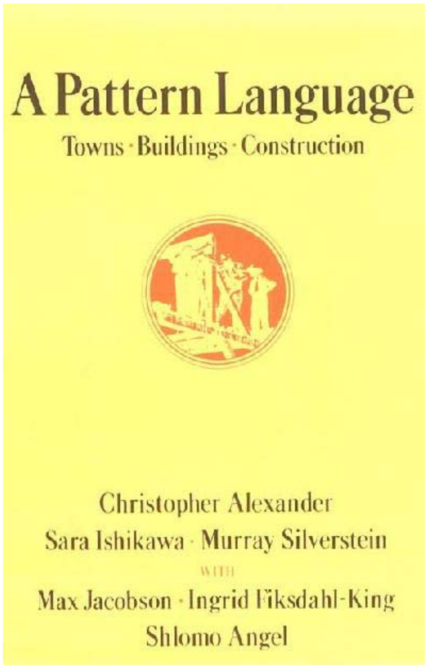
Design patterns

- The concept of a design pattern
- Origins: architecture
- Description of design patterns
- Examples: Composite, Abstract Factory, State
- Design patterns & frameworks
- Antipatterns
- Design patterns & UML
- Summary

The concept of a design pattern

Cristopher Alexander et al.: A Pattern Language, 1977
Cristopher Alexander: The Timeless Way of Building, 1979

- World consists of repeating instances of various patterns
- A pattern is (possibly hidden) design know-how that should be made explicit
- Well known in other engineering areas
- Particularly useful concept in software engineering



A Pattern Language

Towns · Buildings · Construction

Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

Origins of ideas in (buildings) architecture

- Christoffer Alexander: ”Is quality objective”?
- Some buildings possess ”a quality without a name”
  - an *objective* quality, not measured but something you recognize when you see it
  - something to do with *alive, whole, comfortable, exact, eternal*
  - Alexanders Claim: modern archtiecture lacks this quality
- Idea: use~~r~~ centered design
  - let the inhabitants design their own buildings, together with a professional (participatory design), in terms of *patterns*
  - capture the quality in a *pattern lanquage*, which is used to generate the design

Origins of ideas in architecture

- Alexander studied the problem of *objective quality* by making observations of buildings, towns, streets, gardens, any spaces that human beings have built
  - he discovered that high quality constructs had things in common
  - architectural structures differed from each others, even it they were of the same type solving the same problem. Yet different solutions were of high quality.
  - Alexander understood that structures could not be separated from the problem they are solving
- ...so he looked at different sturctures yielding a high quality solution to same problem and extracted the similarity of the structures, the core of the solution, which he calls *a pattern.*

- Alexanders patterns
  - solutions to a problem in a context
  - 253 patterns covering regions, towns, transportations, homes offices, rooms, lighthing, gardens, ...
  - a generative pattern language
    - each pattern defines subproblems solved by other smaller patterns

Alexanders pattern

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

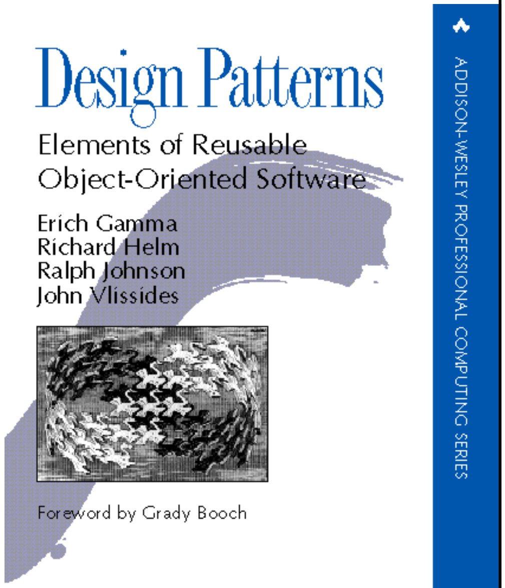
you can use this solution a million times over,

without ever doing it the same way twice

C. Alexander, “*The Timeless Way of Building*”, 1979

Alexander's View of a Pattern

- A pattern is a three part rule that expresses a relation between a certain *context*, a *problem* and a *solution*.
- A pattern is …
- *Element of the world* – a relationship between
  - a context
  - a system of *forces* that occur repeatedly in the context
  - a spatial configuration which allow forces to *resolve* themselves
- *Element of language* – an instruction
  - describes how the spatial configuration can be repeatedly used
  - to resolve the given system of forces
  - wherever the context makes it relevant
- The “*thing – process*” dualism, a pattern is both a thing and a process
  - a thing that happens in the world
  - a process (rule) which will generate that thing



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

From architectural to software design patterns

- Gamma (in GoF book): “[Patterns] are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”
- “A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.”
- So in short:
  - Reusable solutions to general design problems
  - Patterns capture well-proven experience in software development
    - Similar to handbooks in other disciplines
  - The pattern is applied to new situations
  - Basic steps are always the same, but the exact way of applying a pattern is always different.

Software Design Patterns

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - Patterns = **Problem/Solution** pair in **Context**
- Capture *static* and *dynamic structure* and *collaboration* among key participants in software designs
  - key participant – an abstraction that occurs in a design problem
  - useful for articulating the ***how*** and ***why*** to solve *non-functional forces*.
- Facilitate reuse of successful software architectures and design
  - i.e. the "*design of masters*"…

Definition of SW design pattern

A general solution to a frequently occurring architecture/design problem in a context.

General solution...
not specific to language, environment etc.
described as a semiformal document

...frequently occurring...
must be a common problem

...architecture/design problem...
applied at architecture or detailed design level

*...*in a context.
the problem appears in a context that defines certain requirements or forces

Why Design Patterns?

- make hidden design knowledge explicit and available
- can be used to document systems (instances of design patterns)
- name and make explicit a higher-level structure which is not directly supported by a programming language
- can be used as architectural building blocks
- give a common vocabulary for designers
- patterns are particularly useful for articulating how and why to resolve non-functional forces – the design *rationale*
- to sum up patterns...
  1) Reuse solutions - learn from other good designs, not your own mistakes
  2) Estabish common terminology – communication and teamwork
  3) Give a higher-level perspective on the problem and on the porcess of design and object orientation

Higher-level perspective – the greatest benefit?

- A conversation between two carpenters about how to build the drawers for some cabinet:

  ”I think we should make the joints of the drawer by cutting straight down into the wood, then cut back 45 degrees, and then going straight down again, then back up 45 degrees, then again down...”
- What is he saying?

  ”I think we should make a dovetail joint”
- Compare to a software engineer in a code review

  ”And then, I use a while loop here to do ... followed by a series of if statements to do ... and then I call the method ... and handle its return value with a switch statement ...”
- What is he saying?
  – No idea!

Higher-level perspective – the greatest benefit?

- The second carpenter asks ”Should we use a dovetail joint or a miter joint?”
- What is he really asking?
- Should we use a solution that is
  - laborous and expensive to make, requires a skilled carpenter
  - remains solid in changes of temperature and humidity
  - is independent of fastening system, does not require glue or nails
  - is aesthetically pleasing
  - can be sold with a better price
- or a solution that is ...
  - simple and cheap to make
  - weaker, does not hold together under heavy stress
  - inconspicious, the single cut is not very noticeable
  - must be sold on higher quantities due to cheaper price

The question was really about high-level non-functional properties of the design, accompanied with the structural solution

Documenting a pattern

- Gamma et al. used a standard procedure to describe and document design patterns. The use of a standard procedure increases understandability.
- Most books have adopted the same approach. By documenting the design pattern, knowledge becomes explicit, instead of in the designer’s head.
- Patterns are being collected to *pattern cataloques*
- It is essential that a design pattern is presented in a systematic form as a semi-formal document.

- There are however several different forms for describing design patterns, the following are needed in any description.

Description of a design pattern

Essential parts:

Name	Increases design vocabulary
Intent	The purpose of the pattern
Problem	Description of the problem and its context, presumptions, example
Solution	How the pattern provides a solution to the problem in the context in which it shows up
Participants	The entities involved in the pattern
Consequences	Benefits and drawbacks of applying the design pattern. Investigates the forces at play in the pattern
Implementation	Different choices in the implementation of the design pattern, possibly language-dependent

Description of GoF patterns

- Pattern Name and Classification: Convey the essence of the pattern.
- Intent: What does the pattern do? What is its rationale? What design issue does it address?
- Also Known As: Other well-known names for the pattern
- Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. Included to help understand the abstract pattern.
- Applicability: In which situations can the pattern be applied? What are examples of poor designs that the pattern can address? How to recognize these situations

Description of GoF patterns

- Structure: A graphical representation of the classes in the pattern. E.g. use UML Class Diagrams and optionally Sequence Diagrams. Interaction diagrams to illustrate the sequence of requests between objects at runtime.
- Participants: The classes and/or objects in the design pattern and their responsibilities.
- Collaborations: How do the participants collaborate to carry out their responsibilities.
- Consequences: How does the pattern support its objectives? What are the trade-offs? What aspect of the system structure can be varied independently?

Description of GoF patterns

- Implementation: Pitfalls, hints and techniques to use when implementing the pattern. Language specific issues.
- Sample Code: Code fragments that illustrate how you might implement the pattern.
- Known Uses: Examples of the pattern found in real systems. At least two from different domains.
- Related Patterns: What patterns are closely related to this one? What are important differences? With which other patterns should this one be used?

History of Software Patterns

- 1987 Ward Cunningham and Kent Beck: ”Using Pattern Languages for Object Oriented Programming”
  - 5 pattern language for Smalltalk GUIs
  - future expectation: 100-150 patterns could cover OO programming!
- 1990 – 1993 OOPSLA workshops, ideas developed
- 1993 The Hillside Group
- 1994 Start of PLoP conferences (pattern reviews), GoF book
- 1995 the first PLoP book
- 1996 A system of Patterns, Buchmann et. al.)

..... The snowball started rolling

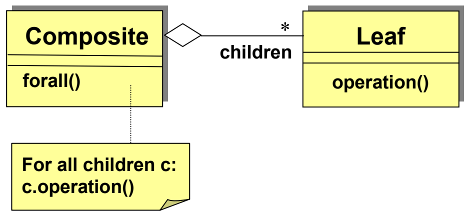
A few notes about GoF patterns

- There are different ”styles” of software patterns, the best known are the GoF patterns
  - 23 in the book, more in PLoP
- GoF patterns are
  - not very problem-specific
  - not in a pattern system
  - small, rather low level patterns
- Empasis on flexibility and reuse through decoupling of classes
- The underlying principles
  1) program to an interface, not to an implementation
  2) favor composition over class inheritance
  3) find what varies and encapsulate it

Example GoF pattern: Composite

Problem: handling structures consisting of parts

First solution: two-levels



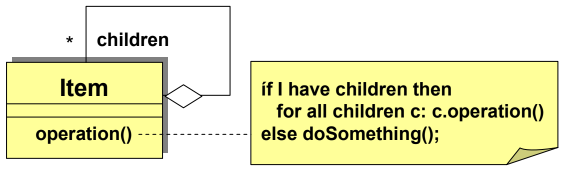
```
…
if X is Composite then
    X.forall()
else X.operation;
...
```

```
…
if X is Composite then
    Op1(…, X, …)
else Op2(…, X, …);
...
```

Symptoms:
- Composite and leafs always treated differently in code
- Difficult to extend (new kinds of leafs or composites, unrestricted depth)

Developing the composite pattern (1)

1) Unified treatment in client and unrestricted depth of parts:

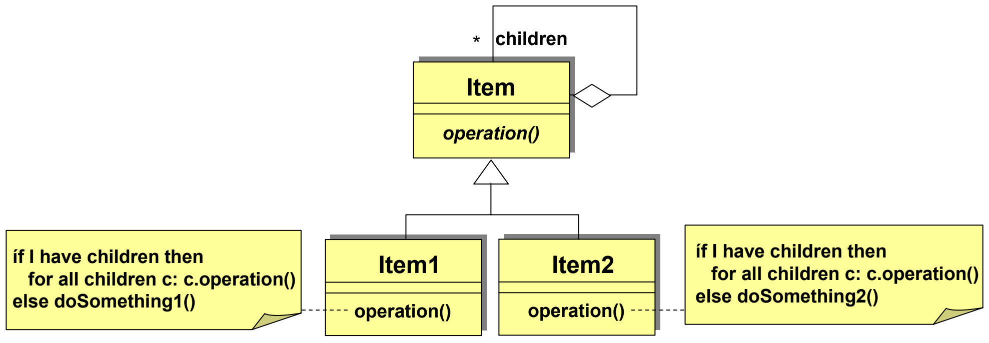


Client:

```
item.operation()
```

Developing the composite pattern (2)

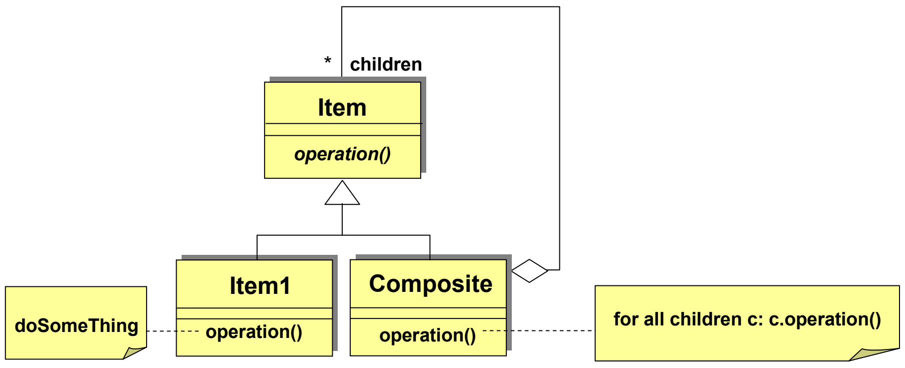
2) New types of elements:



25

Developing the composite pattern (3)

3) Separating the composite class:



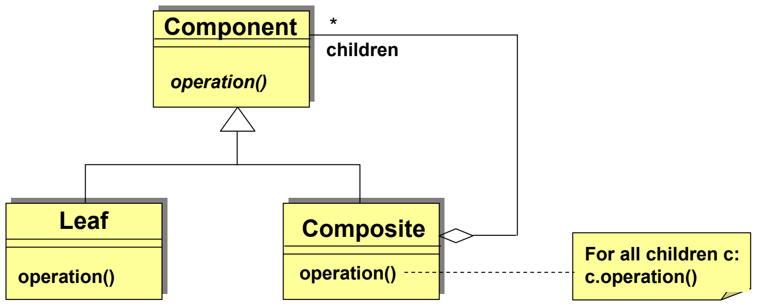
26

Composite design pattern

Name: Composite

Intent: How to organize hierarchical object structures so that the clients are not aware of the hierarchy?

Sturcture:



Composite (cont.)

Applicable when:

- you want to represent part-whole hierarchies of objects
- you want clients to be able to ignore the difference between composite and elementary objects

Consequences:

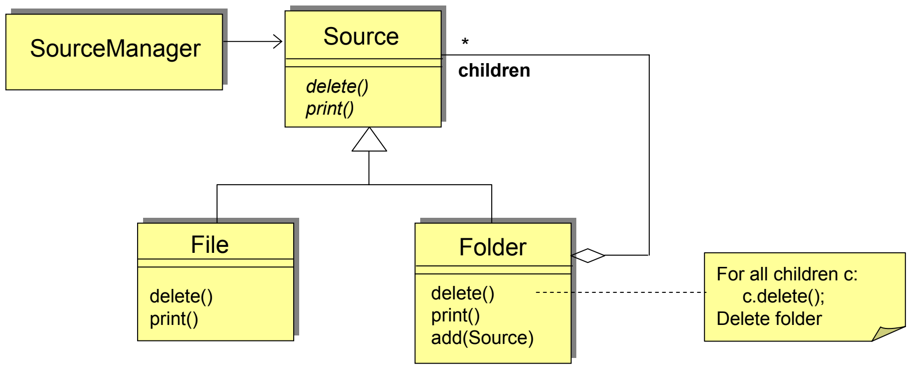
- Where ever a client handles a composite object it can handle an elementary object and vice versa
- Client code becomes simpler – avoid writing switch structures to code handling composites
- Facilitates the adding of new composite and elementary types
- May over generalize: does not support a structure that allows only certain kinds of composition structures. You can not rely on the type system to enforce those constrains, you have to use run-time checks instead.

Composite (cont.)

Implementation concerns

- Both parent and child references needed
  - define parent reference in the component class
  - it is essential to maintain the invariant between child and parent references. The easiest way to do this is ....
- Sharing components: can the same element belong to several composites? How to deal with parent references?
- Common interface for composite and leaf classes, type security, LSP and ISP?
- Where to declare the child management operations? (gives a tradeoff as an answer, transparency vs. safety)
- Are the elements ordered?
- Who creates and destroys the elements?
- Which data structure is used for storing the elements?

Example of applying Composite



Benefits of Design Patterns

- Inspiration
  - *patterns don't provide solutions, they **inspire** solutions*
  - Patterns **explicitly** capture expert knowledge and design tradeoffs and make this expertise widely available
  - ease the transition to object-oriented technology
- Patterns improve developer communication
  - pattern names form a **vocabulary**
- Help document the architecture of a system
  - enhance understanding
- Design patterns enable large-scale reuse of software architectures

Drawbacks of Design Patterns

- Patterns do not lead to direct code reuse
- Some patterns are deceptively simple
- Teams may suffer from patterns overload or pattern abuse
  - Patterns add complexity if applied where they should not
- Integrating patterns into a software development process is a human-intensive activity

Design patterns are not

- parts of a system;
  - not a piece of code
  - usually can be implemented in many ways
- general remedy to improve your system;
  - using wrong design pattern may seriously harm the system
  - at the least, you may be adding needless complexity
- God-given;
  - reject patterns, modify them to fit your needs
- without potential problems:
  - design fragmentation: more classes, more complicated dependencies
  - overkilling problems
  - excessive dynamic binding, potentional performance problem
  - ”object schitzophrenia”, splitting objects
  - implicitness, require careful documentation

Summary

- Design patterns present in a systematic way proven, good design solutions
- Design patterns are framework building blocks
- Use a design pattern only after recognizing the problem
- Design patterns concern experience, not inventions
- Design patterns are about common sense - use them with common sense

Significance to software?

From ”Pattern oriented software architecture” by Buschmann et al.

> ”Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety. ”