Génie Logiciel et Gestion de Projets

Patterns

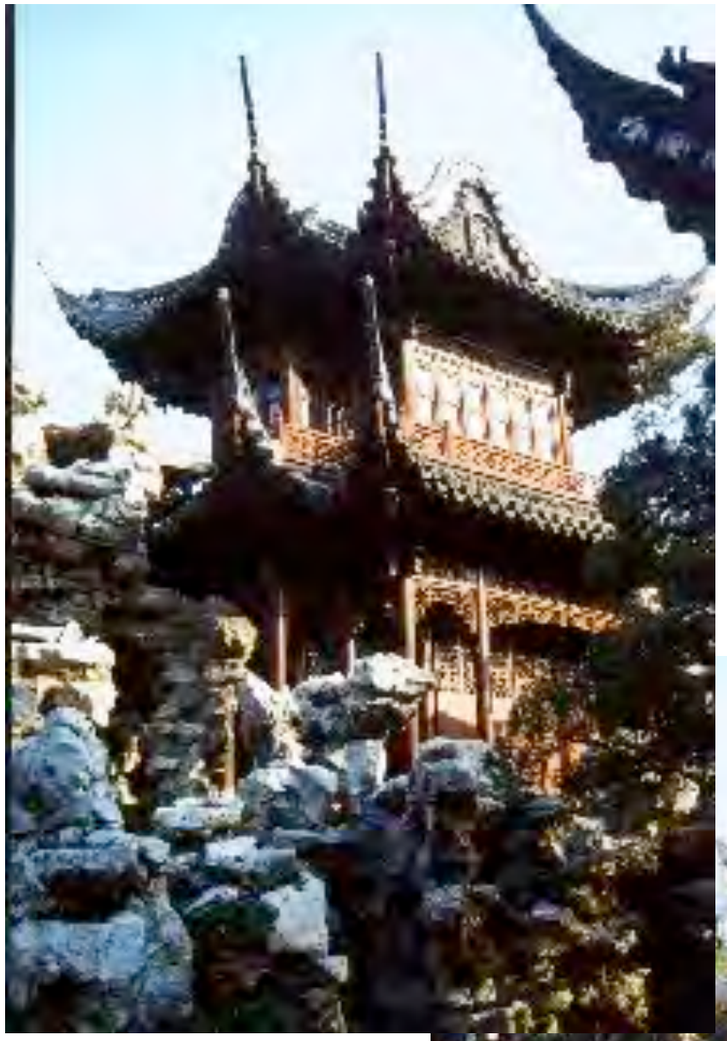
Bit of history...

- Christoffer Alexander
  - "The Timeless Way of Building", Christoffer Alexander, Oxford University Press, 1979, ISBN 0195024028
    - Structure of the book is magnificent
      - There is a copy in the library of DI
- More advanced than what is used in computer science
  - only the simple parts got mainstream

Alexander's patterns

- "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 doing it the same way twice"
- Alexander uses this as part of the solution to capture the "quality without a name"

Illustrating Recurring Patterns...



Essential Elements in a Pattern

- Pattern name
  - Increase of design vocabulary
- Problem description
  - When to apply it, in what context to use it
- Solution description (generic !)
  - The elements that make up the design, their relationships, responsibilities, and collaborations
- Consequences
  - Results and trade-offs of applying the pattern

Roadmap

- Patterns in different contexts:
  - design patterns
  - analysis patterns
  - architectural patterns
  - reengineering patterns

Alert!

- Do not overreact seeing all these patterns!
- Do not apply too many patterns!
- Look at the trade-offs!
- Most patterns makes systems more complex!
  - but address a certain need.
- As always: do good modeling.
  - then see whether patterns can help,
  - and where.

Design Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

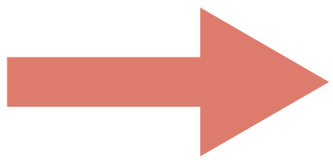
Design Patterns

- A Design Pattern is a *pattern* that captures a solution to a recurring *design* problem
  - It is not a pattern for implementation problems
  - It is not a ready-made solution that has to be applied
    - cfr Rational Modeler, where patterns are available as preconstructed class diagrams, even though in literature the class diagrams are to illustrate the pattern!

Design Patterns

Example:

"We are implementing a drawing application. The application allows the user to draw several kinds of figures (circles, squares, lines, polymorphs, bezier splines). It also allows to group these figures (and ungroup them later). Groups can then be moved around and are treated like any other figure."



Look at Composite Design Pattern

Patterns in Software Design

- A design pattern is a description of communicating objects and classes that are customized to solve a general design problem in a particular context.

Pattern structure

- A design pattern is a kind of blueprint
- Consists of different parts
  - All of these parts make up the pattern!
  - When we talk about the pattern we therefore mean all of these parts together
    - not only the class diagram...

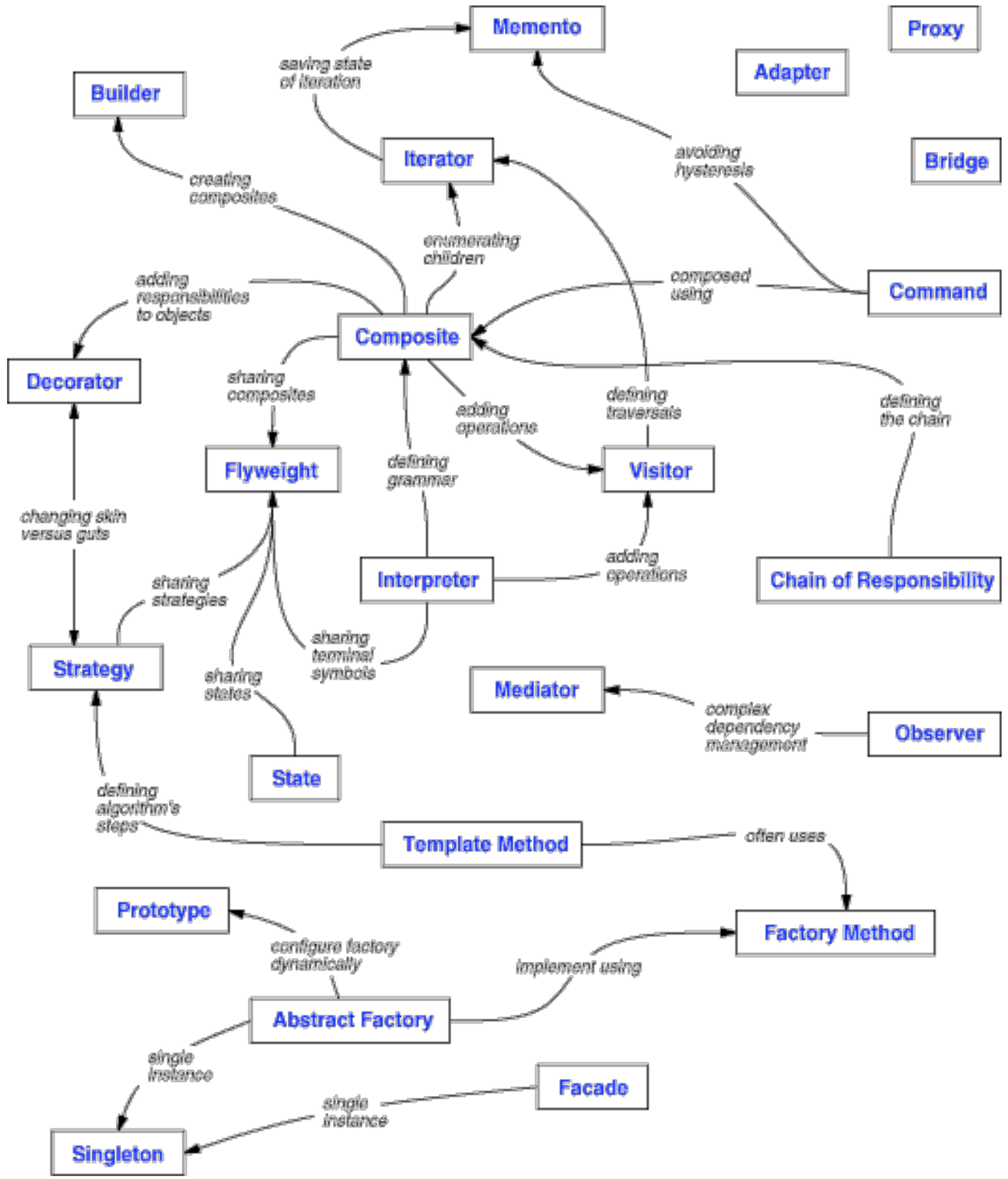
Why Patterns?

- Smart
  - Elegant solutions that a novice would not think of
- Generic
  - Independent on specific system type, language
    - Allthough slightly biased towards C++
- Well-proven
  - Successfully tested in several systems
- Simple
  - Combine them for more complex solutions

Categories of Design Patterns

- Creational Patterns
  - Instantiation and configuration of classes and objects
- Structural Patterns
  - usage of classes and objects in larger structures,
  - separation of interfaces and implementation.
- Behavioural Patterns
  - Algorithms and division of responsibility

Design Pattern Relationships



Design Patterns:
Structural Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

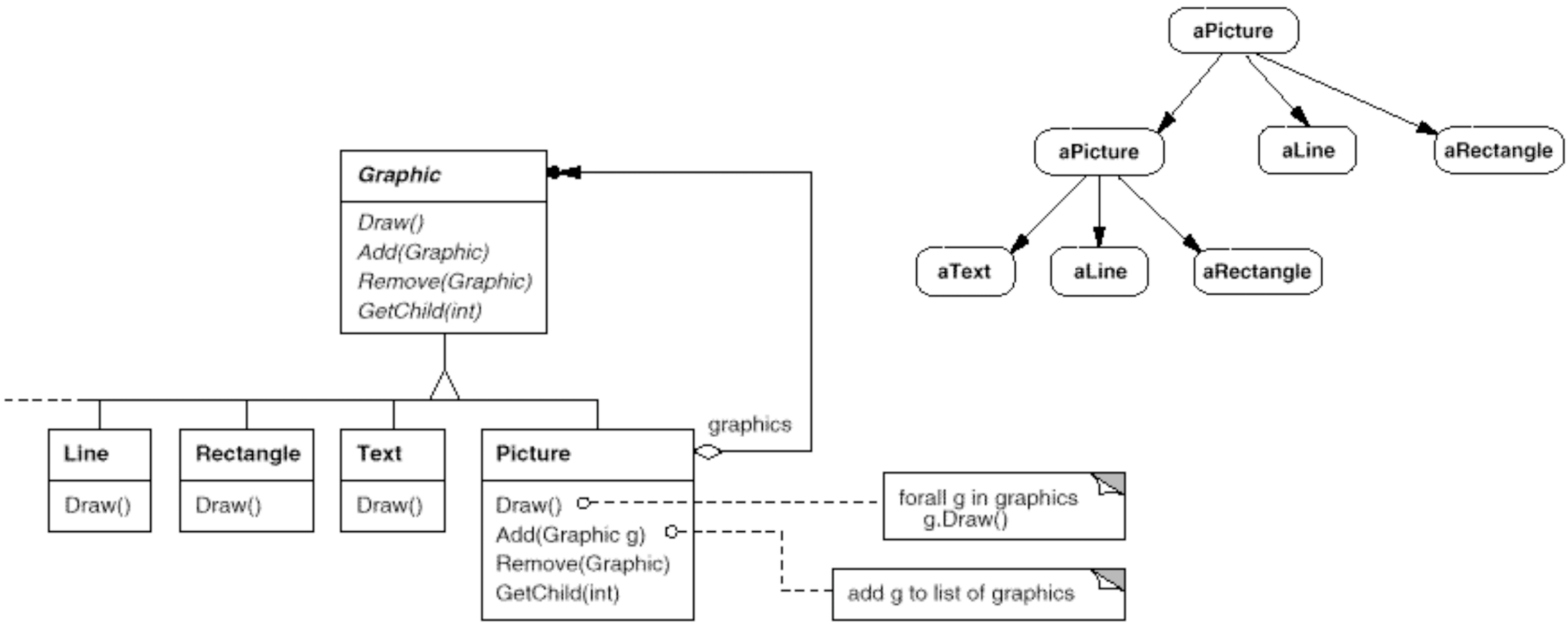
Structural Patterns

- 7 patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

Composite Pattern

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

The Composite Pattern Structure



When to use a Composite?

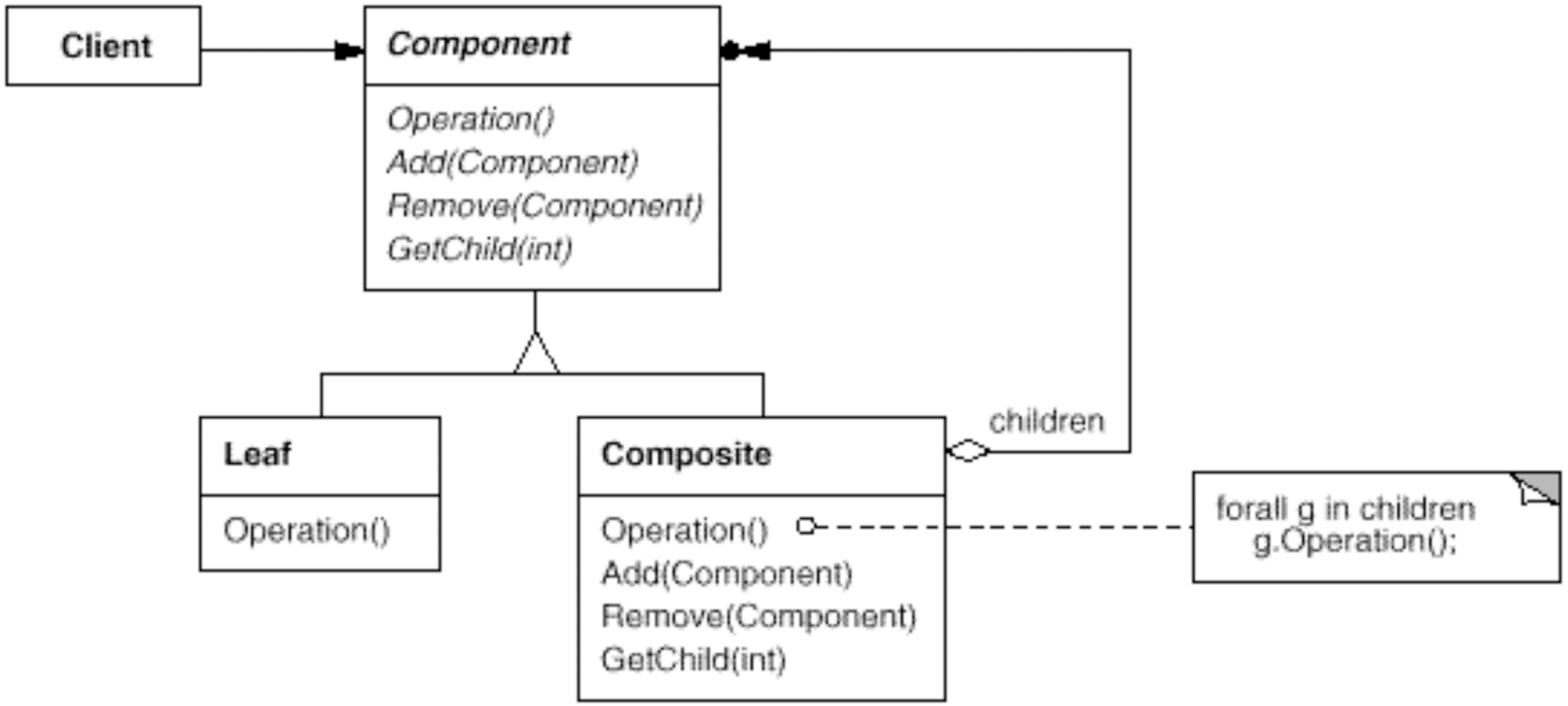
- you want to represent part-whole hierarchies of objects
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Pattern: Participants

- Component:
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all objects
  - declares an interface for accessing and managing child components
  - (optional) defines/implements an interface for accessing a component's parent
- Leaf:
  - defines behavior for primitive objects in the composition

Composite Pattern: Participants

- Composite:
  - defines behavior for components having children
  - stores child components
  - implements child access and management operations in the component interface
- Client:
  - manipulates objects in the composition through the component interface


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

Composite Pattern: Collaboration

- Clients use the Component class interface to interact with objects in the composition
- If the recipient is a Leaf, the request is handled directly
- If the recipient is a Composite the request is usually forwarded to child components, some additional operations before and/or after the forwarding can happen

Composite Pattern: Consequences

- + Makes the Client simple:
  clients can treat composite structures and individual objects uniformly, clients normally don't know and should not care whether they are dealing with a leaf or a composite
- + Makes it easier to add new types of components:
  client code works automatically with newly defined Composite or Leaf subclasses
- - Can make a design overly general:
  the disadvantage of making it easy to add new components is that it is difficult to restrict the components of a composite, sometimes you want a composite to have only certain types of children, with the Composite Patterns you cannot rely on the type system to enforce this for you, you have to implement and use run-time checks

Advanced Composite Discussions

- When looking more closely at the composite and its implementation, we can discuss a number of things in more detail:
  - Who is in charge of child management?
  - Do we need explicit parent references?

Child Management

- Where to define the child management operations?
  - Defining these operations at the root of the class hierarchy gives transparency because you can treat all components uniformly. It costs safety because any attempt to remove or add object from leaves give compilation errors. But loss of transparency because leaves and composites have different interfaces.
  - this must be captured in a default implementation: do nothing or throw error ???

Child Management

- The datastructure for storing children
  - a variety of datastructures is available: lists, trees, arrays, hash tables,
  - the choice depends on efficiency; alternatively each child can be kept in a separate instance variable,
  - all access and management operations must then be implemented on each Composite subclass
- Child ordering:
  - when child order is an issue the child access and management interface must be designed carefully to manage this sequence

Parent References

- Maintaining references from child components to their parents:
  - simplify traversal and management of composite structures;
  - are best defined in the Component class;
  - it is essential to maintain the invariant that all children of a composite have as their parent the composite that in turn has them as children.

Design Patterns:
Behavioural Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

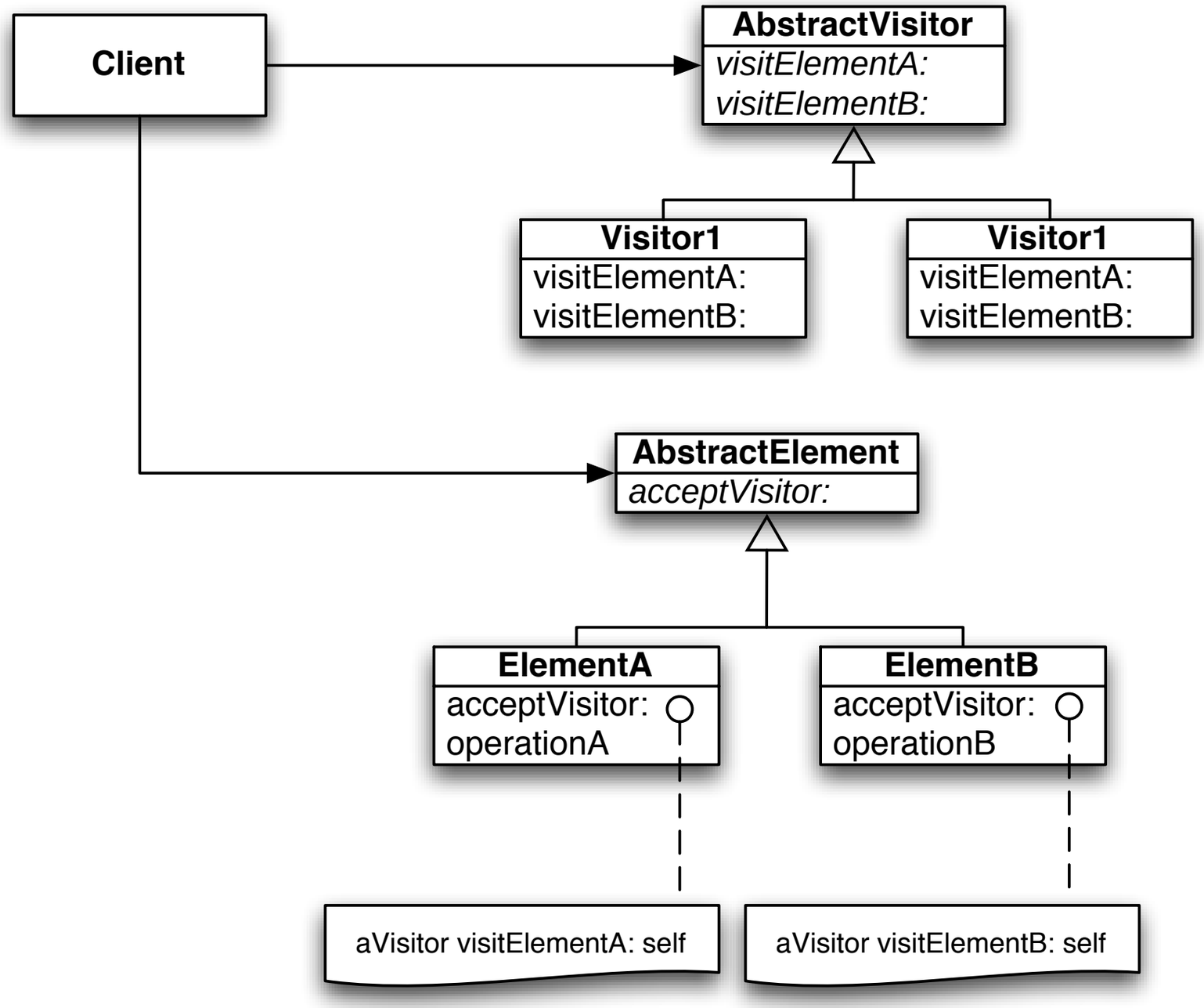
Behavioural Patterns

- 11 patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Visitor Pattern

- Intent: Represent an operation to be performed on the elements of an object structure in a class separate from the elements themselves. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

The Visitor Pattern: Structure



Applying the Visitor

- When reading the intent and seeing the structure we can use the visitor in our design...
- So all our problems are solved, no?
- Well...
  - when to use a visitor
  - control over item traversal
  - choosing the granularity of visitor methods
  - implementation tricks

When to use a Visitor

- Use a Visitor:
  - when the operations on items change a lot.
- Do not use a visitor:
  - when the items you want to visit change a lot.
- Question: But how do we know what to choose up-front?

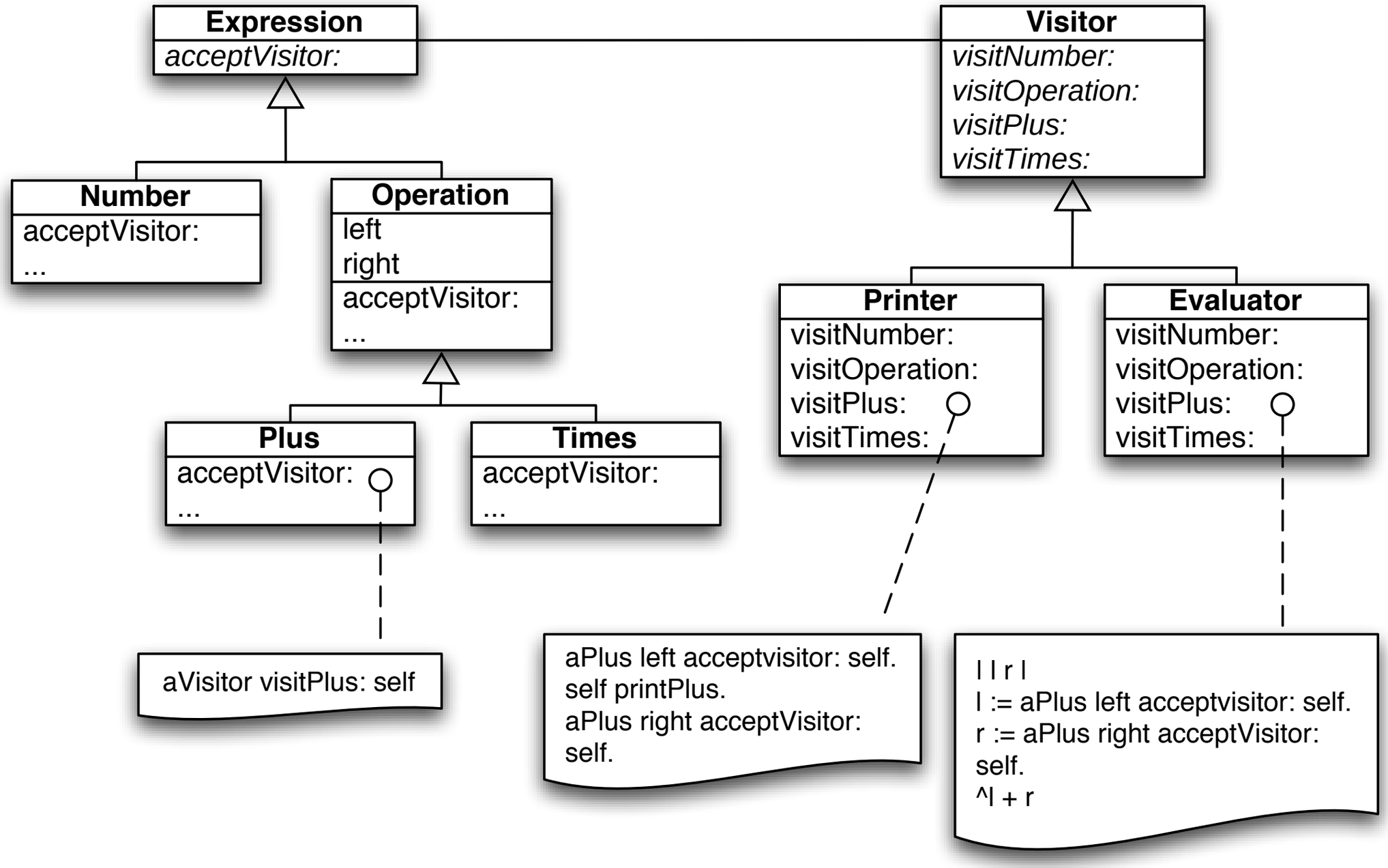
Advanced Visitor Discussions

- When looking more closely at the visitor and its implementation, we can discuss a number of things in more detail:
  - Who controls the traversal?
  - Implementation tricks

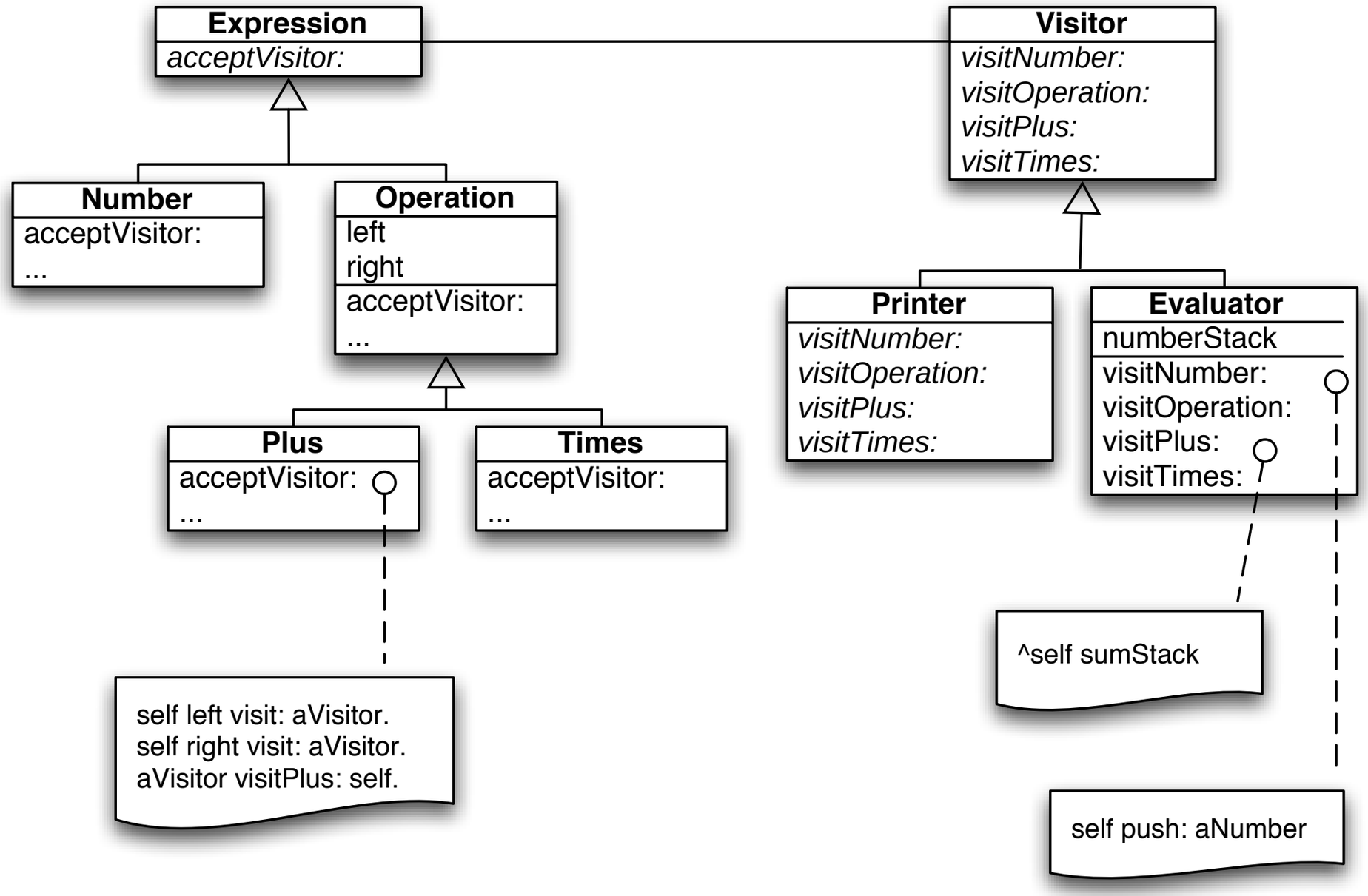
Controlling the traversal

- Somewhere in the visitor, items are traversed.
- Different places where the traversal can be implemented:
  - in the visitor
  - on the items hierarchy

Traversal on the Visitor



Traversel on the items

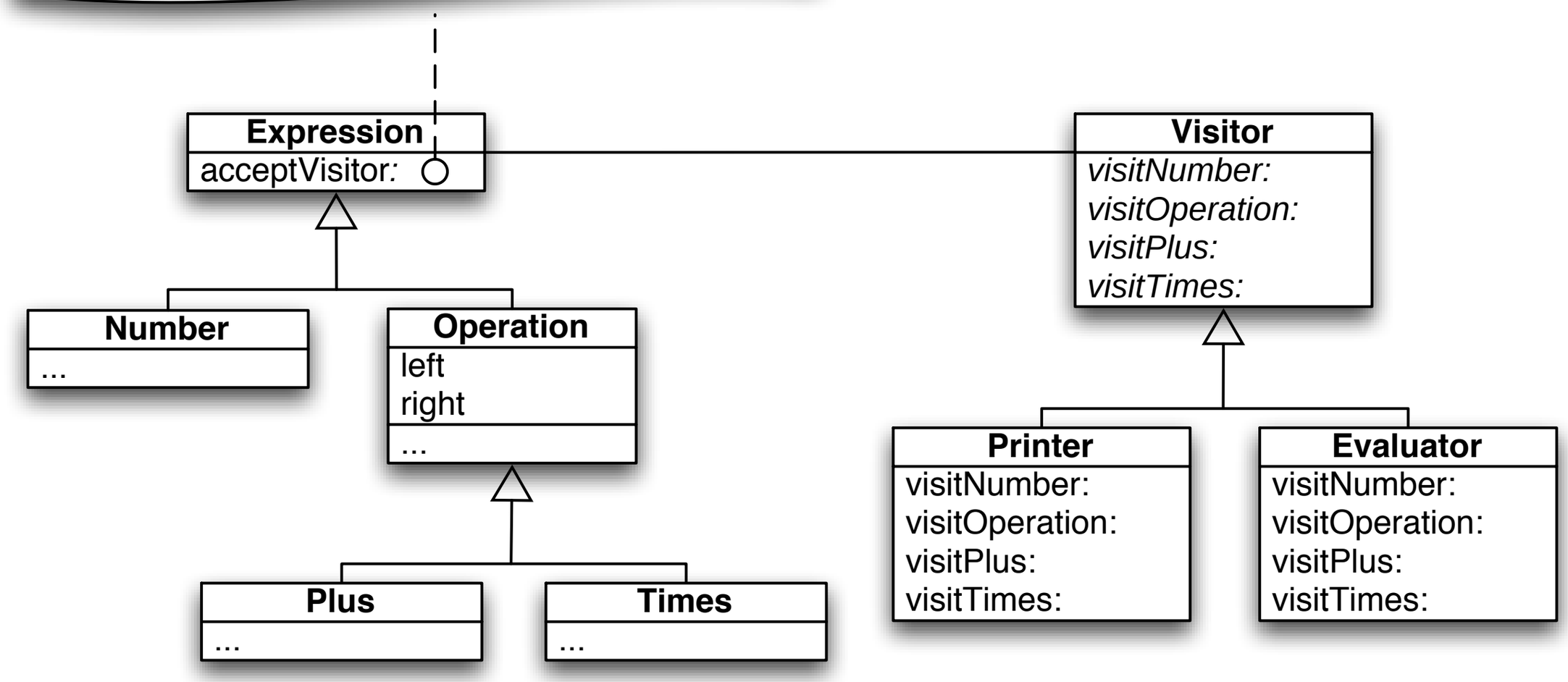


Implementation tricks

- You can implement it as we have shown before.
- But notice the general structure of the methods!
- This can be taken as advantage:
  - code can be generated for a visitor.
  - the method can be *performed/invoked*
- But take care:
  - only works when there is a full correspondence.
  - can make the code hard to understand.

Using #perform:

```
^aVisitor
        perform: ('visit', self class name, ':') asSymbol
        with: self
```



Design Patterns
Creational Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

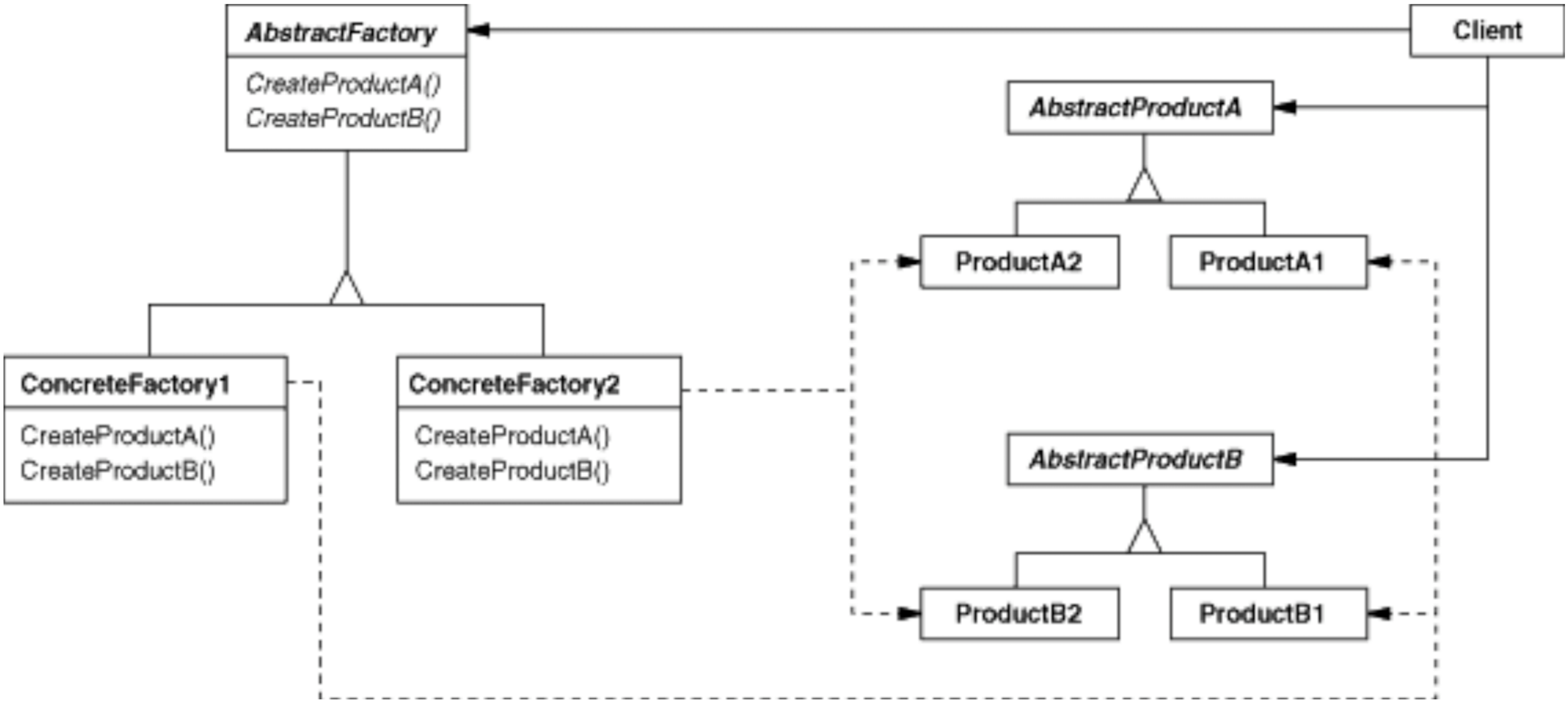
Creational Patterns

- 5 patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

Abstract Factory

- Intent: Provide an Interface for creating families of related or dependent objects without specifying their concrete classes

Abstract Factory: Structure



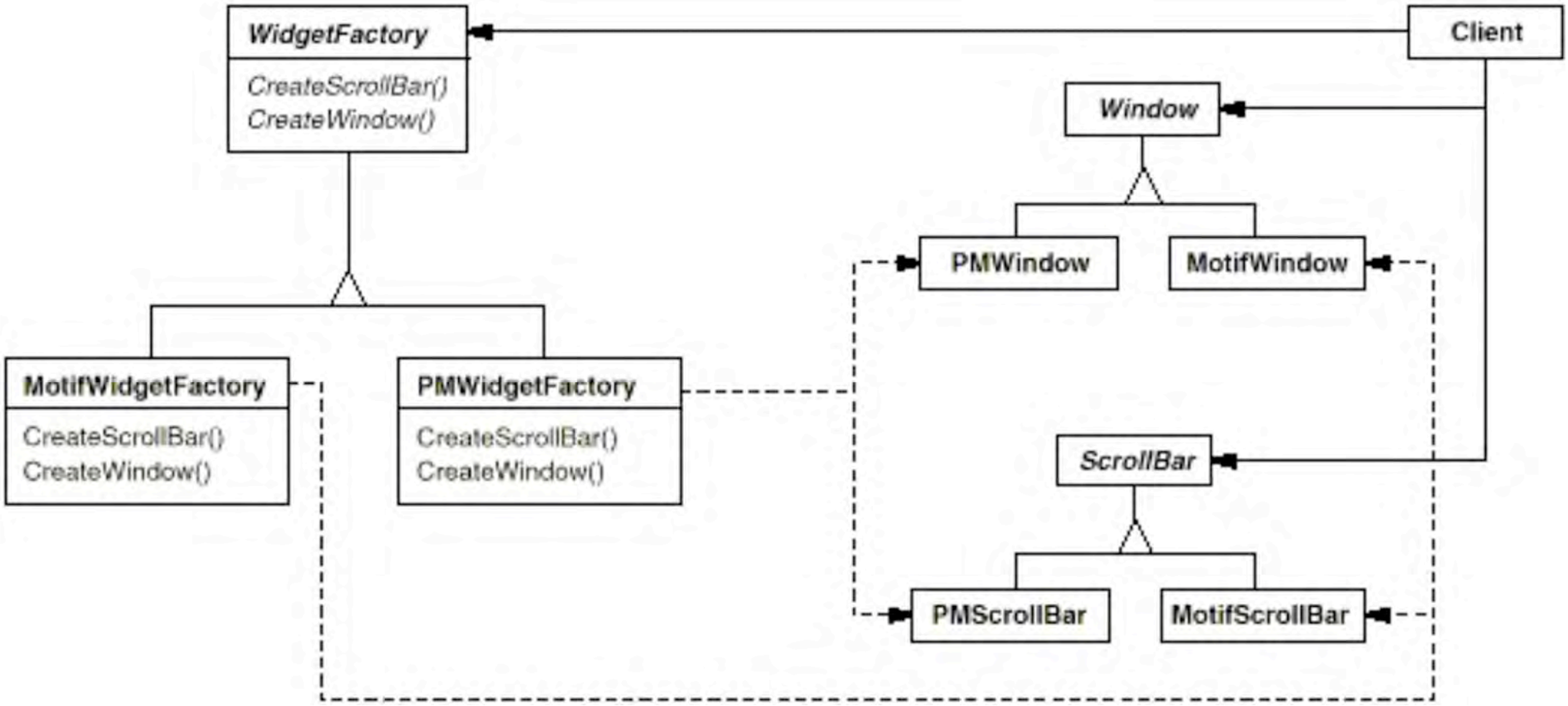
Abstract Factory: Participants

- AbstractFactory
  - declares an interface for operations that create abstract product objects
- ConcreteFactory
  - implements the operations to create concrete product objects
- AbstractProduct
  - declares an interface for a type of product object

Abstract Factory: Participants

- ConcreteProduct
  - defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface
- Client
  - uses only interfaces declared by AbstractProduct and AbstractFactory

Abstract Factory Pattern



Abstract Factory: Consequences

- + Isolates concrete classes:
  the AbstractFactory encapsulates the responsibility and the process to create product objects, it isolates clients from implementation classes; clients manipulate instances through their abstract interfaces, the product class names do not appear in the client code
- + Makes exchanging product families easy:
  the ConcreteFactory class appears only once in an application -that is, where it is instantiated- so it is easy to replace; because the abstract factory creates an entire family of products the whole product family changes at once

Abstract Factory: Consequences

- + Promotes consistency between products:
  when products in a family are designed to work together it is important for an application to use objects from one family only; the abstract factory pattern makes this easy to enforce
- +- Supporting new types of products is difficult:
  extending abstract factories to produce new kinds of products is not easy because the set of Products that can be created is fixed in the AbstractFactory interface; supporting new kinds of products requires extending the factory interface which involves changing the AbstractFactory class and all its subclasses

Defining Extensible Factories

- a more flexible but less safe design is to provide AbstractFactory with a single "make" function that takes as a parameter (a class identifier, a string) the kind of object to create
- is easier to realise in a dynamically typed language than in a statically typed language because of the return type of this "make" operation

Analysis Patterns

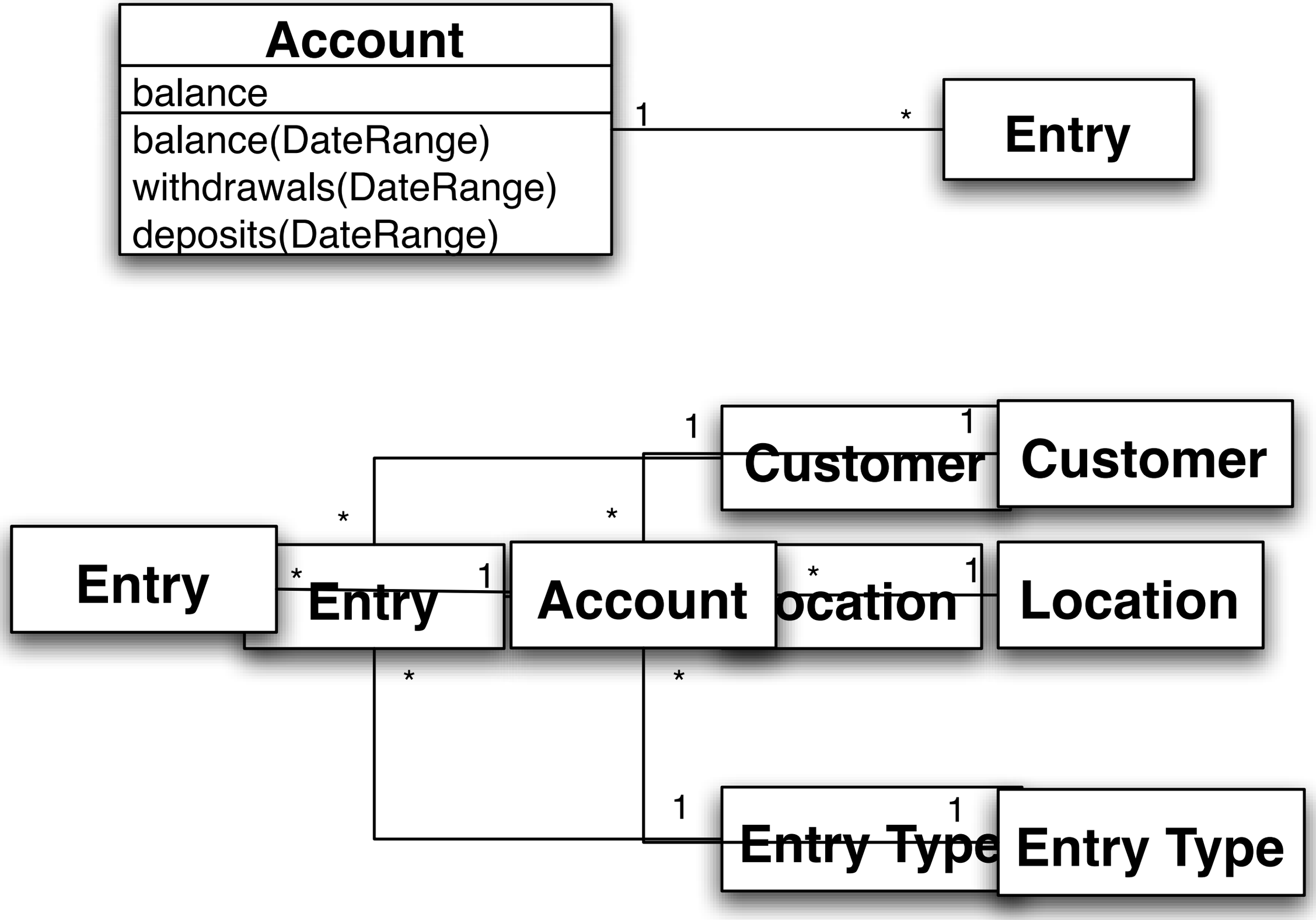
Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Analysis Patterns

- Reusable Object Models
- Groups of concepts that represent common construction in business modelling
  - relevant to one domain or spanning many domains
    - conceptual models -> about the business!!
- Domains considered: health care, accountability, trading, planning, etc.
- Pattern format: context/problem/solutions, make it work, when to use it.

Account Pattern

- Collect together related accounting entries and provide summarizing behaviour.
- Accounts:
  - personal bank accounts,
  - project cost accounts,
- What is an account?
  - container of entries plus behaviour
  - the history of some value
  - tying together all the elements of an entry that describe the kind of entry it is.


Account
balance
balance(DateRange)
withdrawals(DateRange)
deposits(DateRange)
1
*
Entry
1
1
Customer
1
Customer
*
*
Entry
*
1
Entry
Account
*
1
ocation
1
Location
*
*
1
1
Entry Type
1
Entry Type

Account Pattern

- Making it work:
  - Two essential qualities
    - keeping a collection of entries,
    - providing summarizing information over those entries.
  - large amount of entries
    - optimize calculation,
    - cache value,
    - no details required: replace by a single entry
- When to use it?
  - need of current value, historical values and keep a history of changes to that value,
  - pull together all the entries for a common set of descriptors.

Architectural Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

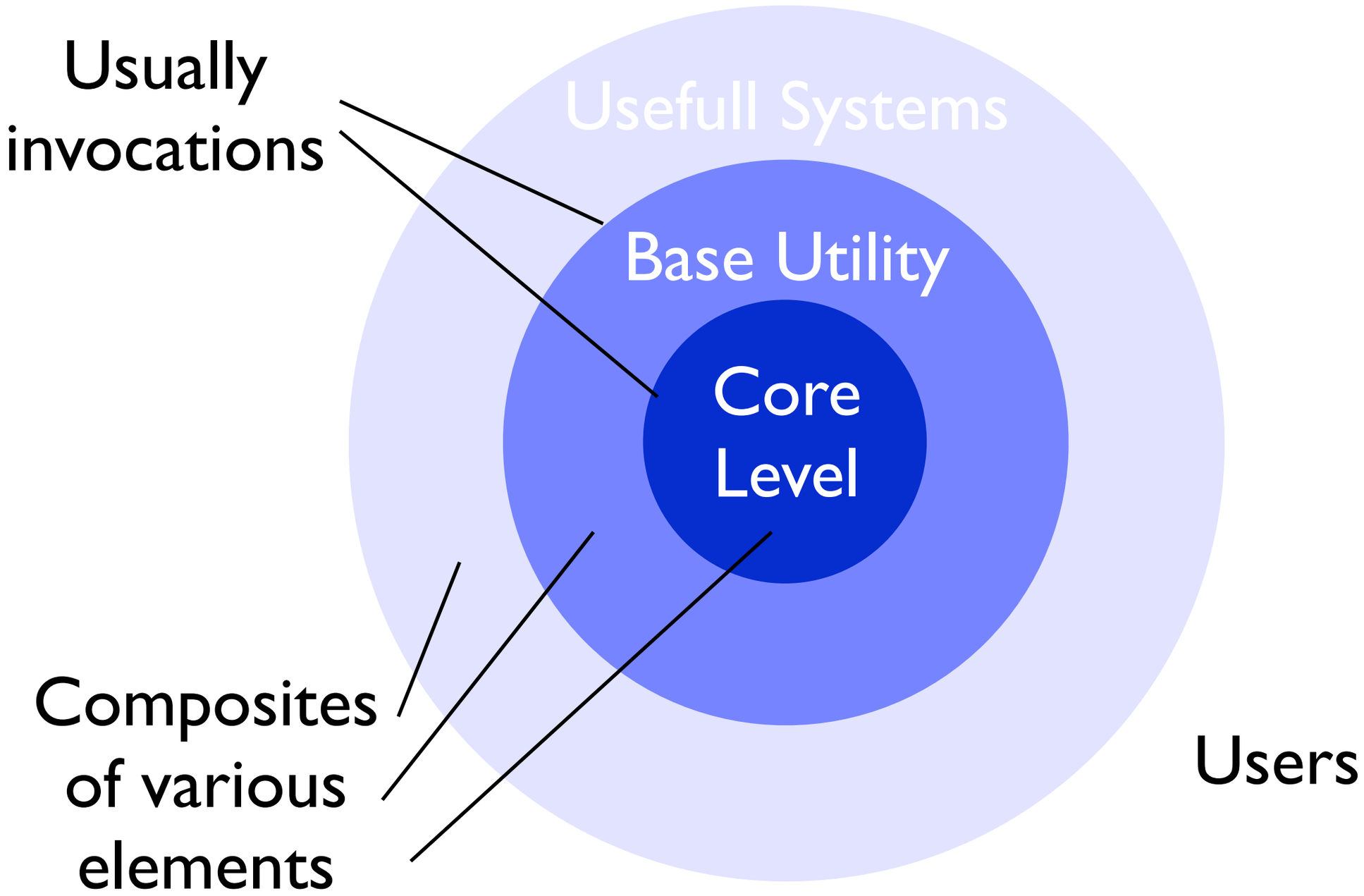
Architectural Patterns

- Architectural patterns often dictate a particular high-level, modular system decomposition.
  - context/problem/solution triplet
  - consequences!!
- Differ from Design Patterns
  - they act more like blueprints
- Some are widely accepted
  - layered architecture
  - pipes and filters architecture
- Improve -ility's!

Layered Architecture

- A layered system divides a system in layers
  - each layer:
    - provides services to layers above
    - requires services from layer below
  - The connectors are defined by the protocols that determine how layers will interact
  - Examples: layered communication protocols, operating systems

Layered system graphically



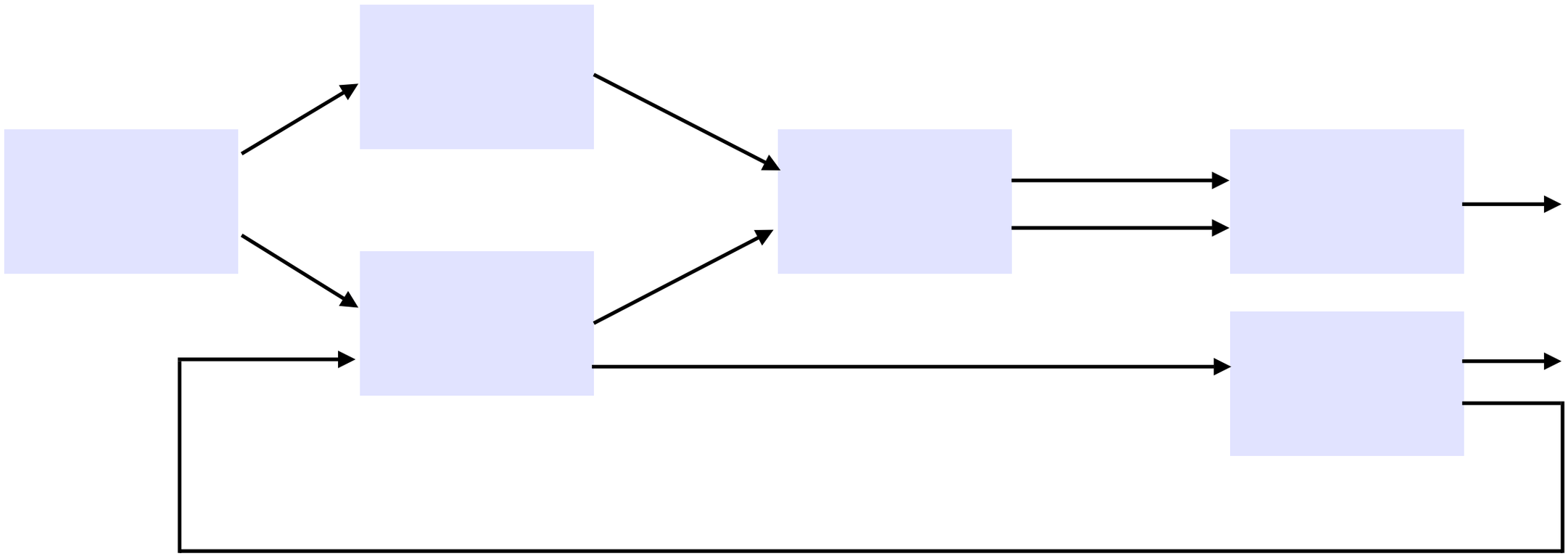
Layered Systems Discussion

- Pros
  - increasing level of abstraction
  - support enhancement & support reuse
- Cons
  - not always applicable
  - performance penalties

Pipes and Filters Architecture

- Filter reads from input & writes to output stream
  - incrementally transforms input to output
    - So output begins for input is complete!
  - are independent (no sharing of state)
- Examples: Unix shell, compilers, DSP, functional programming

Pipes and Filters graphically



Pipes And Filters Discussion

- Pros
  - understand overall behavior a composition of individual filter behaviors
  - support enhancement & reuse
  - specialized analysis possible
- Cons
  - lead to batch processing (interactivity problems)
  - synchronization problems

Plugin

- plugin is a program that interacts with a host application
  - provides certain, usually very specific, function “on demand”.

- reasons:
  - enable third-party developers
    - to create capabilities to extend an application
    - to support features yet unforeseen
  - separate source code from an application because of incompatible software licenses.

Plugin Examples

- Email clients use plugins to decrypt and encrypt email.
- Graphics software use plugins to support file formats and process images (Adobe Photoshop).
- Media players use plugins to support file formats and apply filters (e.g. Winamp)
- Software development environments use plugins to support programming languages. (e.g. Eclipse)
- Web Browsers use plugins to play video and presentation formats (e.g., Flash, QuickTime)

Plugin Discussion

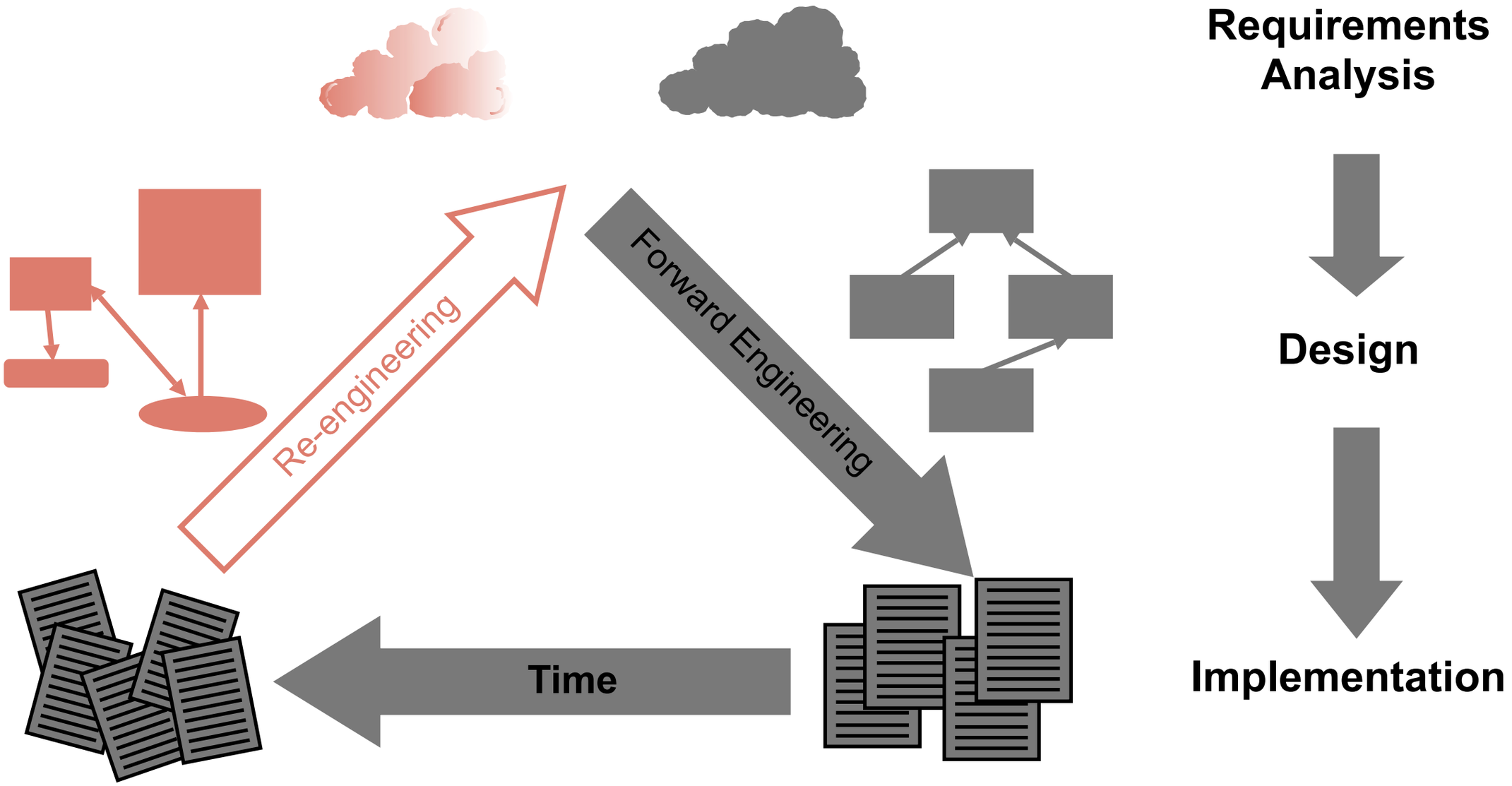
- Pros
  - host application is independent of the plugins.
  - plugins can be added and updated dynamically without changing the host application
- Cons
  - plugins depend on the services of the host application

Reengineering Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Forward and Re-engineering

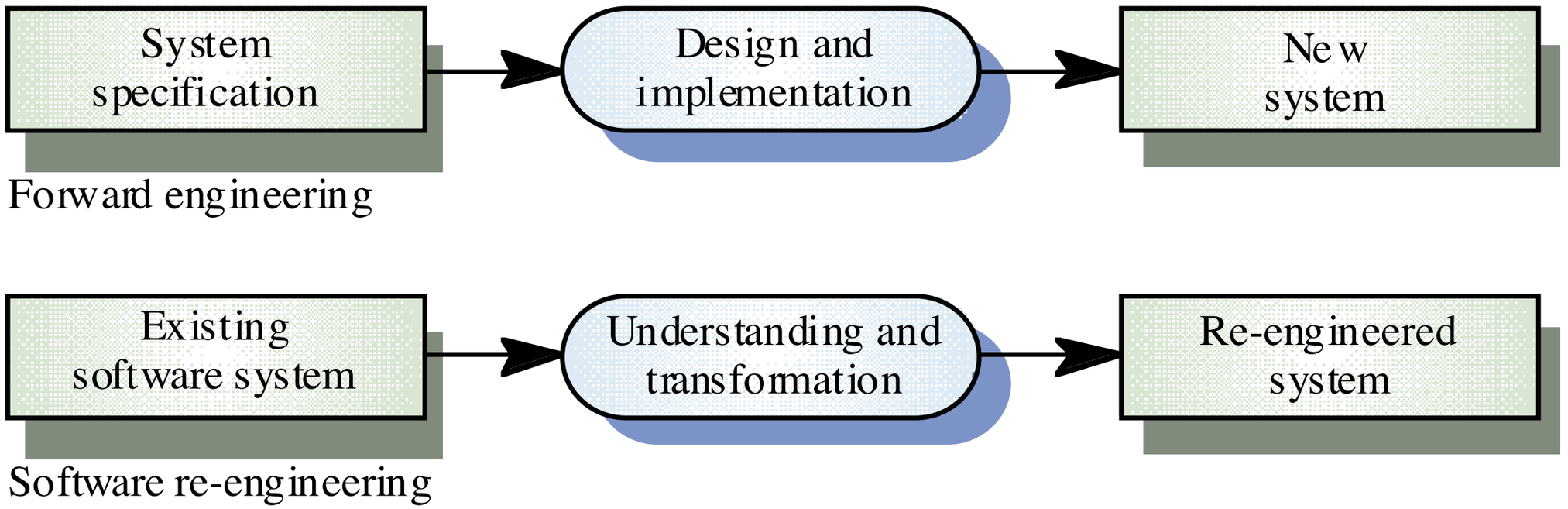
Software development is more than forward engineering alone...



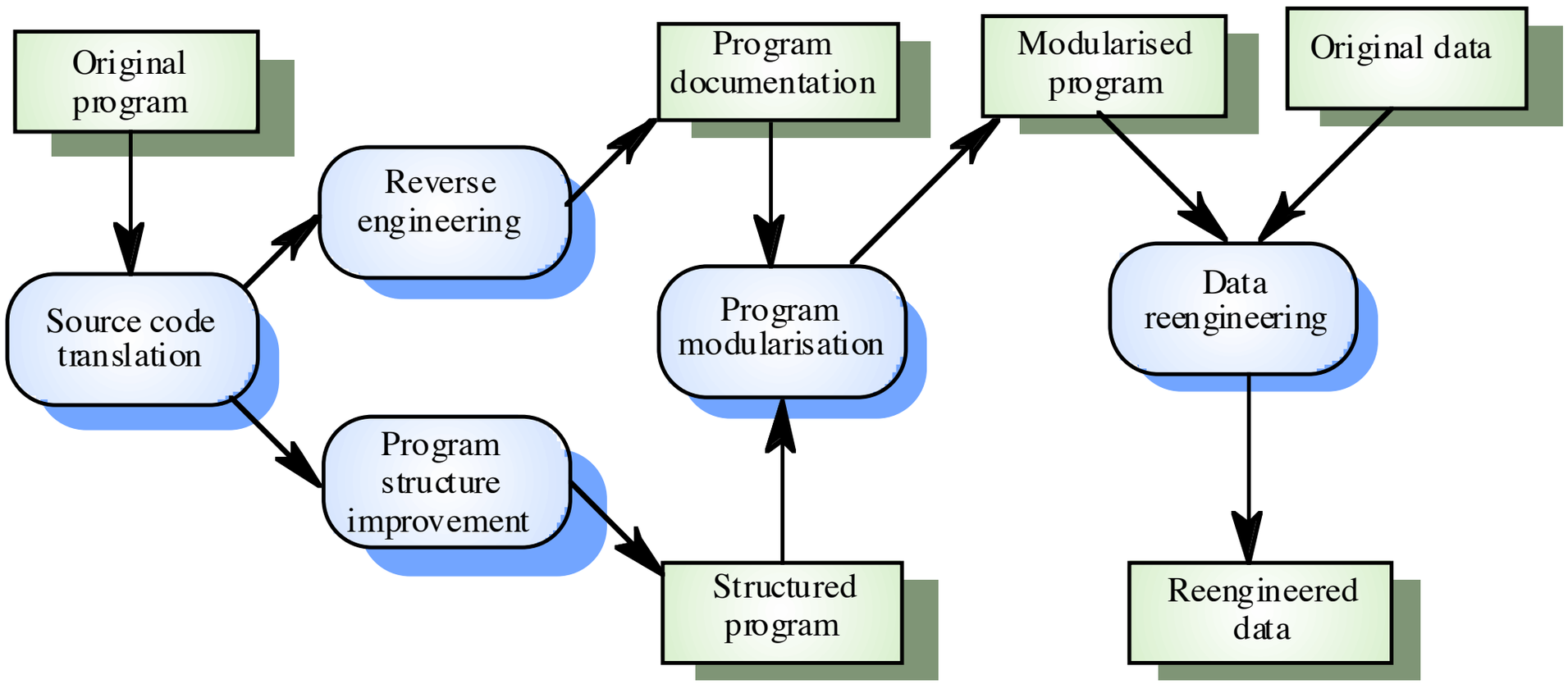
Reengineering

- "Reorganising and modifying existing software systems to make them more maintainable"
  - Re-structuring or re-writing part or all of a legacy system without changing its functionality
  - Applicable where some but not all sub-systems of a larger system require frequent maintenance
  - Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented

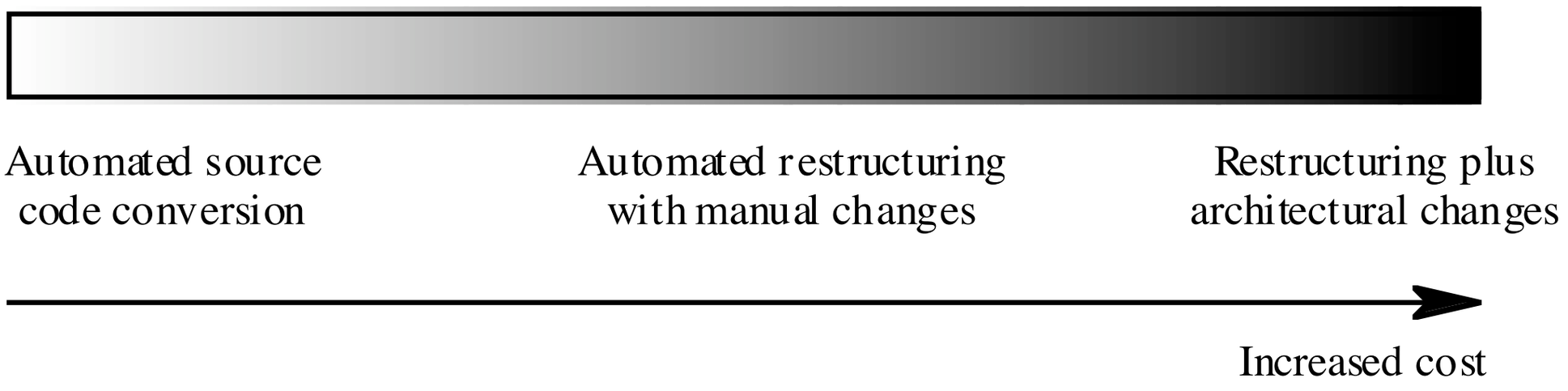
Forward and Re-engineering



Re-engineering Process



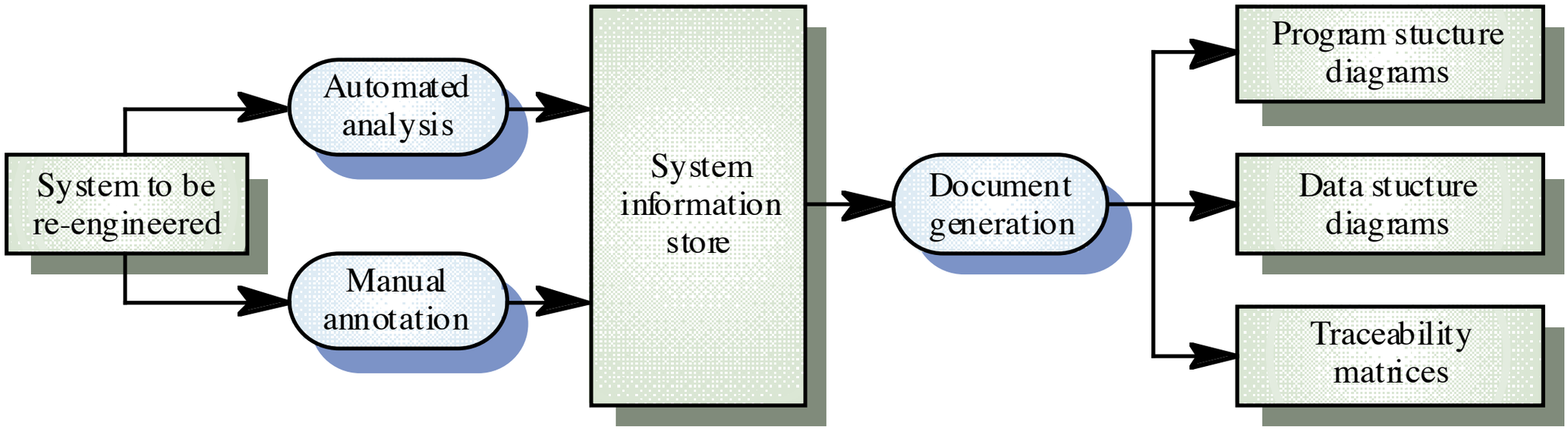
Re-engineering Approaches



Reverse Engineering

- Analysing software with a view to understanding its design and specification
- May be part of a re-engineering process, but may also be used to respecify a system for reimplementation
- Builds a program database and generates information from this
- Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process

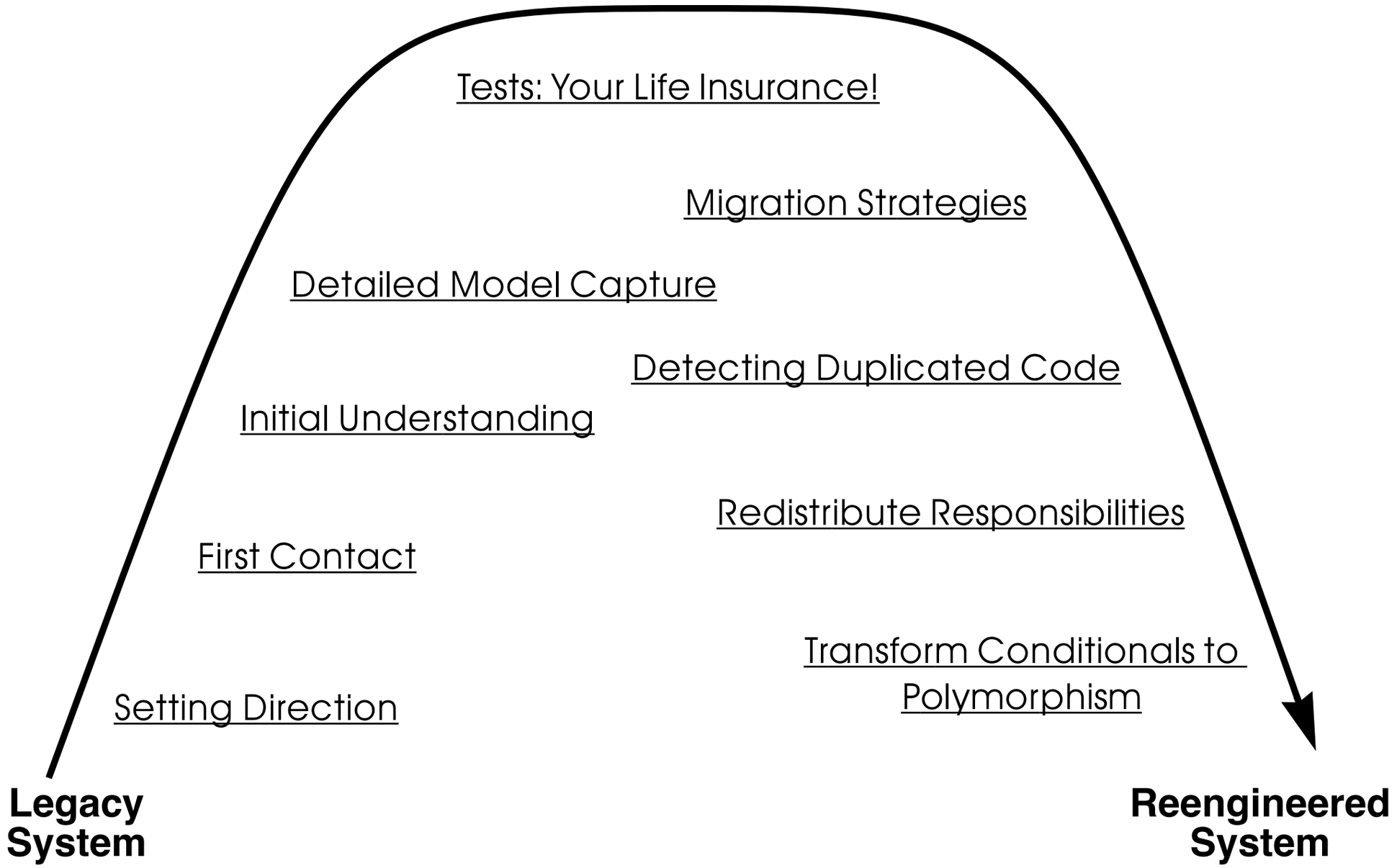
Reverse Engineering Process



Reengineering Patterns

- You might have noticed: Reengineering is difficult.
- Reengineering patterns capture
  - reengineering of *legacy object-oriented systems*
    - planning a reengineering project
    - reverse-engineering
    - problem detection
    - migration strategies
    - software redesign

Patterns in the lifecycle



Reengineering Pattern Structure

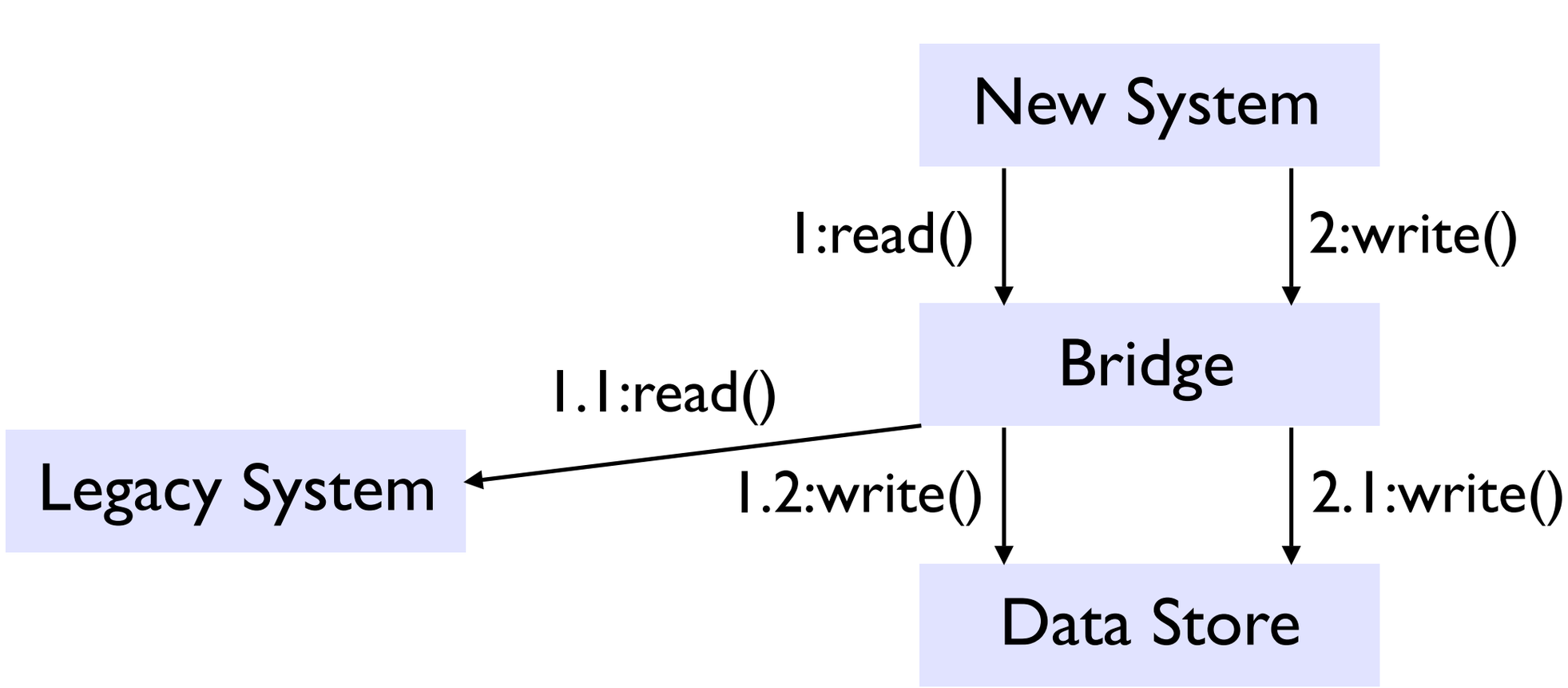
- Name
- Problem(s) it addresses
- Solution
- List of trade-offs
  - the pros
  - the cons
  - the difficulties
- Example
- Reasons for applying the pattern
- Related Patterns

Make a bridge to the new town

- Intent: Migrate data from a legacy system by running the new system in parallel, with a bridge in between them.
- Problem: how do you incrementally migrate data from a legacy system to its replacement while the two systems are running in tandem?

Make a bridge to the new town

Solution: Make a (data) bridge that will incrementally transfer data from the legacy system to the replacement system as new components are ready to take the data over from their legacy counterparts.



Make a bridge to the new town

- Pros
  - you can start using the new system without migrating all the legacy data
- Cons
  - can be tricky to implement
  - once some data is transferred it can be hard to go back
  - the data bridge will add performance overhead
- Known uses, rationale, related patterns

How *not* to use patterns

- Do not apply patterns indiscriminately
  - Flexibility and variability is achieved at a cost
    - typically extra indirections
    - can complicate design or cost performance
- Only apply when flexibility is needed
  - do not overdesign your application
- Study the trade-off issue for each pattern for more information!!

Wrap-up

- Architectures "can't be made, but only generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from the seed." (Alexander)
  - patterns describe such building blocks
  - applying them implicitly changes the overall structure (architecture)
  - whether it is on classes, business concepts, components, ...

References

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

- Frank Buschmann, et al., *Pattern-Oriented Software Architecture — A System of Patterns*, Wiley, 1996

- Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan-Kaufmann, 2002

- Martin Fowler. Analysis Patterns, Reusable Object Models. Addison Wesley, 1997

- www.martinfowler.com

References

- Special issue on Software Patterns, IEEE Software, volume 24, number 4, July/August 2007.

- P. Avgeriou and U. Zdun, Architectural Patterns Revisited - A Pattern Language. Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPLoP), 2005, pp.431 - 470.