Patterns in Object-Oriented Software Development

PATTERNS

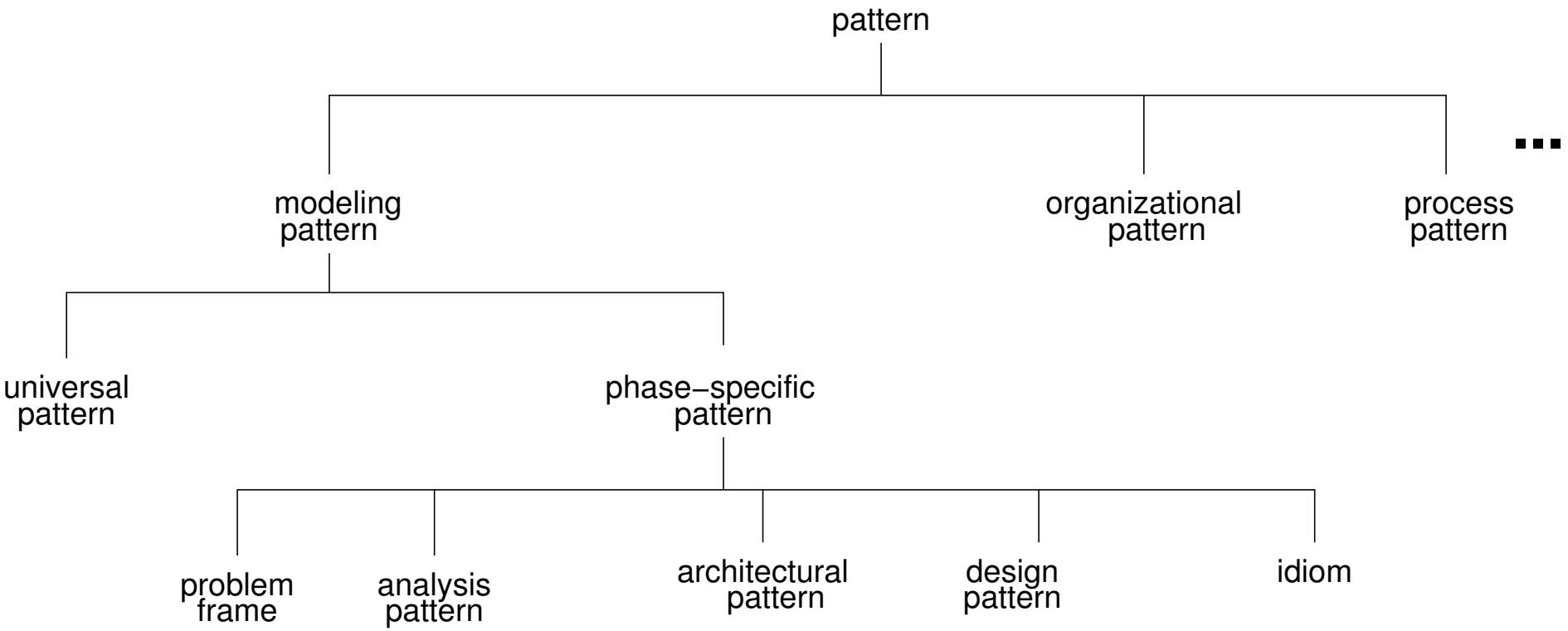
- Are schematic solutions for a class of related problems
- Serve to represent and reuse software development knowledge
- Reuse is achieved by instantiation
- Can be used on different levels of abstraction:
  - Problem Frames to represent software development problems
  - Analysis patterns for object-oriented analysis
  - Architectural patterns for software architectures
  - Design patterns for object-oriented design
  - Idioms for object-oriented programming

Representation of patterns

- name, possibly synonyms
- problem
  - motivation, application domain
- solution(s)
  - structure (class diagram)
  - components (schematic class and object names)
  - description, possibly application flow, e.g sequence diagram
- discussion
  - advantages and disadvantages, dependencies, restrictions
- related patterns (similarities)

> Patterns are a starting point, not a destination.
>
> Martin Fowler

CLASSIFICATION OF PATTERNS

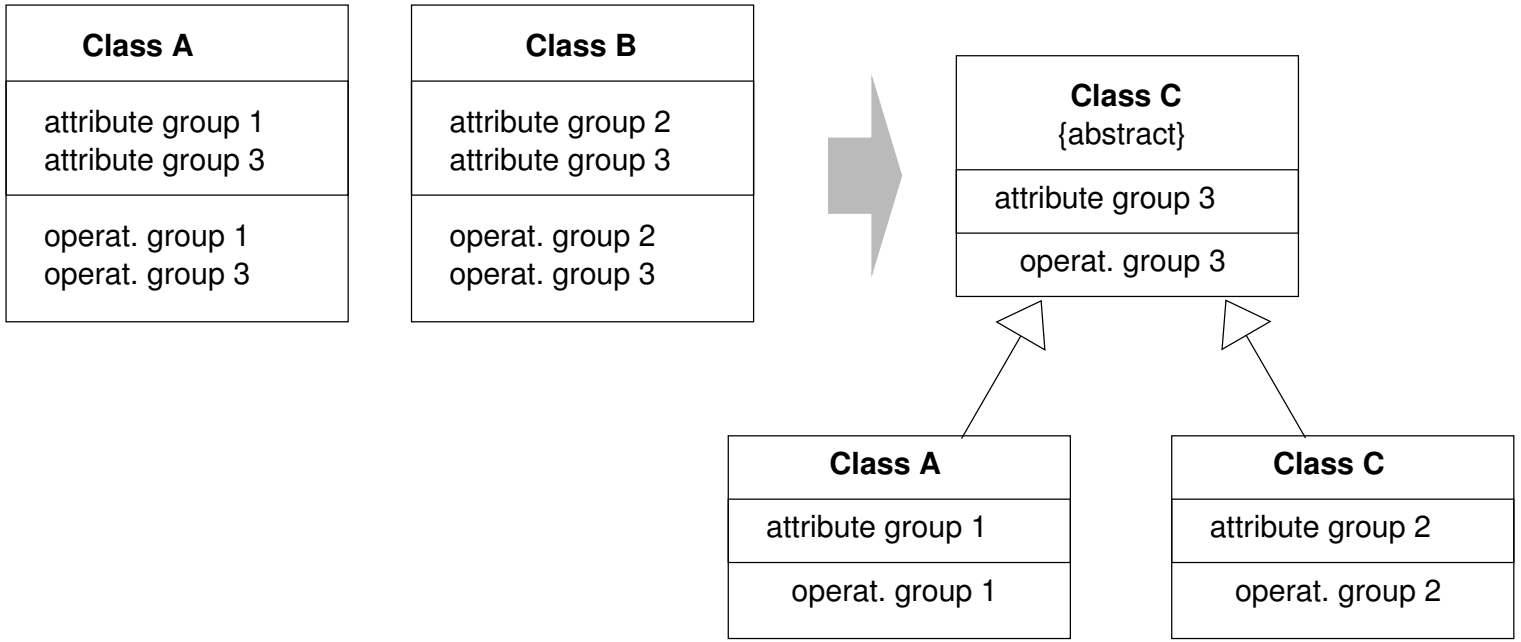


PATTERNS IN THE ANALYSIS PHASE

- Help to set up class models (first model in Fusion OOA)
- Serve to improve class structures (refactoring)
- Depend on application domain
- Are mostly used for business applications

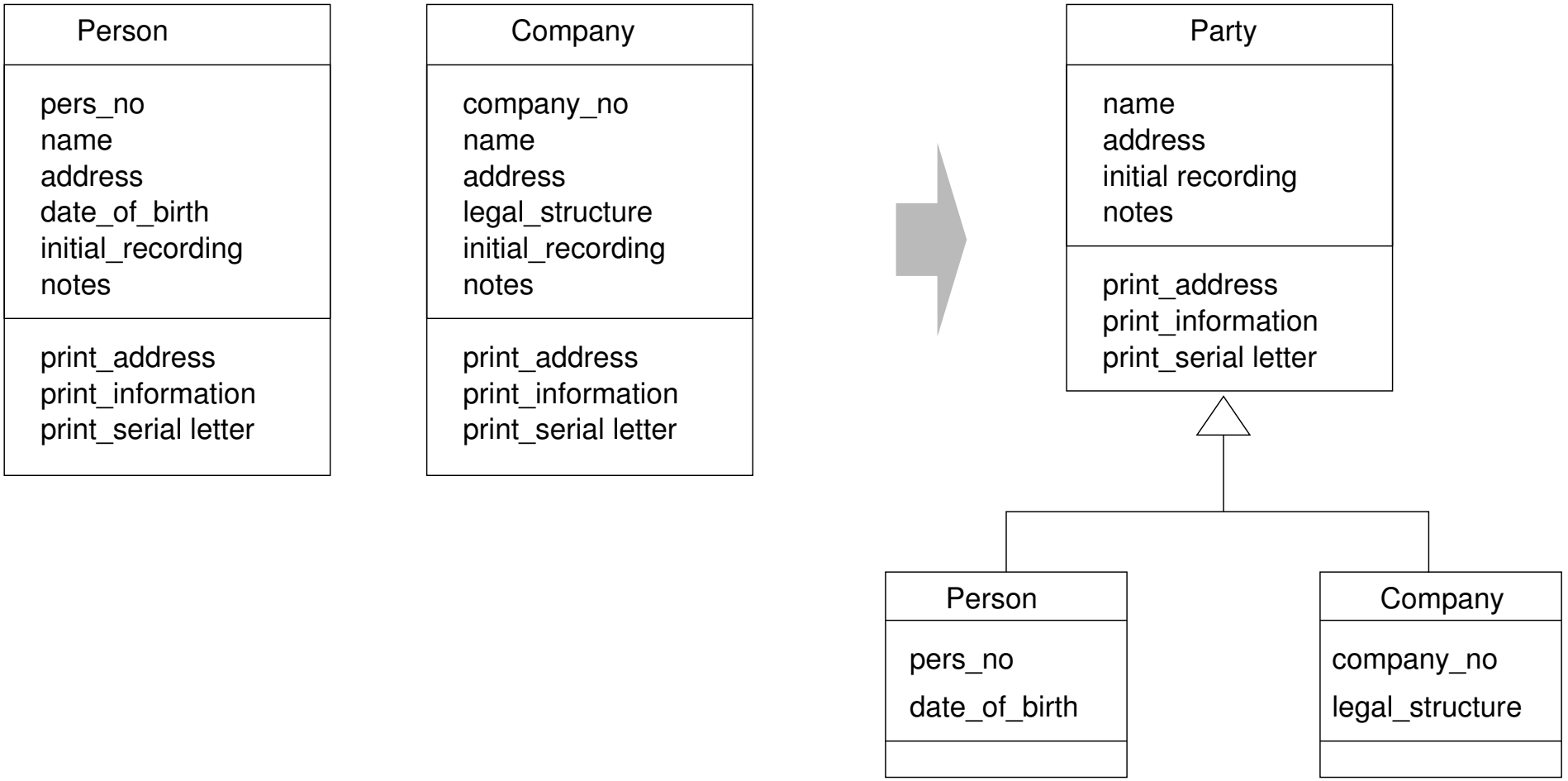
ABSTRACT SUPERCLASS

- Universal pattern
- **Problem:** Classes contain groups of identical attributes and operations.
- **Solution:** Separate the identical features in an abstract superclass.



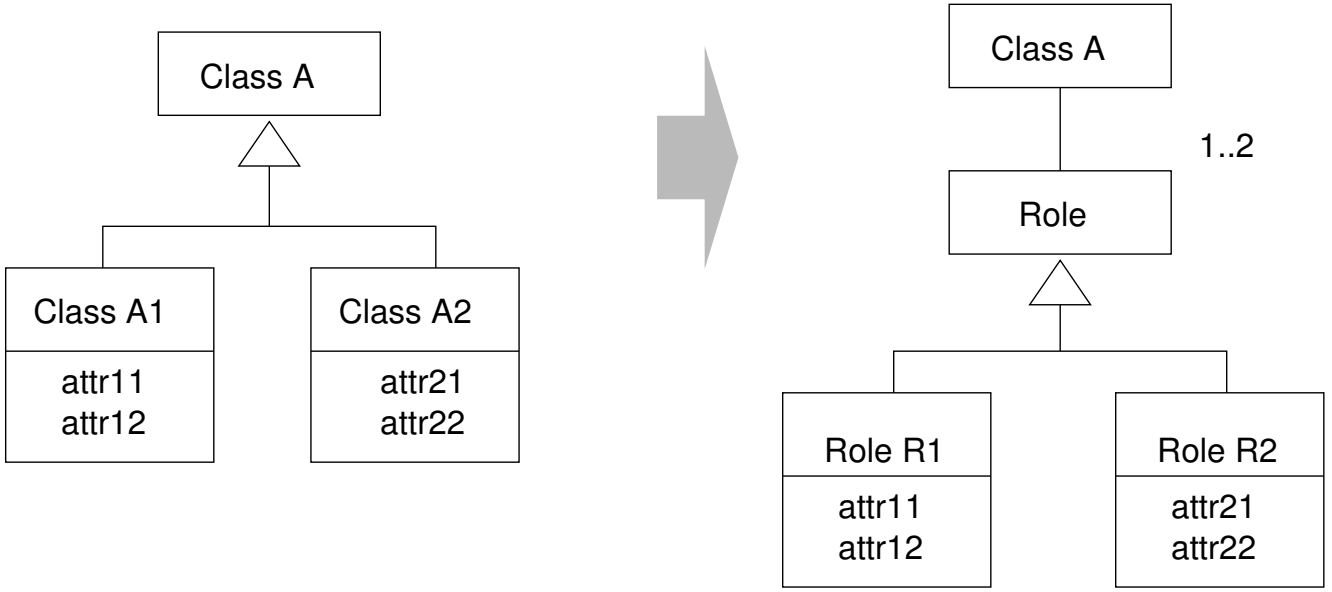
Remark: This pattern is also useful to construct the inheritance model of Fusion OOD.

ABSTRACT SUPERCLASS: EXAMPLE

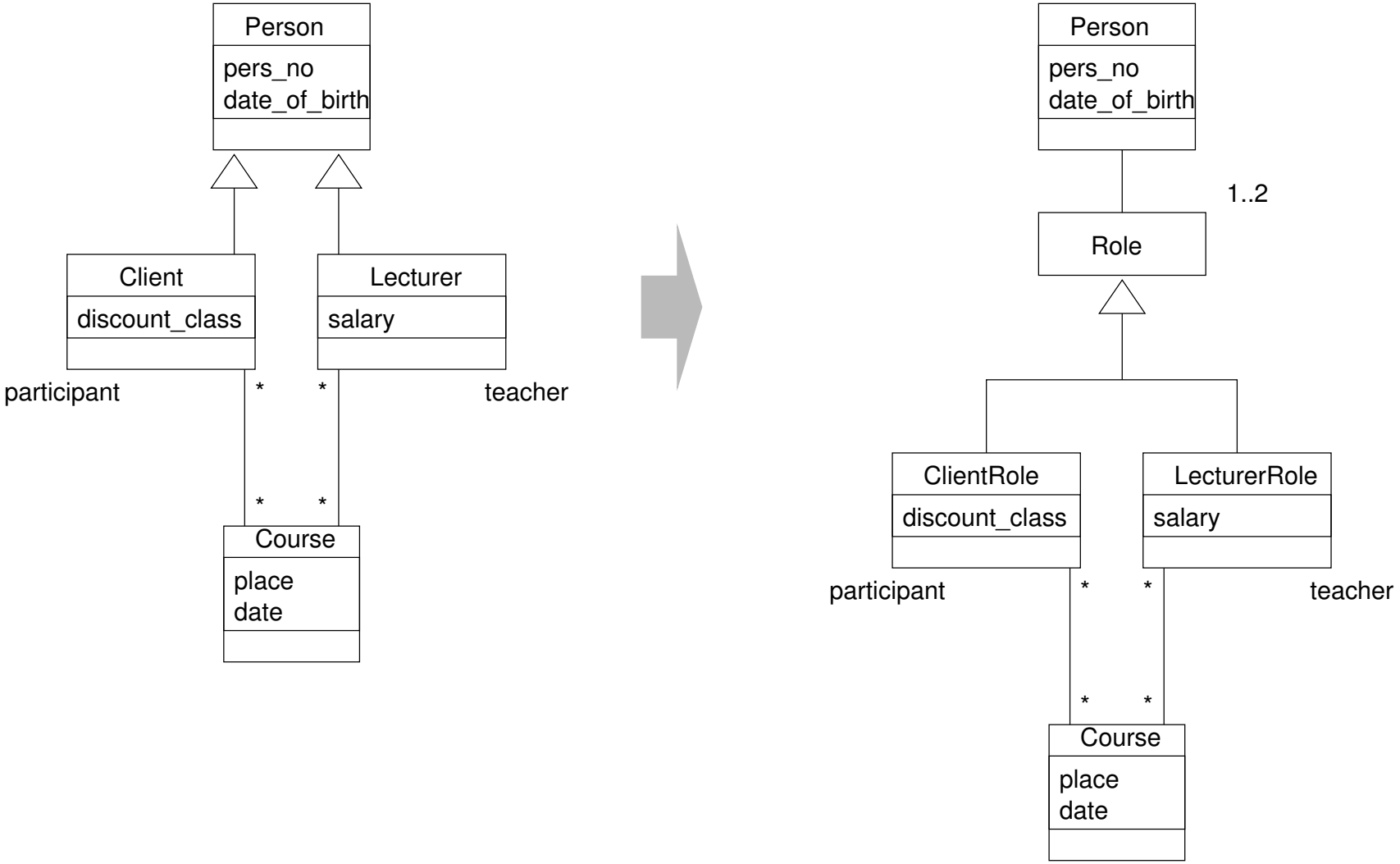


ROLES

- Universal pattern
- **Problem:** Two or more different types of usages exist for a given class. The creation of subclasses would lead to ”overlapping” subclasses.
- **Solution:** Introduce an association for a “role”-class with multiplicity according to the number of usage types of an object. Role classes contain attributes corresponding to the usage types.



ROLES: EXAMPLE



Person can be client and lecturer at the same time.

ANALYSIS PATTERNS: SUMMARY

- Analysis patterns are reusable object-oriented class models
- The following are applied for analysis:
  - universal patterns
  - specific analysis patterns: domain-specific
- universal patterns
  - ”guideline” for modeling
  - components for other patterns
- specific analysis patterns
  - only few catalogues published
    - Fowler: catalogue for business applications
  - company and project specific catalogues

Design Patterns

CHARACTERIZED BY

- usage for detailed design
- object-oriented paradigm
- “Description of a family of solutions for a software design problem” (Tichy)

LITERATURE

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.
Design Patterns – Elements of Reusable Object-Oriented Software.
Addison Wesley, 1995.

(“Gang of Four”, GoF)

John Vlissides.
Pattern Hatching – Design Patterns Applied.
Addison Wesley, 1998.

Slide collection of Walter F. Tichy, University of Karlsruhe
Slides of the TU Dresden (H. Humann, Th. Santen)

TYPES OF DESIGN PATTERNS (GOF)

- *creational*

  concern the process of object creation

- *structural*

  deal with the composition of classes or objects

- *behavioral*

  characterize the ways in which classes or objects interact and distribute responsibility

Second criterion: *scope*

specifies whether the pattern applies primarily to classes or to objects.

TYPES OF DESIGN PATTERNS (TICHY)(1)

- Coupling/decoupling patterns

  System is divided into units that can be changed independently from each other

  e.g. Iterator, Facade, Proxy

- Unification patterns

  Similarities are extracted and only described at one place.

  e.g. Composite, Abstract Factory

- Data-structure patterns

  Process states of objects independently of their responsibilities

  e.g. Memento, Singleton

TYPES OF DESIGN PATTERNS (TICHY)(2)

- Control flow patterns

  Influence the control flow; provide for the right method to be called at the right time

  e.g. Strategy, Visitor

- Virtual machines

  Receive programs and data as input, execute programs according to data

  e.g. Interpreter

  (Remark: no clear boundary to architectural styles)

Advantages of Design Patterns (Tichy)

- Improvement of team communication

  Design pattern as “short formula” in discussions

- Compilation of essential concepts, expressed in a concrete form

- Documentation of the “state of the art”

  Help for less experienced designers, not constantly reinventing the wheel

- Improvement of the code quality

  Given structure, code examples

Description of Design Patterns (GoF) (1)

Name and Classification A good name is important, because it will become part of the design vocabulary.

Intent What does the pattern do? Which problems does it solve?

Also Known As Other familiar names.

Motivation Scenario which illustrates the design problem and how the pattern solves the problem.

Applicability What are the situations in which the design pattern can be applied? How can one recognize these situations?

Structure Class and interaction diagrams.

Participants Classes and objects, which are part of the pattern, as well as their responsibilities.

DESCRIPTION OF DESIGN PATTERNS (GOF) (2)

Collaborations How do the participants collaborate to carry out their responsibilities?

Consequences What are the trade-offs and results of using the pattern? What aspect of system structure does it let one vary independently?

Implementation What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there any language-specific issues?

Sample Code Code fragments in C^{++} or Smalltalk.

Known Uses At least two examples of applications taken from existing systems of different fields.

Related Patterns Similar patterns and patterns that are often used in combination with the described pattern.

EXAMPLE: "COMPOSITE"

Classification object/structural

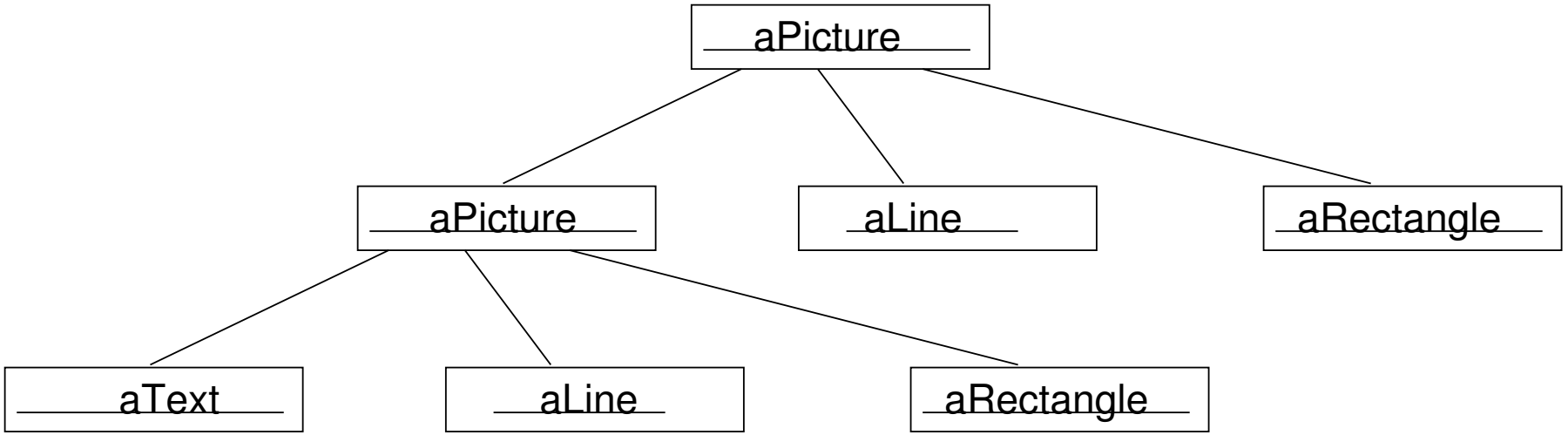
Intent Compose objects into tree structures to represent part-whole hierarchies. Composite lets you treat individual objects and compositions of objects uniformly.

Also Known As —

Motivation Users can build complex diagrams out of simple components by using graphics applications.

Problem: Code that uses the corresponding classes must treat primitive and container objects differently, even if most of the time the user treats them identically. The Composite pattern describes how a recursive composition can be designed so that the client does not have to distinguish between primitive objects and containers.

Example:

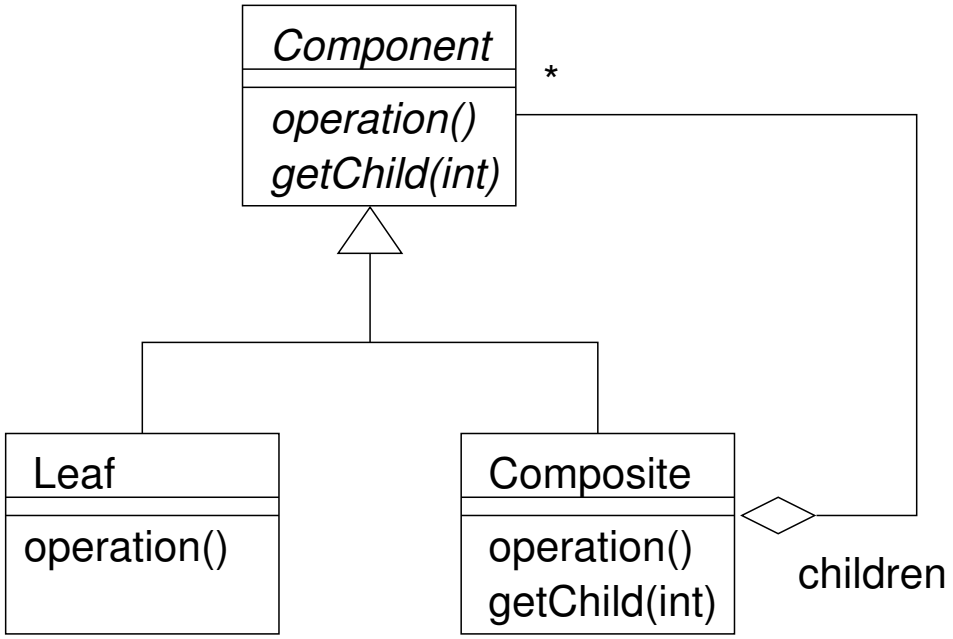


common operations: *draw*(), *move*, *delete*(), *scale*()

Applicability Use the Composite pattern when

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

Structure (abstract classes and operations are noted in *italics*)



Participants

- *Component* (graphic)
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all classes, as appropriate
  - declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
- *Leaf* (rectangle, line, text etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition

- *Composite* (picture)
  - defines behavior for components having children
  - stores child components
  - implements child-related operations in the *Component* interface
- *Client* (not contained in the class diagram)
  - manipulates objects in the composition through *Component* interface

Collaborations Clients use the *Component* class interface to interact with objects in the composite structure. If the recipient is a leaf, then the request is handled directly. If the recipient is a composite, then it usually forwards the request to its child components, possibly performing additional operations before and/or after forwarding.

Consequences The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects.

  Whenever client code expects a primitive object, then it can also take a composite object.

- makes the client simple

  Clients can treat composite structures and individual objects uniformly.

  Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite object. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

- makes it easier to add new kinds of components.

  Newly defined subclasses of *Composite* or *Leaf* work automatically with existing structures and client code. Clients don't have to be changed for new component classes.

- can make your design overly general

  The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Implementation GoF considers the following aspects:

1. Explicit parents references

   Should be defined in the *Component* class.

2. Sharing components

   Can be useful to reduce storage requirements, but destroys tree structure.

3. Maximizing the *Component* interface

   Necessary to make clients unaware of the specific *Leaf* or *Composite* classes they are using. Default implementation in *Component* can be overwritten in subclasses.

4. Declaring the child management operations

   Declaration of *add*- and *remove*-operations in the class *Component* results in transparency; all components can be treated uniformly. Costs safety, because meaningless operations can be called, e.g. adding to objects to leafs.

   Defining child management in the *Composites* class gives safety, but is at the expense of transparency (leaves and composites have different interfaces).

5. Should *Component* implement a list of components?

   Incurs a space penalty for every leaf.

6. Child ordering

   When child ordering is an issue, applying the Iterator pattern is recommended.

7. Caching to improve performance

   Useful, if the compositions have to be traversed or searched frequently.

8. Who should delete components?

   In languages without garbage collection, it's usually best to make a composite responsible for deleting its children when it's destroyed.

9. What's the best data structure for storing components?

   Depends on aspects of efficiency.

Sample Code Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies.

```
class Equipment {
public:
    virtual ~Equipment();
    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
```

```
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

Equipment declares operations that return the attributes of a piece of equipment, like its power consumption and cost. A *CreateIterator*-operation returns an iterator for accessing its parts.

Further classes such as *FloppyDisk* as class for leaves and *CompositeEquipment* for composite equipment are defined in the GoF-book.

Known Uses

- *View*-class in Model/View/Controller
- Composite structure for parse trees
- portfolio containing assets

Related Patterns

- Often the component-parent link is used for a Chain of Responsibility.
- Decorator is often used with composites. When decorators and composites are used together, they will usually have a common parent class.
- Flyweight lets you share components, but they can no longer refer to their parents.
- Iterator can be used to traverse composites.
- Visitor localizes operations and behavior that would otherwise be distributed across *Composite* and *Leaf* classes.

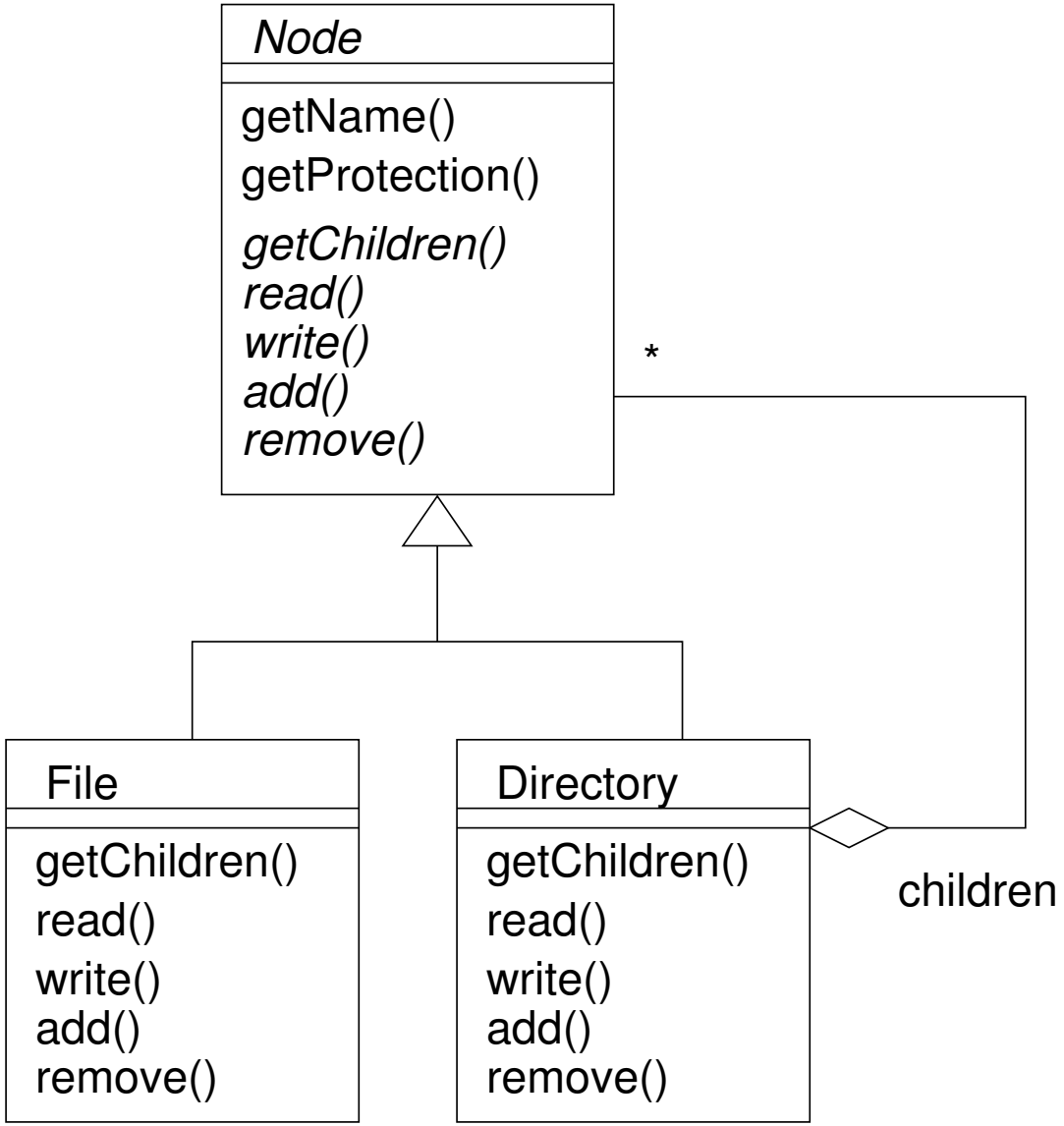
Example

User Interface of a File System

REQUIREMENTS (1)

- File system should be able to handle file structures of any size and complexity.
- *Directories* and (basic) *files* should be distinguished.
- The code, e.g. for selecting the name of a directory should be the same as for files. The same holds for size, access rights, etc.
- It should be easy to add new types of files (e.g. symbolic links).

APPLICATION OF THE COMPOSITE PATTERN



DESIGN PATTERNS: SUMMARY

- Design patterns are object oriented patterns at detailed design level.
- They are closer to implementation than analysis patterns.
- According to the GoF classification, there are behavioral, creational and structural patterns.
- Design patterns support achieving desirable properties in implementing object-oriented software, e.g. independent modification of parts, limitation of communication paths etc.