Software Engineering I (02161)

Week 9

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2011

Recap

- Project planning
  - Classical project planning:
    - Divide the project into milestones and deliverables
    - Define the tasks that are needed to reach milestone and deliverables
    - → Schedule tasks using, e.g., Gantt charts
  - Project estimation techniques
    - from experience
    - algorithmic based (e.g. Constructive Cost Model / COCOMO)
  - Agile project planning: fixed project structure with time boxed releases and iterations; assign user stories to iterations
- Version Control
- Introduction to the project

What does this function do? Specification (Contract)

```
Vector<Integer> f(Vector<Integer> a)
```

Precondition: a is not null

Postcondition: For all $result, a \in$ Vector$<$Integer$>$:

$result == f(a)$

if and only if

P(result) and Q(a,result)

where

P(result) if and only if

for all $0 \leq i, j < result.size()$:

$i \leq j$ implies $result.elementAt(i) \leq r.elementAt(j)$

and

Q(a,result) if and only if

for all $i \in Integer$: $count(a, i) = count(result, i)$

What does this function do?

Implementation

```
public Vector<Integer> f(Vector<Integer> vector) {
   if (vector.size() <= 1) return vector;

   int k = vector.elementAt(0);

   Vector<Integer> l1 = new Vector<Integer>();
   Vector<Integer> l2 = new Vector<Integer>();
   Vector<Integer> l3 = new Vector<Integer>();

   h(k,vector,l1,l3,l2);

   Vector<Integer> r = f(l1);

   r.addAll(l3);
   r.addAll(f(l2));

   return r;
}

 public void h(int k, Vector<Integer> vector,
               Vector<Integer> l1,
               Vector<Integer> l2,
               Vector<Integer> l3) {
   for (int i : vector) {
      if (i < k) l1.add(i);
      if (i == k) l3.add(i);
      if (i > k) l2.add(i);
   }
}
```

Function descriptions by name and implementation

Main question

Given a signature of a function, what does this function do?

- Answer 1: Look at the implementation
  - In most cases one relies on the name of the function: e.g. qsort
    - But is the name correct? Does qsort really mean quick sort?
    - What are the preconditions for the method to work? E.g. qsort requires that the vector is not null!)
    - Currently with most libraries one actually relies on the function name and the manual to explain what the function does
  - Then one looks at the implementation
    - the quick-sort algorithm in this simple version is easy to recognise
    - What about insertion sort, heap sort, bubble sort, more modern variant of quick sort, optimised implementations, . . . ?

Use of intention revealing names

- Use names expressing the intend of a piece of code / expression / variable

```
Vector<Integer> sort(Vector<Integer> a)
```

Precondition: a is not null

Postcondition: For all $result, a \in$ Vector$<$Integer$>$:

$result == sort(a)$

if and only if

isSorted(result) and sameElements(a,result)

where

isSorted(result) if and only if

for all $0 \leq i, j < result.size()$:

$i \leq j$ implies $result.elementAt(i) \leq r.elementAt(j)$

and

sameElements(a,result) if and only if

for all $i \in Integer$: $count(a, i) = count(result, i)$

Function description by contract

- Answer 2: Look at it behaviour specification
  - → also called contract of the function
  - Formal description of
    - What values are allowed?
    - → Precondition
    - What is the desired outcome?
    - → Postcondition
  - One does not need to understand the algorithm used in the implementation to understand what the program does
    - → Design by contract
    - Use the method / class as a black box
    - Rely on its contract instead of its implementation

Design by contract

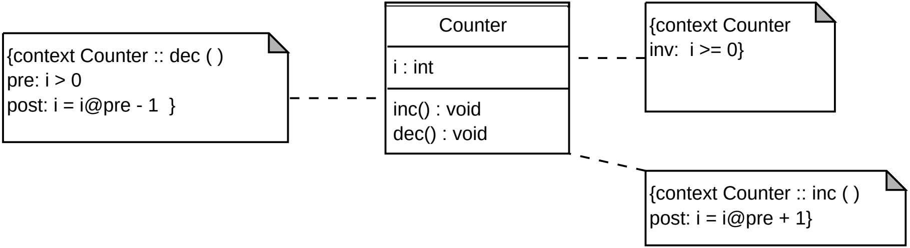
Design by contract is a term coined by Bertrand Meyer for describing a design method based on pre- and postconditions

Contract

A contract is an agreement between the caller of a method and the method itself

- The caller must ensure the precondition (and the invariant) of a method.
- The method must ensure the postcondition (and the invariant), but only when the caller has done his part of the contract (i.e. ensuring the precondition and the postcondition)

Example Counter



Example Counter

```
public T n(T1 a1, .., Tn an, Counter c)
    ...
    // Here the precondition of c has to hold
    // to fulfil the contract
    c.dec();
    // Before returning from dec, c has to ensure the
    // postcondition of dec
    ...
```

- Method *n* is responsible for ensuring that it calls `dec` only when $c.i > 0$
- If $c.i$ has the value k then after calling $c.dec()$ $c.i$ has the value $k - 1$.
- Note that is not important if *c* is created in *n* or passed as an argument

Precondition

Precondition of a method

A precondition is a condition on the state that needs to be satisfied whenever a method is called

- Preconditions can refer to
  - States of the object before executing the method
  - States of objects reachable from the state before executing the method
  - Method arguments

Postcondition

Postcondition of a method

A postcondition binary condition (= binary predicate) describing what has to hold in the state of the system after the execution of a method relating to the state of the system before executing the method

- Postconditions can refer to
  - States of the object before and after executing the method
  - To refer to the state before executing the method, one can use, e.g., *@pre*, i.e. x@pre
  - States of objects reachable from the state of the object before and after executing *m*
  - Method arguments
  - Result of the method (e.g., via `result`)

Invariants

Invariant

An invariant is a condition that restricts the set of possible system states.

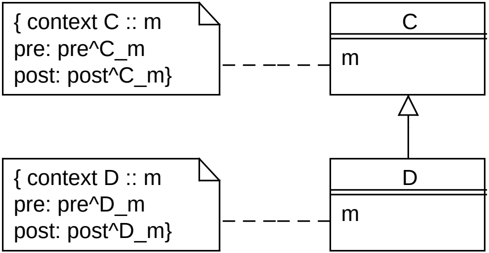
- The system can only be in a state defined by the invariant(s)

```
context Counter
inv: i >= 0
```

- Operations can assume that the invariant holds and must itself establish the invariant

Contracts and inheritance (I)

- Assume that D is a subclass of C, overriding method m of class C
  - class C has a method m with precondition pre_m^C and postcondition $post_m^C$.
  - subclass D of C overrides m with an implementation satisfying the precondition pre_m^D and postcondition $post_m^D$.



Contracts and Inheritance (II)

Liskov / Wing Substitution principle:

At every place, where one can use objects of the superclass C, one can use objects of the subclass D

```
public T n(T1 a1, .., Tn an, C c)
   ...
   // n has to ensure the precondition Pre^C_m for calling m
   c.m();
   // n can rely on that m ensures the postcondition Post^C_m
   ...
```

- If $c_1 = new\ C()$ and $d = new\ D()$ then one should be able to use $n(a_1, \ldots, a_n, d)$ instead of $n(a_1, \ldots, a_n, c_1)$

$\rightarrow$ Subclasses methods must honour the contracts of the superclass methods

Inheritance and preconditions

```
public T n(T1 a1, .., Tn an, C c)
   ...
   // n has to ensure the precondition Pre^C_m for calling m
   c.m();
   // n can rely on that m ensures the postcondition Post^C_m
   ...
```

- We want to call $n(a_1, \ldots, a_n, d)$ (d instance of class D)
- According to the contract n has with method m of class C, it only has to establish the precondition of m for class C: Pre_m^C
- Thus, the implementation of m in class D can only rely on Pre_m^C
  - This means Pre_m^D is either the same as Pre_m^C or weaker

  $\rightarrow$ pre_m^C implies pre_m^D

Inheritance and postconditions

```
public T n(T1 a1, .., Tn an, C c)
   ...
   // n has to ensure the precondition Pre^C_m for calling m
   c.m();
   // n can rely on that m ensures the postcondition Post^C_m
   ...
```

Calling $n(a_1, \ldots, a_n, d)$ (d instance of class D)

- n can assume that c has established the postcondition $Post_m^C$
- Thus m in D has to establish either $Post_m^C$ or a more restrictive postcondition

$\rightarrow$ $post_m^D$ implies $post_m^C$

Why relying on contracts and not directly on the implementation?

- Contracts describe what a method should be doing and not how
  - Implementations can be rather complex compared to contracts:

  → e.g. sort example
- Encapsulation
  - The implementation can be treated as a black box
  - When reasoning about the program one can rely on the contract and don't need to know about the implementation
  - This is the basic principle to construct larger system

    → who knows how an XML parser works or how the Vector datatype is implemented?
- Make explicit the assumptions under which a method works → precondition → robust programming

But . . .

- Contracts can be trivial
  - → e.g. when the implementation has the same complexity as the contract
  - Sometimes it is easier to say how something is done than to say what the result is
    - → e.g. contract of getter/setter functions
- Contracts can be unsatisfiable
  - E.g. if one writes something corresponding to $\mathit{true} = \mathit{false}$ (e.g. contradictions)
- Writing easy to read and error free contracts is far from trivial
  - In my experience: the complexity of writing a correct formal contract is in the order of writing the program

Implementation of Design by Contract (DbC)

- Use a programming language that directly supports design by contract, e.g. Eiffel or Spec#
- Many languages provide an assert statement which helps to implement DbC (e.g. Java)

```
void dec() {
   assert i > 0; // Precondition
   int prei = i; // Remember the value of the counter
                 // to be used in the postcondition
   i--;
   assert i == prei-1; // Postcondition
   assert i >= 0; // Invariant
}
```

- Note that assert bexp; is not the same as assertTrue(bexp);
  - assertTrue is a JUnit library function used for tests
  - assert is a Java language construct used in production code
- Often, assertions are enabled during development and disabled in the field
  - In Java assertion checking is disabled by default
  - `java -enableassertions Main` to enable assertion checking

Implementing DbC with assertions

```
void dec() {
   assert i > 0; // Precondition
   int prei = i; // Remember the value of the counter
                 // to be used in the postcondition
   i--;
   assert i == prei-1; // Postcondition
   assert i >= 0; // Invariant
}
```

- One assert statement for the precondition
- Possible code to remember the previous state of the object
- One assert statement for the postcondition
- One assert statement for the precondition
- Why not check the for the invariant at the beginning of the method?
  - The invariant is established by the constructor and each method (checked through assertions)

  → Encapsulation means that nobody else then constructor and methods can change the state of the object

  → The invariant has to be established at the beginning of each method

Distinction between defensive programming and design by contract

- Design by contract relies on the fact that the client fulfils the contract of the method

  $\rightarrow$ The precondition is not checked in the method
  - Example: Counter dec method
    - Precondition: $i > 0$
    - Postcondition: $i = i@pre - 1$
    - Don't check for the precondition

```
void dec() { i--; }
```

Distinction between defensive programming and design by contract

- Defensive programming does not trust the client to fulfil the contract of the method, instead the method checks itself that it works correctly
  - → Check all input parameters
    `void dec() { if (i > 0) i--; }`
  - The contract for this version of dec is:

    ```
    pre:  true
    post:  (i <= 0 implies i=0) and (i > 0 implies i = i@pre-1)
    ```

    → Why is the above method a correct implementation of this post condition?
  - Alternatively, using under specification we leave open what happens when $i <= 0$

    ```
    pre:  true
    post:  i > 0 implies i = i@pre-1
    ```
  - Note that in both cases the client is not obliged to fulfil $i > 0$

Defensive Programming vs. Design by Contract

- Design by Contract
  - Use explicit preconditions when you have control over the caller to make sure that he honours the contract
  - Include assertions for preconditions

```
void dec() {
    assert i > 0;
    if (i > 0) i--;
}
```

  - Don't turn of these preconditions (only if you can prove that the precondition will never be violated and even then be careful

    → Ariane 5
    - `http://www.youtube.com/watch?v=kYUrqdUyEpI`
    - For more information why it happened `http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html`)
- Defensive Programming
  - Use defensive programming when you cannot assume anything from the caller
  - For example for public methods in a library

Design Patterns (II)

Pattern

A pattern is a solution to a problem in context

- Design patterns book by ”Gang of Four” (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)
- So far: Template Method, Observer Pattern, State Pattern, Composite Pattern
- Today: Visitor Pattern, Facade, Strategy / Policy, Decorator, Adapter / Wrapper

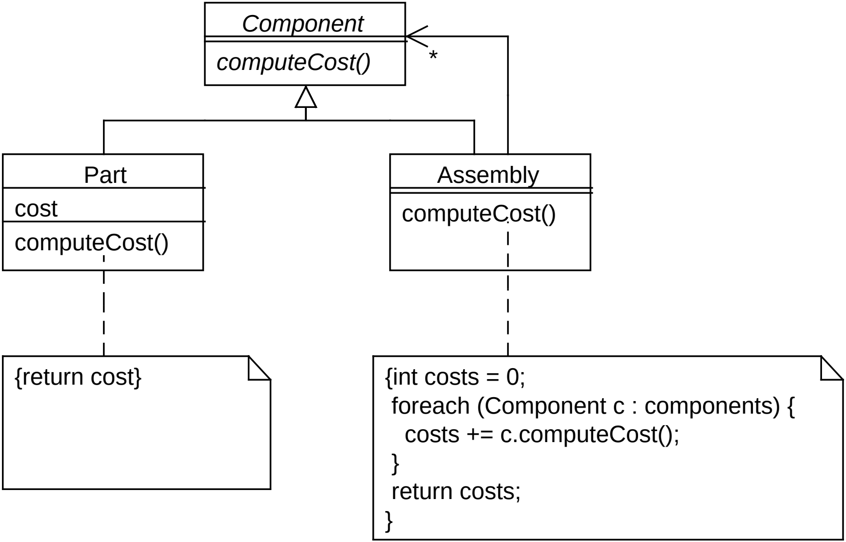
Visitor Pattern

Visitor Pattern

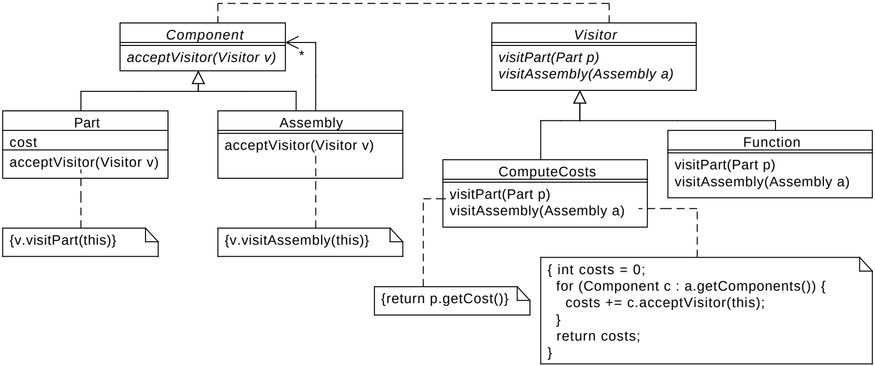
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- Instead of of adding a new function to each of a class hierarchy (which could be based on the composite pattern), add a new class Function that encapsulates the function
- The object structure (e.g. based on a composite pattern) provides access to itself through a set of methods

Example: compute costs for components



Example: compute costs as a visitor



- The method computeCosts() is made into a class ComputeCosts
- Still, through dynamic dispatch, we avoid code like
  `if (c instanceof Assembly) ...`

→ Easy to adapt to changes in composite structure

Visitor pattern

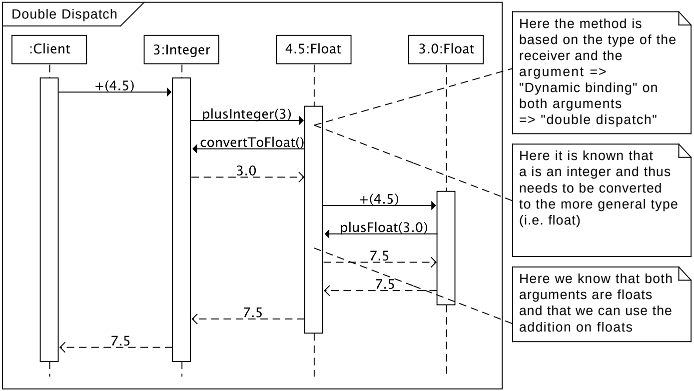
- The trick of the visitor is to use double dispatch
  - add type information to the method name
    - acceptVisitor → visitPart, visitAssembly
- Use the visitor pattern if
  - The functions don't belong to the concept of the object structure: e.g. generator functions
  - One should be able to do traverse an object structure without wanting to add operations to the object structure
  - One has several functions almost the same. Then one can use the visitor pattern and inheritance between the visitors do define slight variants of the functions (e.g. only overriding visitPart)
- Do not use it
  - if the complexity of the visitor pattern is not justified
  - if the functions belongs conceptually to the object structure
  - If the flexibility of the visitor is not needed, e.g. if one only wants to add one function

Double Dispatch

Compute $3 + 4.5$

- Assume numbers are first class objects (e.g. like in Ruby or Smalltalk but unlike Java)

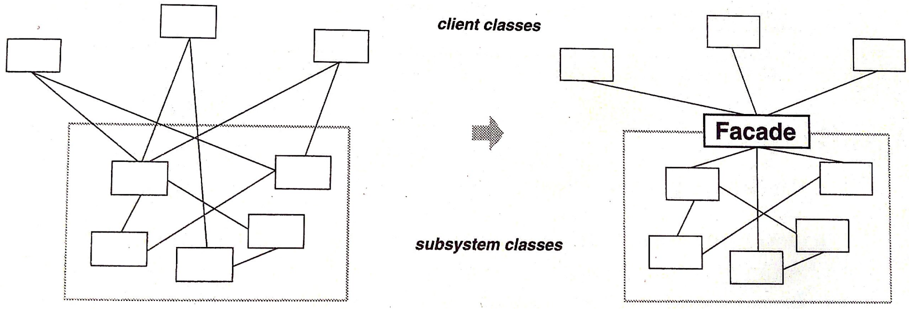
$\rightarrow$ the message $+$ is sent to the integer object 3 (i.e. $3. + (4.5)$)



Facade

Facade

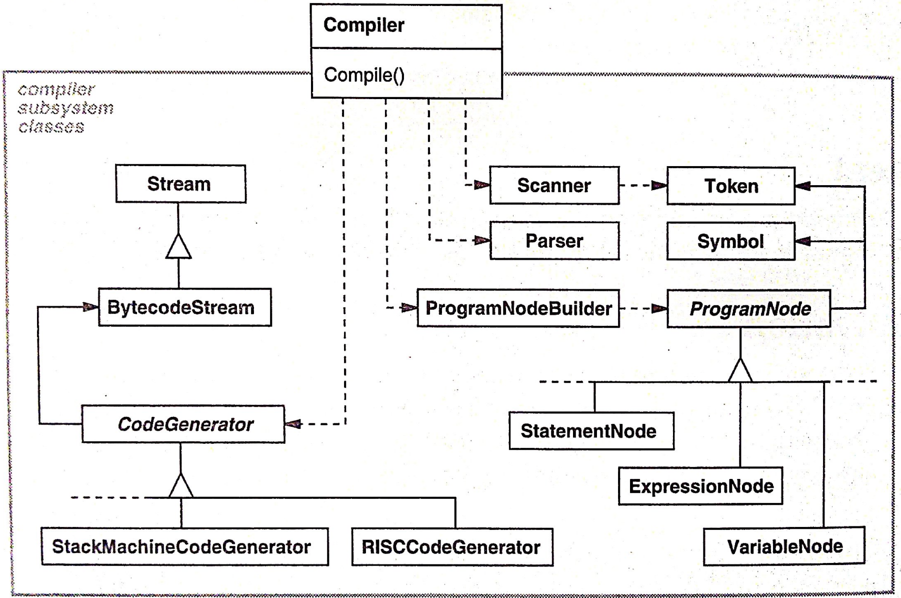
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystems easier to use.



- Achieves low coupling

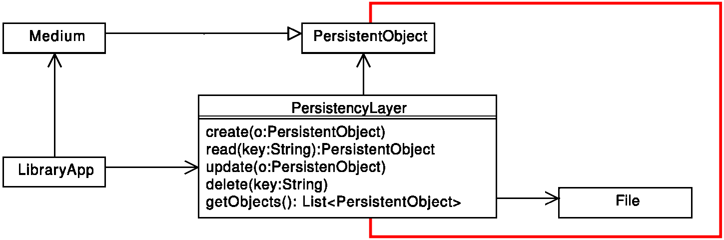
Design Patterns, Addison-Wesley, 1994

Example Compiler

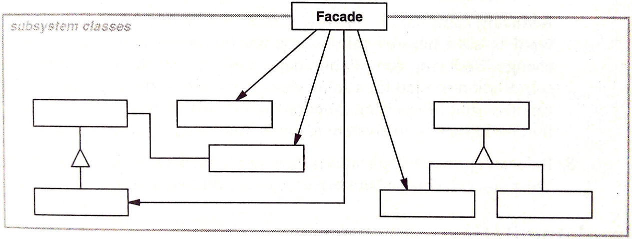


Example Persistency Layer

- PersistencyLayer acts as a facade to the persistent story
- We have used simple files to store the objects, but we could have used, e.g., relational databases to do the storage
- LibraryApplication uses the persistent storage only through the operations offered by the PersistencyLayer



Applicability



Use the Facade pattern to

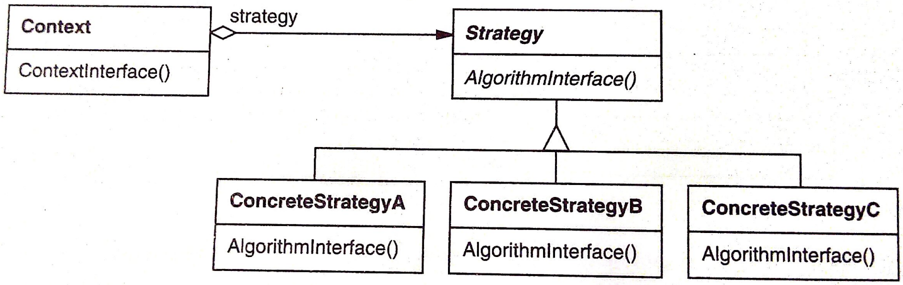
- provide a simple interface to a complex subsystem
- decouple between clients from the implementation class of an abstraction
- implement a layered structure
  - → Layered Architecture
  - Sometimes the application layer is also called application facade

Design Patterns, Addison-Wesley, 1994

Strategy / Policy

Strategy / Policy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Design Patterns, Addison-Wesley, 1994

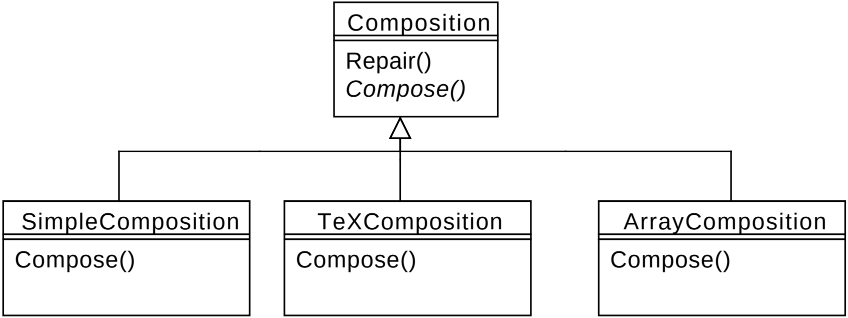
Example: Text formatting

E.g. text formatting algorithm that uses several strategies to create line breaks:

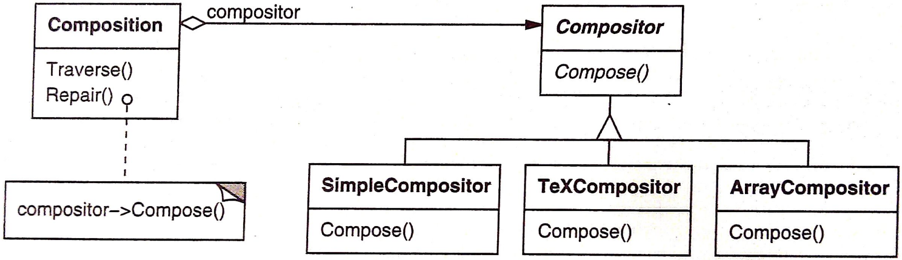
- Simple compositor: Simple algorithm where each component does not have a stretchability (e.g. white space has a constant width)
- TeX compositor: creates line breaks based on looking at the whole paragraph; incorporates stretchability information of components (e.g. white spaces between words and between sentences); tries to produce line breaks with result in an even ”colour” of the paragraph
- Array compositor: Fills lines to *n* characters regardless of word boundaries

```
public void repair() {
   ...
   if (strategy = "simple")
        // Do simple linebreak
   else if (strategy = "tex")
      // Use TeX's algorithm
   else if (strategy = "array")
      // Do array style linebreak
   ..
}
```

Possible solution using inheritance (= Template Method)



Possible solution using distributed control (= Strategy Pattern)



Design Patterns, Addison-Wesley, 1994

Applicability

Use the Strategy pattern to

- configure a class with one of many behaviours (each behaviour is represented by one class)
- use different variants of the same

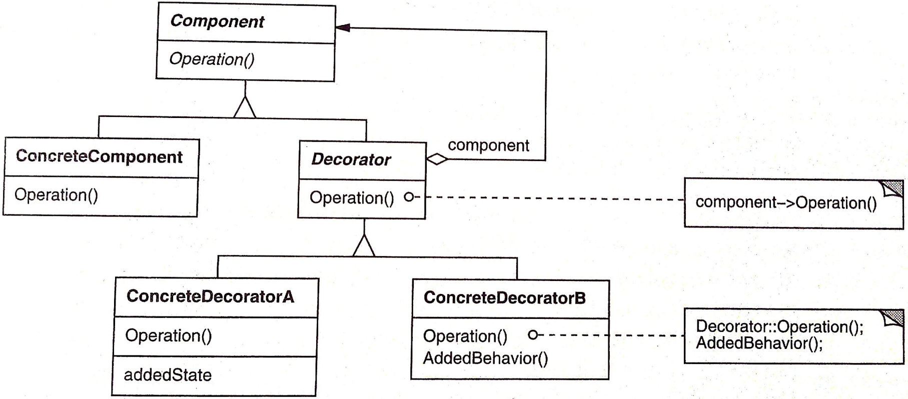
Template method can only be used ”once” (e.g. each subclass represents one strategy)

- Strategy pattern can be applied several times to the same class (e.g. replacing more than one part of the computation, e.g. strategy for line break, strategy for page break, ...)

Decorator

Decorator

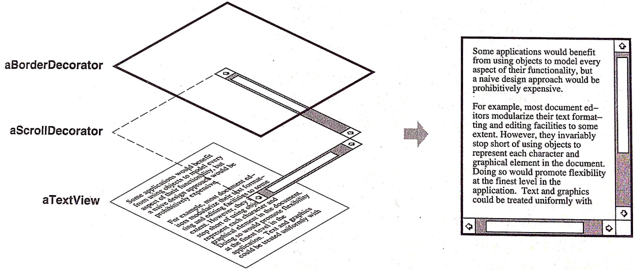
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

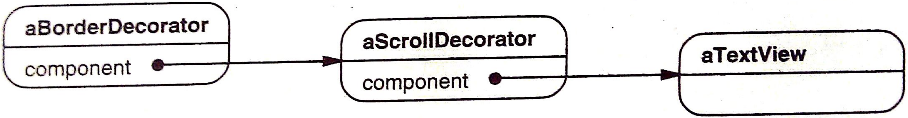


Example: Window decorators

Problem: To add to a window in a window system the capability to have boarder and to be able to scroll

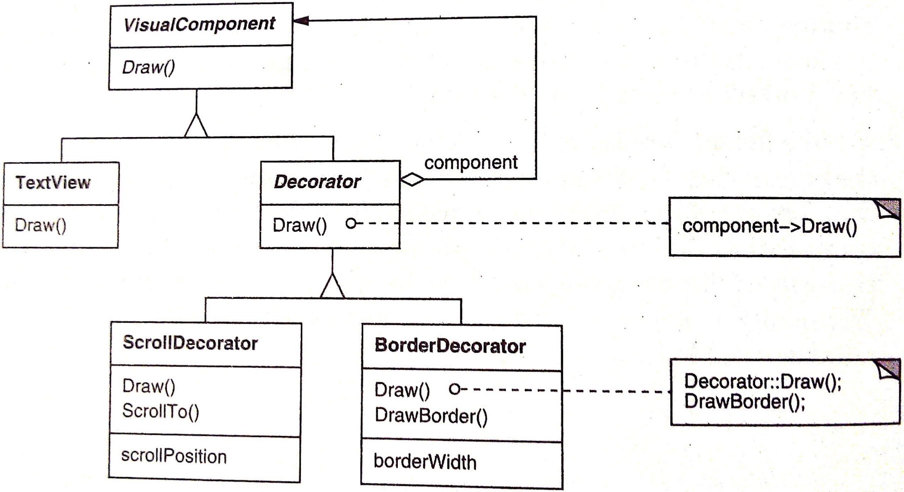
- New functionality: drawing a boarder and to scroll to a position





Design Patterns, Addison-Wesley, 1994

Example: Window decorators

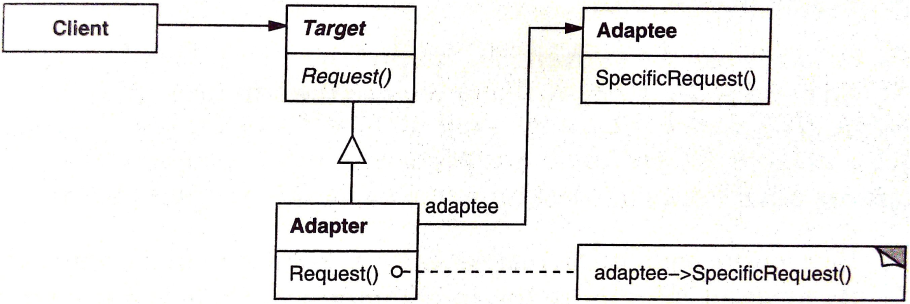


Design Patterns, Addison-Wesley, 1994

Adapter / Wrapper

Adapter / Wrapper

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



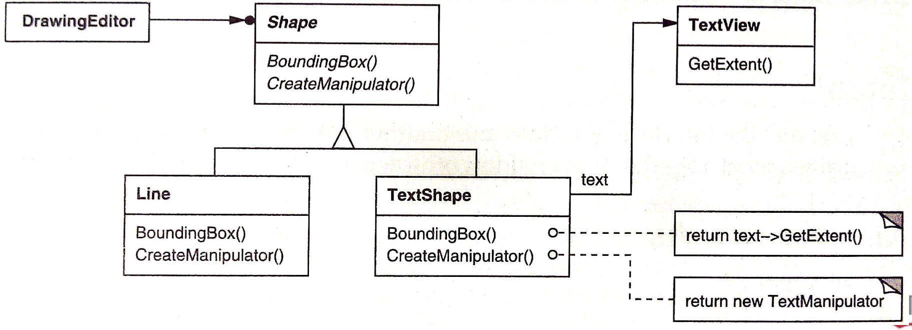
Design Patterns, Addison-Wesley, 1994

Example: Using text views in a graphics editor

Problem: To reuse a text view as a special kind of shape in a graphics editor

- But, the text view itself is not a shape

Solution: define a class TextShape that has the shape interface and that translates requests to TextShape to requests that the text view understands



Design Patterns, Addison-Wesley, 1994

Summary: Good Design and Patterns

- There is a difference between good and bad design
  - Good design is important
- Some basic principles:
  - DRY, KISS
- Patterns capture knowledge
  - → Design patterns capture common object oriented design principles
- Low Coupling and High Cohesion
  - → Leads to modularised / object oriented design
- Layered Architecture
  - → application of the low coupling and high cohesion principle
  - supports independence of application from UI
  - supports automated tests

→ Good design is a life long learning process
  - e.g. when or when not to apply certain patterns
  - think about what and why you are doing it!

Summary

- Design by contract (DbC)
  - Contract between client of a method and the method:
    - Client ensures precondition, method ensures postcondition
  - Implementation of DbC in Java: using the assert statement (for precondition, postcondition, and class invariant)
    - Usually assertion checking is enabled during development but not in the field (default is that assertion checking is disabled)
  - Defensive programming: The method does not rely on the client to provide correct data
    - e.g. with public library functions
- Design Patterns (II)
  - Visitor, Facade, Strategy / Policy, Decorator, Adapter / Wrapper
  - Places to find more patterns:
    - Wikipedia `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`
    - Portland Pattern repository `http://c2.com/cgi/wiki?PeopleProjectsAndPatterns` (since 1995)
    - Wikipedia `http://en.wikipedia.org/wiki/Category:Software_design_patterns`