Design Patterns:
History, Theory and Practice

Drs. Vadim V. Zaytsev

7 September 2004

Technical questions

- This course is given by Dr.ing. Ralf Lämmel.
- All lectures are given in English.
- Ask small questions preferably during the lecture, big ones in the break or via e-mail.
- Up-to-date information about the course: rescheduling issues, slides, papers ... — `http://www.cs.vu.nl/~ralf/oo/lecture-2004/`
- There will be a lecture by Ralf on the 14^{th} and no lecture on the 21^{st} of September.
- These slides incorporate some of the work by Ralf Lämmel, Mehmet Akşit, Brian Malloy and Lodewijk Bergmans.

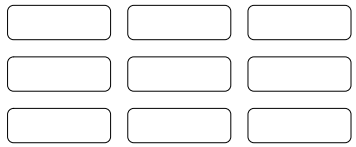
What is a *design pattern*?

Background: brief history

- First step: **object-oriented programming languages** (1982).
  - classes and objects as first-class entities
  - insufficient for creating object-oriented software
- Second step: **object-oriented methods** (1988).
  - explore "real-world" domain concepts and apply object-oriented modelling techniques
  - did not make design knowledge explicit
- Third step: **object-oriented design patterns** (1992).

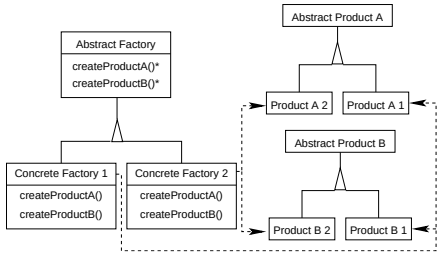
Discovering object-oriented design patterns

- Determine the basic building blocks:



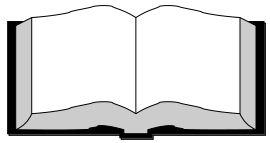
  - *"these are the object-oriented concepts"*

- Find a pattern for the problem:



  - *"this is the pattern"*

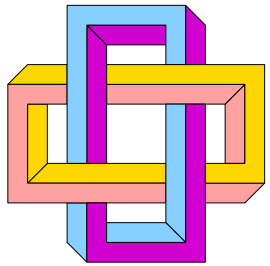
- Record the pattern:



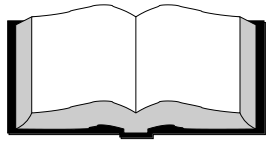
  - *"this is the definition"*

Applying design patterns

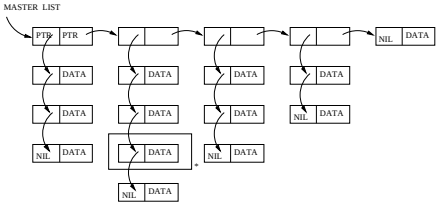
- Face a problem and search for a pattern:
  - *"this is my problem"*



- Understand the pattern:
  - *"this is the suitable pattern"*



- Implement solution according to the pattern:
  - *"this is the implementation"*



Design patterns

- Design pattern is a named abstraction for a recurring solution to a particular design problem.

- Patterns are used to represent design knowledge to solve a particular type of problems.

- The concept is due to Christopher Alexander (1979), now mostly by the book of Erich Gamma et al ("Gang'o'Four", 1995) — explores basic object-oriented model features.

- Patterns have explicit structure.

Why one should use design patterns?

- A language everyone speaks
- Design language, independent of a programming language
- Most design elements are covered by patterns
- Patterns make design resistant to re-design. . .

Robustness to change

- **Algorithmic dependencies:** Template Method, Visitor, Iterator, Builder, Strategy.

- **Creating an object by specifying a class explicitly:** Abstract Factory, Factory Method, Prototype.

- **Dependence on specific operations:** Command, Chain of responsibility.

- **Dependence on object representation or implementation:** Abstract Factory, Bridge, Proxy.

Robustness to change

- **Tight coupling:** Abstract Factory, Command, Façade, Mediator, Observer, Bridge.

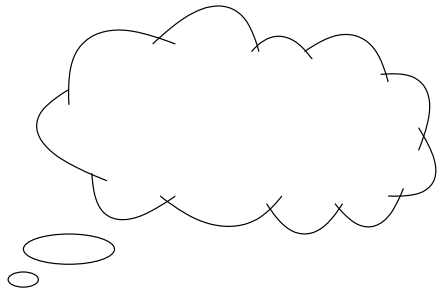
- **Extending functionality by subclassing:** Command, Bridge, Composite, Decorator, Observer, Strategy.

- **Inability to alter classes conveniently:** Visitor, Adapter, Decorator.

Where and how do they arise?

"Patterns are discovered, not invented"
Richard Helm

- 23 mostly used patterns come from the GoF book
- lots of domain/language/... specific patterns exist
- what is so difficult?
  - encapsulation
  - flexibility, extensibility
  - adaptability, ease of modification
  - performance
  - evolution
  - reusability
  - ...



Patterns classification by purpose

purpose reflects what a pattern does

- **creational** — facilitate object creation
- **structural** — help to compose classes or objects
- **behavioural** — introduce ways objects interact and distribute responsibility

Patterns classification by scope

scope specifies if the pattern applies to classes or objects

- **class patterns** deal with relationships between classes or subclasses established through inheritance, while these relationships are fixed at compile time

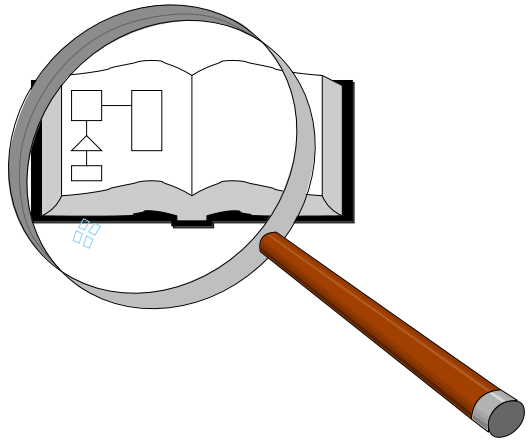
- **objects patterns** deal with object relationships that can be changed at runtime

How to write down patterns?

- **Portland form** — narrative unstructured form
- **Alexandrian form** — problem, forces, solution
- **GoF-1 form** — name, problem, solution, consequences
- **GoF-2 form** — name, classification, intent, aka, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, related patterns
- No perfect form (yet).

How to choose a pattern

- Formulate your problem well
- Know the basic catalogue
- Scan the “intent” section
- Study related patterns
- Always have redesign case in mind

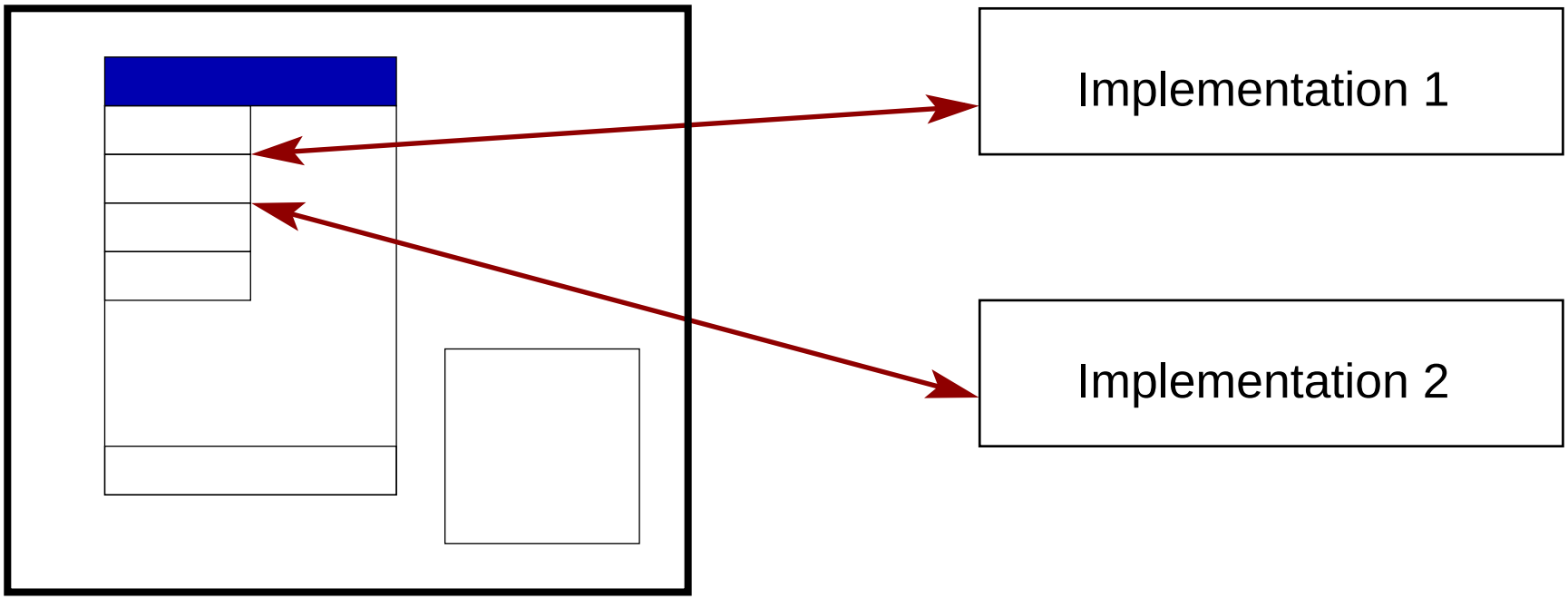


Common problems

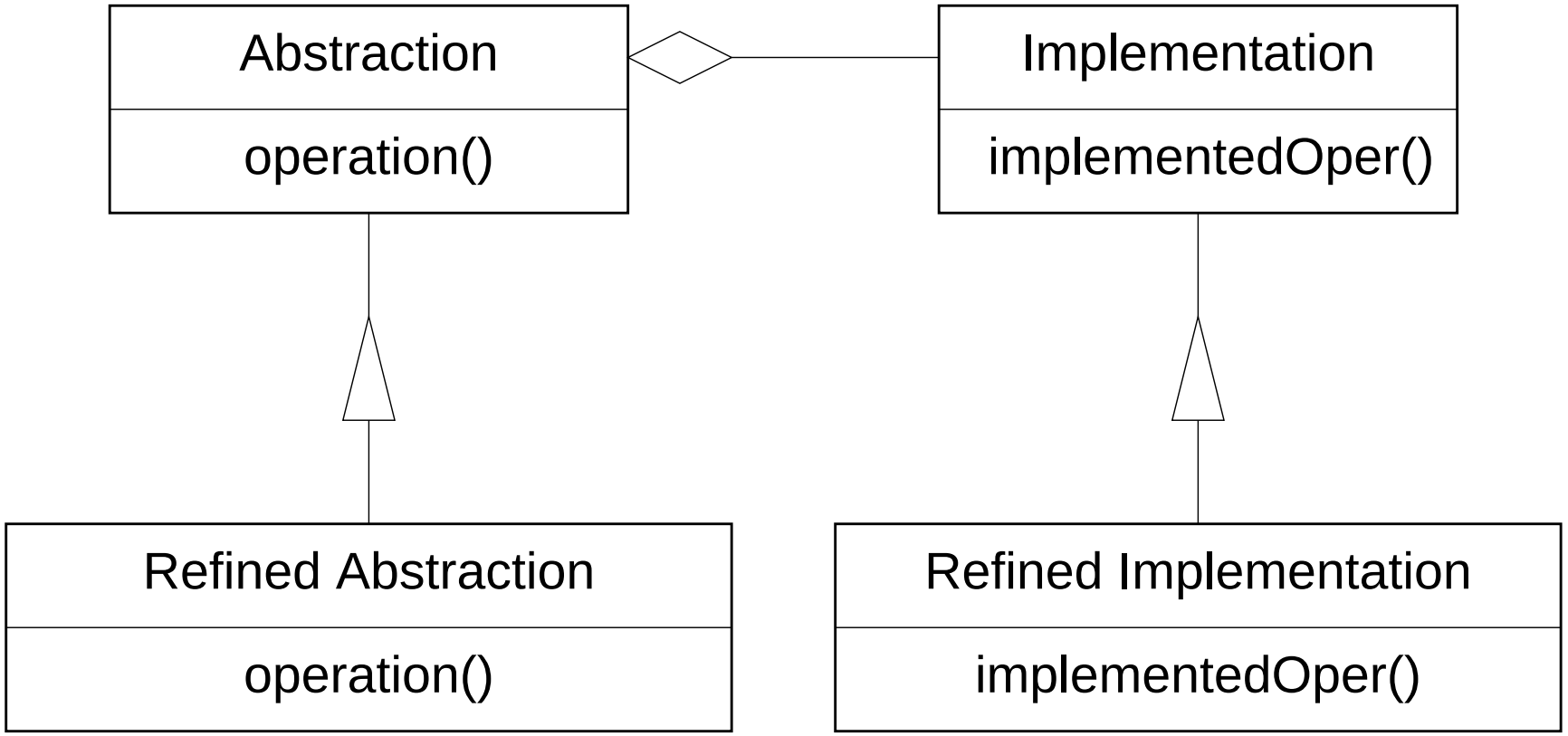
- Getting the right pattern (even within those with the same UML diagram)
- Taming over-enthusiasm (use all the patterns in one application; use only the pattern last learned)
- Remember: the client does not give a damn about design patterns
- Do not forget your own head

Pattern example: Bridge (p.151)

- **Motivation:** to avoid permanent binding between abstraction and implementation $\Rightarrow$ make the method implementation an object itself!



Bridge: solution in UML



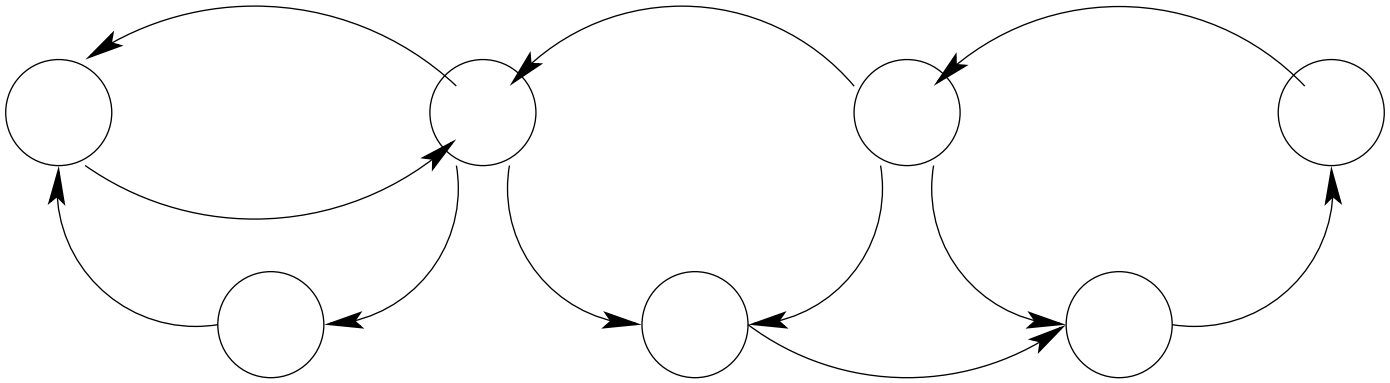
Bridge: features

- Easy to switch to an alternative implementation
- Both parts are extensible
- Implementation details are invisible for clients
- All methods should be declared at the interface: no run-time extension
- Loss of state in the implementation: working & switching

When to use: different GUIs

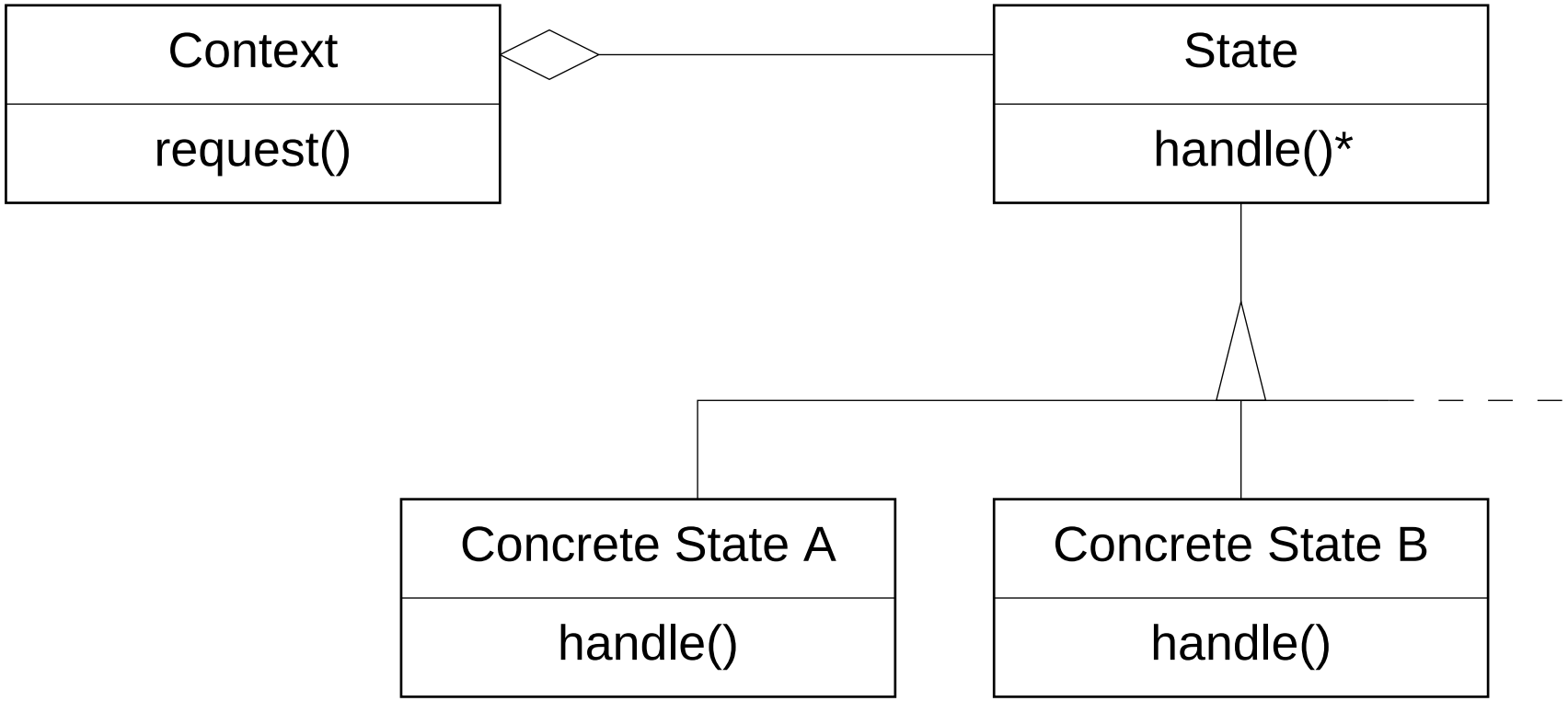
Pattern example: State (p.305)

- **Motivation:** state-dependent behaviour $\Rightarrow$ we should be able to change state at a run-time!



- **Alternative ways:** FSM is usually programmed with a table or with (nested) conditional statements. The first method is bad if the table is sparse, the second is far from efficient (and maintainable) if complex.

State: solution in UML

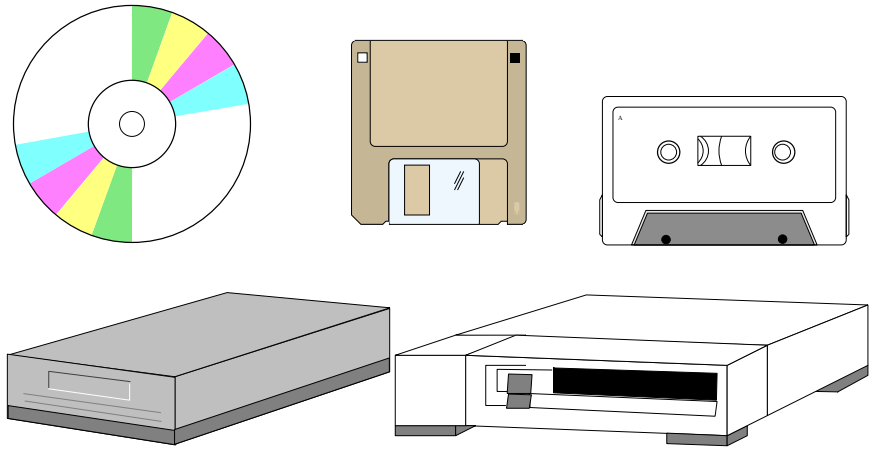


State: features

- Localises state-specific behaviour and treat each state separately
- Makes state transitions explicit (may be perfect, may be awful)
- State objects can be shared (`static`, `fixed`, whatever)
- Creating and destroying state objects can be nasty
- **When to use:** TCP protocol

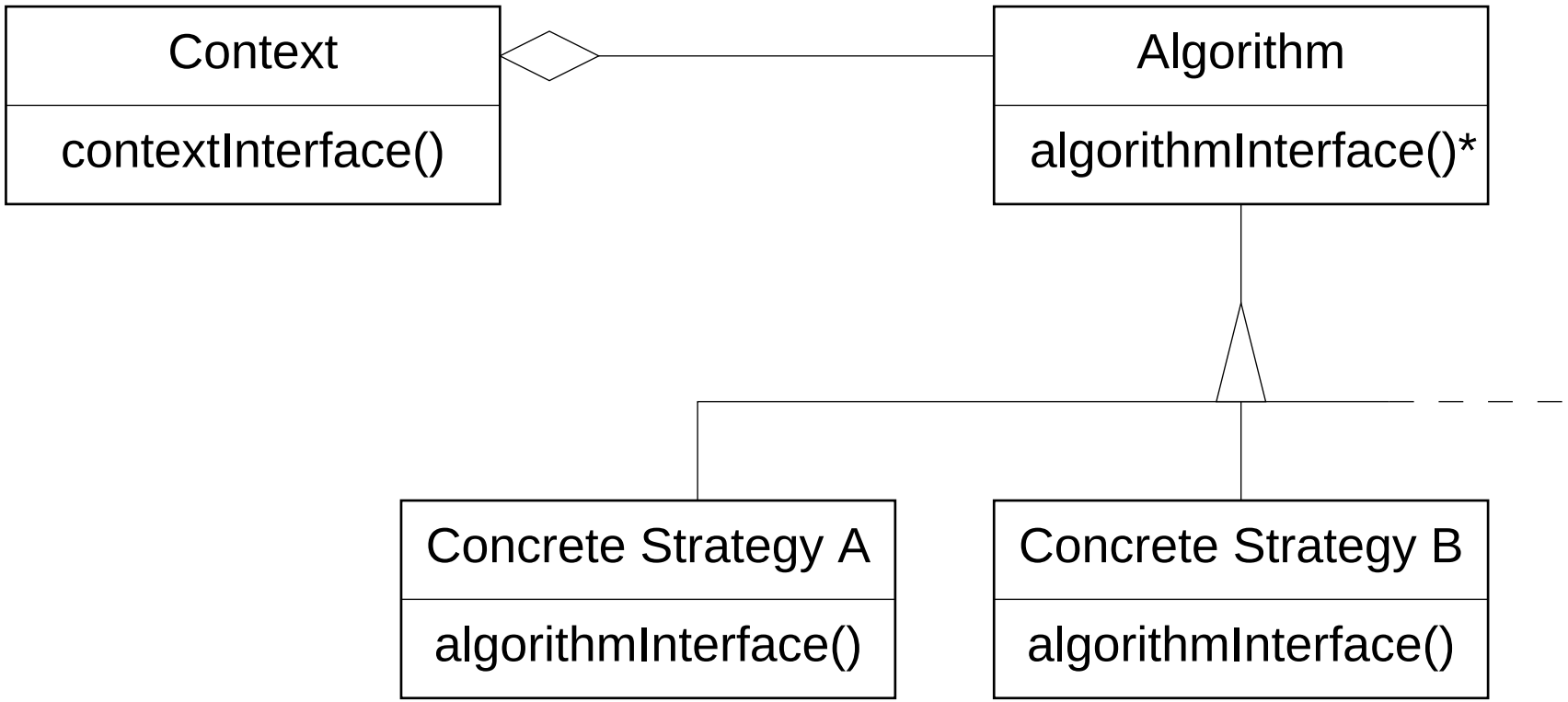
Pattern example: Strategy (p.315)

- **Motivation:** dynamically changing algorithms (there are classes which differ only in their behaviour)
- Different variants of an algorithm are needed



- **Solution:** make the algorithm an object!

Strategy: solution in UML



Strategy: features

- Families of related algorithms
- Alternative to **subclassing**
- Choice of implementations of the same behaviour
- All methods have to be declared explicitly
- Overhead in communication; in the number of objects
- **When to use:** mail composer

What is going on?

Things to be inferred:

- There are different design patterns with very similar solutions.
- Please recall: design patterns are pieces of knowledge, not just class diagrams.
- It makes right design pattern choice even more challenging.
- However, not all design patterns are about making new classes outta the old ones' parts!

Pattern example: Abstract Factory (p.87)

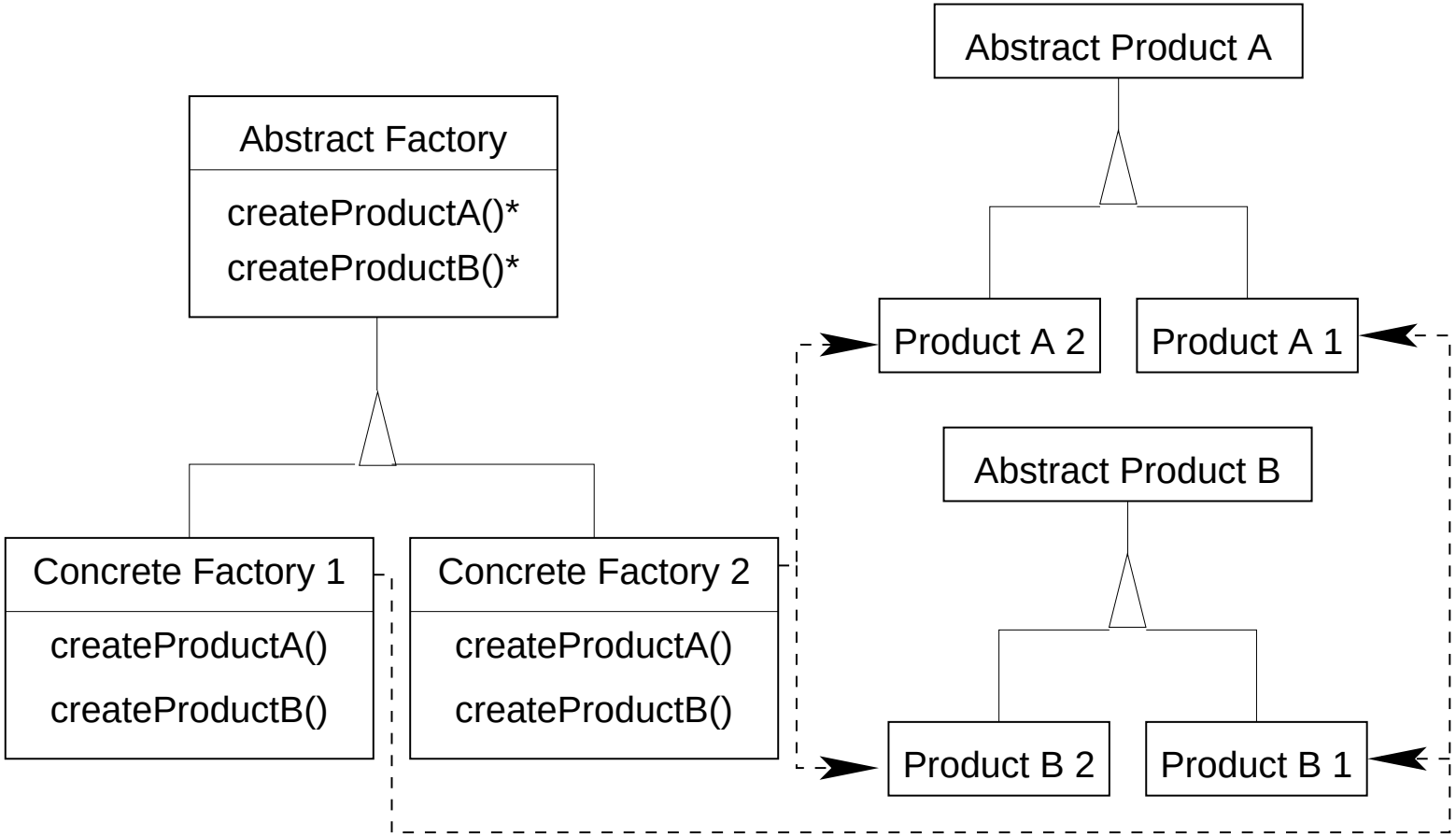
- **Motivation:** independent creation and utilisation of products.

- *"I know how to put objects together, I do not know how to make them"*

- Multiple **families** of products may be used by the same client, who is able to switch between factories.

- Related products should be used together.

Abstract Factory: solution in UML



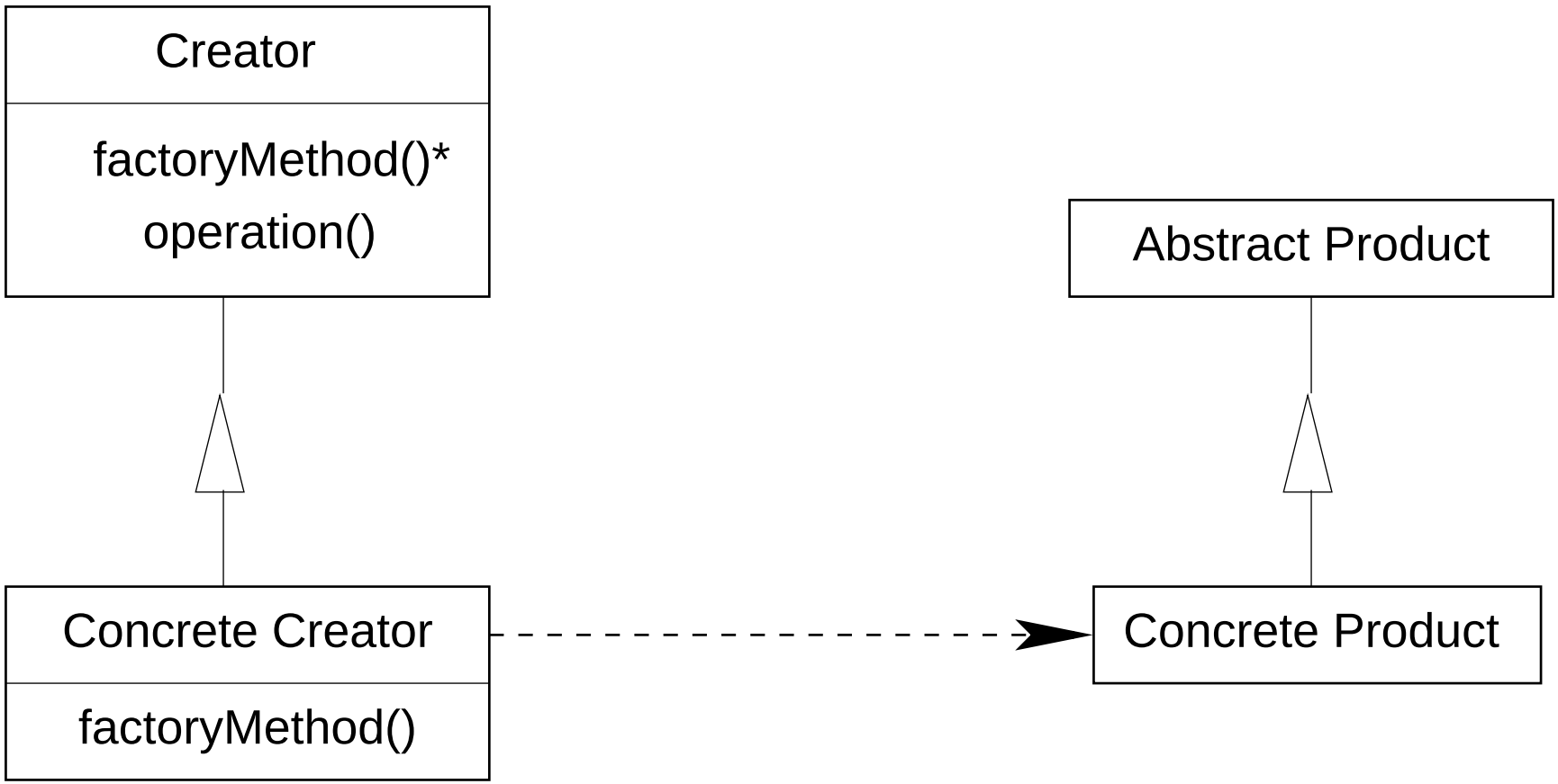
Abstract Factory: features

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products
- Only one instance of the factory is need (most often)
- Creating new products: new subclass for each family
- Creating new families: recompilation
- A new factory is needed for any consistent composition
- **When to use:** interface themes/skins

Pattern example: Factory Method (p.107)

- **Motivation:** inheritance-based design pattern for creating objects
- A class **should not know** the class of objects it creates!
- *"I do not know the exact products, but I can explain how to do something with them"*
- The same knowledge can be used to put other objects together.

Factory Method: solution in UML



Factory Method: features

- Provides hooks for subclasses
- A factory method can be either **abstract** (no default implementation) or **concrete** (with a default implementation)
- Product class should have a generic interface for all the creator classes
- All creations should be realised through the Factory Methods
- **When to use:** multipurpose editors

Overview on creational patterns

- We have seen: `Abstract Factory`[87] (independent creation and utilisation), `Factory Method`[107] (inheritance-based creating).

- We have not seen: `Prototype`[117] (very flexible object creation), `Singleton`[127] (global access to the only class instance), etc.

- All creational design patterns concern the process of object creation.

- **When to use:** if you want a good "patterned" way to create objects of some special sort.

Overview on structural patterns

- We have seen: `Bridge`[151] (decouple interface and implementation).
- We have not seen: `Composite`[163] (uniform part-whole structures), `Decorator`[175] (extending behaviour), etc.
- All structural design patterns deal with the composition of classes and objects.
- **When to use:** if you want to organise a lot of classes or objects in a good and "patterned" way.

Overview on behavioural patterns

- We have seen: `State`[305] (FSM-like behaviour), `Strategy`[315] (dynamically changing algorithms).
- We have not seen: `Command`[233] (objects with dynamically changing number of methods), `Mediator`[273] (expressing co-ordination between objects), `Visitor`[331] (non-uniform data structures), etc.
- All behavioural design patterns characterise the ways in which classes or objects interact and distribute responsibility.
- **When to use:** only in the cases described! And even then be careful.

Discussion on problems

- **Methodological:** OO development methods are not pattern-driven; high-level hacking; not generative enough; large number of similar patterns; multiple patterns incorporation.

- **Programming:** patterns reverse engineering; adaptability is not always the most important issue; the GoF book is bound to the OO model.

- **Organisational:** patterns need time; not always gladly supported.

The End.