LINGI2252 – PROF. KIM MENS

SOFTWARE MAINTENANCE & EVOLUTION

LINGI2252 – PROF. KIM MENS

SOFTWARE PATTERNS

LINGI2252 — PROF. KIM MENS

A. PATTERNS

WHERE DOES THE "PATTERNS" IDEA COME FROM?

Influential but controversial architect Christopher Alexander

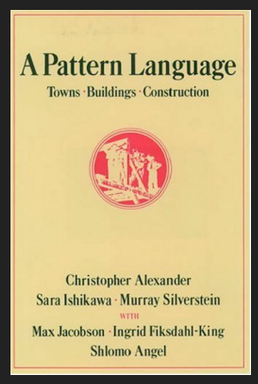
proposed patterns as a way to *reuse* extensive *architectural experience*

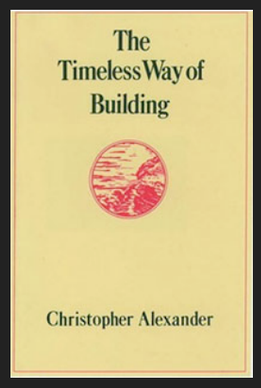
expressed as pre-canned architectural fragments

Several interesting books:

A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977.

The Timeless Way of Building. Oxford University Press, 1979.





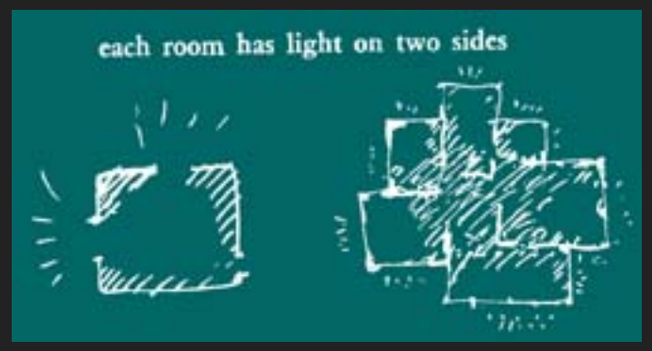
WHAT IS A PATTERN?

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

Christopher Alexander

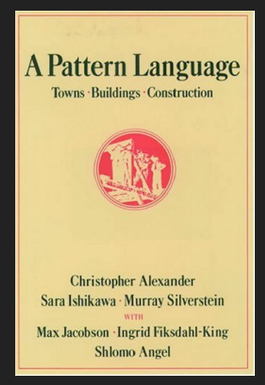
EXAMPLE 1: "LIGHT ON TWO SIDES OF EVERY ROOM"

"When they have a choice, people will always gravitate to those rooms which have light on two sides, and leave the rooms which are lit only from one side unused and empty.



Therefore:

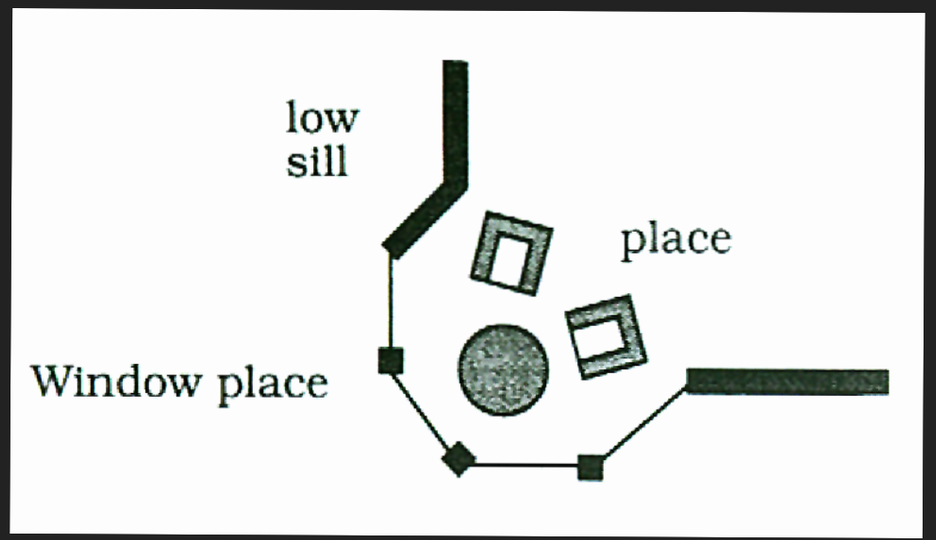
Locate each room so that it has outdoor space on at least two sides, and then place windows in these outdoor walls so that natural light falls into every room from more than one direction."



Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

http://www.patternlanguage.com/apl/aplsample/apl159/apl159.htm

EXAMPLE 2:"WINDOW PLACE"



ADOPTION OF CHRISTOPHER ALEXANDER'S IDEAS

Christopher Alexander's ideas were later adopted in other disciplines

(often much more heavily than the original application of patterns to architecture)

in particular in the software engineering community :
design patterns, *architectural patterns*, *anti patterns*, ...

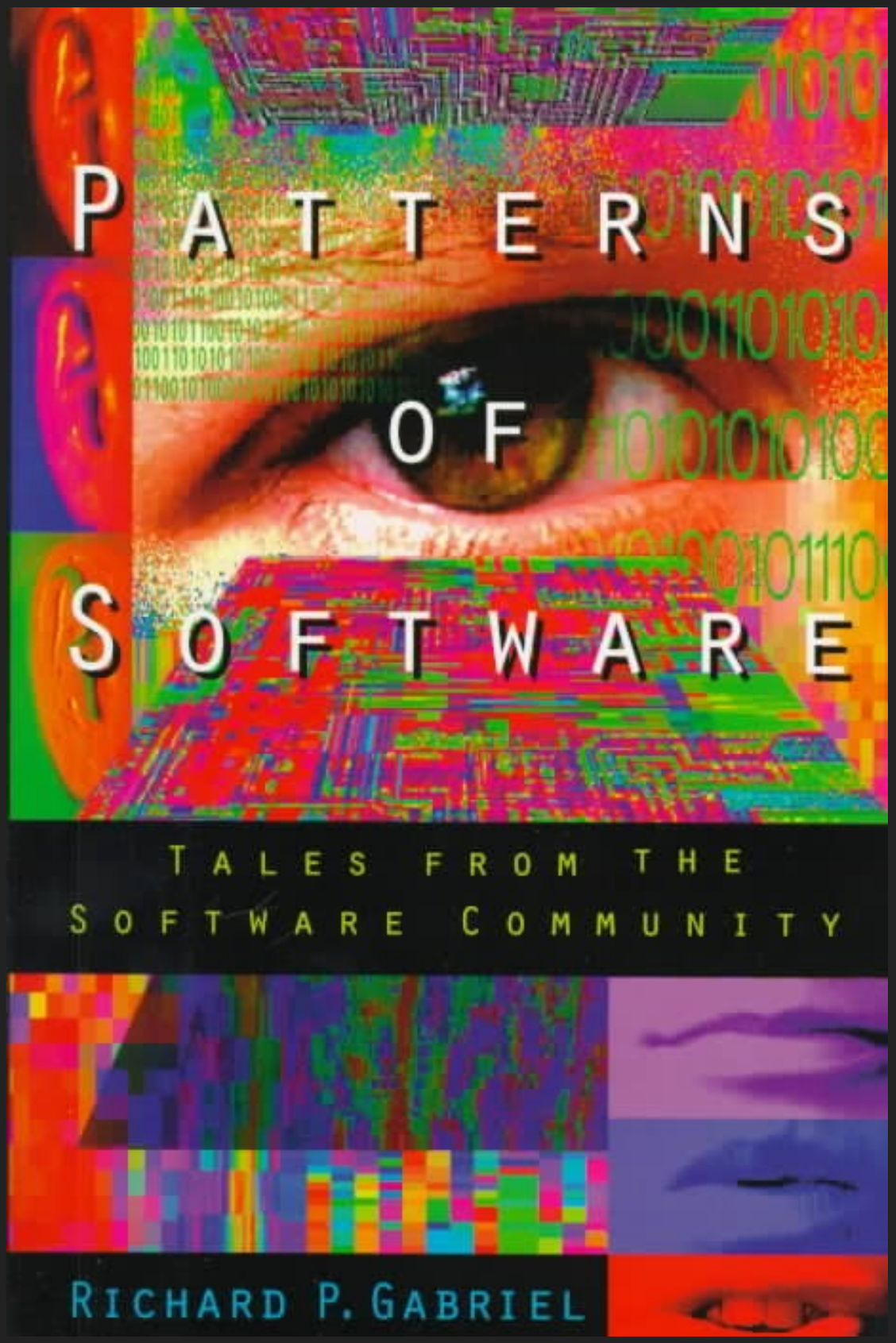
Keynote talk at OOPSLA 1996 on "Patterns in Architecture"

video link + keynote text

ADDITIONAL READING

- A.o., a good introduction to the analogy between the work of Christopher Alexander on patterns in architecture and its applications in software development.

- Richard P. Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1998. ISBN 0195121236



LINGI2252 – PROF. KIM MENS

B. PATTERNS IN SOFTWARE

MANY KINDS OF PATTERNS IN SOFTWARE ENGINEERING

Best Practice Patterns

Programming Styles and Idioms

Design Patterns

Architectural Patterns

Anti Patterns

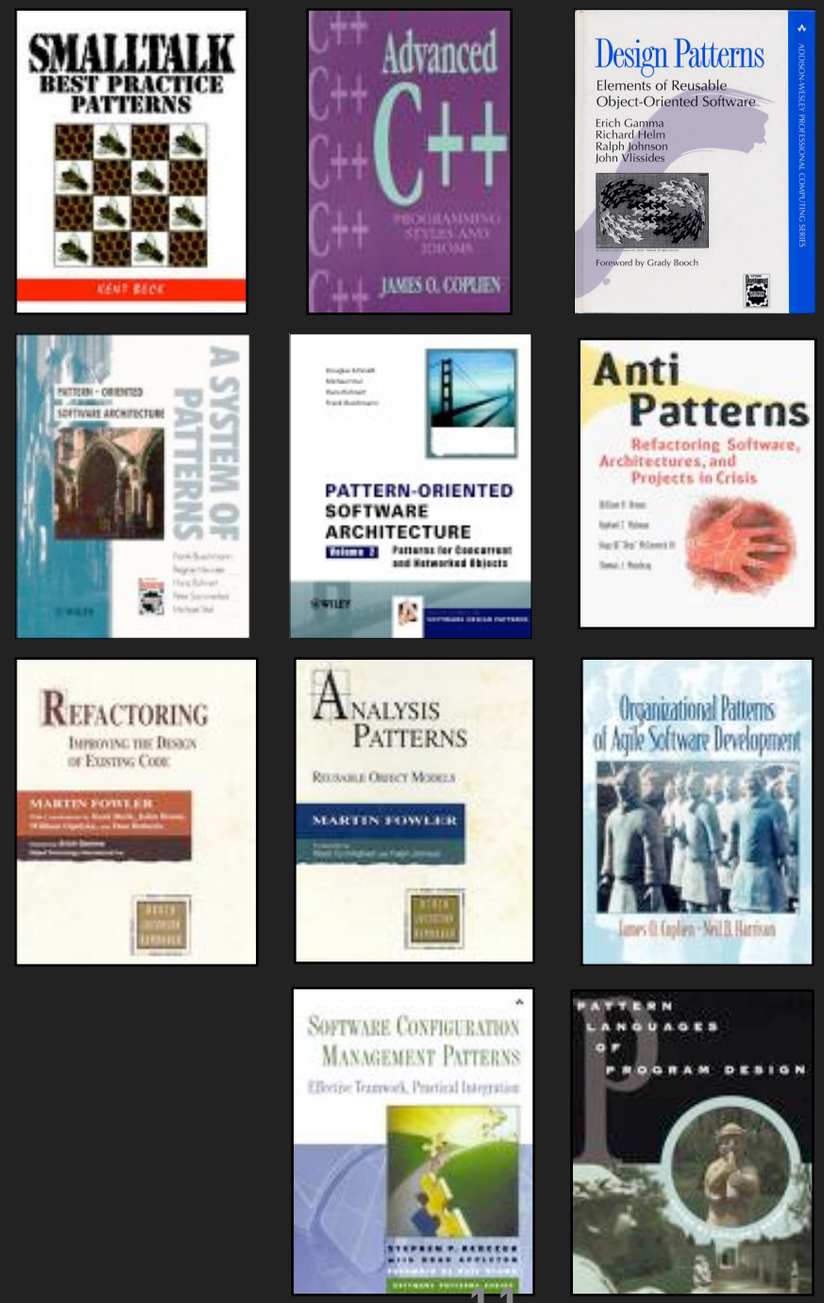
Bad Smells and Refactoring Patterns

Analysis Patterns

Organisational Patterns

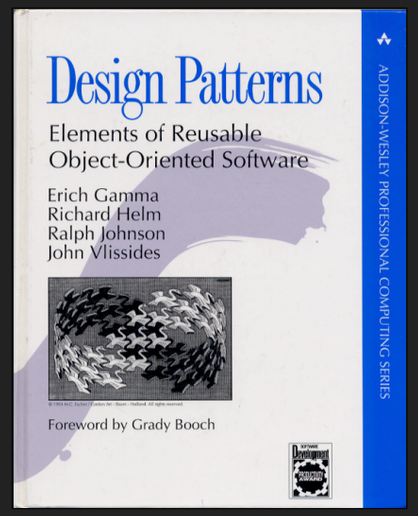
Software Configuration Management Patterns

Pattern Languages of Program Design



SOFTWARE DESIGN & ARCHITECTURAL PATTERNS

E. Gamma, R. Helm, R. Johnson & J. Glissades. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.



F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad & M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

D. Schmidt, M. Stal, H. Rohnert & F. Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Wiley, 2001.

WHY USE ARCHITECTURAL & DESIGN PATTERNS ?

Patterns provide guidance in the analysis & design process.

- *Capture proven* (design) *solutions* from prior experiences.
- *Better structure and design*, *improved confidence* in design.
  - Improve key *software quality* factors.
    - Maintainability, *reusability, extensibility*, … => better products.
    - Favour reuse at analysis, architecture & design level.

Codify "good" analyses & designs.

- Capture and disseminate know-how.
- Help novices in applying good practices.
- Encourage abstraction.

WHY USE ARCHITECTURAL & DESIGN PATTERNS?

Common vocabulary

Provide explicit name for each pattern, e.g. MVC pattern.

Constitutes a common terminology; reduced complexity.

Within and across teams.

Documentation

Provide explicit representation for each pattern, preserve properties.

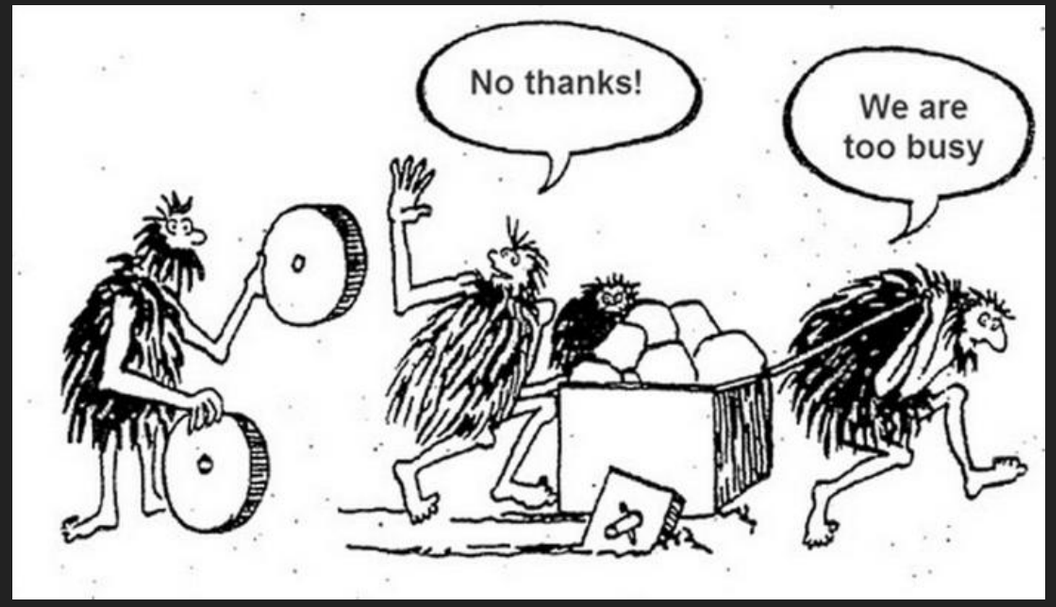
Effective, concise documentation.

DESIGNING REUSABLE OBJECT-ORIENTED SOFTWARE IS HARD

Reusable solutions should solve a specific problem, but at the same time be general enough to address future problems and requirements.

Everything boils down to:

- Aha-erlebnis / déjà-vu feeling
- Don’t reinvent the wheel
- But don’t reinvent the flat tire either ;-)



LINGI2252 – PROF. KIM MENS

C. ARCHITECTURAL SOFTWARE PATTERNS

DESIGN AND ARCHITECTURAL PATTERNS

Document "proven" solutions to a particular design problem.

Are discovered, not invented:

Takes time to emerge, trial and error;

Requires experience;

Based on practical solutions from existing applications.

Example:

Architectural pattern: Model-View-Controller

Design patterns: Factory, Visitor, Composite, Iterator, …

ARCHITECTURAL STYLES VS. ARCHITECTURAL PATTERNS

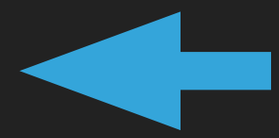
Idiomatic templates for software architecture fragments

Architectural style: more global

Examples: pipes-and-filters, client-server, layered

Architectural pattern: more local

Example: Model-View-Controller



A template made of pre-arranged:

modules / components

relationships / connectors

EXAMPLE: THE MVC PATTERN (MODEL–VIEW–CONTROLLER)

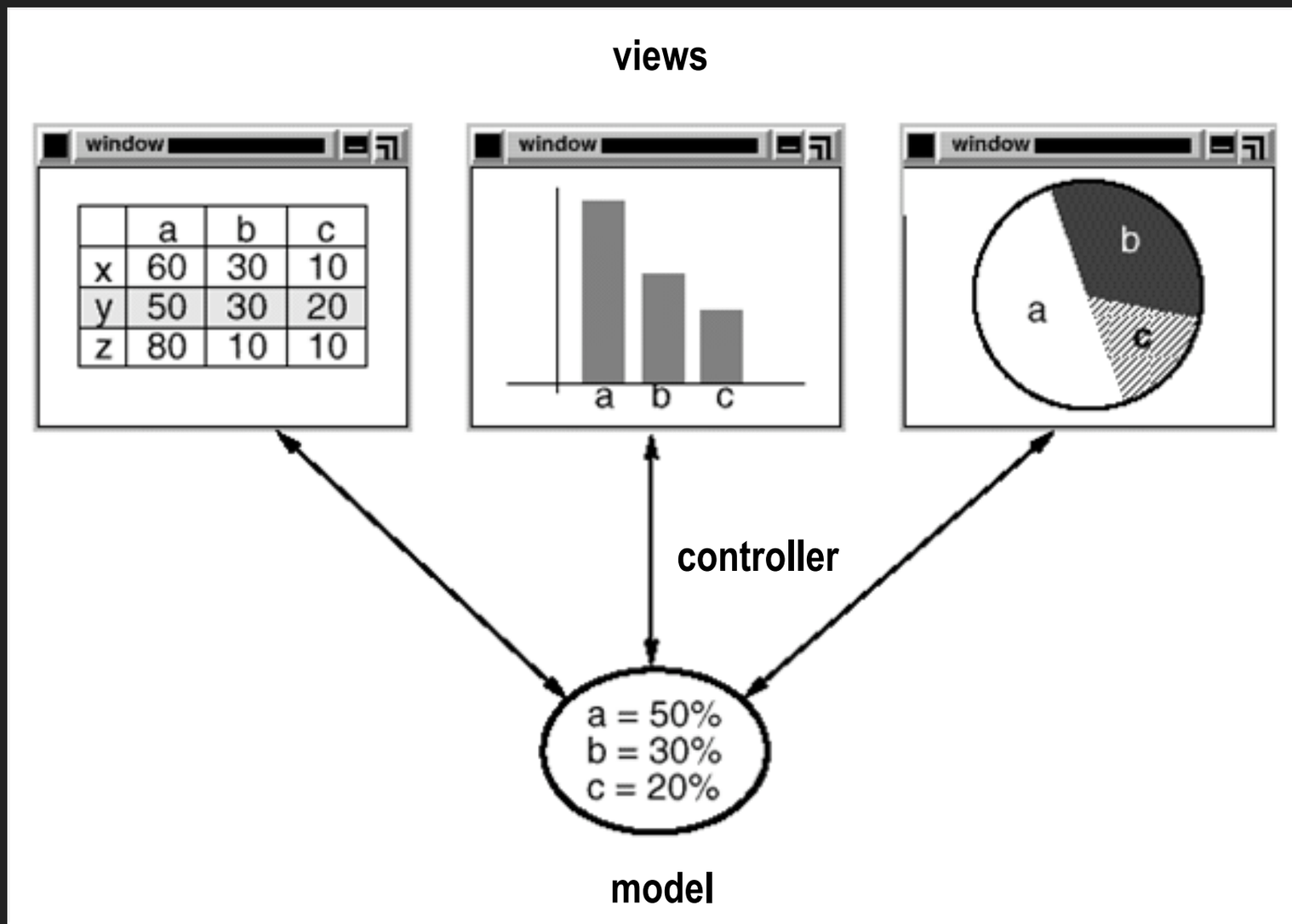
Goal: uncouple application and user interaction

Applicability: any interactive application

Example:

3 main components:

- model
- view
- controller



EXAMPLE: THE MVC PATTERN

"Model" component

provides *functional core* of the application

registers dependent views and controllers

notifies dependent components *about data changes*

EXAMPLE: THE MVC PATTERN

"View" component

creates & initialises its associated controller

displays information to the user

retrieves data from model

implements updates for change propagation

EXAMPLE: THE MVC PATTERN

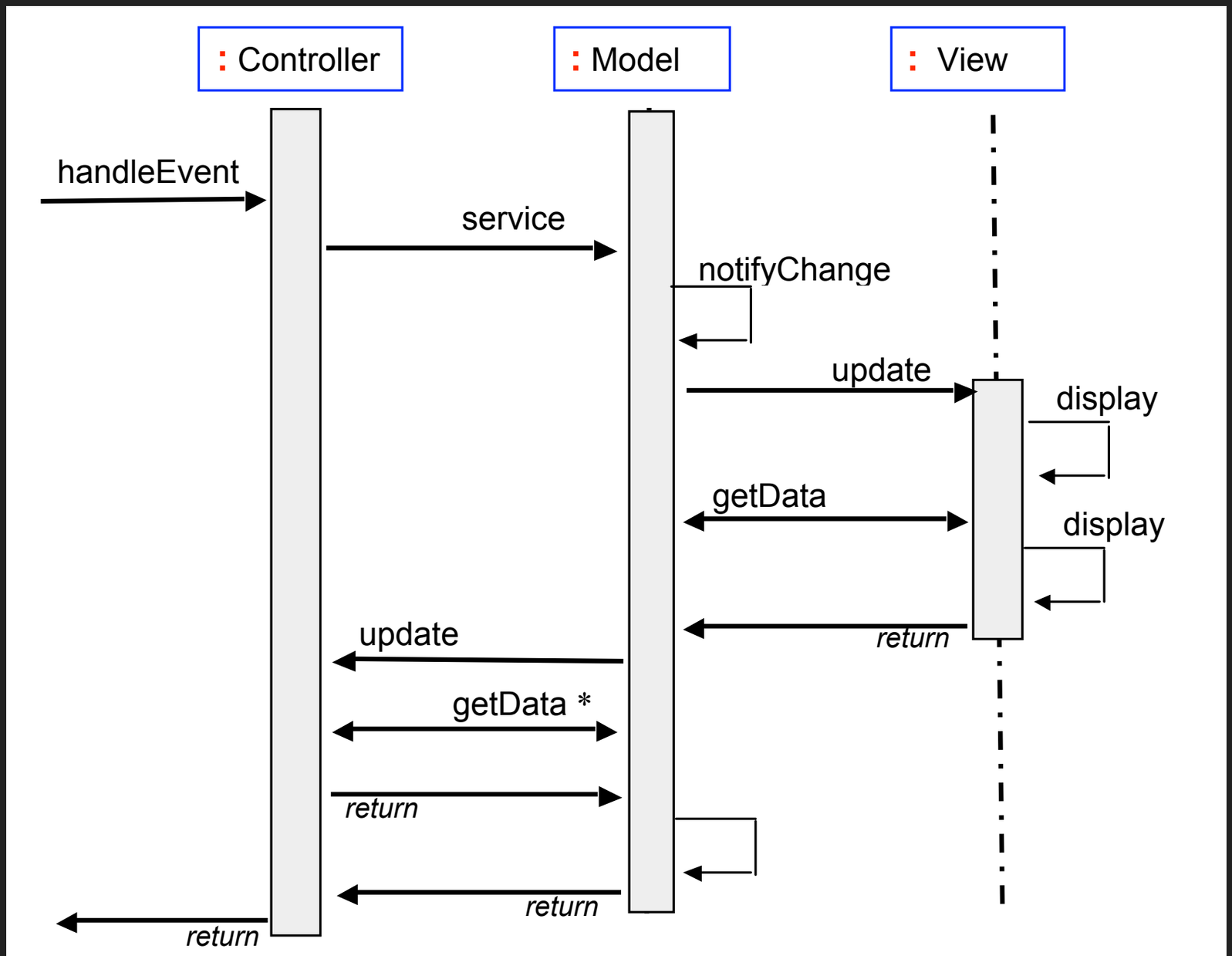
"Controller" component

accepts user inputs as events

translates events to service requests for model or display requests for view

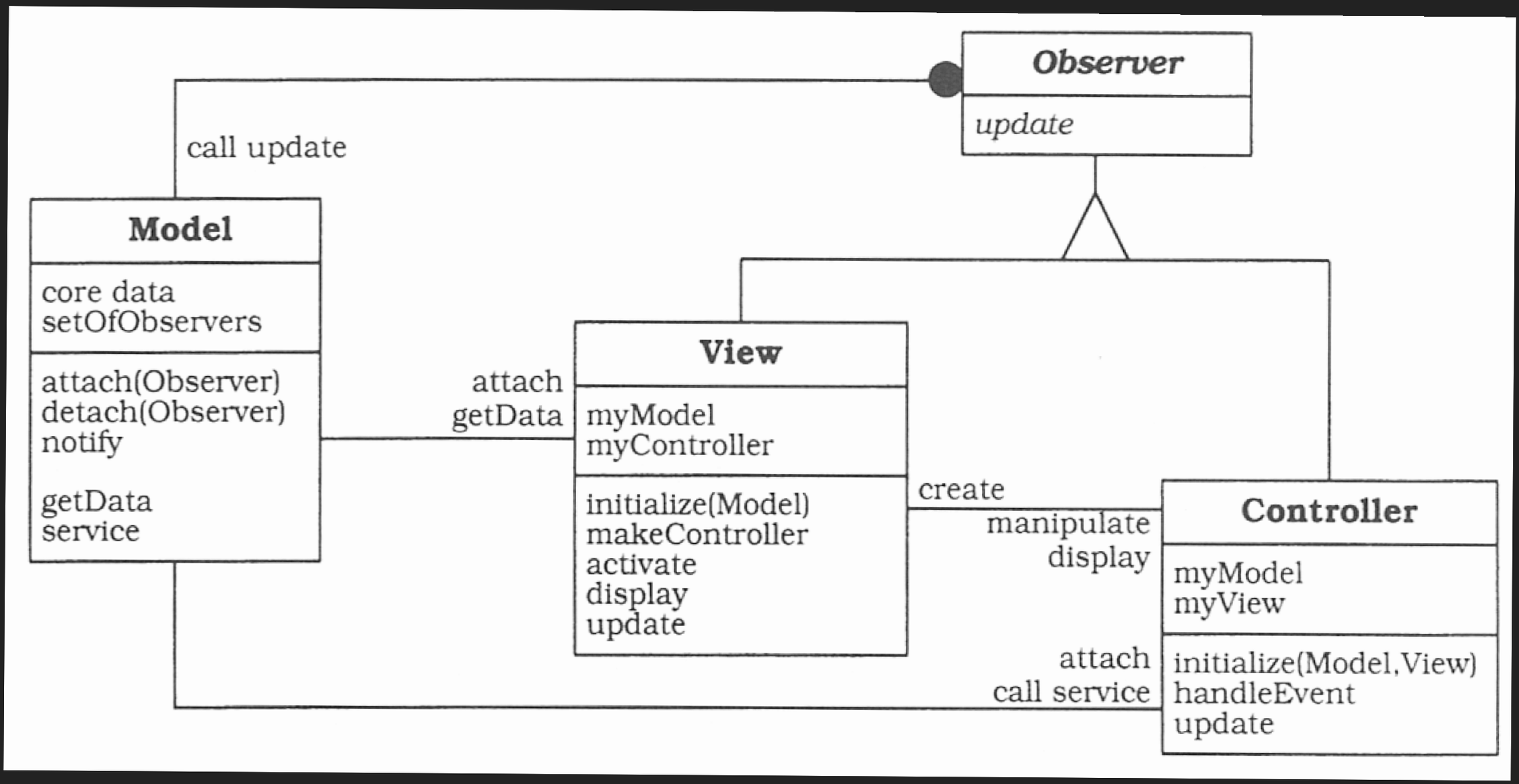
implements updates for change propagation (if required)

THE MVC PATTERN: INTERACTION PROTOCOL



* to enable/disable user functions
(e.g. save when data change)

THE MVC PATTERN: STRUCTURE



Add note that this slides are largely based on Tom’s slides.

LINGI2252 – PROF. KIM MENS

D. DESIGN PATTERNS*

* Slides party reused from slides by Prof. Tom Mens at UMons, Belgium

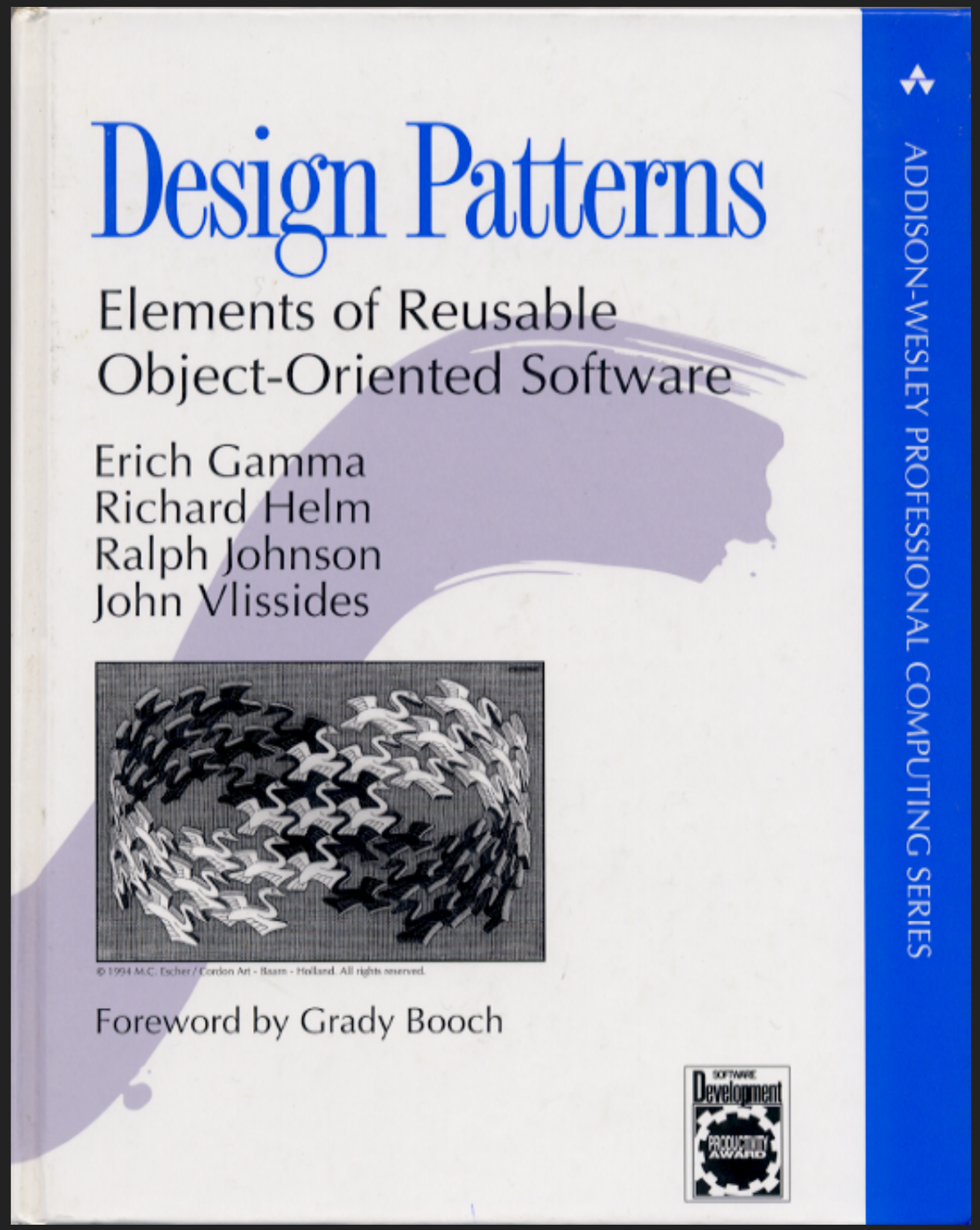
KEY REFERENCE

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
(a.k.a. *the "Gang of Four"* or "GoF")

Addison-Wesley, 1995
ISBN: 0-201-63361-2



ONLINE REFERENCES

http://www.tutorialspoint.com/design_pattern/index.htm

http://www.oodesign.com/

https://sourcemaking.com/design_patterns

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/ (in french)

WHAT IS A DESIGN PATTERN?

> DESIGN PATTERNS ARE DESCRIPTIONS OF COMMUNICATING OBJECTS AND CLASSES THAT ARE CUSTOMISED TO SOLVE A GENERAL DESIGN PROBLEM IN A PARTICULAR CONTEXT
>
> Gang of Four

SOFTWARE DESIGN PATTERNS

Describe in a reusable way

a recurring design problem and

a best practice solution to this problem.

Are about *reusing* good designs and solutions

that have worked previously for similar problems.

The *intent* or *rationale* of a design pattern is a crucial part of its definition

as well as its applicability, trade-offs, consequences, related patterns.

SOFTWARE DESIGN PATTERNS

Identify participating classes and objects, their roles and collaborations, distribution of responsibilities.

Described in an abstract way:

- abstract away from concrete designs;
- the way a particular design pattern is implemented may vary.

Based on practical solutions discovered in main-stream applications implemented in Smalltalk, Java and C++.

DESIGN COVERAGE

- Large portion of design covered by patterns
  - Most classes play role in some patterns
  - Improved understanding
- Seductively simple to implement patterns
- You still have to write functionality
  - Common mistake is to think design patterns solve all your problems

A FIRST EXAMPLE: THE ABSTRACT FACTORY DESIGN PATTERN

Problem

Solution

Participants

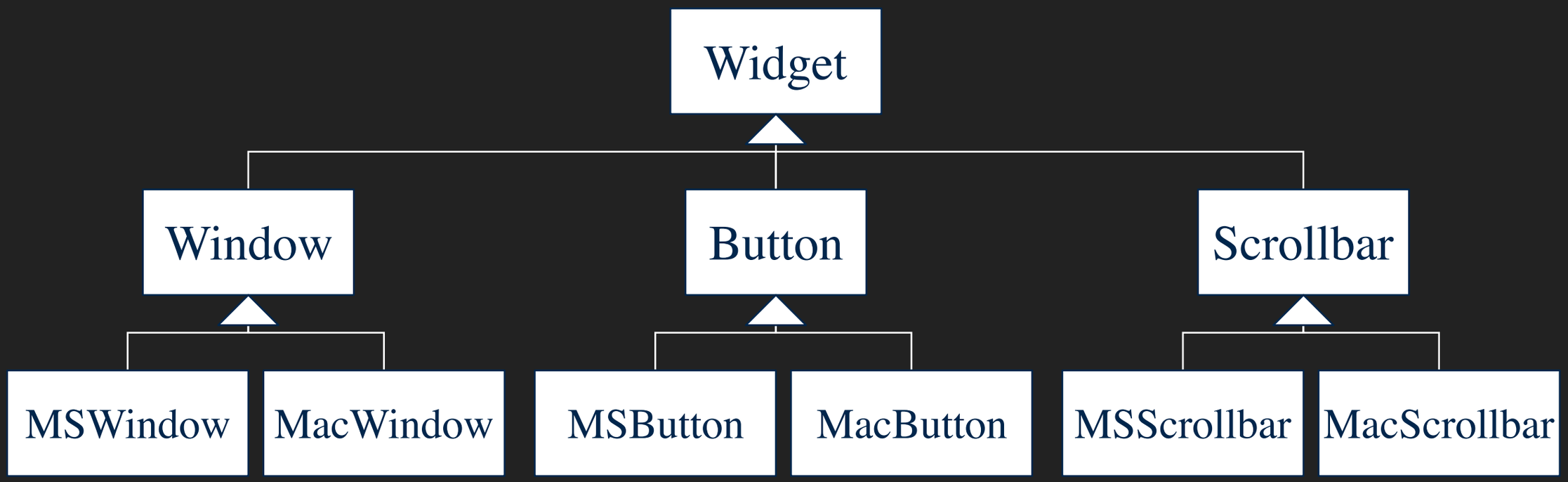
Structure

Applicability

ABSTRACT FACTORY: THE PROBLEM

Entangled hierarchies

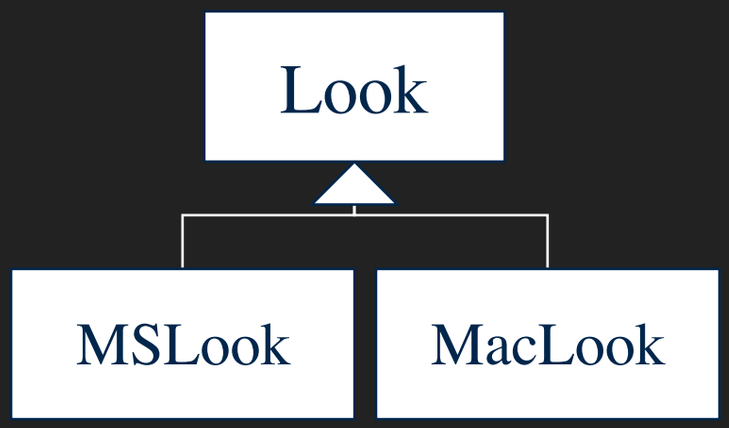
- behaviour and visualisation of UI objects are combined in same inheritance structure

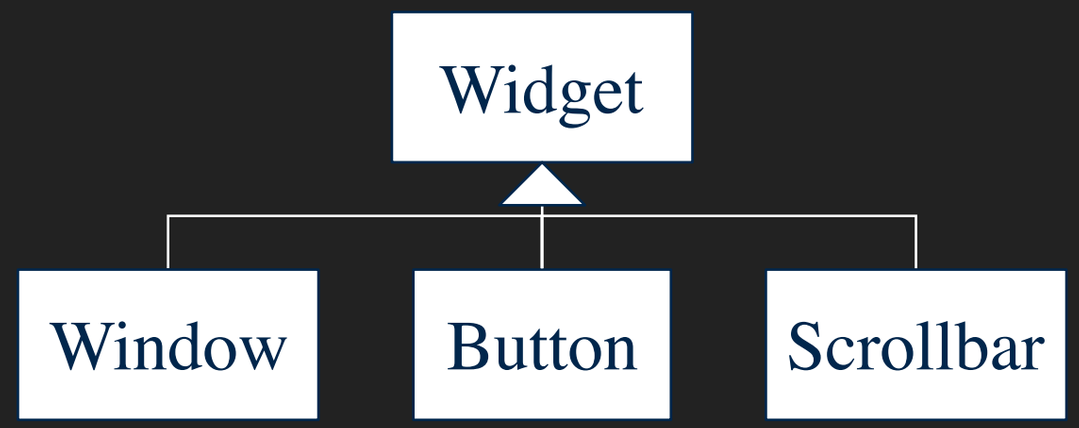


ABSTRACT FACTORY: THE SOLUTION

Idea of the solution

separate behaviour of UI objects from their visualisation

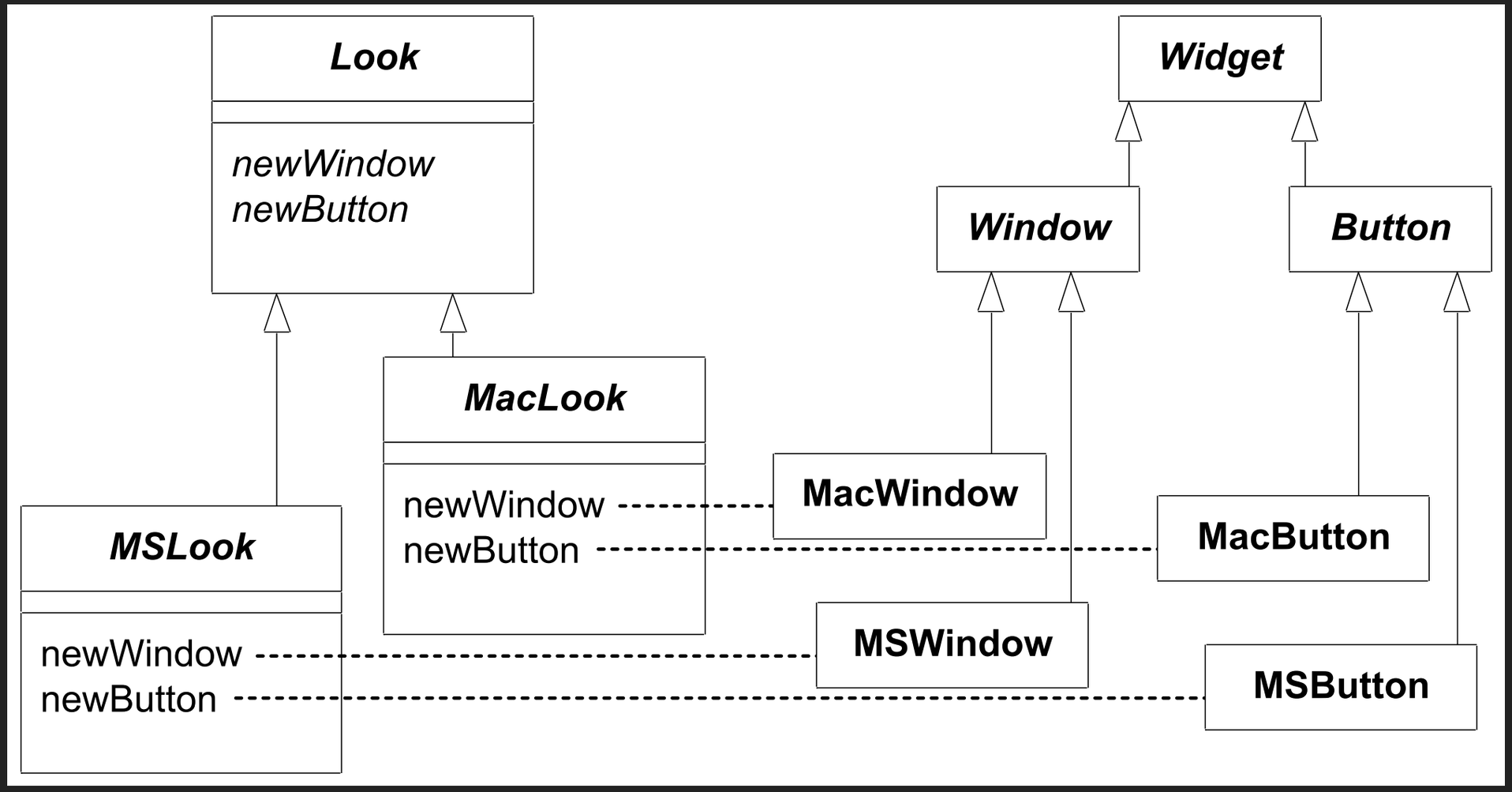




ABSTRACT FACTORY: THE SOLUTION

The *Abstract Factory* design pattern

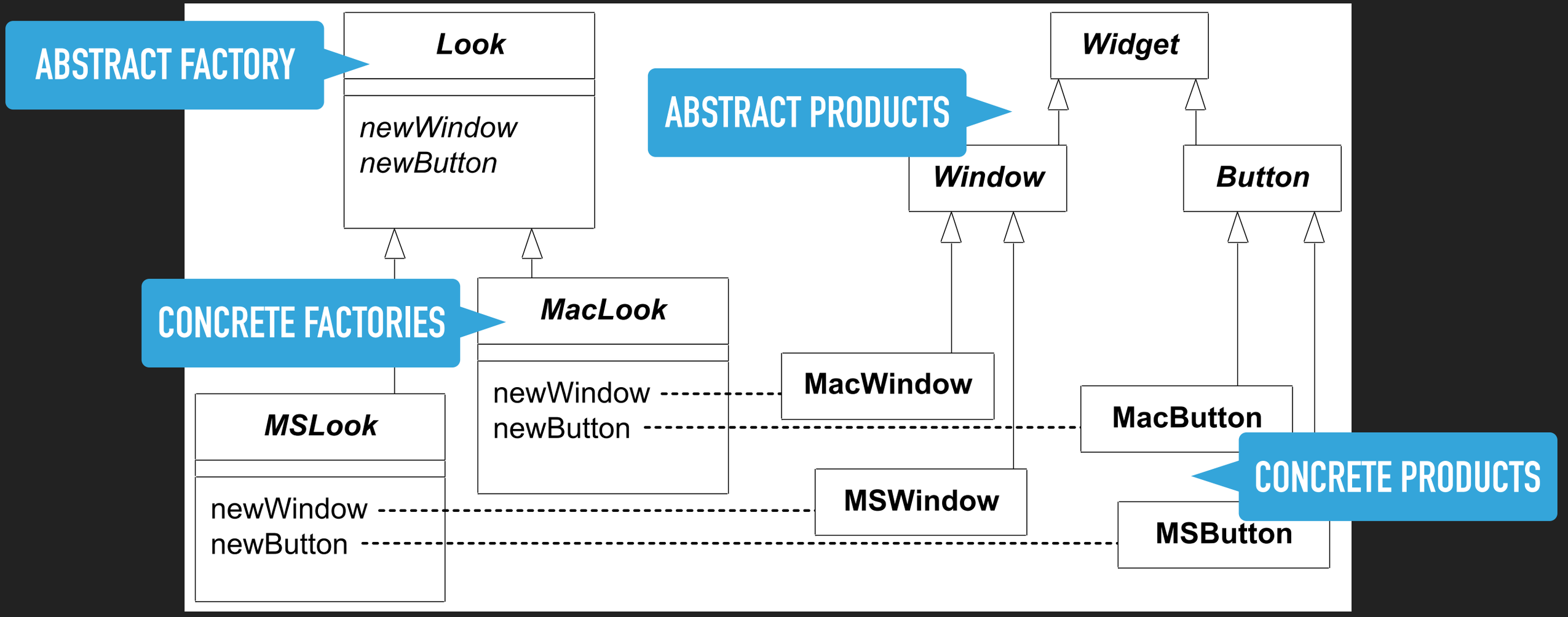
separate behaviour of UI objects from their visualisation



ABSTRACT FACTORY: PARTICIPANTS

The *Abstract Factory* design pattern

separate behaviour of UI objects from their visualisation



ABSTRACT FACTORY: PARTICIPANTS

AbstractFactory (Look)

declares an interface for operations that create abstract product objects

ConcreteFactory (MSLook, MacLook)

implements the operations to create concrete product objects

AbstractProduct (Window, Button)

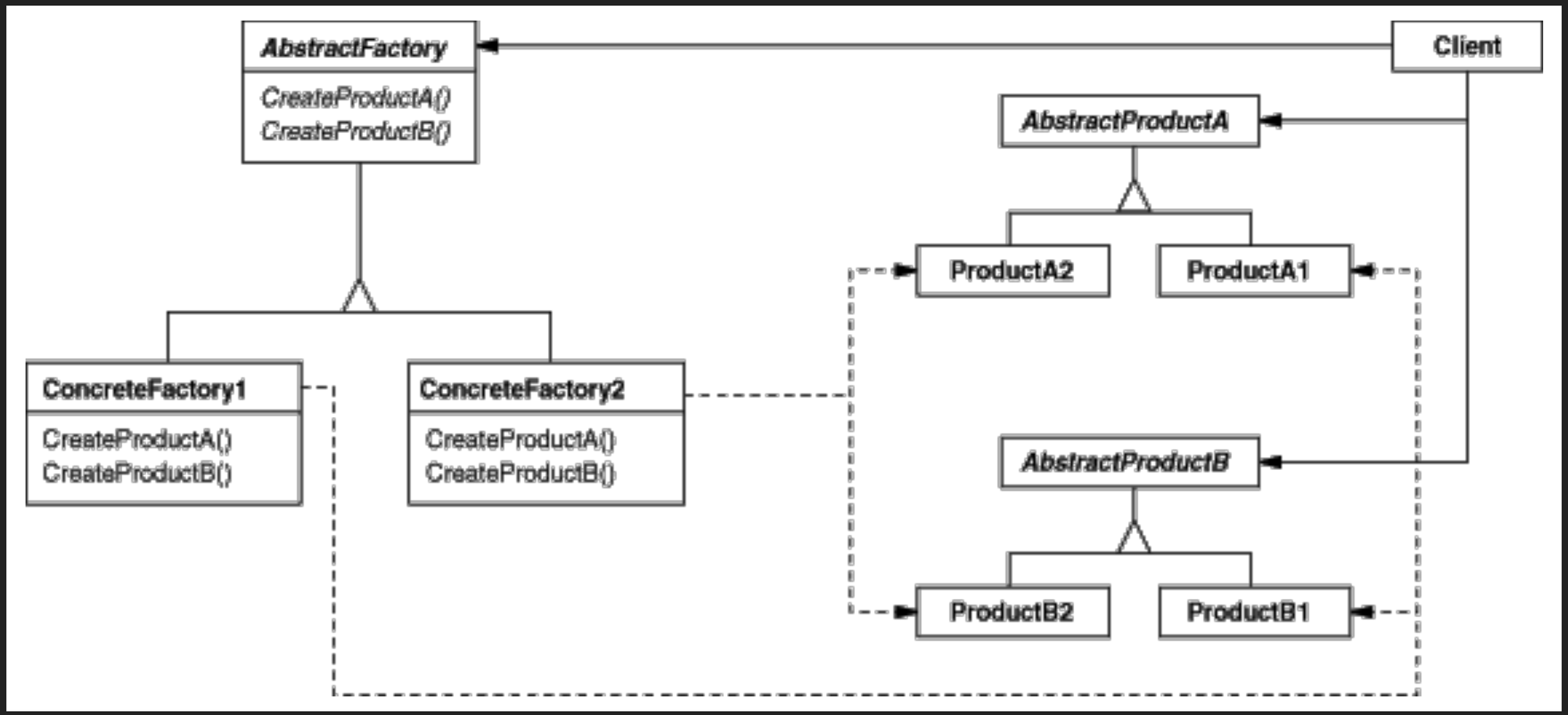
defines a product object to be created by the corresponding concrete factory

ConcreteProduct (MacWindow, MSButton, …)

implements the AbstractProduct interface

Client uses only interfaces declared by AbstractFactory and AbstractProduct classes

ABSTRACT FACTORY: STRUCTURE



Normally a single instance of ConcreteFactory is created at run-time

AbstractFactory defers creation of product objects to its ConcreteFactory subclass

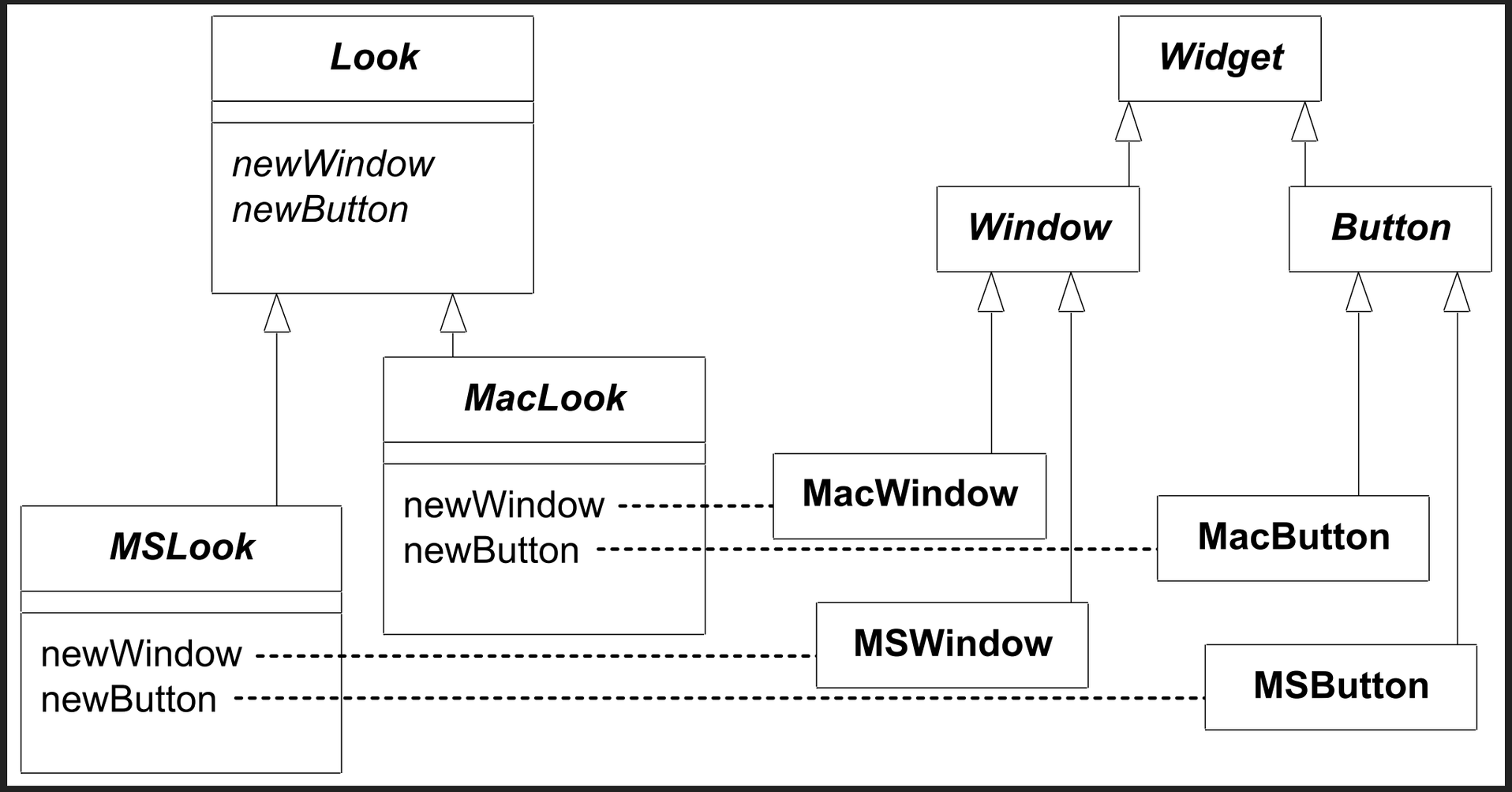
ABSTRACT FACTORY: APPLICABILITY

This solution can be applied in general when

- a system should be independent of how its products are created, composed, and represented
- a system should be configured with families of products
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations
- a family of related product objects is designed to be used together, and you need to enforce this constraint

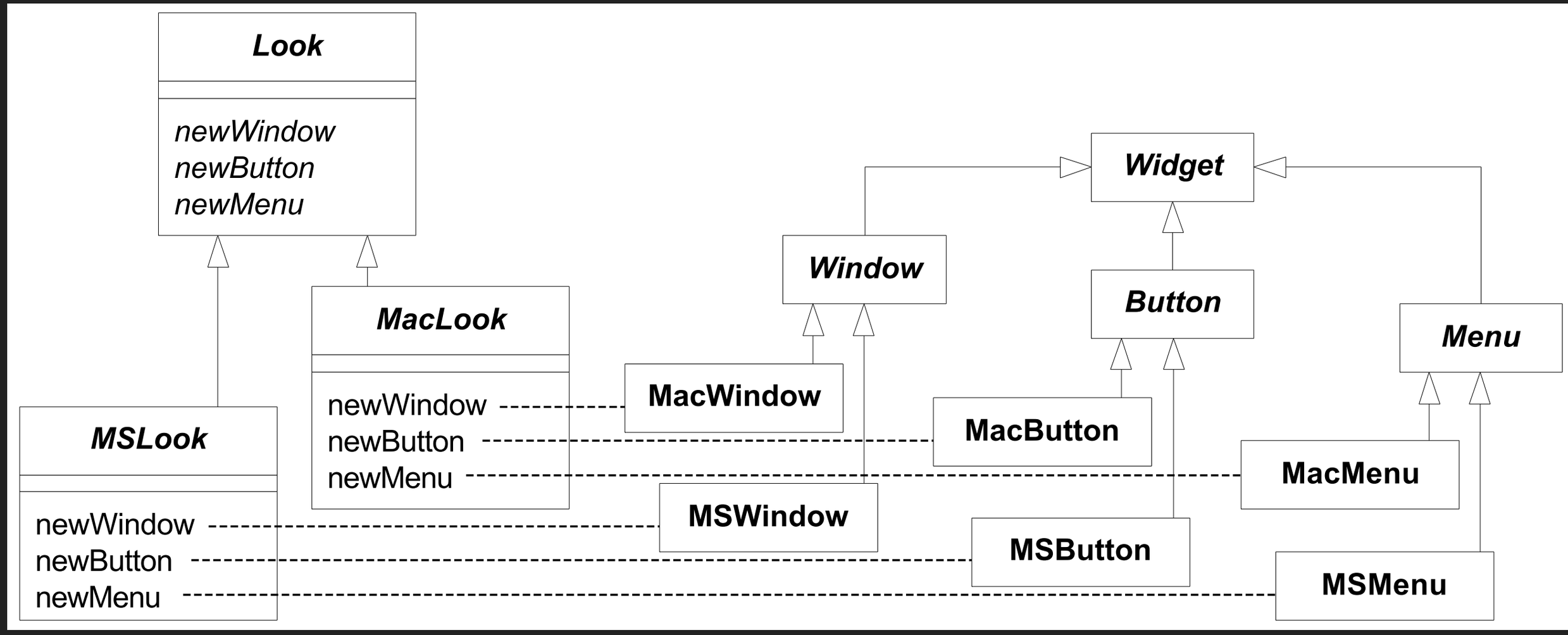
ABSTRACT FACTORY: EVOLVABILITY (1)

Adding a new product Menu



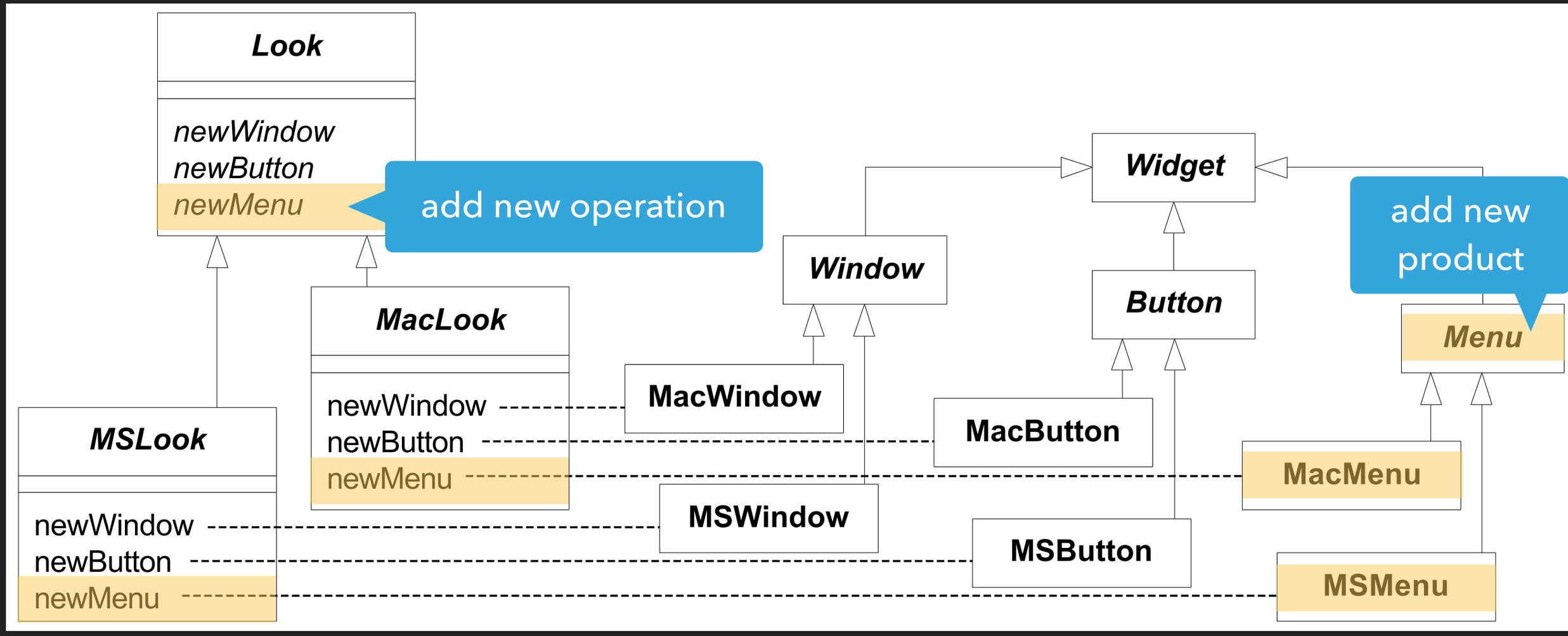
ABSTRACT FACTORY: EVOLVABILITY (1)

Adding a new product Menu



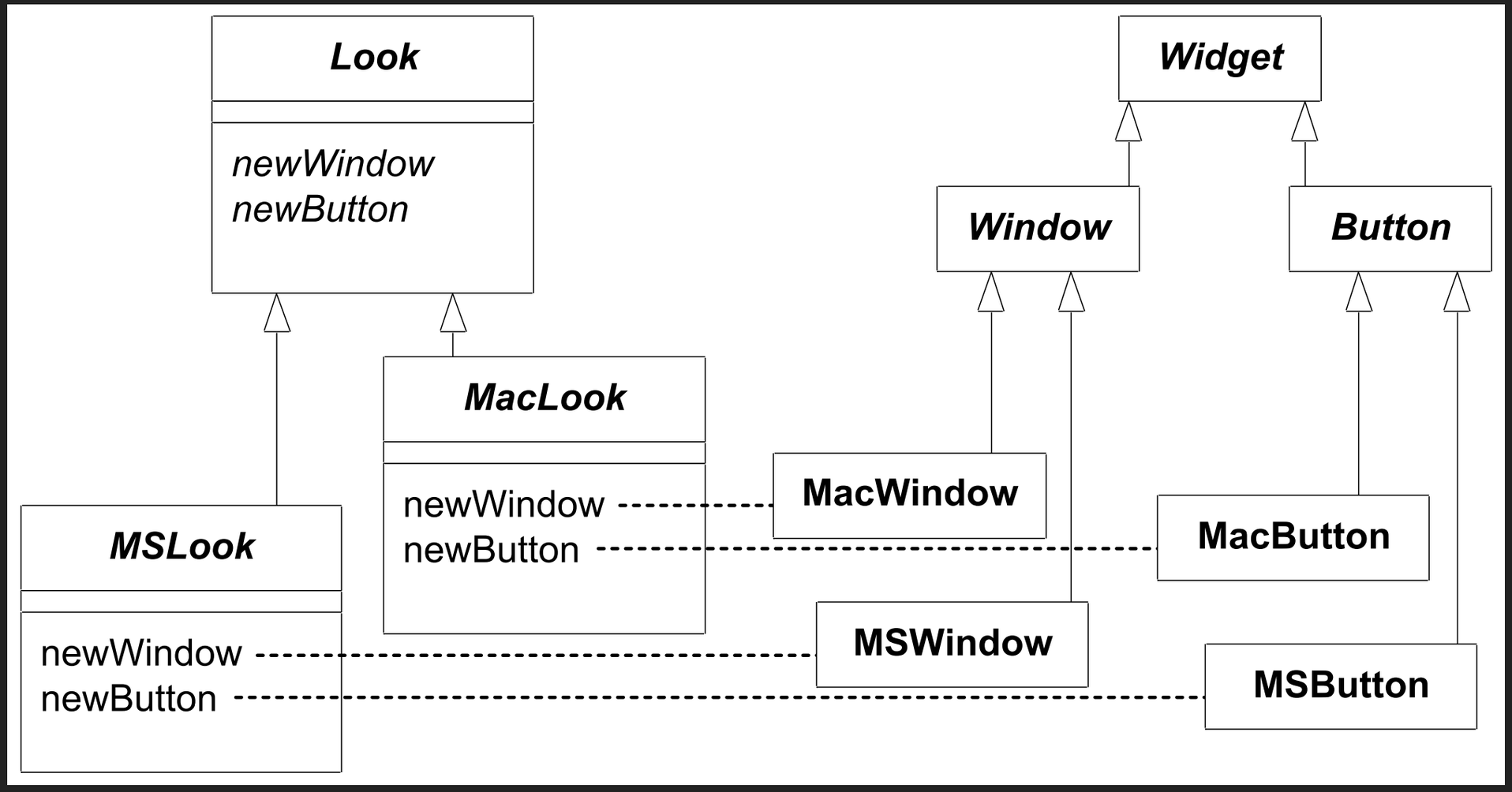
ABSTRACT FACTORY: EVOLVABILITY (1)

Adding a new product Menu



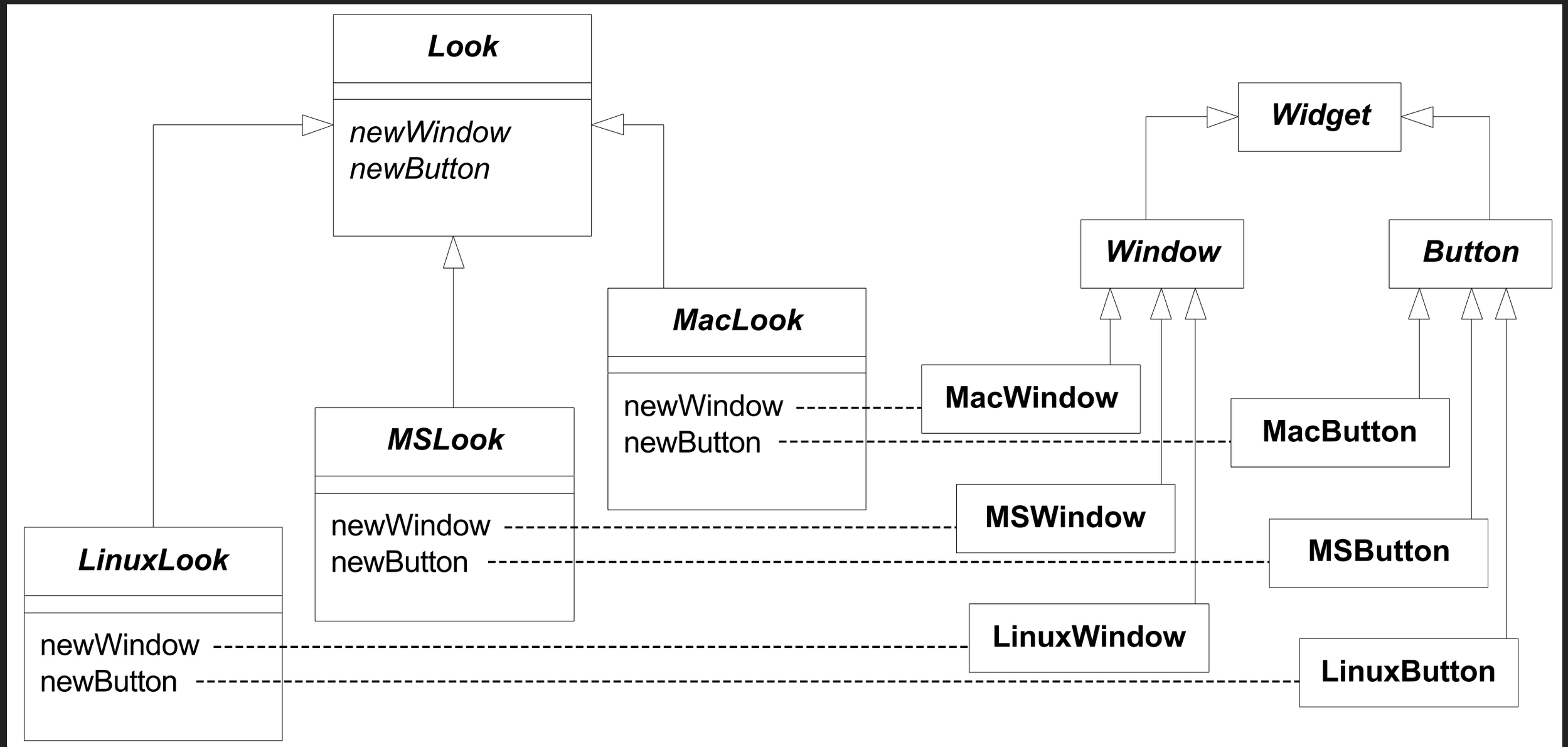
ABSTRACT FACTORY: EVOLVABILITY (2)

Adding a new concrete factory LinuxLook



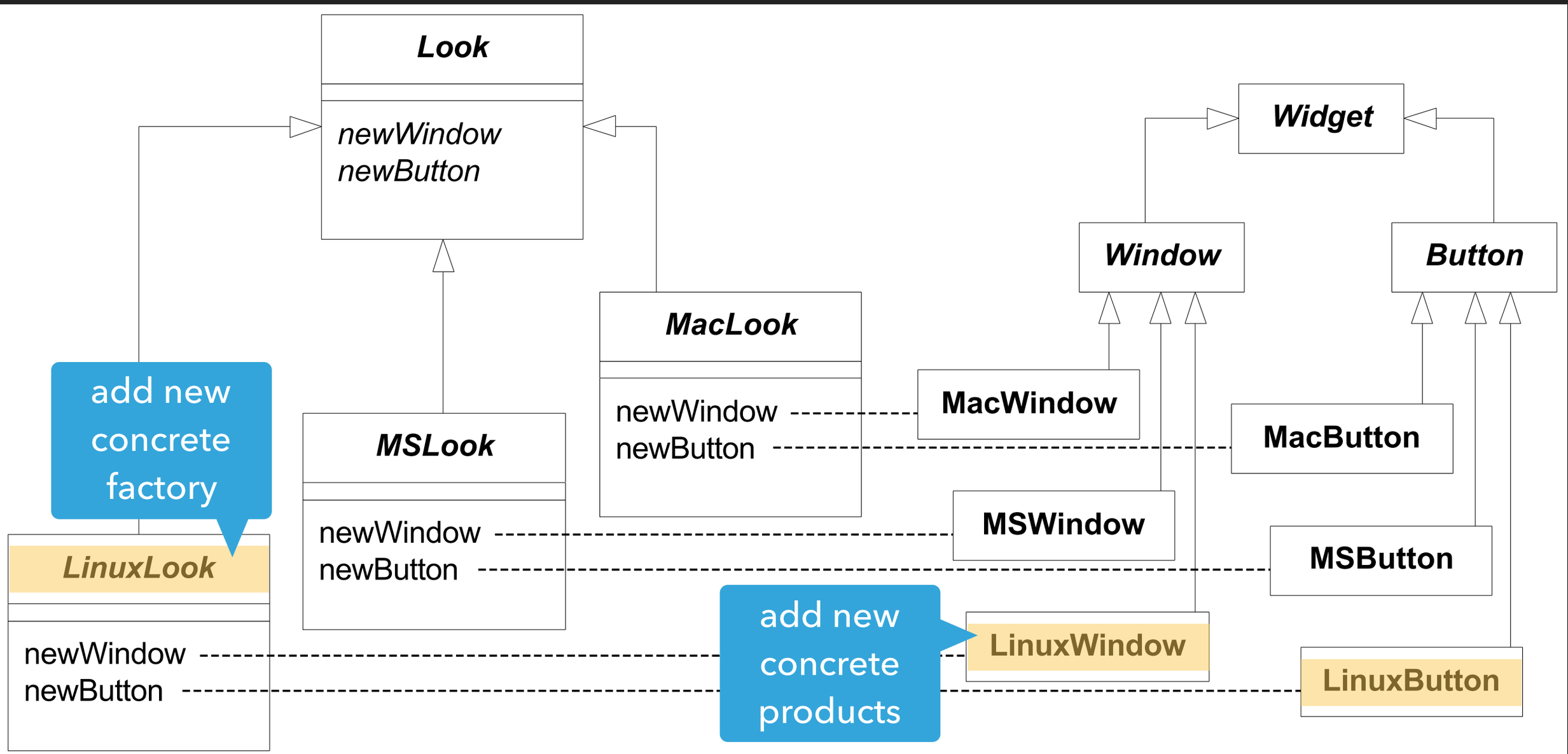
ABSTRACT FACTORY: EVOLVABILITY (2)

Adding a new concrete factory LinuxLook



ABSTRACT FACTORY: EVOLVABILITY (2)

Adding a new concrete factory LinuxLook



LINGI2252 – PROF. KIM MENS

E. SELECTED DESIGN PATTERNS*

* Slides party reused from slides by Prof. Tom Mens at UMons, Belgium

A SELECTION OF DESIGN PATTERNS

- Design pattern catalogue
- Factory Method
- Template Method
- Strategy
- Observer
- Decorator
- Visitor
- Singleton
- Composite
- Builder
- Iterator

DESIGN PATTERN CATALOGUE

DESIGN PATTERN CATALOGUE . . .

Design Patterns can be classified based on two criteria:

Purpose

what a pattern *does*

Scope

whether the pattern applies primarily to classes or to objects

DESIGN PATTERN CATALOGUE : PURPOSE [GOF]

Creational design patterns

are concerned with the process of *object creation*

Structural design patterns

are concerned with how to compose classes and objects in larger structures

Behavioural design patterns

are concerned with algorithms and the separation of responsibilities between objects or classes

DESIGN PATTERN CATALOGUE : SCOPE

Class Patterns

deal with relationships between classes (and their subclasses)

relationships are static since they are fixed at compile time

Object Patterns

deal with relationships between objects

relationships are more dynamic since they can be changed at run-time

DESIGN PATTERN CATALOGUE . . .

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	**Factory Method**	Adapter (Class)	Interpreter Template Method
	Object	**Abstract Factory** Builder Prototype Singleton	Adapter (Object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

CREATIONAL PATTERNS

Abstract the instantiation/object creation process

- *Encapsulates* knowledge about which concrete classes to use
- *Hides* how class instances are created and put together

Give a lot of flexibility in *what*, *who*, *how* and *when* things get created

A creational class pattern

- typically uses inheritance to *vary* the class to be instantiated
- and delegates instantiation to another object

STRUCTURAL PATTERNS

How to compose classes and objects to form larger structures

- When/how should you use *inheritance*?
- When/how should you use *aggregation*, *association*, *composition*?

A *structural* ***class*** *pattern* uses *inheritance* to compose interfaces or implementation

- Compositions are fixed *at compile time*

A *structural* ***object*** *pattern* describes ways to compose objects to realise new functionality

- Added flexibility: ability to change the composition *at run-time*

BEHAVIOURAL PATTERNS

Concerned with algorithms and the assignment of responsibilities between objects

A *behavioural* ***class*** *pattern* uses *inheritance* to distribute behaviour between classes

A *behavioural* ***object*** *pattern* uses object composition rather than inheritance

- describes how groups of objects cooperate to perform tasks no single object could carry out by itself
- is concerned with encapsulating behaviour in an object and delegating requests to it

DESCRIPTION OF A DESIGN PATTERN

- Catchy name — A DESCRIPTIVE AND UNIQUE NAME TO HELP IDENTIFY AND REFER TO THE PATTERN
- Classification
- Intent — A DESCRIPTION OF THE GOAL BEHIND THE PATTERN AND THE REASON FOR USING IT
- Also known as — OTHER NAMES FOR THE PATTERN
- Motivation — A SCENARIO CONSISTING OF A PROBLEM AND A CONTEXT IN WHICH THIS PATTERN CAN BE USED
- Applicability — SITUATIONS IN WHICH THIS PATTERN IS USABLE; THE CONTEXT FOR THE PATTERN
- Structure — GRAPHICAL REPRESENTATION OF THE PATTERN. CLASS DIAGRAMS AND INTERACTION DIAGRAMS CAN BE USED FOR THIS PURPOSE
- Participants — LIST OF CLASSES AND OBJECTS USED IN THE PATTERN AND THEIR ROLES IN THE DESIGN
- Collaboration — HOW CLASSES AND OBJECTS USED IN THE PATTERN INTERACT WITH EACH OTHER
- Consequences — RESULTS, SIDE EFFECTS, AND TRADE OFFS CAUSED BY USING THE PATTERN
- Implementation — IMPLEMENTATION OF THE PATTERN; THE SOLUTION PART OF THE PATTERN
- Sample Code — ILLUSTRATION OF HOW THE PATTERN CAN BE USED IN A PROGRAMMING LANGUAGE
- Known Uses — EXAMPLES OF REAL USAGES OF THE PATTERN
- Related Patterns — OTHER PATTERNS THAT HAVE SOME RELATIONSHIP WITH THE PATTERN; DISCUSSION OF THE DIFFERENCES BETWEEN THE PATTERN AND SIMILAR PATTERNS

HOW TO SELECT A DESIGN PATTERN

Study the *intent* section of the design pattern:
what problem does it solve?

Consider *how* the design patterns solves the problem

Study related patterns of similar purpose

Consider what should be variable in your design

Introduce a pattern to *refactor* the design

HOW TO USE A DESIGN PATTERN

Read pattern overview for its Applicability and Consequences

Study the Structure, Participants and Collaboration sections

Look at the Sample Code

Choose names for the Participants (classes and methods) meaningful in your context

Define the classes and appropriate interfaces

Implement the operations to carry out the responsibilities and collaborations in the pattern

WHEN TO USE A DESIGN PATTERN

Avoid overstructuring

Only use design patterns when there is a real *need*.

Introducing design patterns *too early* may be counterproductive.
It makes the design structure unnecessarily complex.

Do *not* waste the effort of introducing a design pattern if you are not sure that it will be needed.

WHEN TO USE A DESIGN PATTERN

Avoid understructuring

If there is a need for design patterns, use them!

Use refactorings to introduce design patterns in existing code.

SELECTED DESIGN PATTERNS

FACTORY METHOD

http://www.oodesign.com/factory-method-pattern.html

A SELECTION OF DESIGN PATTERNS

- Design pattern catalogue
- Factory Method
- Template Method
- Strategy
- Observer
- Decorator
- Visitor
- Singleton
- Composite
- Builder
- Iterator

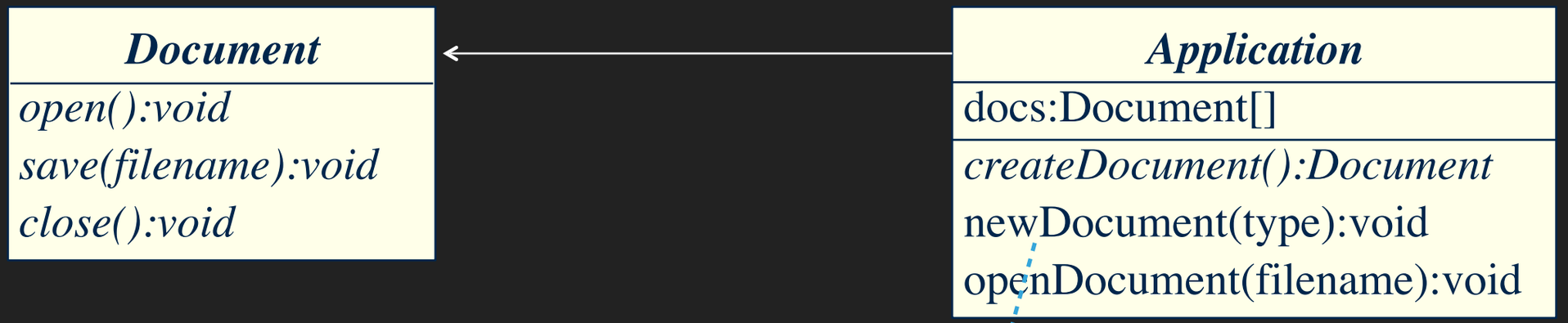
FACTORY METHOD

Classification: Class Creational.

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. *Factory Method* lets a class defer instantiation to subclasses.

Motivating example: Application framework for handling documents in a desktop application.

FACTORY METHOD : MOTIVATION




```
public void newDocument(String type) {
   Document doc = this.createDocument(type);
   docs.add(doc);
   doc.open();
}
```

Consider an application framework for creating desktop applications.

Such applications work with documents.

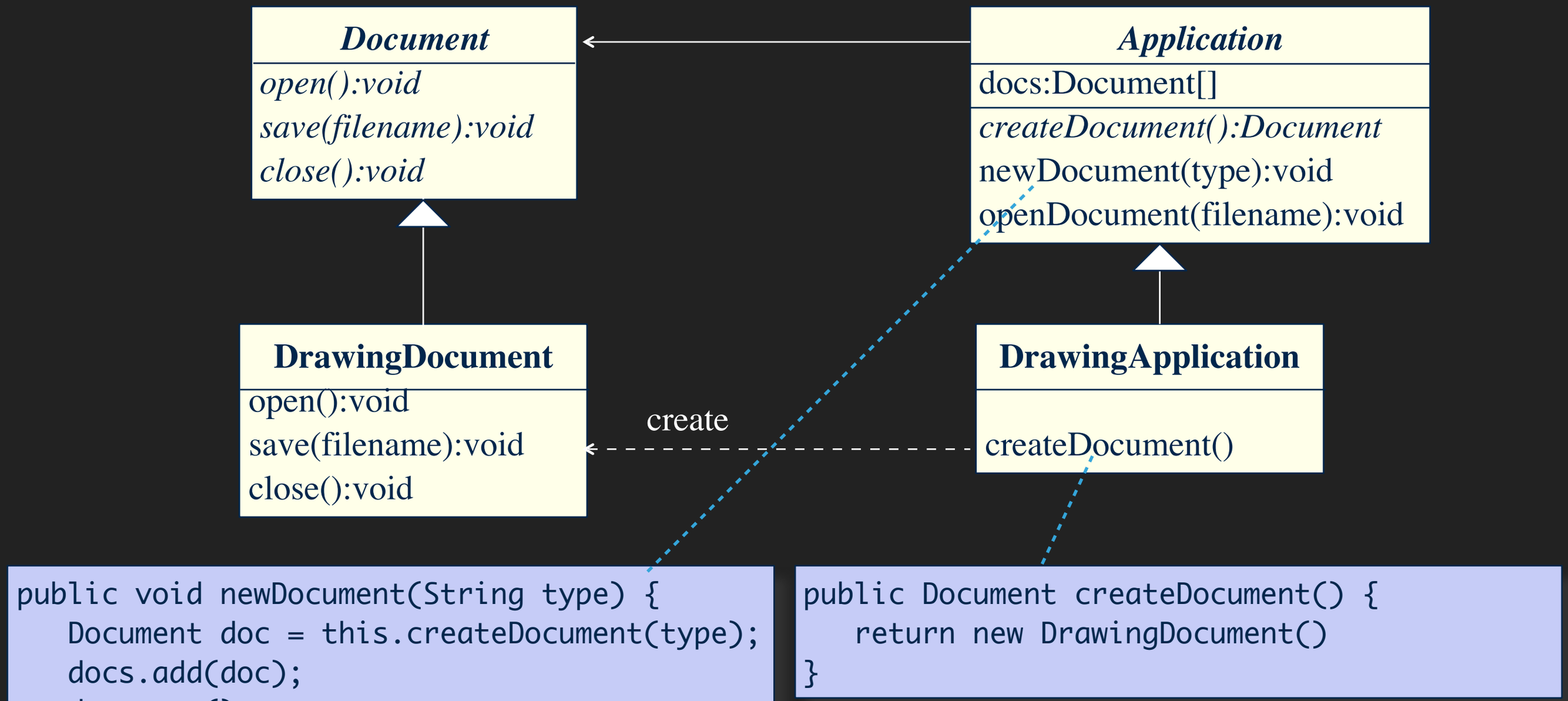
The framework contains basic operations for opening, closing and saving documents.

Specific applications (e.g., subclasses of Application) can create their own kinds of documents (subclasses of Document)

For example, a DrawingApplication creates DrawingDocuments.

To create application-specific documents, the abstract Application class needs to delegate to its subclasses.

FACTORY METHOD : MOTIVATION

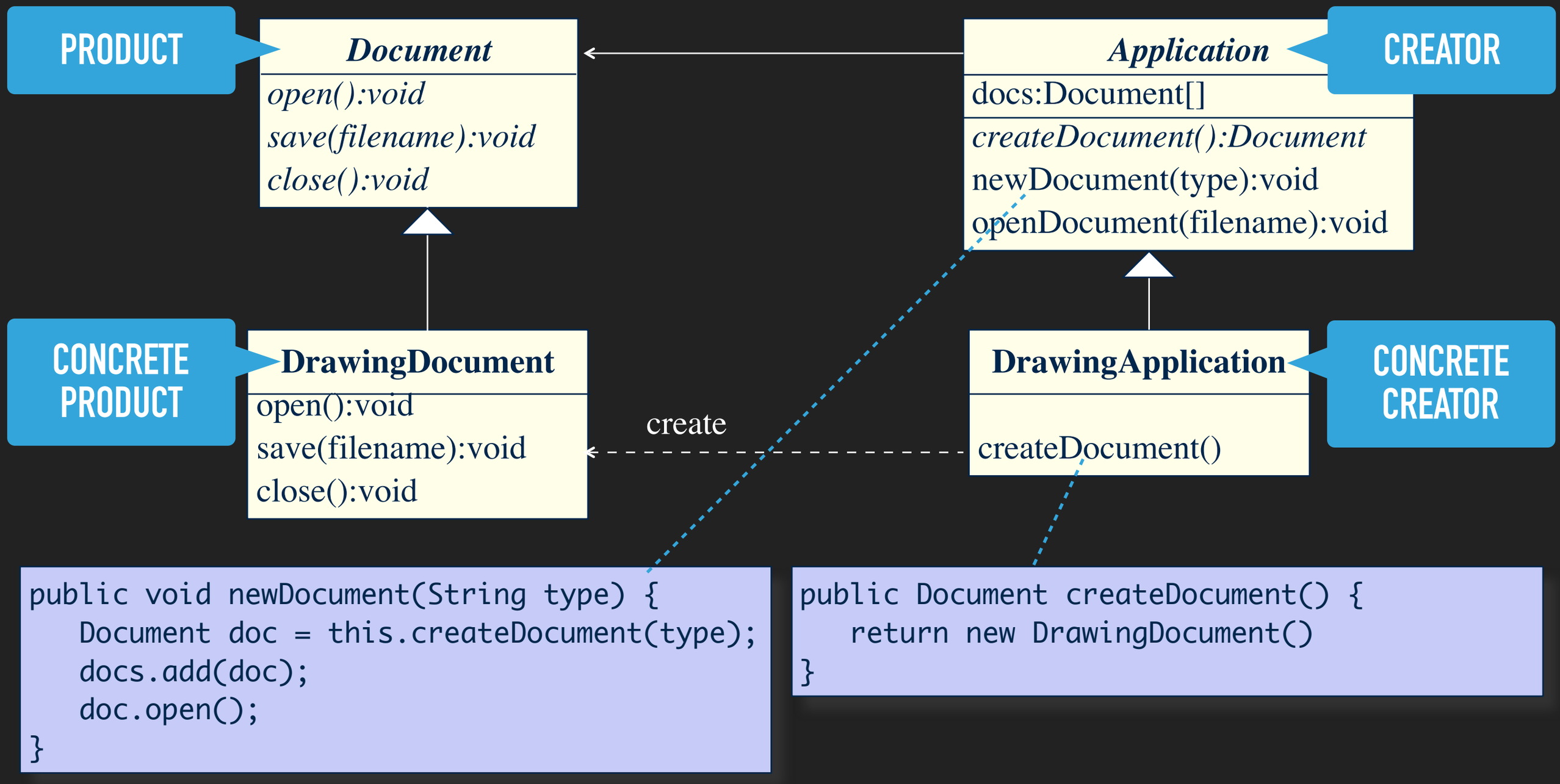


FACTORY METHOD : APPLICABILITY

Use the Factory Method design pattern when:

- A class can’t anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Classes delegate responsibility to one of several helper subclasses, and you want to localise the knowledge of which helper classes to use

FACTORY METHOD : PARTICIPANTS



FACTORY METHOD : PARTICIPANTS

Product (Document)

Defines the interface of objects the factory method creates

ConcreteProduct (DrawingDocument)

Implements the Product interface

Creator (Application)

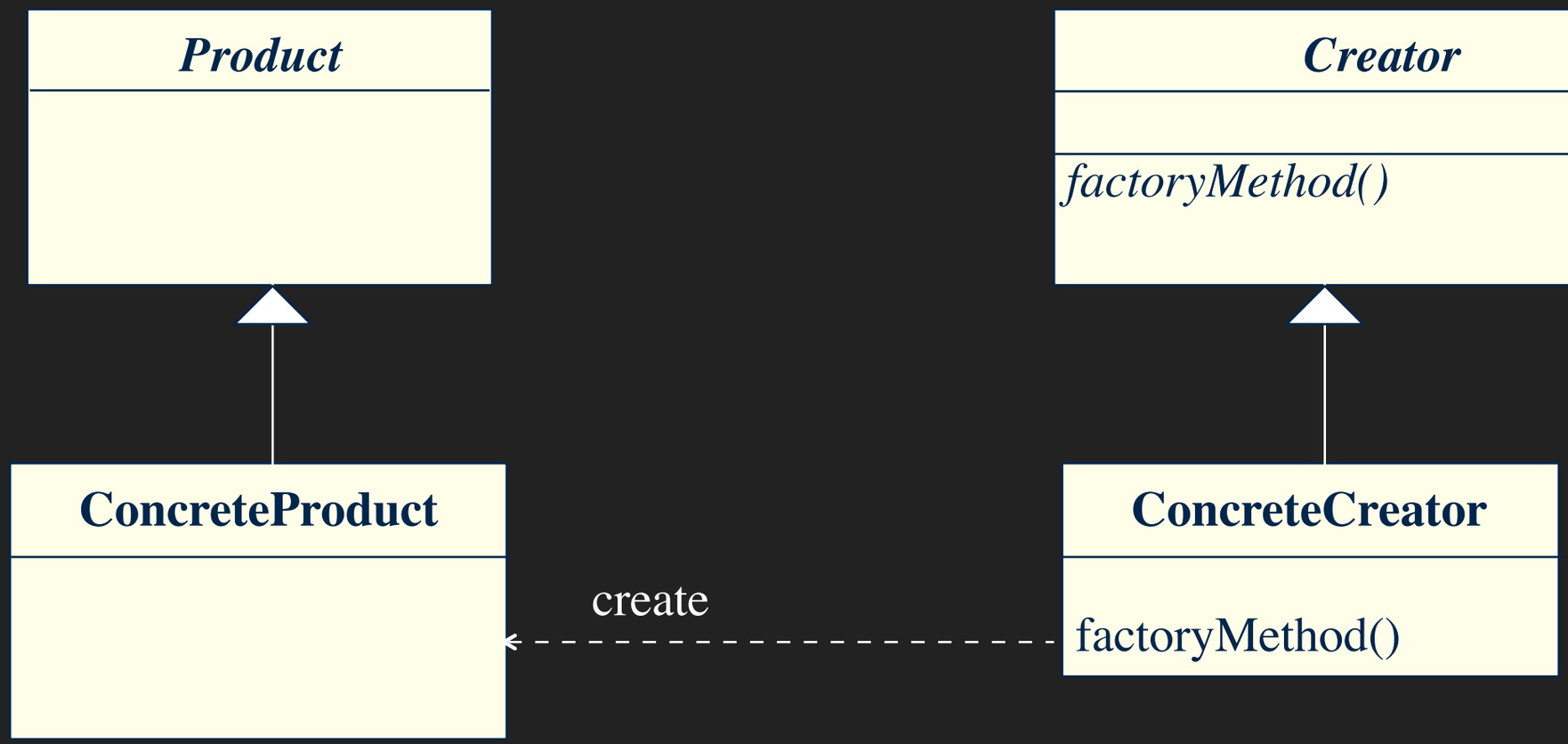
Declares the abstract factory method (createDocument)

May call the factory method to create a Product object

ConcreteCreator (DrawingApplication)

Overrides factory method with a concrete one to return a ConcreteProduct instance

FACTORY METHOD : STRUCTURE



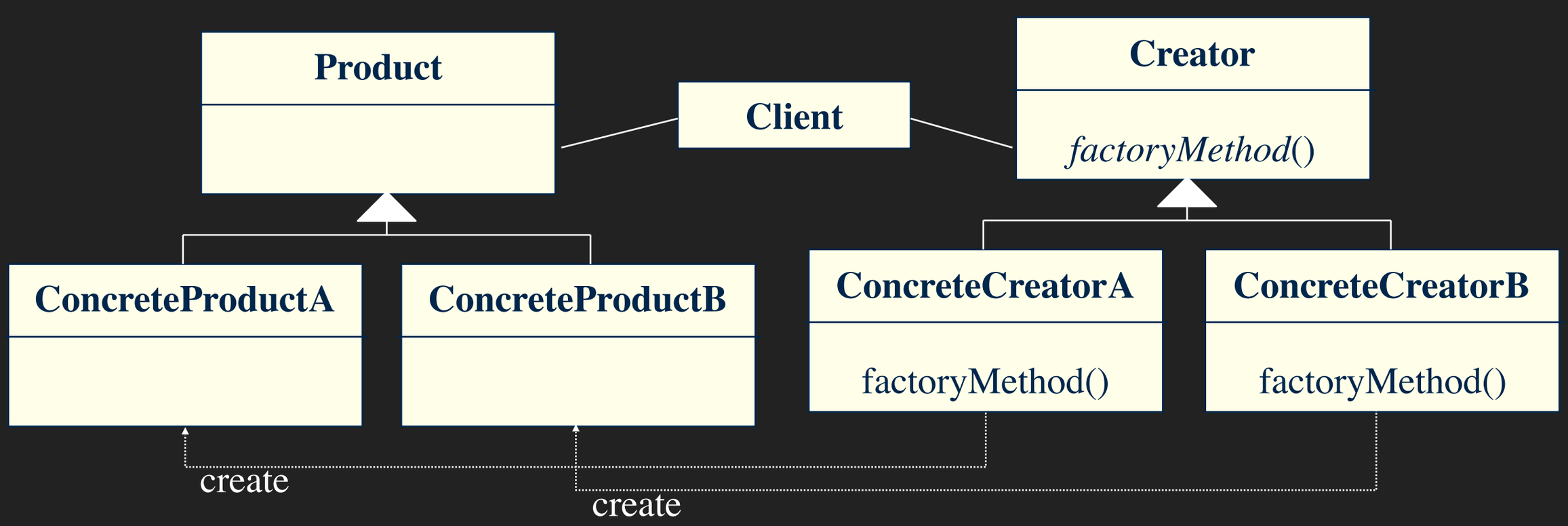
FACTORY METHOD : COLLABORATIONS

Creator relies on its ConcreteCreator subclasses
to specify the concrete factory method so that it returns an
instance of the appropriate ConcreteProduct

FACTORY METHOD: CONSEQUENCES

More reusable than creating objects directly

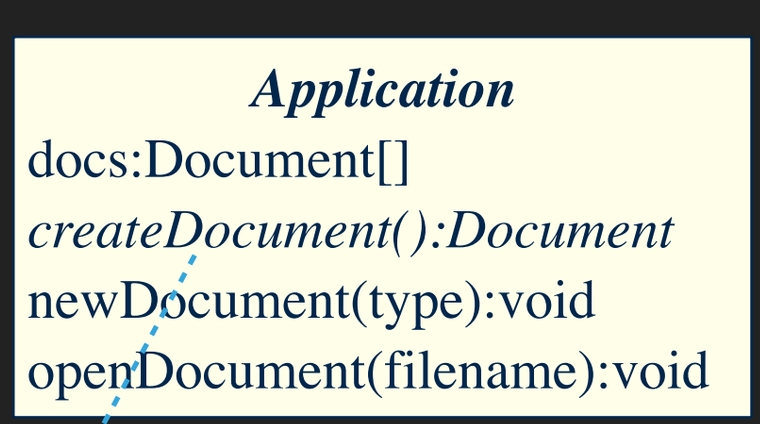
Connects parallel class hierarchies



FACTORY METHOD: IMPLEMENTATION

Variation 1:

- Some factory methods can provide a *default*. In that case the factory method is not abstract.




```
public Document createDocument(String type){
    if (type.isEqual("drawing"))
        return new DrawingDocument();
    …
}
```

FACTORY METHOD: IMPLEMENTATION

Variation 2:

> Factory methods can be *parameterised* to return multiple kinds of products.
>
> e.g. pass an extra parameter to createDocument to specify the kind of Document to create.

SELECTED DESIGN PATTERNS

TEMPLATE METHOD

A SELECTION OF DESIGN PATTERNS

- Design pattern catalogue
- Factory Method
- *Template Method*
- Strategy
- Observer
- Decorator
- Visitor
- Singleton
- Composite
- Builder
- Iterator

WILL BE DISCUSSED LATER WHEN TALKING ABOUT APPLICATION FRAMEWORKS

SELECTED DESIGN PATTERNS

STRATEGY

A SELECTION OF DESIGN PATTERNS

- Design pattern catalogue
- Factory Method
- Template Method
- Strategy
- Observer
- Decorator
- Visitor
- Singleton
- Composite
- Builder
- Iterator

STRATEGY

Classification: Object Behavioural.

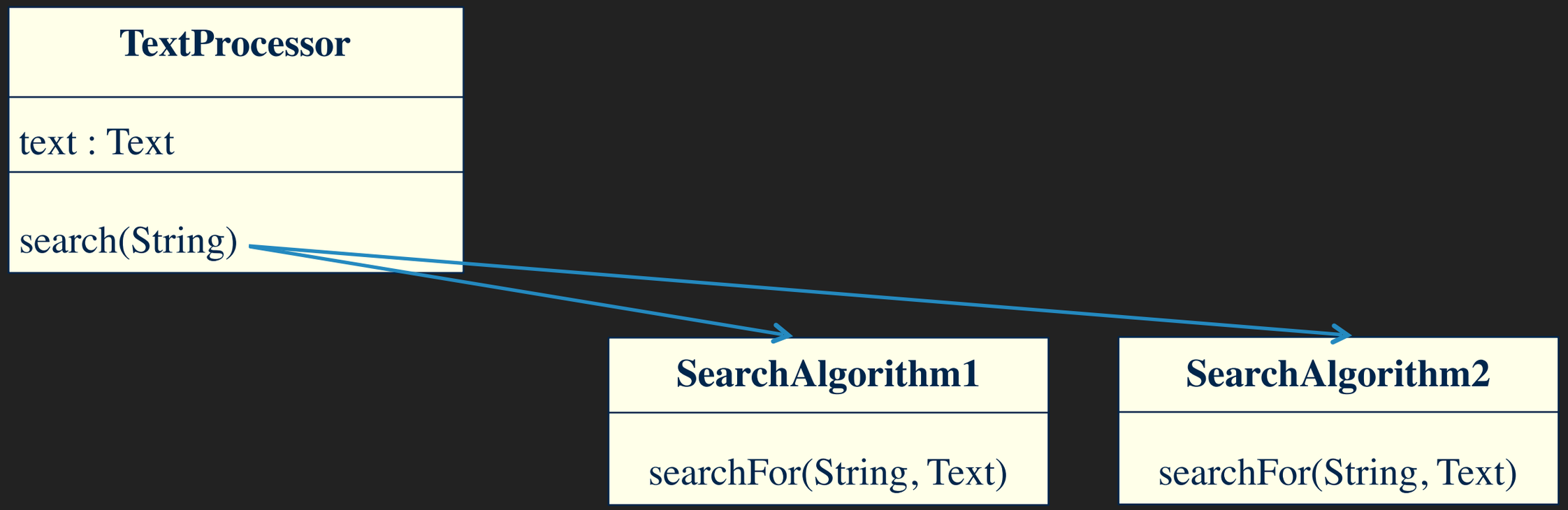
Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. *Strategy* lets the algorithm vary independently from clients that use it.

Motivating example:

- Different sorting algorithms
- String search in a text processor

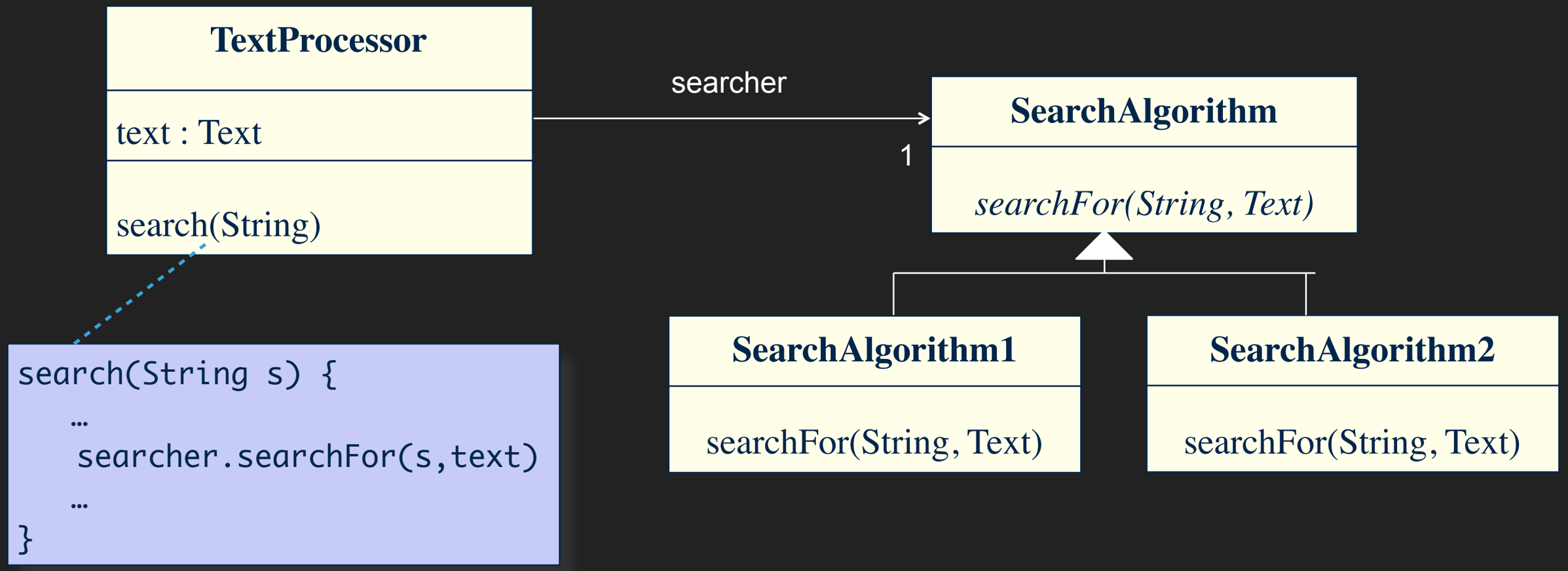
STRATEGY: MOTIVATION

There are common situations when classes differ only in their behaviour. For such cases it is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.



STRATEGY: MOTIVATION

There are common situations when classes differ only in their behaviour. For such cases it is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.



STRATEGY: APPLICABILITY

Use the Strategy design pattern when

- Many related classes that differ only in their behaviour
- You need different variants of an algorithm
- An algorithm uses data that clients should not know about
- A class defines many behaviours

STRATEGY: PARTICIPANTS

Strategy (SearchAlgorithm)

Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy

ConcreteStrategy (SearchAlgorithm1, SearchAlgorithm2)

Implements the algorithm using the Strategy interface

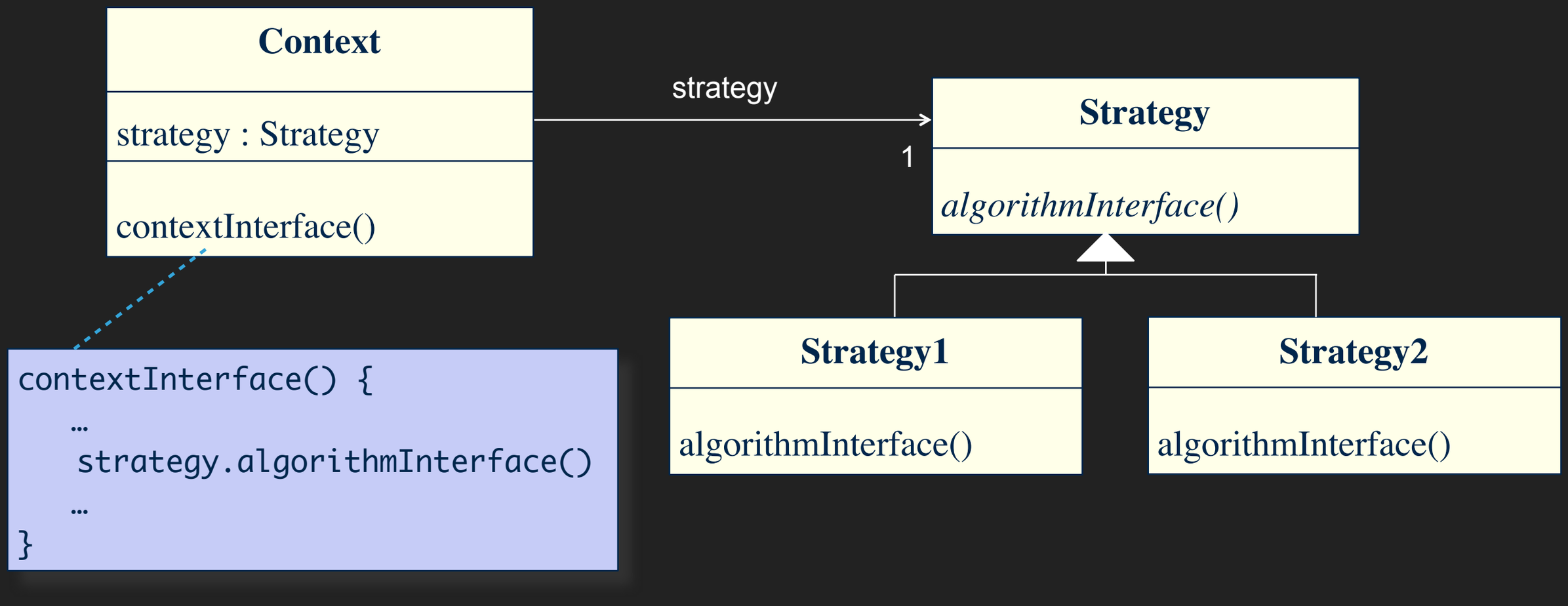
Context (TextProcessor)

Is configured with a ConcreteStrategy object

Maintains a reference to a Strategy object

May define an interface that lets Strategy access its data

STRATEGY: STRUCTURE



STRATEGY: COLLABORATIONS

Strategy and Context interact to implement the chosen algorithm

- Context may pass all data required by the algorithm to the strategy when the algorithm is called
- Context can pass itself as an argument to Strategy operations

A Context forwards requests from its clients to its strategy

- Clients usually create and pass a ConcreteStrategy object to the Context. From then on, they interact with the Context exclusively.

STRATEGY: CONSEQUENCES

Families of related algorithms

An alternative to subclassing

Strategies eliminate conditionals

Provide a choice of implementation

- allow for different implementations of same behaviour

Overhead involved

- Communication between Context and Strategy
- Increased number of objects

STRATEGY: IMPLEMENTATION

The Strategy interface must provide enough information to ConcreteStrategy

Default behaviour can be incorporated in the Context object. If no strategy object is present, this default behaviour can be used.

SELECTED DESIGN PATTERNS

DECORATOR

A SELECTION OF DESIGN PATTERNS

- Design pattern catalogue
- Factory Method
- Template Method
- Strategy
- Observer
- Decorator
- Visitor
- Singleton
- Composite
- Builder
- Iterator

DECORATOR

Classification: Object Structural.

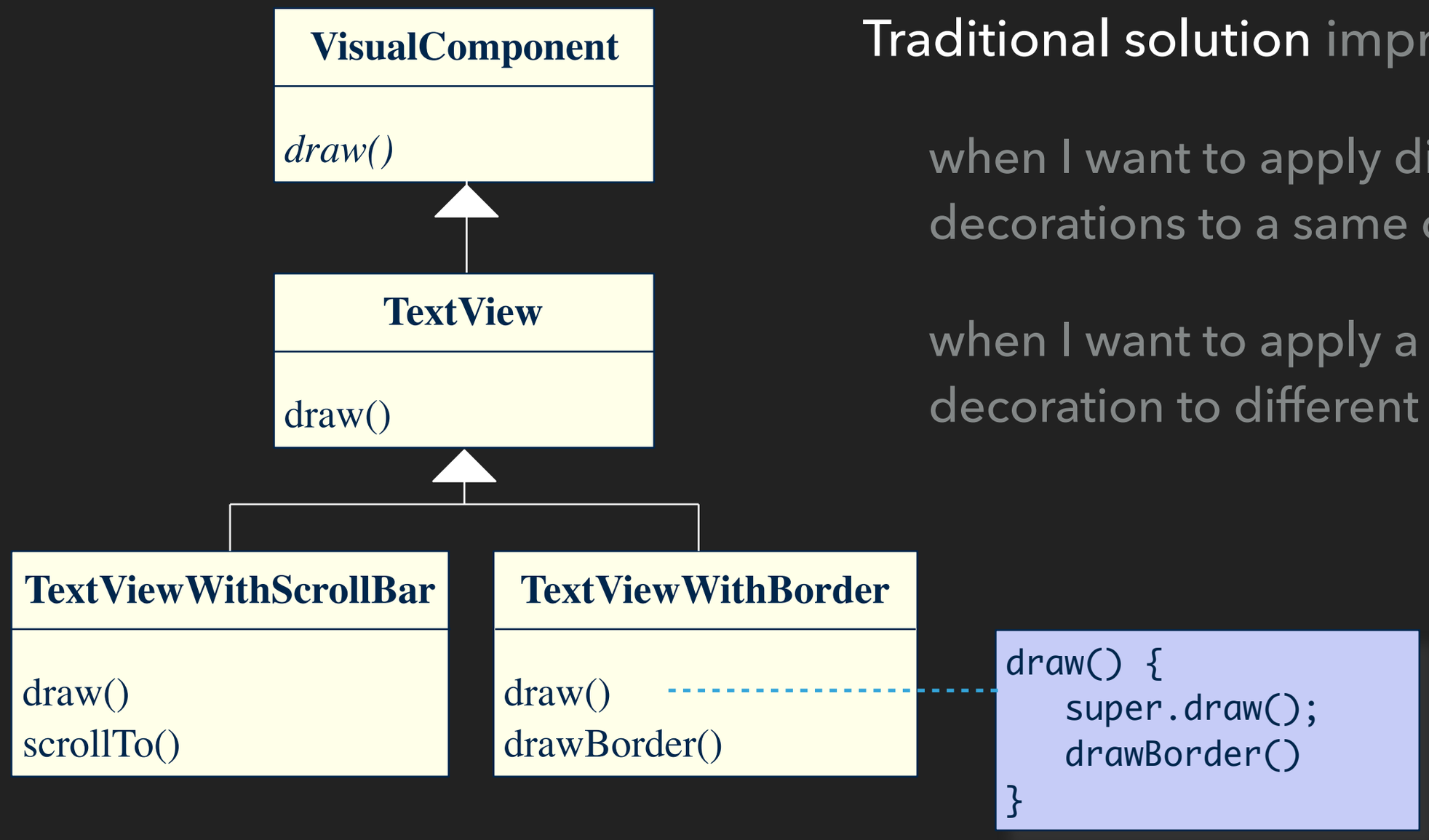
Intent:

Attach additional responsibilities to an object dynamically.
Decorators provide a dynamic alternative for subclassing.

Motivating example:

Adding borders/scrollbars/... to a visual component

DECORATOR: MOTIVATION

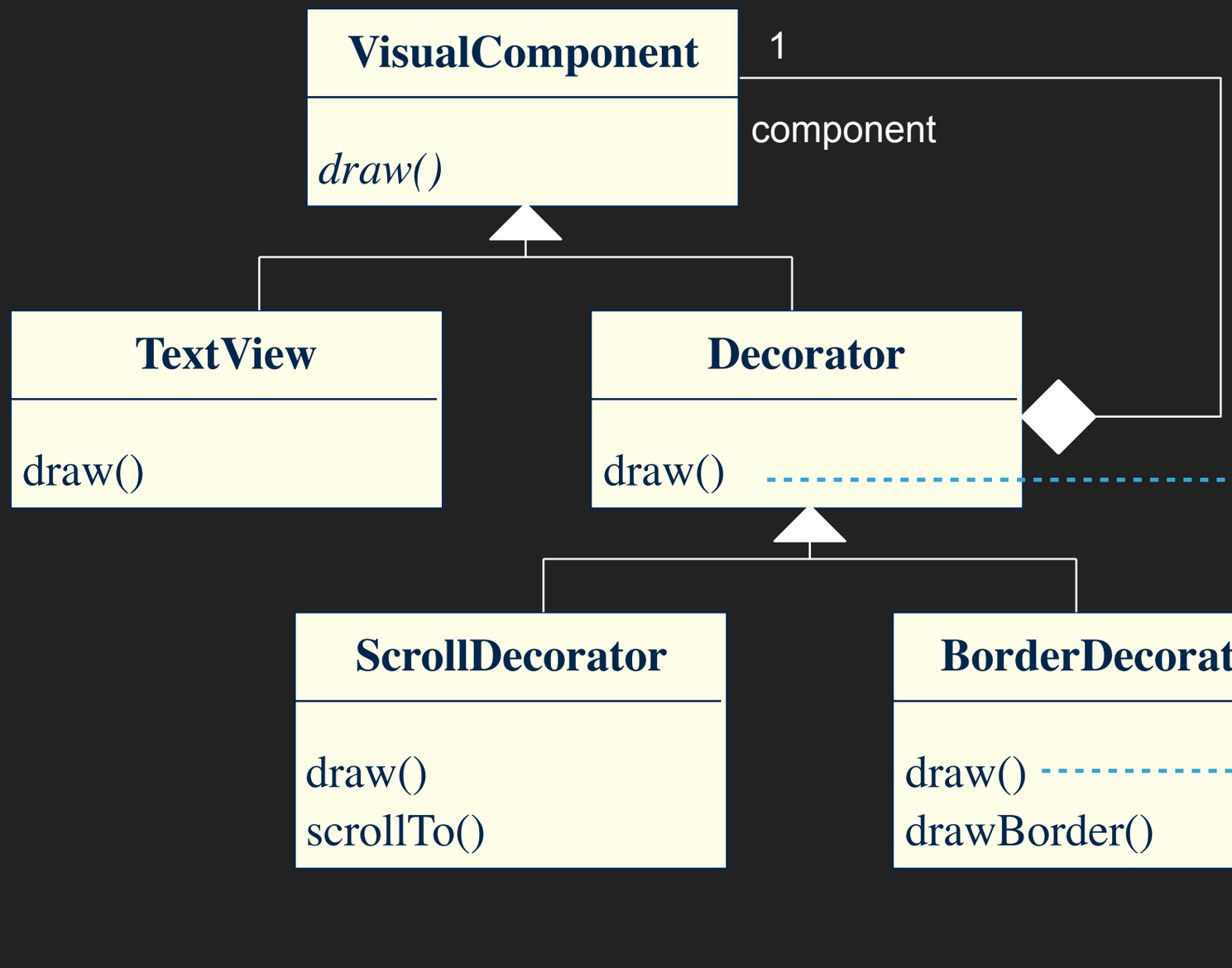


Traditional solution impractical

when I want to apply different decorations to a same component

when I want to apply a same decoration to different components

DECORATOR: MOTIVATION



Solution with decorator design pattern

```
draw() {
    …
     component.draw()
    …
}
```

```
draw() {
    super.draw();
    drawBorder()
}
```

DECORATOR: APPLICABILITY

Use the Decorator design pattern

- To add responsibilities to individual objects dynamically and transparently without affecting other objects
- For responsibilities that can be withdrawn
- When extending by subclassing is impractical

DECORATOR: PARTICIPANTS

Component (VisualComponent)

Defines the interface for objects that can have responsibilities added to them dynamically

ConcreteComponent (TextView)

Defines an object to which we want to attach additional responsibilities

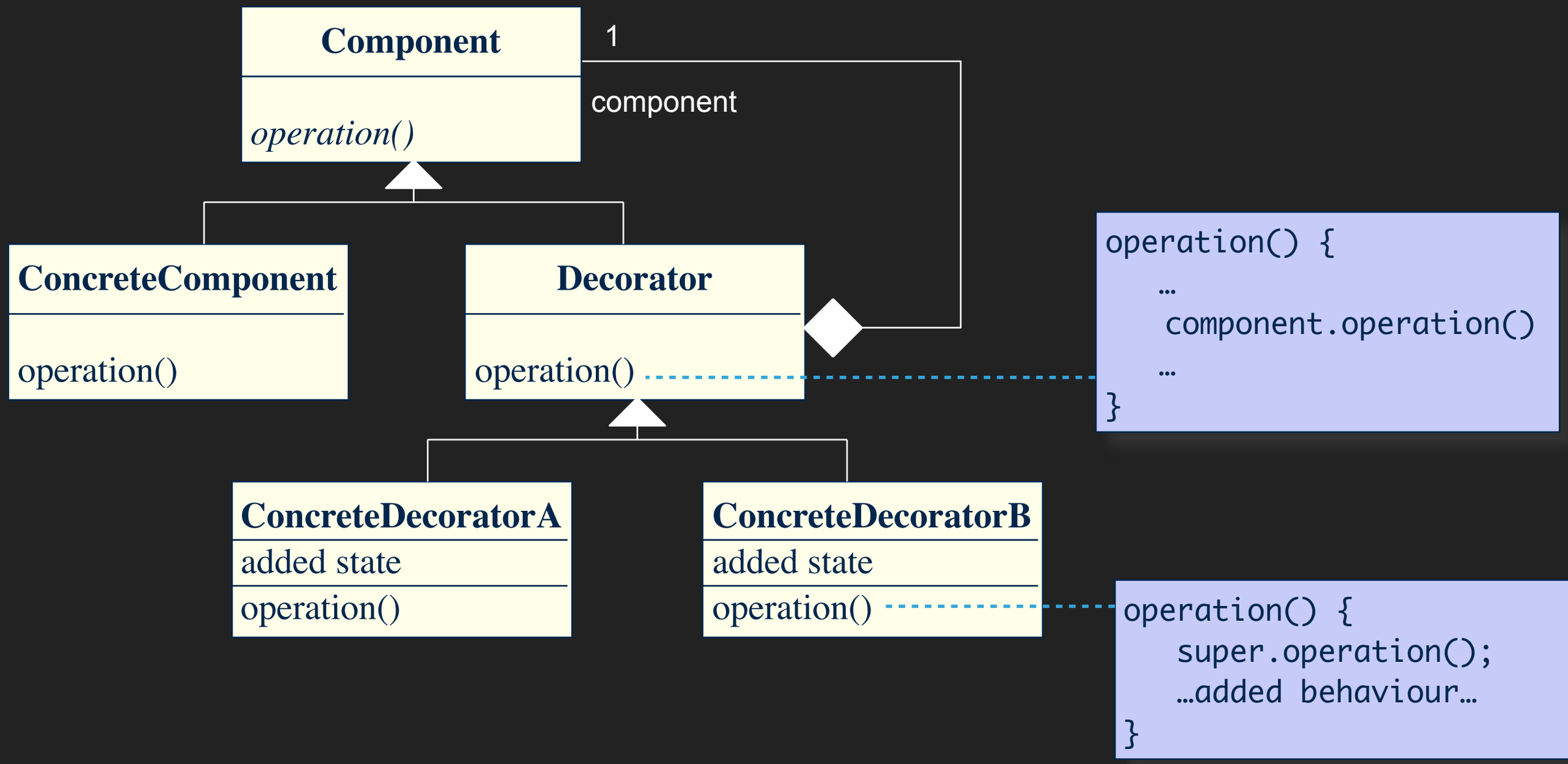
Decorator (Decorator)

Maintains a reference to a Component object and defines an interface that conforms to the Components interface

ConcreteDecorator (BorderDecorator)

Adds responsibilities to the component

DECORATOR: MOTIVATION



DECORATOR: COLLABORATIONS

Decorator forwards requests to the Component object. It may optionally perform additional behaviour before and after forwarding the request

DECORATOR: CONSEQUENCES

More flexible than static inheritance

- Possible to dynamically add or withdraw responsibilities
- Avoids classes with a lot of features

Lots of little objects

DECORATOR: IMPLEMENTATION

Object identity can be a problem

- a decorated component is not identical to the object itself

LINGI2252 – PROF. KIM MENS

F. ANTIPATTERNS*

* Slides party reused from slides by Prof. Tom Mens at UMons, Belgium

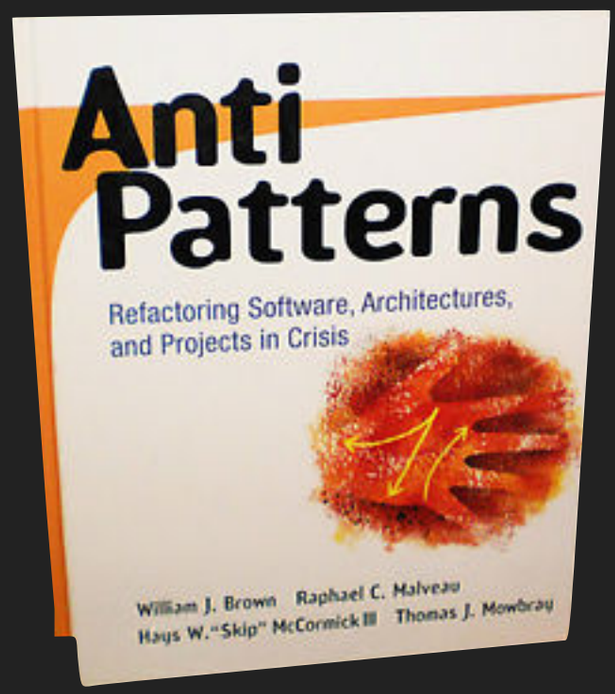
KEY REFERENCE

Anti Patterns

Refactoring Software, Architectures and Projects in Crisis

W. J. Brown
R. C. Malveau
H. W. McCormick
T.J. Mowbray

John Wiley & Sons, 1997



WHAT ARE ANTIPATTERNS?

Design patterns identify *good working practices*

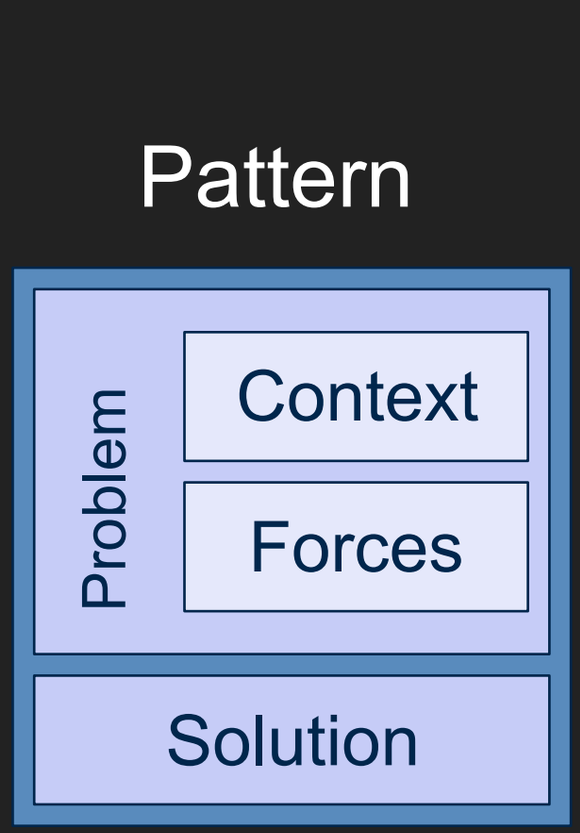
Anti patterns identify *common mistakes* and how these mistakes can be overcome in *refactored* solutions.

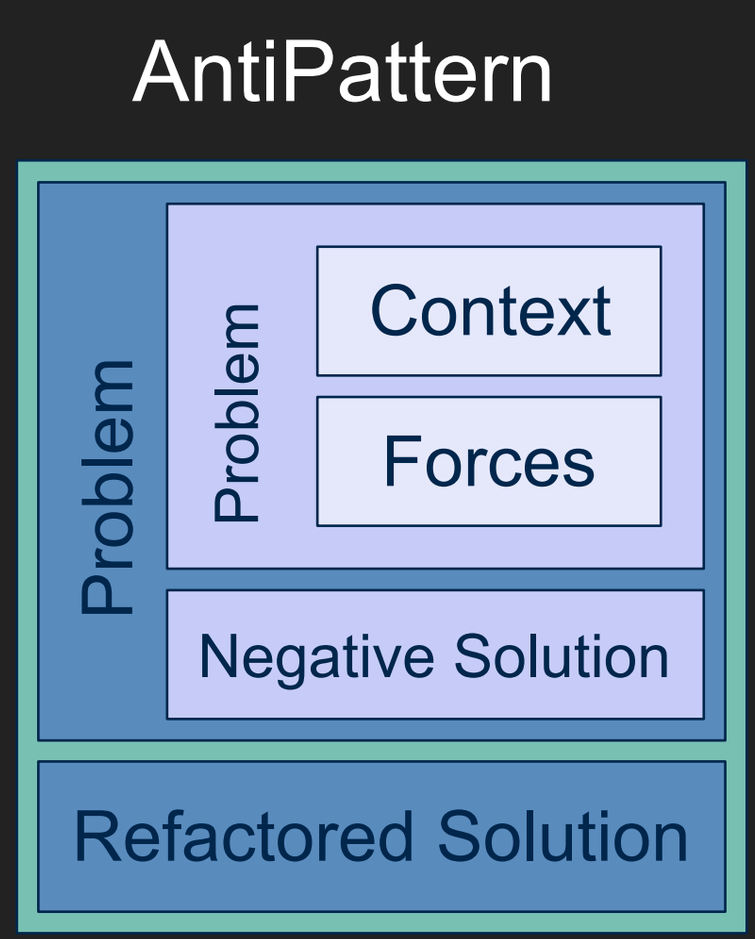
An AntiPattern is

> *"a commonly used solution to a problem that generates decidedly negative consequences"*

WHAT ARE ANTIPATTERNS?

An AntiPattern is a special (negative) design pattern which features an extra refactored solution to the problem.





7 DEADLY SINS

1. Haste : Hasty decisions lead to compromises in software quality. Especially testing is a victim.

   “Just clean up the code, we ship tomorrow…”

2. Apathy : Not caring about solving known problems

   “Reuse? Who’s ever gonna reuse this crappy code?”

3. Narrow-mindedness : Refusal to practice solutions that are widely known to be effective.

   “I don’t need to know, and… I don’t care to know”

7 DEADLY SINS

4. Sloth (lazyness) : Making poor decisions based upon easy answers (lazy developers)
5. Avarice (greed) : The modeling of excessive details, resulting in overcomplexity due to insufficient abstraction

   “I’m impressed ! The most complex model ever done !”
6. Ignorance : Failing to seek understanding

   “100 pages… let’s find a one page summary on the net”
7. Pride : Reinventing designs instead of reusing them.

CATEGORIES OF ANTIPATTERNS

Development AntiPatterns

technical problems/solutions encountered by programmers

Architectural AntiPatterns

identification of common problems in system structures

Managerial AntiPatterns

addresses common problems in software processes and development organisations

DEVELOPMENT ANTIPATTERNS

The Blob

Continuous Obsolescence

Lava Flow

Ambiguous Viewpoint

Functional Decomposition

Poltergeists

Golden Hammer

Boat Anchor

Dead End

Spaghetti Code

Minefield Walking

Cut-and-Paste

EXAMPLE: THE BLOB

Category : Software Development

Also Known As : The God Class

Scale : Application

Refactored Solution Name : Refactoring of Responsibilities

Root Causes : Sloth, Haste

THE BLOB: GENERAL FORM

Designs where one class monopolises the processing, and other classes primarily encapsulate data

- Key problem: majority of responsibilities allocated to a single class.

In general it is a kind of procedural design

- conflicts with OO paradigm

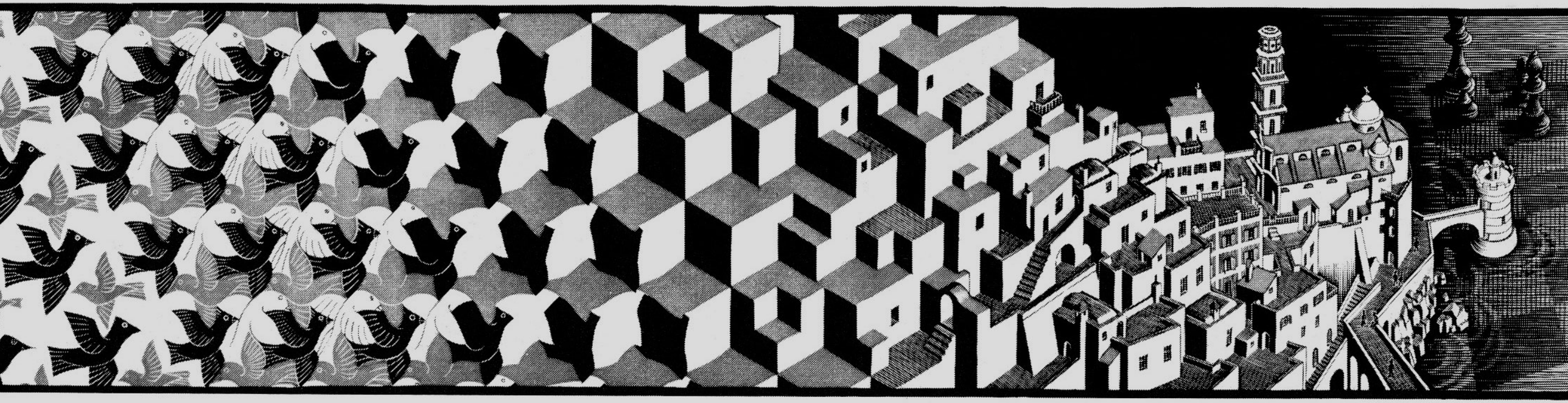
THE BLOB: REFACTORED SOLUTION

Identify or categorise related attributes and operations

Look for ‘natural homes’ for these collections of functionality

Apply OO design techniques (e.g., inheritance, …)

Apply refactorings to bring code back in shape



LINGI2252 – PROF. KIM MENS

G. CONCLUSION

SOFTWARE PATTERNS . . .

capture successful solutions to recurring problems that arise during software construction

define a common vocabulary amongst software developers

increase reuse of design as opposed to reuse of implementation

help to improve software quality

can be introduced by refactoring

should be used with care! (anti patterns)

ARCHITECTURAL AND DESIGN PATTERNS ARE

Smart

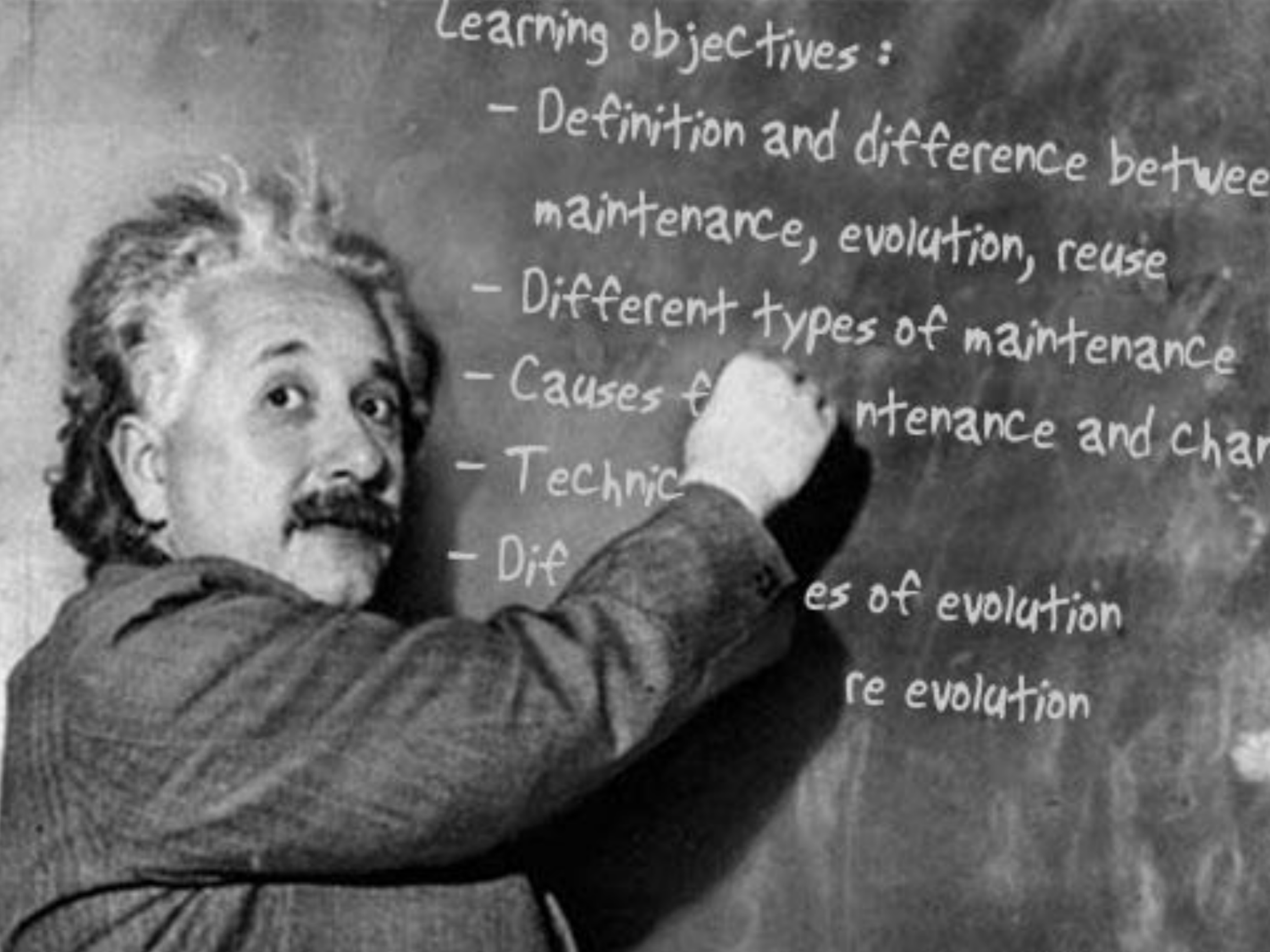
provide elegant solutions that a novice would not think of

Generic

independent of a specific system or programming language

Well-proven

successfully tested and applied in several real-world applications


Learning objectives :
- Definition and difference betwee
maintenance, evolution, reuse
- Different types of maintenance
- Causes f ntenance and char
- Technic es of evolution
- Dif re evolution

POSSIBLE QUESTIONS (1)

- What do Christopher Alexander's (building) architectural patterns and (software) design patterns have in common? Explain with a concrete example.
- Give a definition of the notion of "design pattern".
- What are design patterns good for and why? Why cannot you say "I have invented a design pattern"?
- Explain the different parts of which a design pattern description typically exists.
  - (Catchy name, Classification, Intent, Also known as, Motivation, Applicability, Structure, Participants, Collaboration, Consequences, Implementation, Sample Code, Known Uses, Related Patterns)

POSSIBLE QUESTIONS (2)

- Explain and illustrate the Abstract Factory design pattern in detail. Clearly mention its problem, solution, participants, structure and applicability.
- Explain the Factory Method design pattern in detail. Clearly mention its Intent, Motivation, Applicability, Participants, Collaboration, Consequences and Implementation.
- Same question for the Strategy and Decorator design patterns.
- What is an antipattern and how does it compare to a design pattern? What purpose does it serve?
- Explain at least 4 of the 7 deadly sins related to antipatterns.
- Explain The Blob antipattern.

CLASS... IS... DISMISSED.