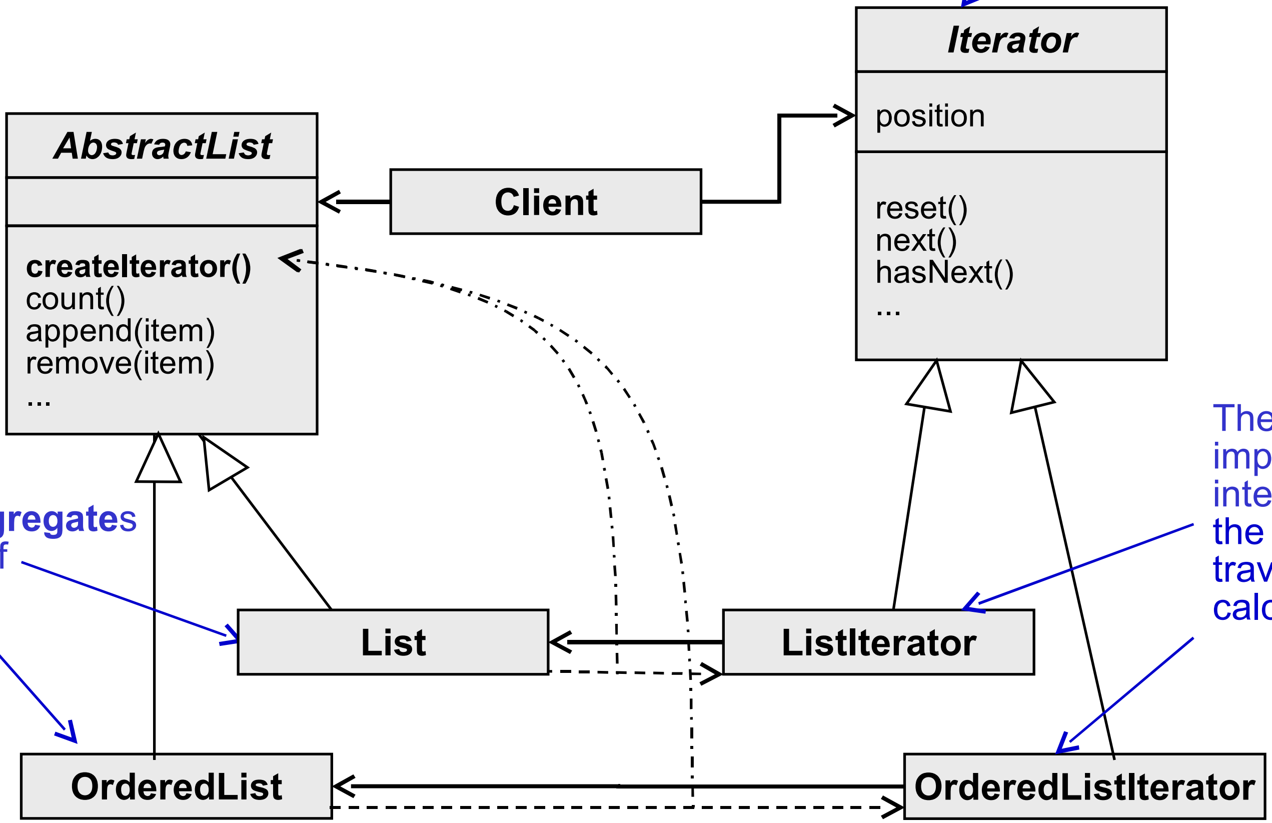


Iterator

- Structure
  - Polymorphic Iteration

The *Iterator* defines the interface to access and traverse elements

The *Aggregate* defines an interface, so that *Iterator* objects could be generated



The **ConcreteAggregate**s return instances of the proper **ConreteIterator**

The **ConreteIterator**s implement the interface, the next element in traversal could be calculated

Iterator

- Structure – Coding
  - Polymorphic Iteration
    - The code of the client to traverse an ***Aggregate*** (e. g. ***AbstractList***) is always the same, independent of the **ConcreteAggregate** in use

```
List myList = new OrderedList();
Iterator i = myList.createIterator();
while (i.hasNext()) {
   Object e = i.next();
}
```

A Factory Method returns an OrderedListIterator

Iterator

- Code example

Internal Iterator (Smalltalk):

```
parts do:[:part | part draw].
```

Collection

Iterator

Operation to be executed

Internal Iterator with filter function (Smalltalk):

```
newParts := parts select:[:part | part isNew].
```

External Iterator (Java):

```
Vector parts = ...;
Iterator i = parts.iterator();
while (i.hasNext()) {
   ((Part)i.next()).draw();
}
```

Iterator

- Consequences

+ Iterators support variations in the traversal of an aggregates
  - Different Iterators could support different traversal variants
+ All traversal algorithms are implemented in one location
+ Several iterations could traverse a collection at the same time, as the different traversal states could be tracked

Iterator

- Implementation
  - Iterator has to know implementation details of the corresponding collection owing the circumstances, especially in static typed languages
  - Who controls the iteration?
    Who implements the traversal algorithm?
    - Iterator controls the iteration → Internal Iterator
    - Client controls the iteration → External Iterator

Iterator

- Implementation
  - Who controls the iteration?
    - Internal Iterator:
      - can encapsulate different kind of iterations
      - is easier to use, as the client has not to care about how the iteration loop is specified
      - more work to implement
    - External Iterator:
      - more flexible in use, allows for example the comparison of two collections
      - better to use in programming languages without anonymous functions like C++

Iterator

- Known Uses
  - Most collection class libraries offer iterators
    - In Smalltalk e. g.:
      - Collection (internal)
      - Stream (external)
    - `java.util.Collection`

Iterator

- Related Patterns
  - Composite
    - Iterators are often used for recursive structures such as Composites
  - Factory Method
    - Factory Methods are used by Iterators to instantiate the indicated Iterator subclass.
  - Memento
    - Memento and Iterator are often combined – An Iterator could use a Memento to gather the state of an iteration