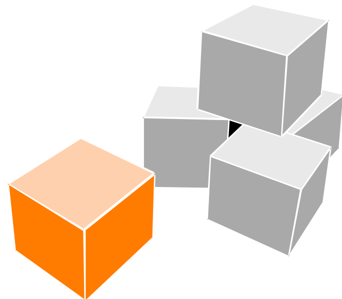


Software
Technology
Group
TU Darmstadt | FB Informatik

Software Engineering Design

4. Design Patterns An Introduction

Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

Dipl.-Ing. Michael Haupt

A Definition

A 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.

Christopher Alexander

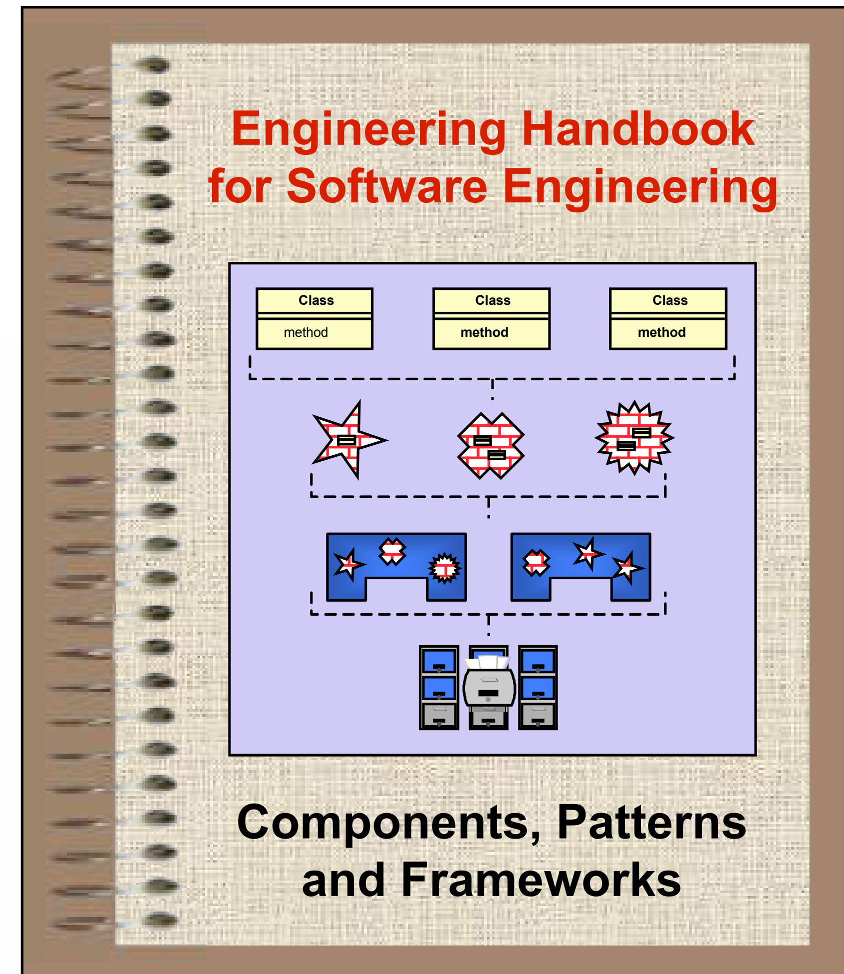
Why Patterns?

- designing reusable software is hard
- novices are overwhelmed
- experts draw from experience
- some design solutions reoccur
- understanding reoccurring solutions is beneficial in several ways
  - know when to apply
  - know how to establish it in a generic way
  - know the consequence (trade-offs)

Why Patterns?

Systematic software-development

- documenting expert knowledge
- use of generic solutions
- raising the abstraction level



What is a Pattern? - Occurrence

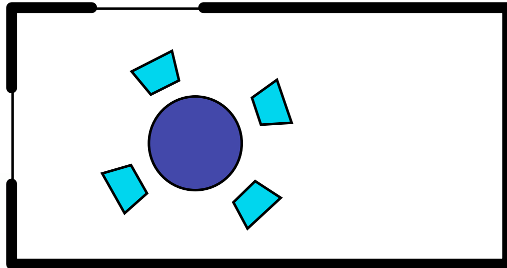
- fishing
- chess
- literature
- agriculture
- architecture
- software design

- anecdotal documentation
- from rules to expertise
- oldest reference
- wisdom vs. science
- pioneer work

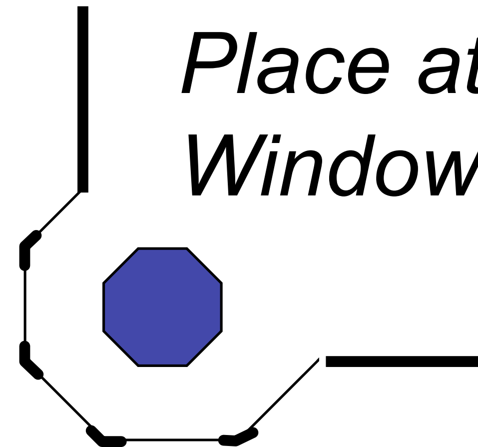
Patterns in Architecture

Christopher Alexander

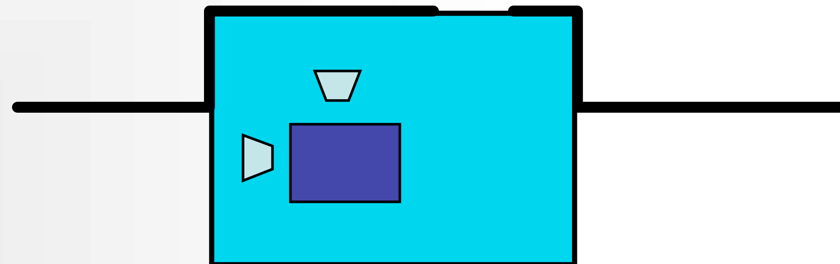
Light from two sides



Place at Window



Deep terrace



What is a pattern? - Proven

- “aggressive disregard for originality”
- rule of three:

Once is an event,
twice is an incident,
thrice it's a pattern.

Jerry Weinberg

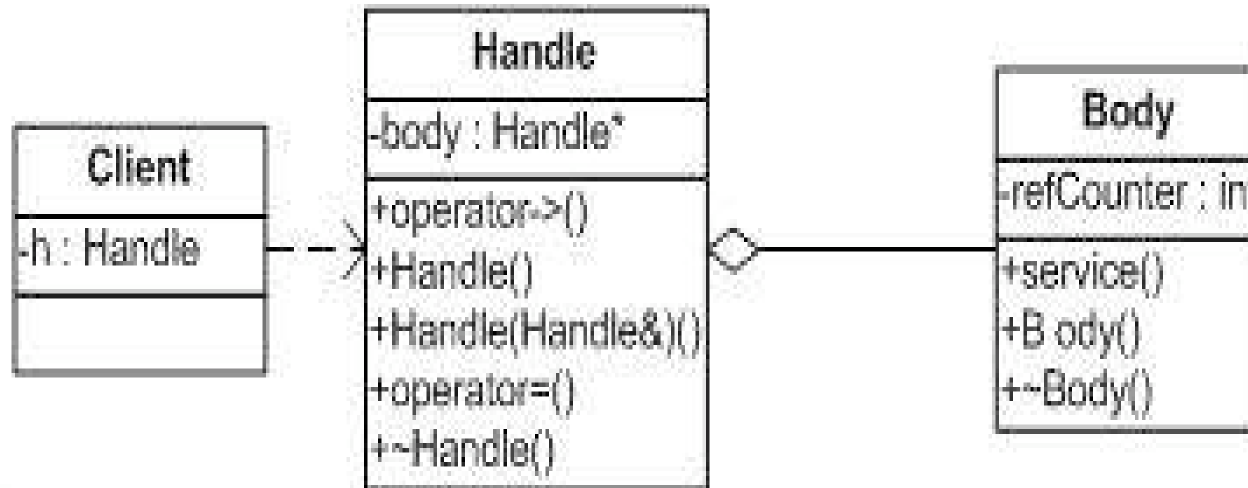
Pattern Categories in Sw. Design

- architectural patterns
  - Buschmann et al POSA 1, Garlan/Shaw ‘96
- design patterns
  - GoF ‘95
- idioms
  - Coplien ‘91
- analysis patterns
  - Fowler ‘96
- organisational patterns
  - http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?OrganizationalPatterns
- anti-patterns
- ...

Idioms

- low-level pattern specific to a programming language
- describes how to implement particular aspects of components or their relationships using the features of the given language
- example:
  - indented control flow
  - while(item = item->next()) { ... }
  - *counted pointer* idiom

Counted Pointer Idiom




```
class Body {
public:  void service();
private: friend class Handle;
         Body(/*...*/) { }
         ~Body() { }
         int refCounter;
};
```

Counted Pointer Idiom

```
class Handle {
public:  Handle(/*...*/) {
            body = new Body(/*...*/);
            body->refCounter = 1;
         }
         Handle(const Handle& h) {
            body = h.body; body->refCounter++;
         }
         ~Handle() {
            if(--body->refCounter <= 0) delete body;
         }
         Body* operator->() { return body; }
         Handle &operator=(const Handle &h) {
            h.body->refCounter++;
            if(--body->refCounter <= 0) delete body;
            body = h.body;
            return *this;
         }
private: Body* body;
};
```

Counted Pointer Idiom

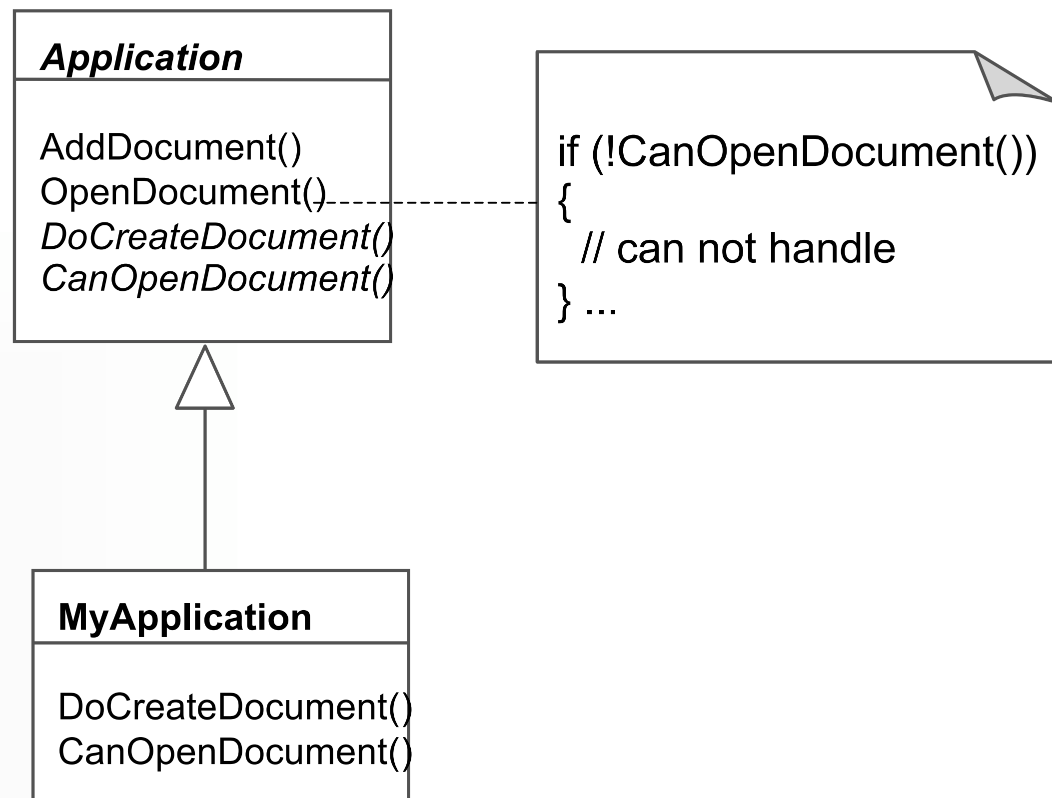
```
// Usage:

...

{
  Handle h(/*some parameters*/);
  {
    Handle g(h);
    h->service(); g->service();
  } // g goes out of scope, decreases rc

  ...

   ...
   h->service();
} // body instance is deleted automatically
```

Design Patterns

- a design pattern describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context
- medium-scale patterns
  - smaller than architectural patterns
  - no effect on fundamental structure of a software system but may have strong influence on subsystems
  - relatively independent of a particular programming language

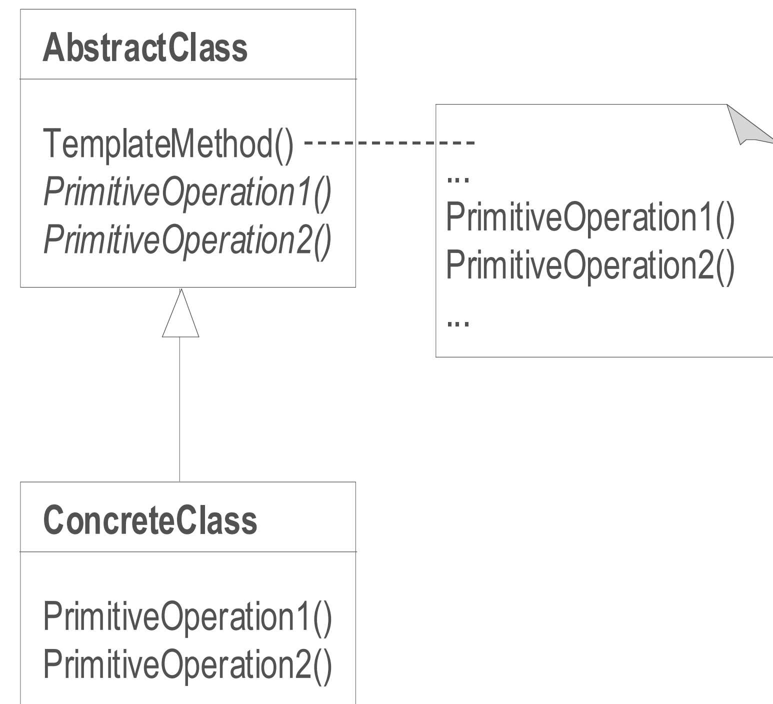
Example: Template Method

- often found in frameworks
- abstract classes
  - define interfaces
  - contain code



Example: Template Method

- avoid code duplication
- separate invariant and variant parts
- separate policies from mechanisms
- control subclass extensions



Example: Composite - Intent

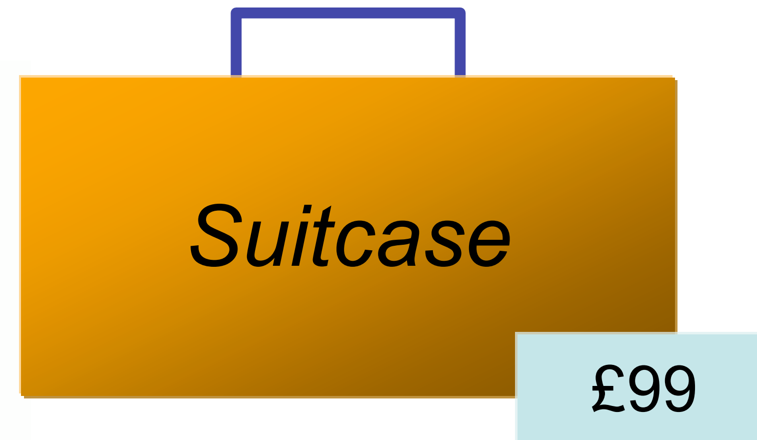
Compose objects into tree structures to represent part-whole hierarchies.

Composite lets clients treat individual objects and compositions of objects uniformly.

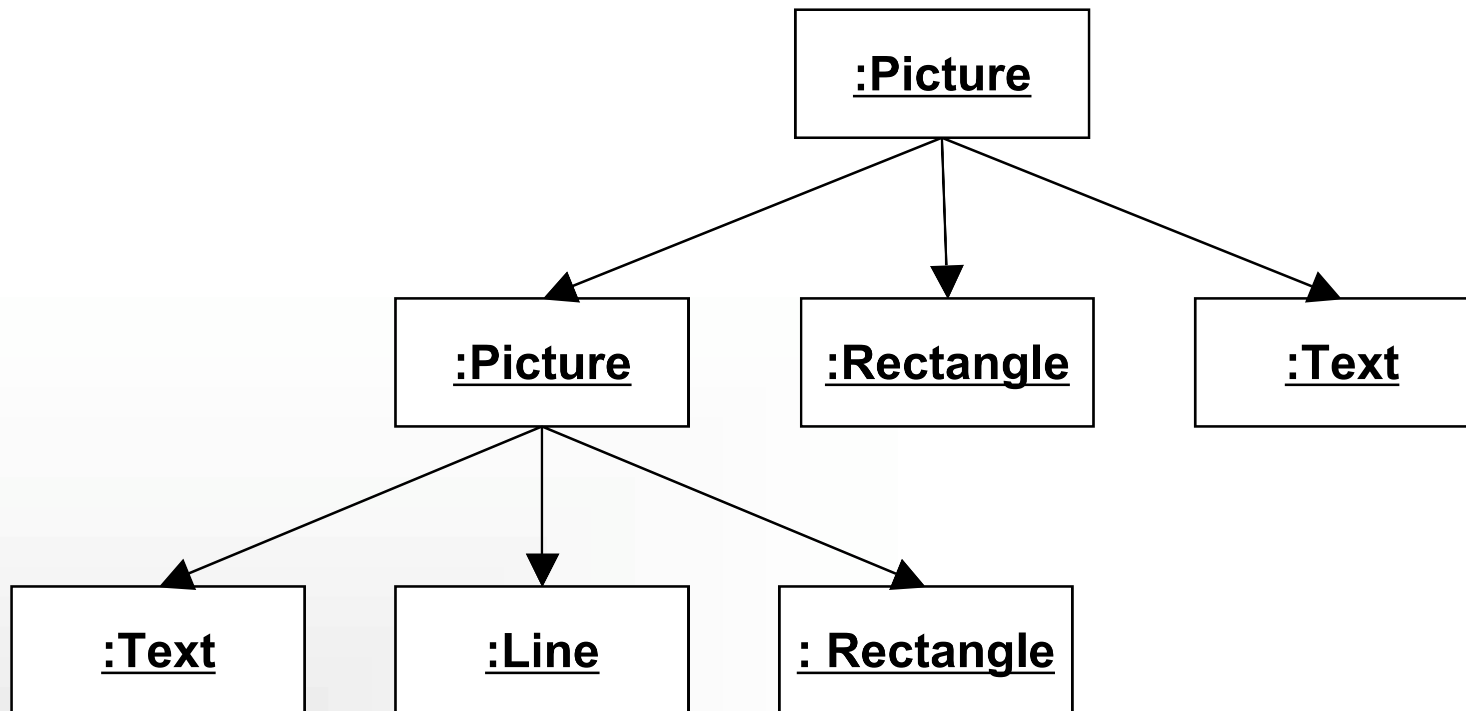
Composite: Motivation

Drawing editor

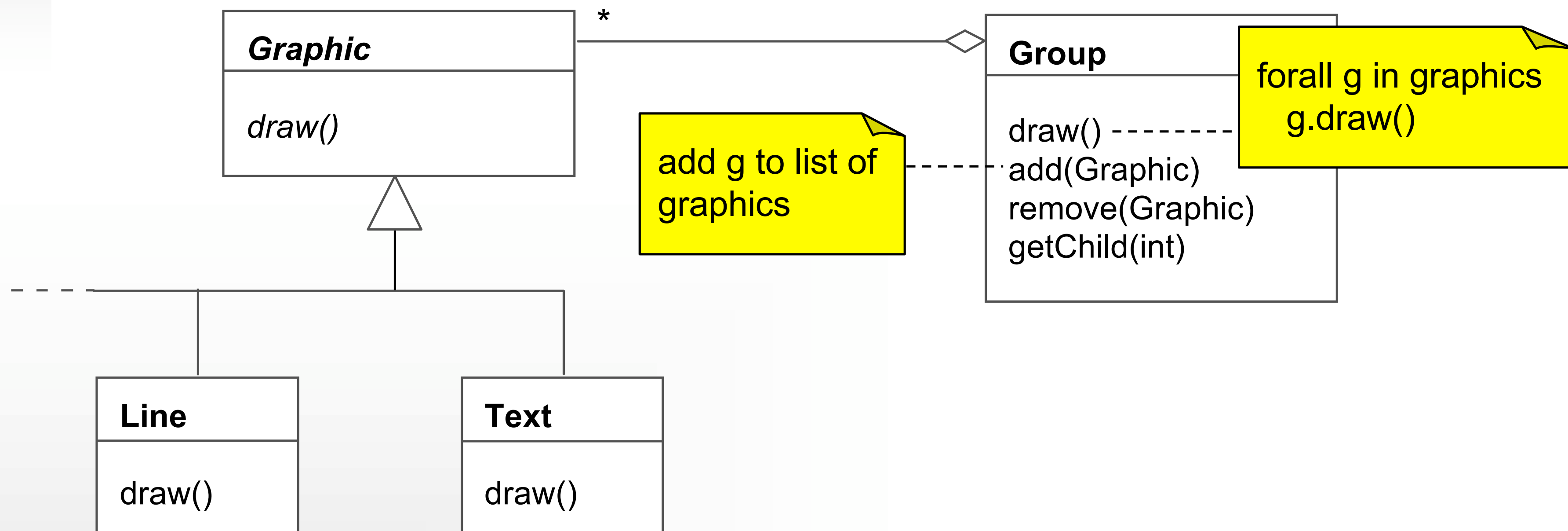
- pictures contain elements
- elements can be grouped
- groups can contain other groups



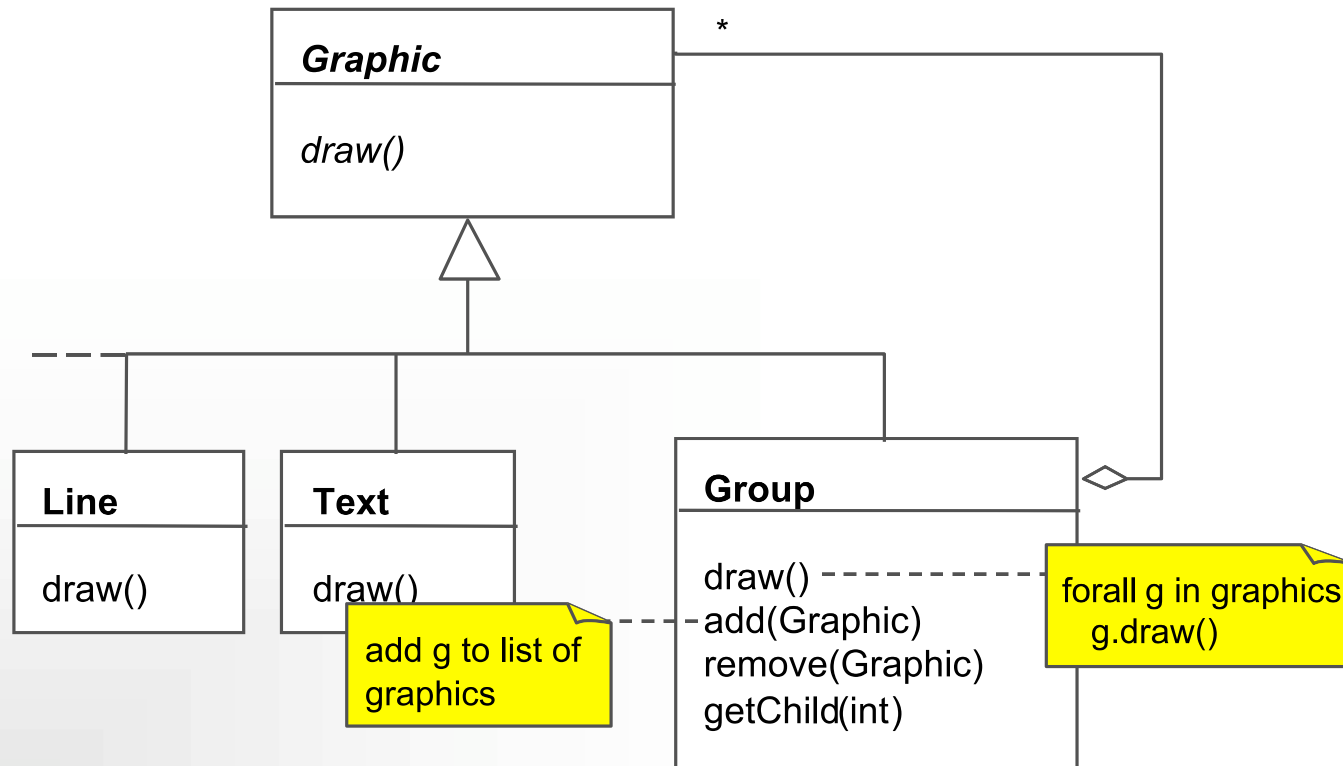
Composite: Motivation



Composite: Motivation



Composite: Motivation

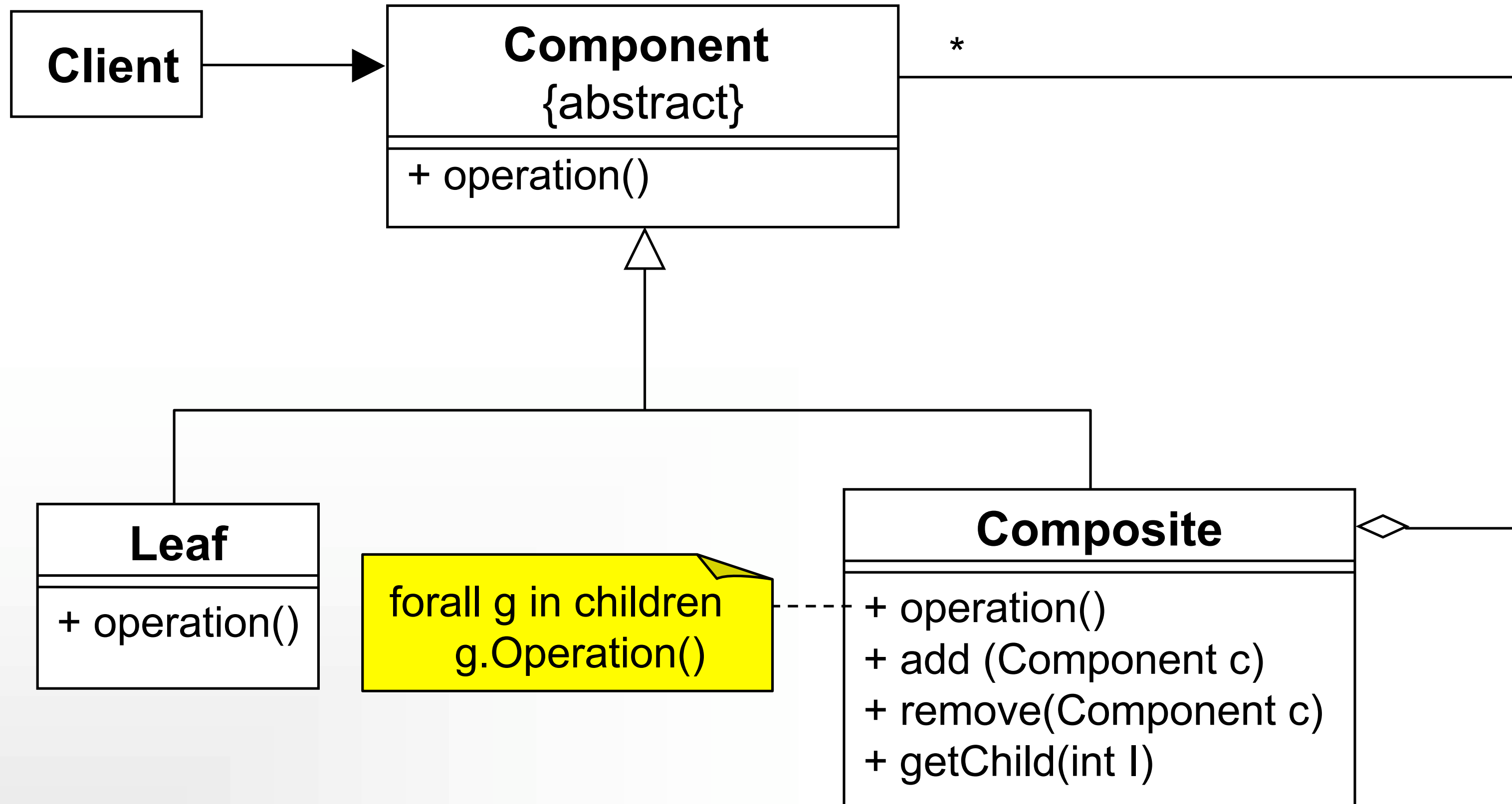


Composite: Applicability

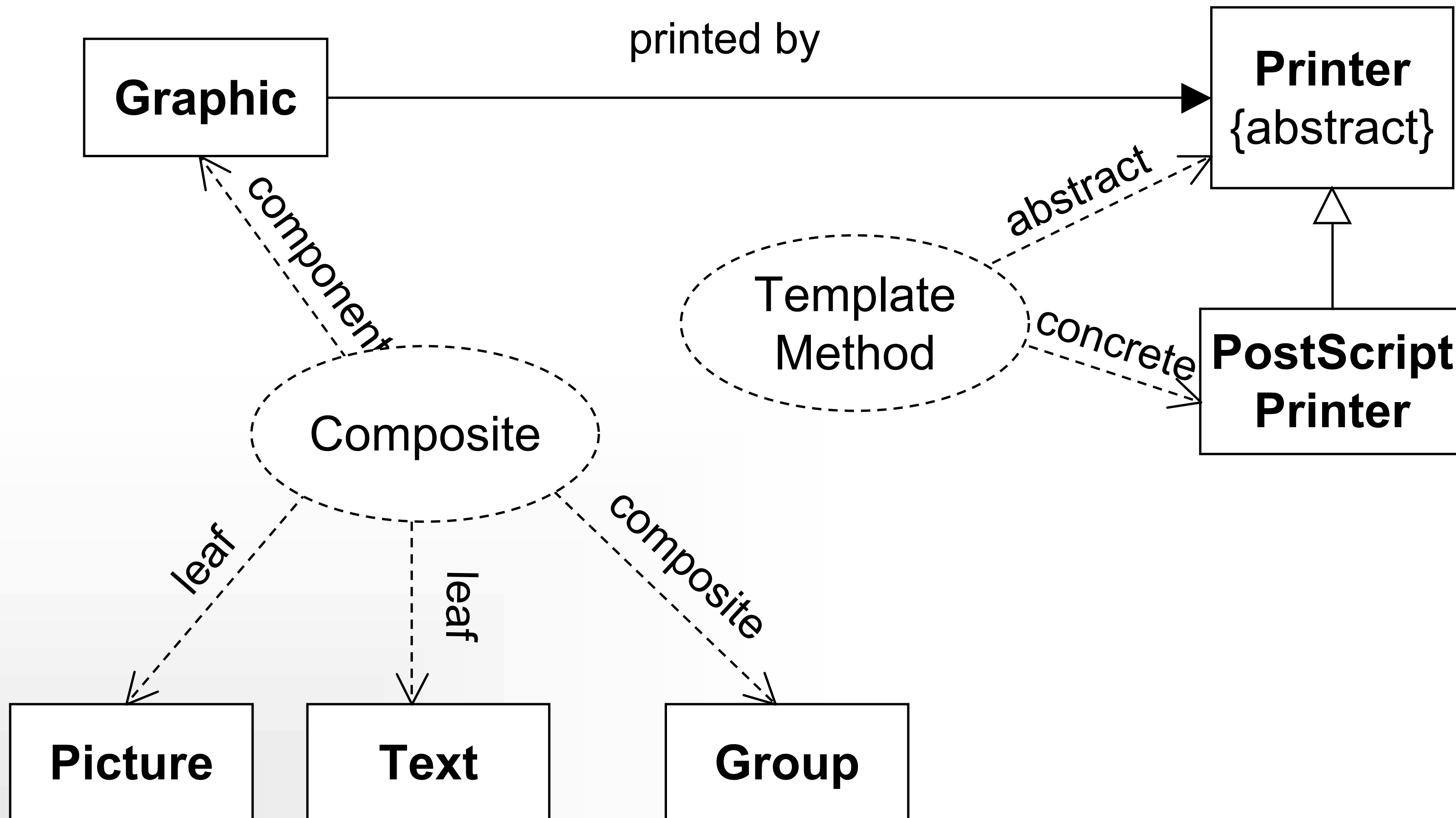
Use Composite when...

- you want to represent part-whole hierarchies of objects
- clients should be able to ignore the difference between individual and composed objects

Composite: Structure



Documenting Patterns



Documenting Patterns

A pattern is a collaboration

- object diagram for the context
  - which *roles* are involved?
- sequence diagram for interaction
  - what is the order of method calls?
- collaboration diagram for context & interaction

Composite: Participants

- component
  - declares interface / default behaviour
- leaf
  - atomic elements / primitive behaviour
- composite
  - stores children / composite behaviour
- client
  - accesses via Component interface

Composite: Collaborations

- clients interact with objects through the Component interface
- leaf recipients react directly
- composites forward requests to their children, possibly adding before/after operations

Composite: Consequences

- primitive objects can be recursively composed
- clients can treat composites and primitives uniformly
- new components can easily be added

- design may become overly general

Composite: Implementation Tradeoffs

- explicit parent references?
  - traversal easier, need to maintain consistency
- sharing components
  - reduce memory, incompatible with parent references
- maximising the Component interface
  - should have all common methods
    - not all methods may make sense for every component
- placing child management operations
  - at the root (Component)
    - less safety
  - in Composite
    - safe, but less uniform

Composite: Sample Code

```
abstract class Equipment {
  String       name() { return name; }
  abstract int  price();

  abstract void add(Equipment eq);
  abstract void remove(Equipment eq);

  private String name;
}
```

Composite: Sample Code

```
class FloppyDisk extends Equipment {
  int price()
  {
    return 50;
  }

  ...
}
```

Composite: Sample Code

```
class CompositeEquipment extends Equipment {
 int price()   { ...
  for (i=0; i<equipment.length; i++)
       total += equipment[i].price();
 }

 void add(Equipment eq)    {...};
 void remove(Equipment eq) {...};

 private Equipment[] equipment;
}
```

Composite: Sample Code

```
chassis.add(new FloppyDisk("3.5in Floppy"));

Bus bus = new Bus("PCI Bus");
bus.add(new Card("ATM Network card"));

chassis.add(bus);
cabinet.add(chassis);

System.out.println("Total price: " +cabinet.price() );
```

Composite: Known Uses

- view class of Model/View/Controller
- application frameworks & toolkits
  - ET++, 1988
  - Graphics, 1988
  - Glyphs, 1990
  - InterViews, 1992
  - Java AWT, Swing
  - Java files
  - abstract syntax trees in compilers

Composite: Related Patterns

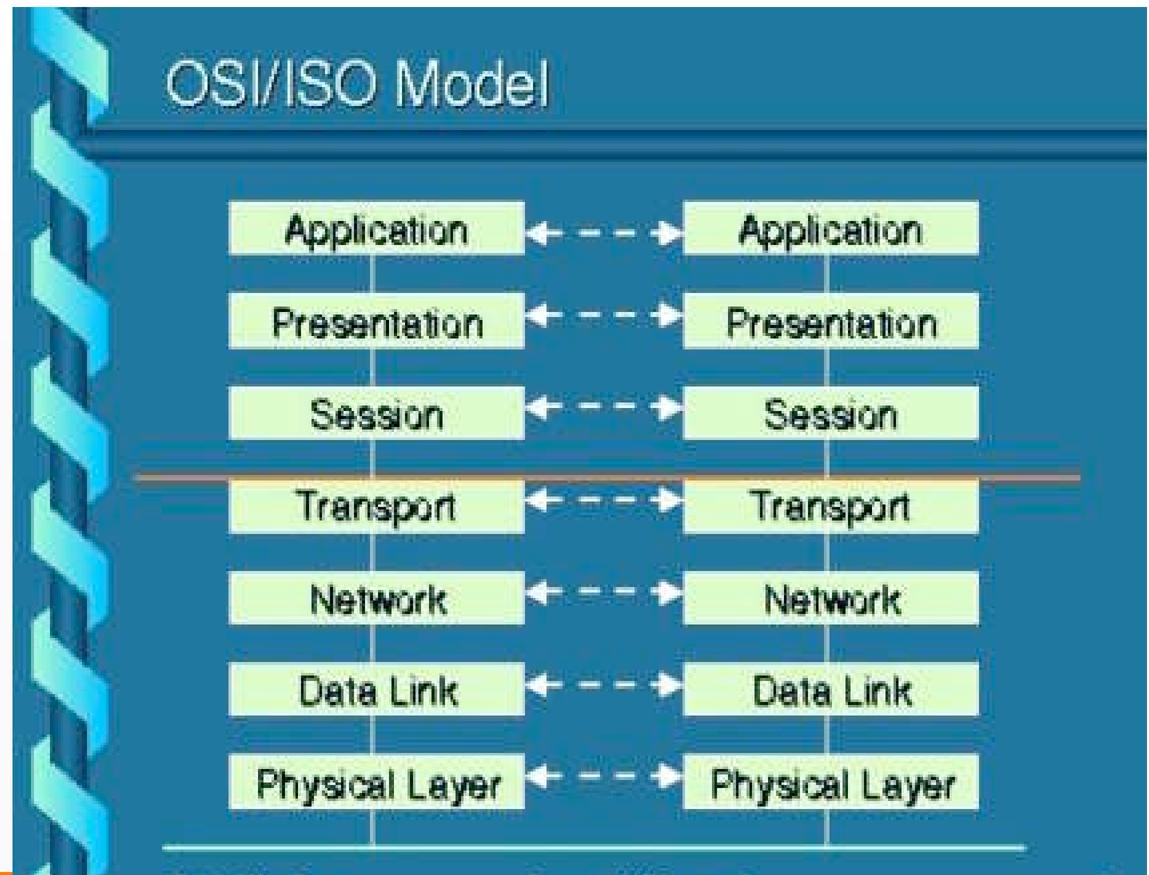
- Iterator
  - traverse Composite
- Visitor
  - localise operations on tree
- Chain of Responsibility
  - use chain of components to parents
- Flyweight
  - sharing of components
- Decorator
  - use together with Composite

Architectural Patterns

- an architectural pattern expresses a fundamental structural organisation schema for software systems
  - provides set of predefined subsystems and specifies their responsibilites
  - includes rules and guidelines for organising the relationship between them
- are templates for concrete software architectures
  - hence fundamental design decision
- examples: MVC, layering, blackboard architecture

Example: Layers

- structure applications by decomposing them into groups of subtasks whereby each group of subtasks is at a particular level of abstraction



Layers

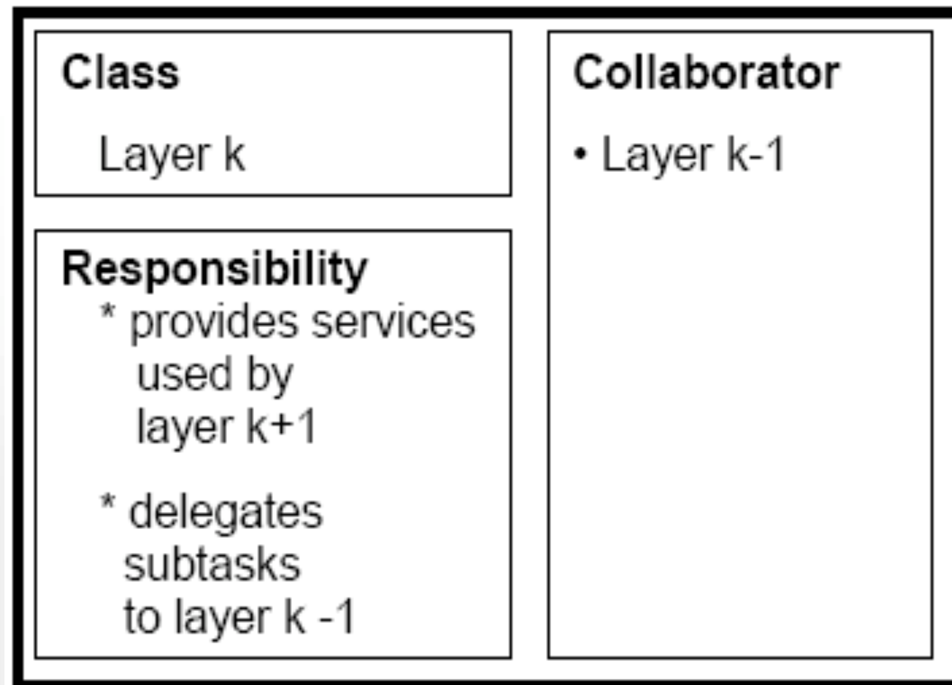
- problem
  - system is a mix of low- and high-level issues, the latter rely on the former
  - typical pattern of communication flow: requests moving from high to low-level, answers/notifications traveling in opposite direction
  - portability is desired
  - system needs horizontal structuring in addition to vertical structuring

Layers

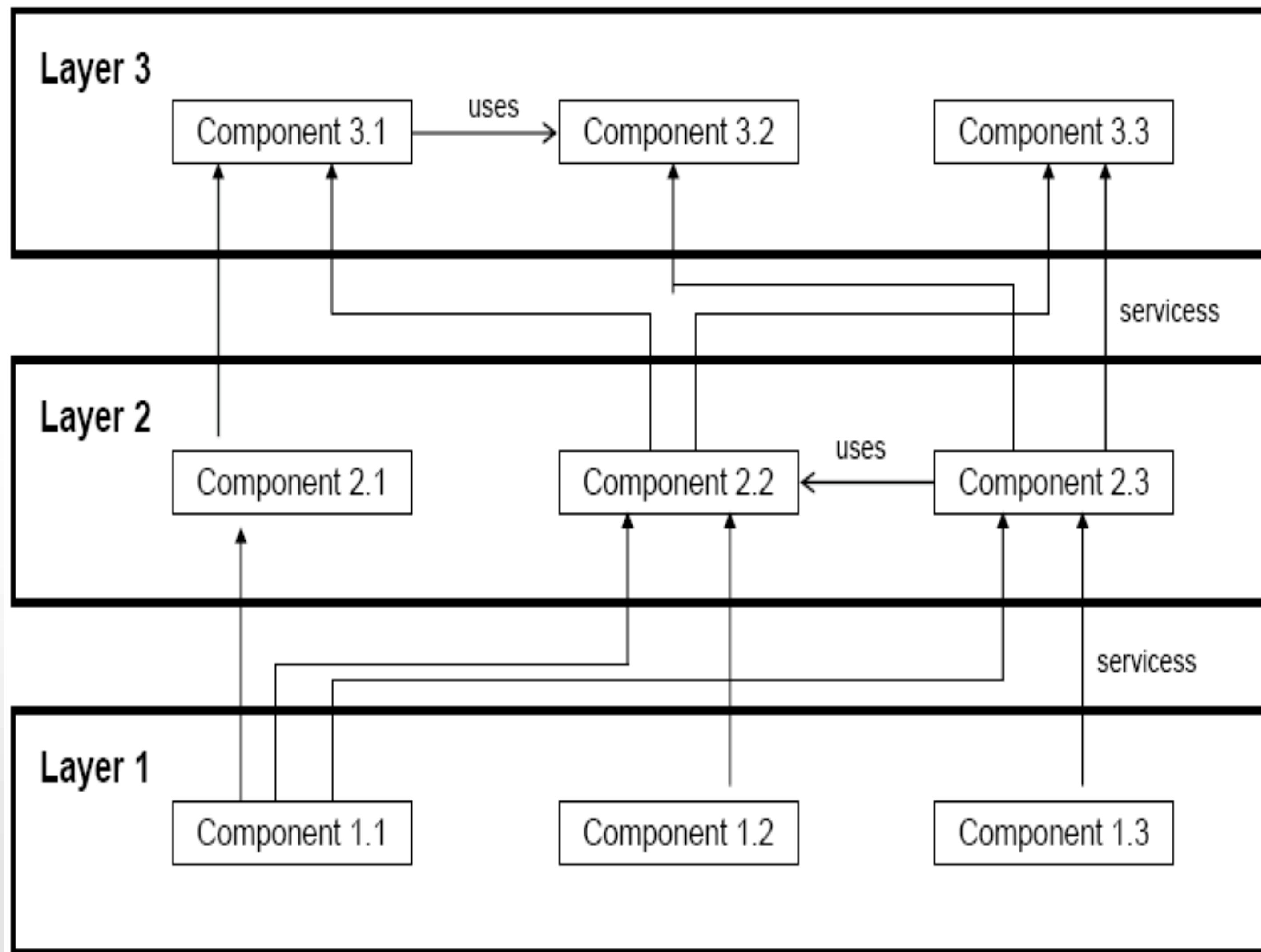
- forces
  - source changes should not ripple through the system, should be confined to one component
  - interfaces should be stable
  - parts of the system should be exchangeable
    - replace components without affecting the rest of the system
  - crossing component boundaries may impede performance
  - system is built by a team, work has to be divided along clear boundaries

Layers

- solution:
  – structure a system into an appropriate number of layers, such that each lower level layer provides services to its immediate upper level neighbour



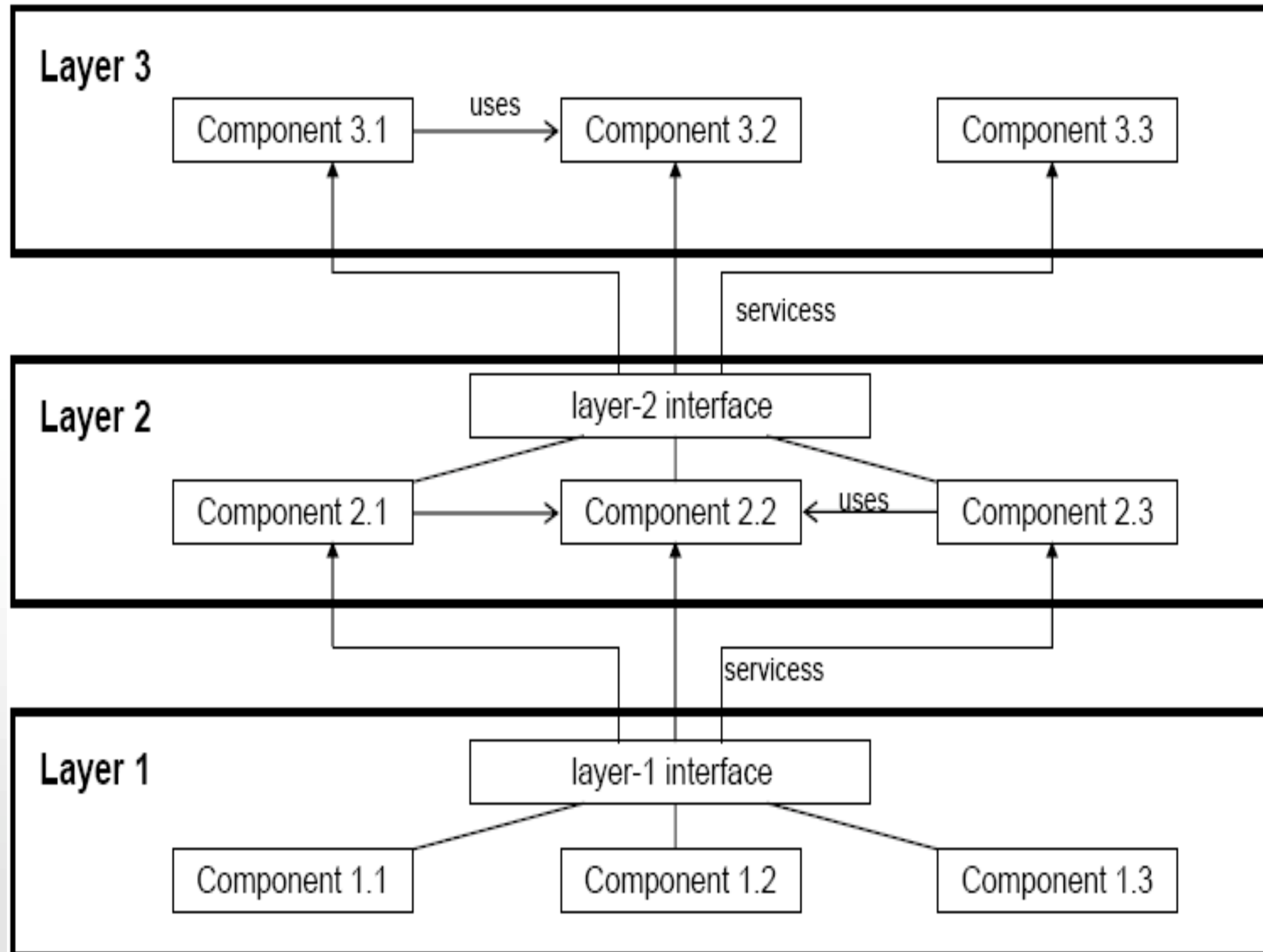
Layers



Layers: Implementation

- define abstraction criterion for layering
  - “conceptual distance from the platform”
- determine number of abstraction levels
- name layers, assign tasks to each of them
- specify the services
- refine the layering (iterate over steps 1-4)
- specify an interface for each layer (protocols; layer k-1 should be a black box for layer k)
- structure individual layers
- specify the communication between layers
  - push or pull?
- decouple adjacent layers (lower layer should not be aware of its users)
  - use callbacks, commands, events etc.
- design an error-handling strategy

Layers with Interfaces



Layers: Variants

- relaxed layered system
  - layer k communicates with layer 1...k-1
  - adds flexibility and performance
  - loss of maintainability
- layering through inheritance
  - class of layer k is subclass of layer k-1 class
  - advantage: higher layers can modify lower-layer services
  - disadvantage: higher coupling between layers
    - fragile base class problem

Layers

- known uses
  - network protocol stack
  - virtual machines
    - source-code, byte-code, intermediate code, machine code
  - information systems
    - 3-tier architecture
  - operation systems, e.g., Windows XP
    - hardware, hardware abstraction layer, kernel, resource management layer, system services

Layers

- benefits
  - reuse of layers
  - support for standardisation
  - dependencies kept local
  - exchangeability
    - e.g., install new graphics driver
- liabilities
  - cascades of changing behaviour
  - lower efficiency
  - unnecessary work
    - e.g., unnecessary error checking, encryption
  - difficulty of establishing correct granularity of layers
    - too many layers: unnecessary complexity, overhead
    - too few layers: does not fully exploit pattern’s potential

Design Pattern Formats

Gang of Four

- Name
- Motivation
- Applicability
- Structure
- Participants
- Collaboration
- Consequences
- Related Patterns

Alexandrian

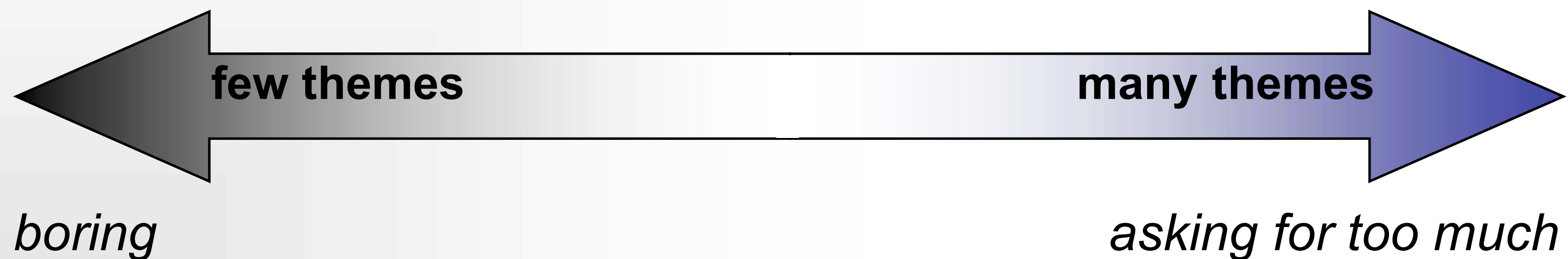
- Name
- Context
- Forces
- Solution
- Trade Offs
- Related Patterns

Design Forces

Context composing a song

Problem keep the attention of the listener without asking for too much

Forces the number of themes is critical



boring *asking for too much*

Pattern Literature

- various description approaches
  - “forces” are a central aspect
  - context description is important
- building blocks of an engineering discipline
  - capturing expertise
  - solutions are generic
  - raising the abstraction level: *"pattern language"*

What is a pattern? - Promise

- reusable software
- reuse of design
- documentation
- communication
- teaching
- language design

- patterns foster reusability
- rather than code
- information chunks
- design vocabulary
- passing on culture
- high level languages

Pattern Languages

no traffic city core

Pattern Languages

no traffic city core

satellite centers

public transport

Pattern Languages

no traffic city core

satellite centers

public transport

cheap housing

student ticket

Conclusion

If you reuse code,
you'll save a load,
but if you reuse design,
your future will shine.

Ralph E. Johnson

References

- Design Patterns, E. Gamma et al.,
- Pattern-Oriented Software Architecture, Buschmann et. al.
- Pattern Languages of Program Design,
  PLoP & EuroPLoP conferences
- Patterns Home Page & Mailing List
  http://hillside.net/patterns
  patterns-discussion@cs.uiuc.edu