Design Patterns

January 26th, 2019

Object-oriented Software Design Patterns

Design Patterns help solving problems in object-oriented software design.

Code complexity in class hierarchy

Naming (controller, manager, handler, coordinator, executor, provider)

Everyone uses their own coding patterns

Data flow all over the place

Spaghetti code

Dependencies

Components

God classes

Context

Packages

Responsibilities of classes

Software architecture

Structures of class hierarchies

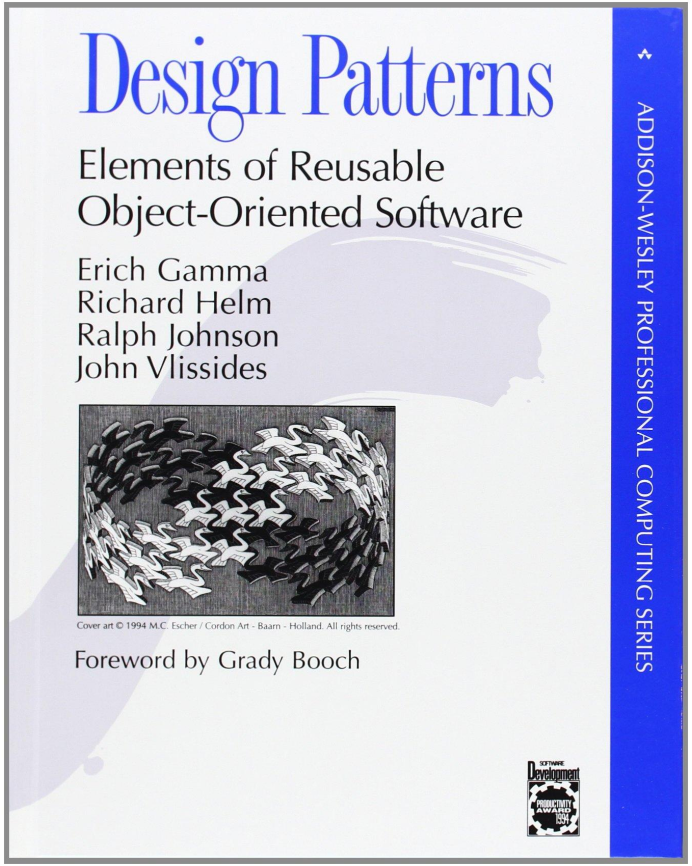
Design Patterns

Design Patterns help solving problems in object-oriented software design.

Design Patterns

Gamma, Helm, Johnson, Vlissides:
Design Patterns. Elements of Reusable Object Oriented Software.
1995 Pearson Education

„Gang of Four“
„GoF design patterns“



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

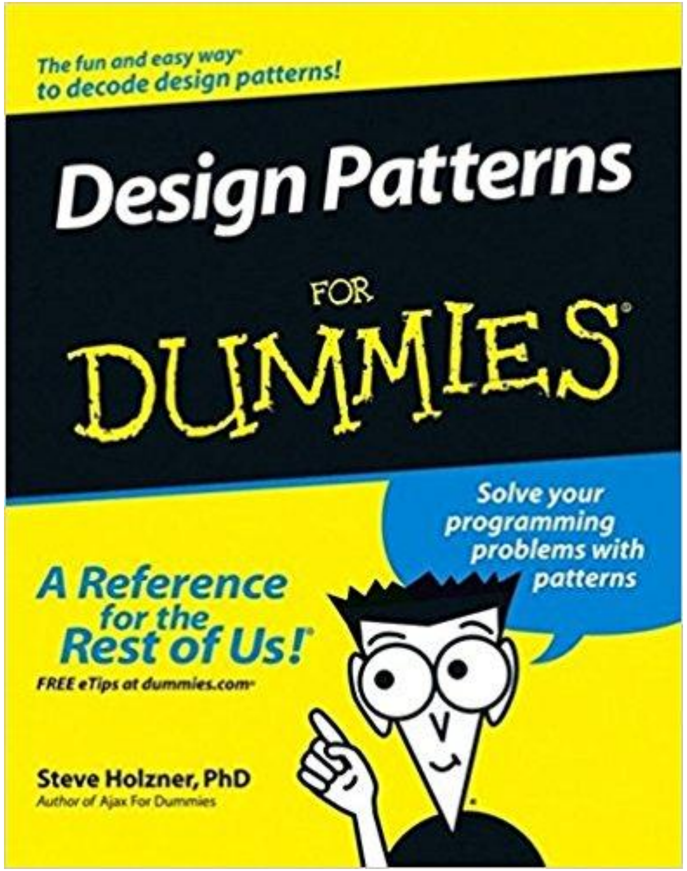
Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns

Holzner:
Design Patterns for Dummies.
2006 Wiley Publishing

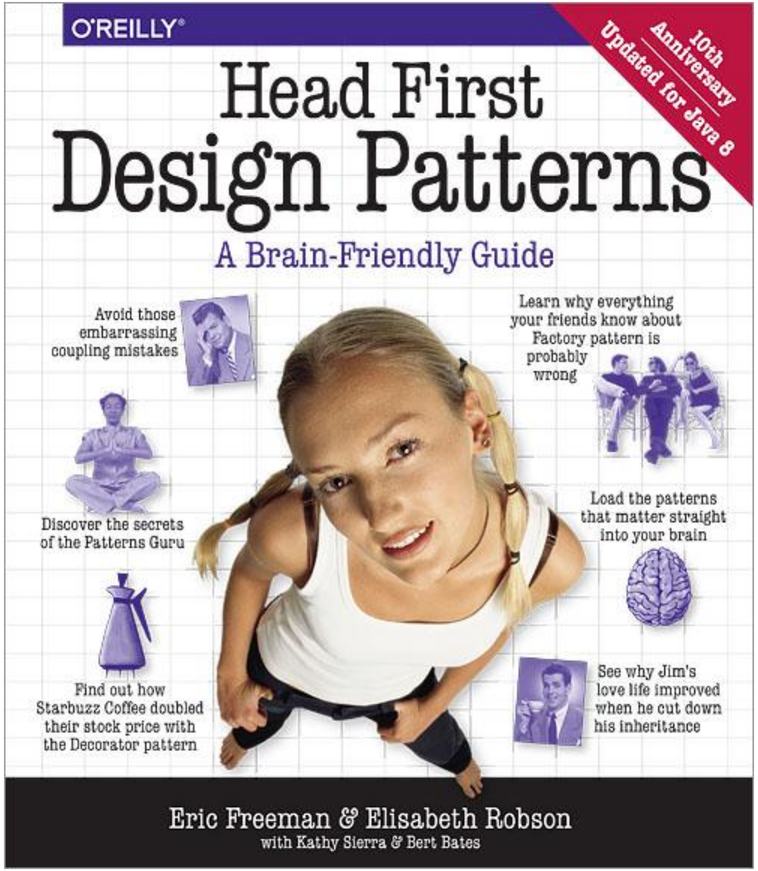
„Dummies“


The fun and easy way to decode design patterns!
Design Patterns
FOR DUMMIES
Solve your programming problems with patterns
A Reference for the Rest of Us!
FREE eTips at dummies.com
Steve Holzner, PhD
Author of Ajax For Dummies

Design Patterns

Freeman, Robson, Bates, Sierra:
Head First Design Patterns:
A Brain-Friendly Guide.
2004 O'Reilly

„Head First“


O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

Design Patterns

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“
>
> ~ GoF / Christopher Alexander

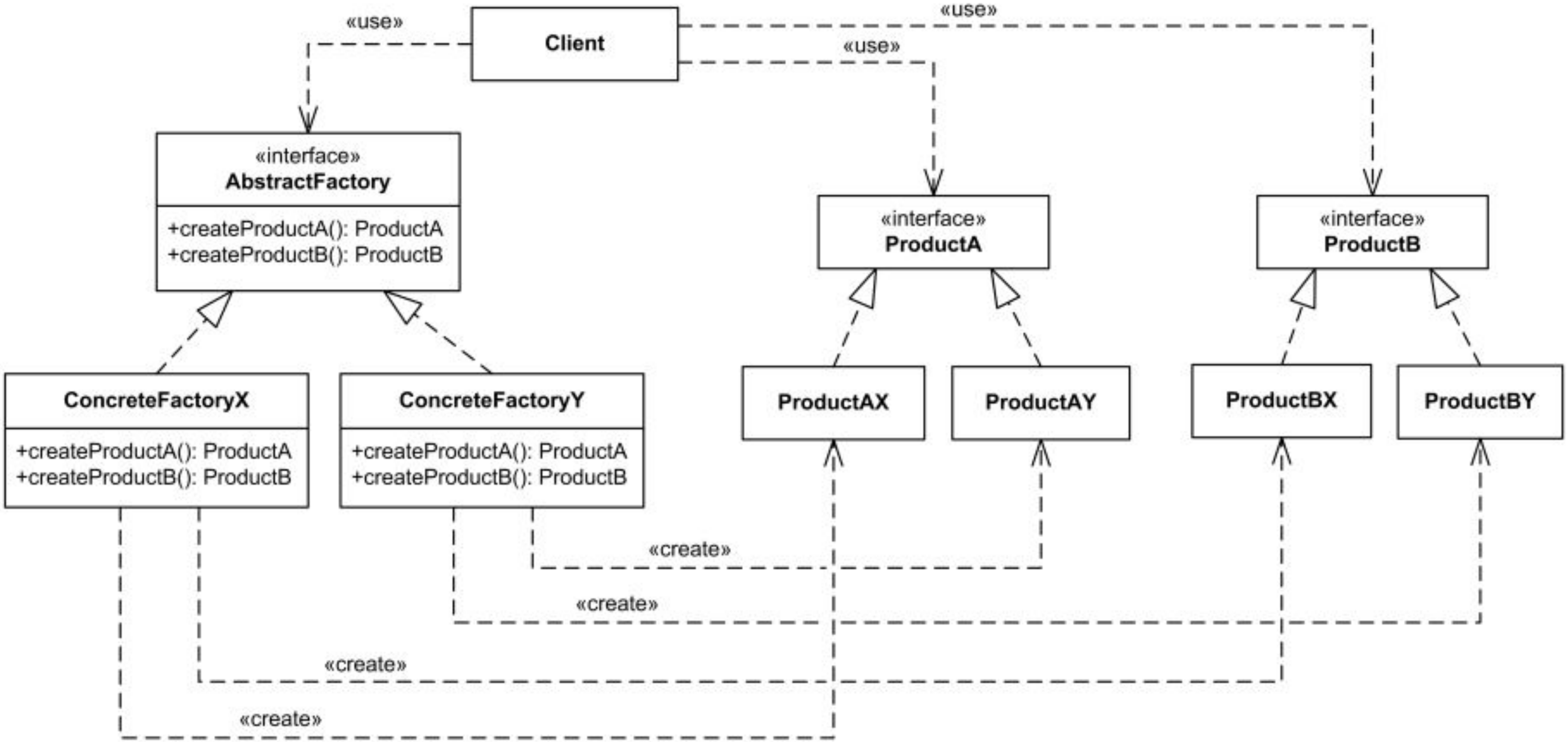
Design Patterns

What is a pattern?

> “Congratulations, your problem has already been solved.“
>
> ~ Dummies

Design Patterns

What is a pattern in object-oriented software?


«use»
Client
«use»
«use»
«interface»
AbstractFactory
+createProductA(): ProductA
+createProductB(): ProductB
«interface»
ProductA
«interface»
ProductB
ConcreteFactoryX
+createProductA(): ProductA
+createProductB(): ProductB
ConcreteFactoryY
+createProductA(): ProductA
+createProductB(): ProductB
ProductAX
ProductAY
ProductBX
ProductBY
«create»
«create»
«create»
«create»

Design Patterns

UML Diagrams

Creational Patterns	Structural Patterns	Behavioral Patterns
Singleton	Composite	Observer
Factory Method	Flyweight	Command
Abstract Factory	Decorator	Mediator
Builder	Proxy	Strategy
Prototype	Adapter	Chain of Responsibility
	Facade	Visitor
	Bridge	Interpreter
		Iterator
		Memento
		State
		Template Method

Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
Singleton	**Composite**	Observer
Factory Method	Flyweight	Command
Abstract Factory	Decorator	Mediator
Builder	Proxy	Strategy
Prototype	Adapter	Chain of Responsibility
	Facade	**Visitor**
	Bridge	Interpreter
		Iterator
		Memento
		State
		Template Method

Creational Patterns	Structural Patterns	Behavioral Patterns
Singleton	**Composite**	Observer
Factory Method	Flyweight	Command
Abstract Factory	Decorator	Mediator
Builder	Proxy	Strategy
Prototype	Adapter	Chain of Responsibility
	Facade	**Visitor**
	Bridge	Interpreter
		Iterator
		Memento
		State
		Template Method

Design Patterns - January 26th, 2019

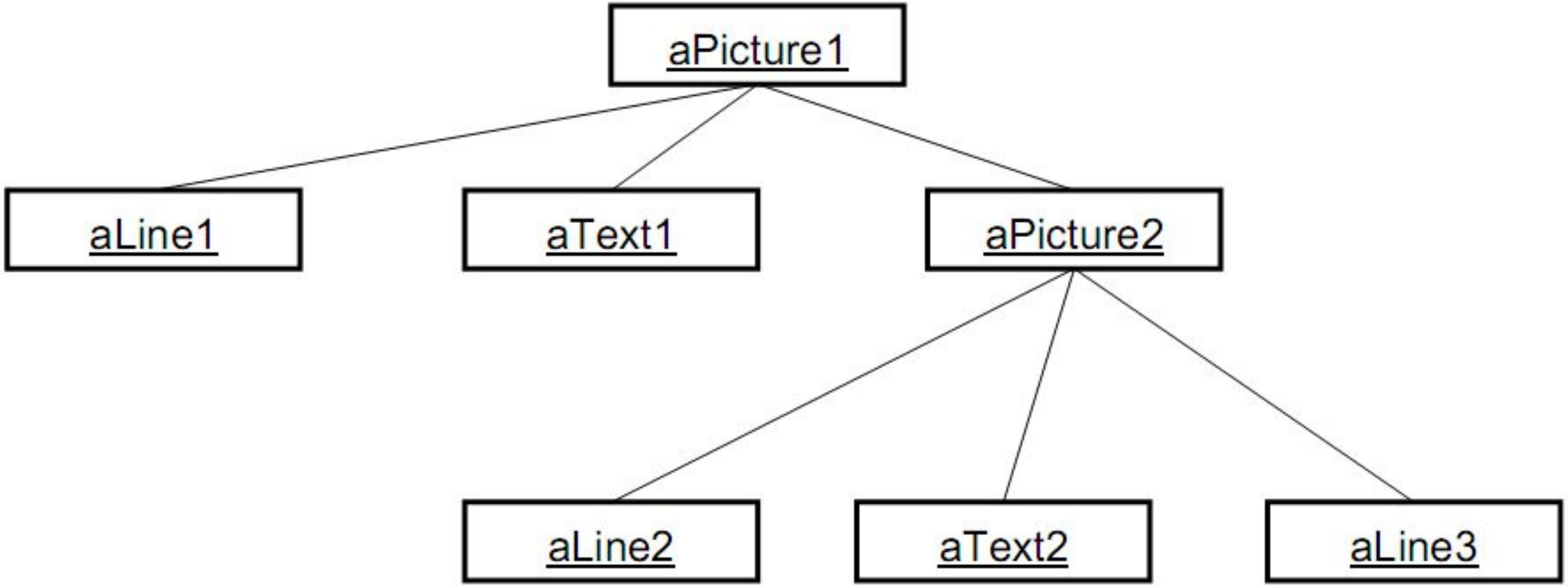
1. `Composite`
2. `Visitor`
3. `Bridge`
4. `Singleton`
5. `Factory Method`
6. `Abstract Factory`

For each Pattern

1. The Problem
2. The Pattern
3. Implement it in Java
4. Use if...
5. Consequences, open discussion

Composite

Composite



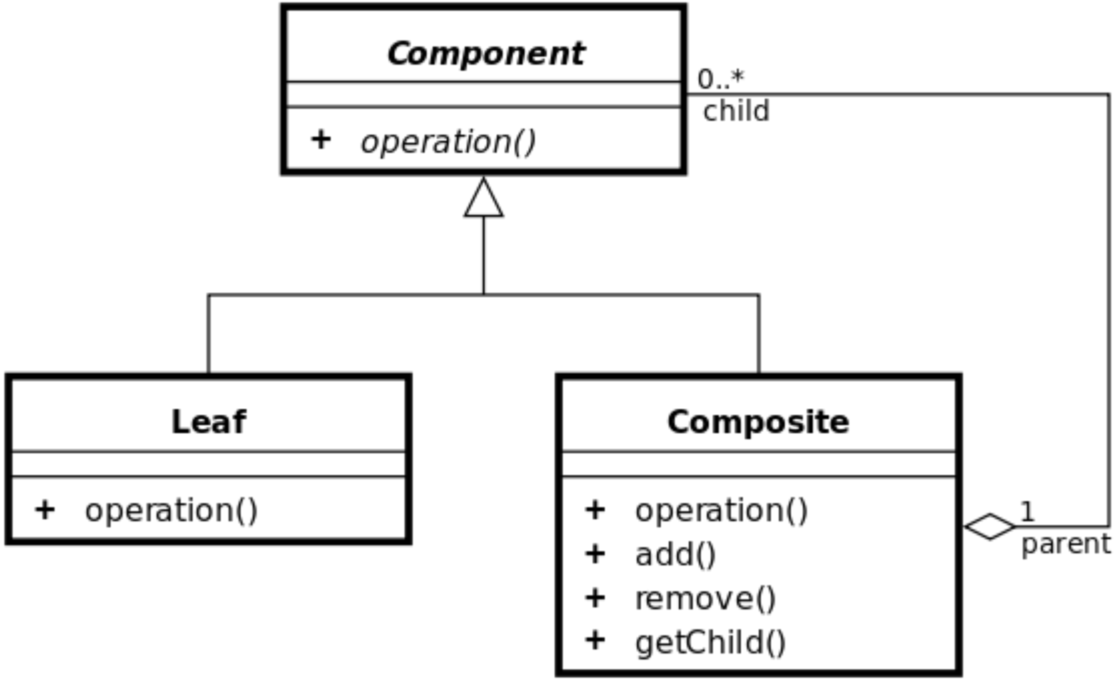
Composite

The Problem:

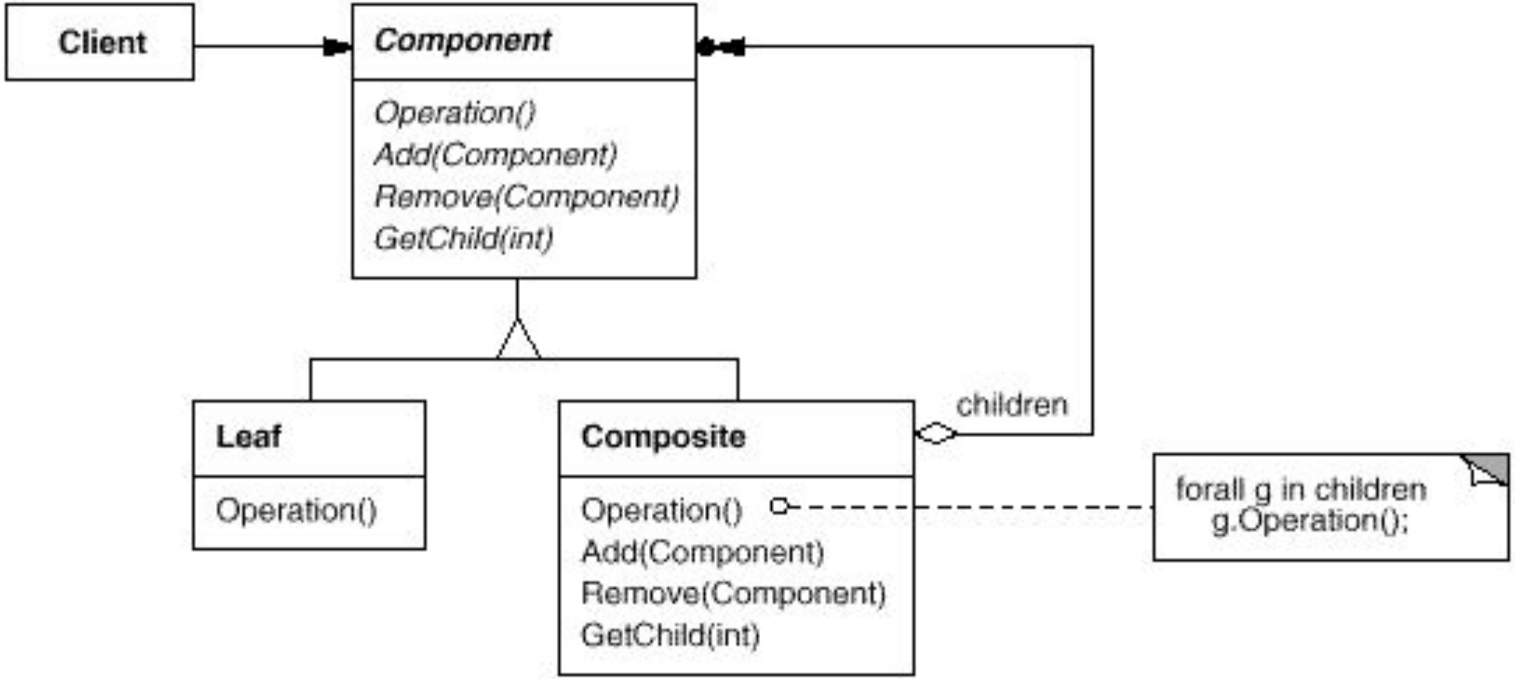
You want to create a structure that has recursive properties: parent objects that can have children that again are parents. You need to build a tree of objects, where objects can be both nodes and leaves.

Necessary e.g. for building composed graphics elements, or for building nested layout components.

Composite


Component
+ *operation()*
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

Composite


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

Composite

The Pattern:

1. Create either an abstract class / interface, which defines the methods that both leaf and node have in common.
2. Create a class for the leaf that implements this abstract class / interface.
3. Create a class for the node that also implements the abstract class / interface. The node has a list of your abstract class / interface as member.
4. If you implement the methods, iterate over the children and call their methods *recursively*.

Composite

Let’s touch some code.

Composite

Use if…

… you want objects to be connected in a tree structure.

… the client should be able to use the composed parent object the same way as leaf objects.

Composite

Consequences:

Allows for class hierarchies from primitive and composite objects.

Makes the access for a client object easier.

Simplifies adding of new composite or leaf objects.

Drawback:

It’s hard to limit the breadth or depth of the hierarchy (needs runtime checks).

Visitor

Visitor

The Problem:

You want to add extra functionality to an object structure without modifying it.

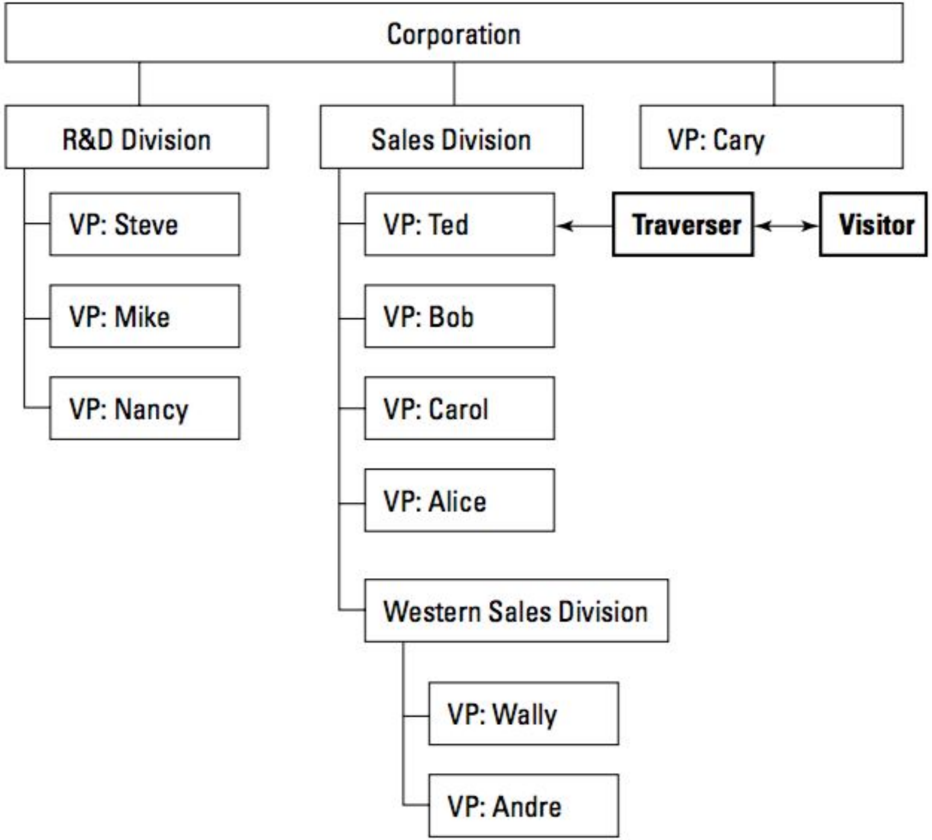


Figure 11-8: A visitor traversing an object structure.

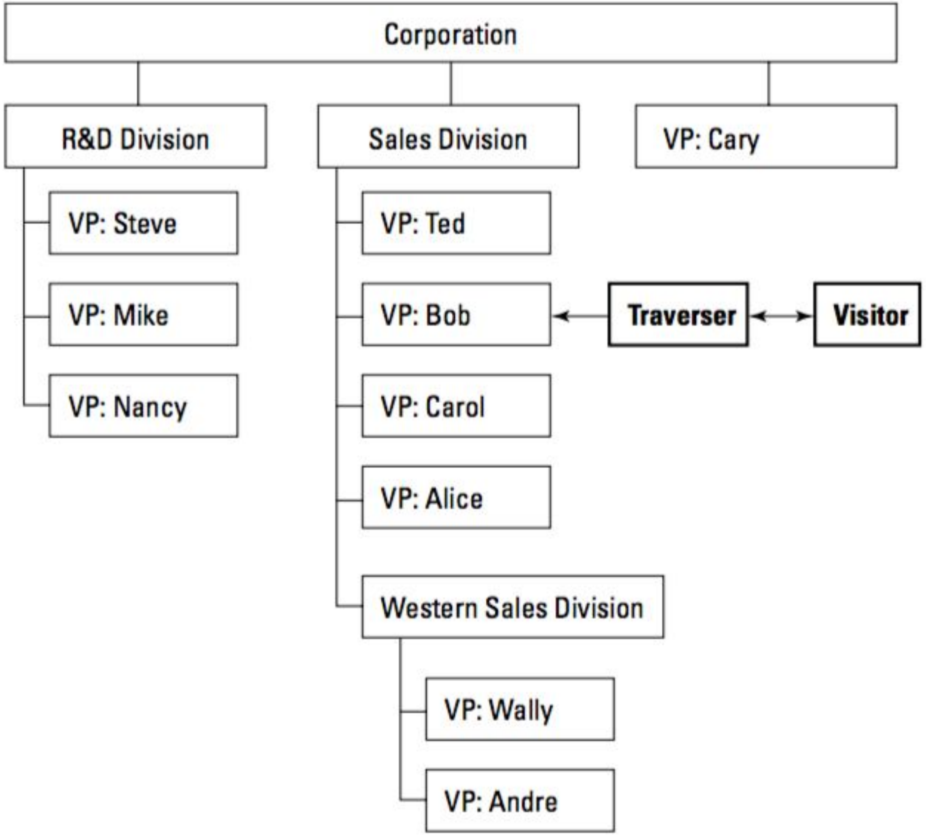
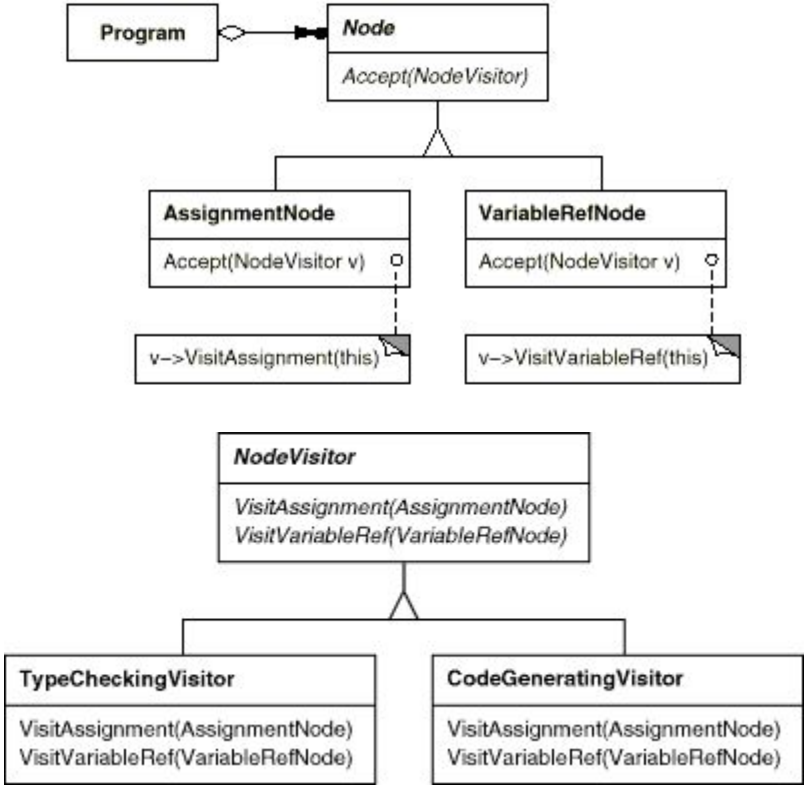


Figure 11-9: The visitor is visiting another object.

Visitor



Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

Visitor

The Pattern:

1. Create an interface `Visitor` with a method `visit(Item item)`.
2. For each functionality that you want to add, create a concrete visitor class that implements the `Visitor` interface. Add the functionality by implementing `visit(Item item)`.
3. Create an interface `Visitable` with method `accept(Visitor visitor)`.
4. Let `Item` implement `Visitable`. Implement `accept(Visitor visitor)` by calling `visitor.visit(this)`.

Visitor

Bonus:

5. If your data structure is a composite, let it implement Visitable. Implement `accept(Visitor visitor)` by calling `element.accept(visitor)` for each child node.

Visitor

Let’s touch some code.

Visitor

Use if…

… your object collection contains many objects of different classes, and you want to perform operations on these objects that depend on their concrete classes.

… many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid polluting their classes with these operations.

Visitor

Consequences:

Makes adding new operations easy.

A visitor gathers related operations and separates unrelated ones.

Drawback:

Adding new ConcreteElement classes is hard, since the abstract visitor needs to add a new abstract method and you need to change all its implementations.

Bridge

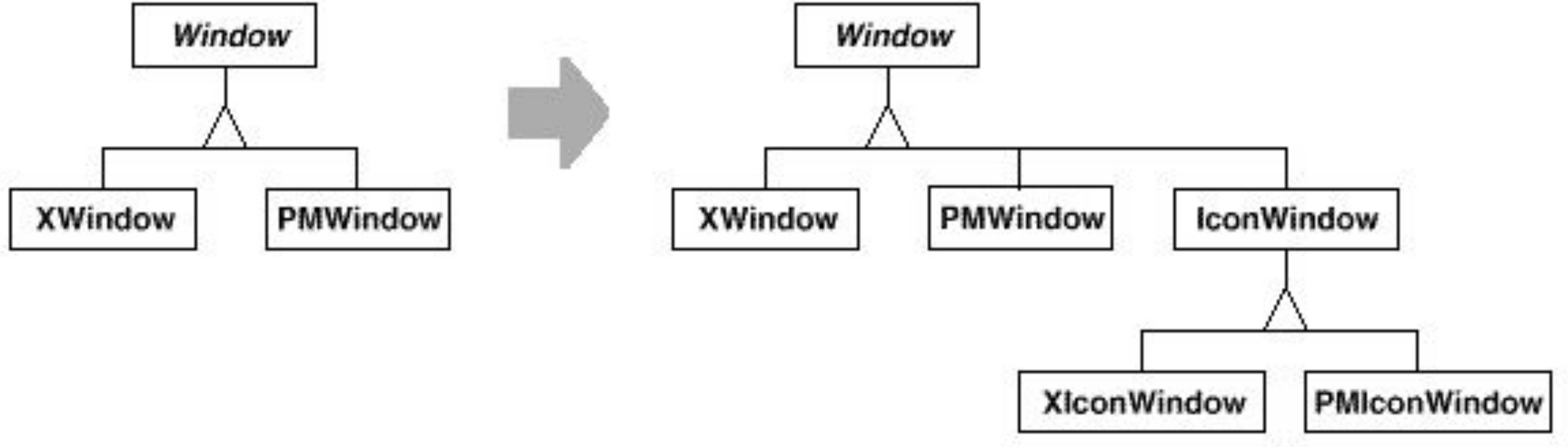
Bridge

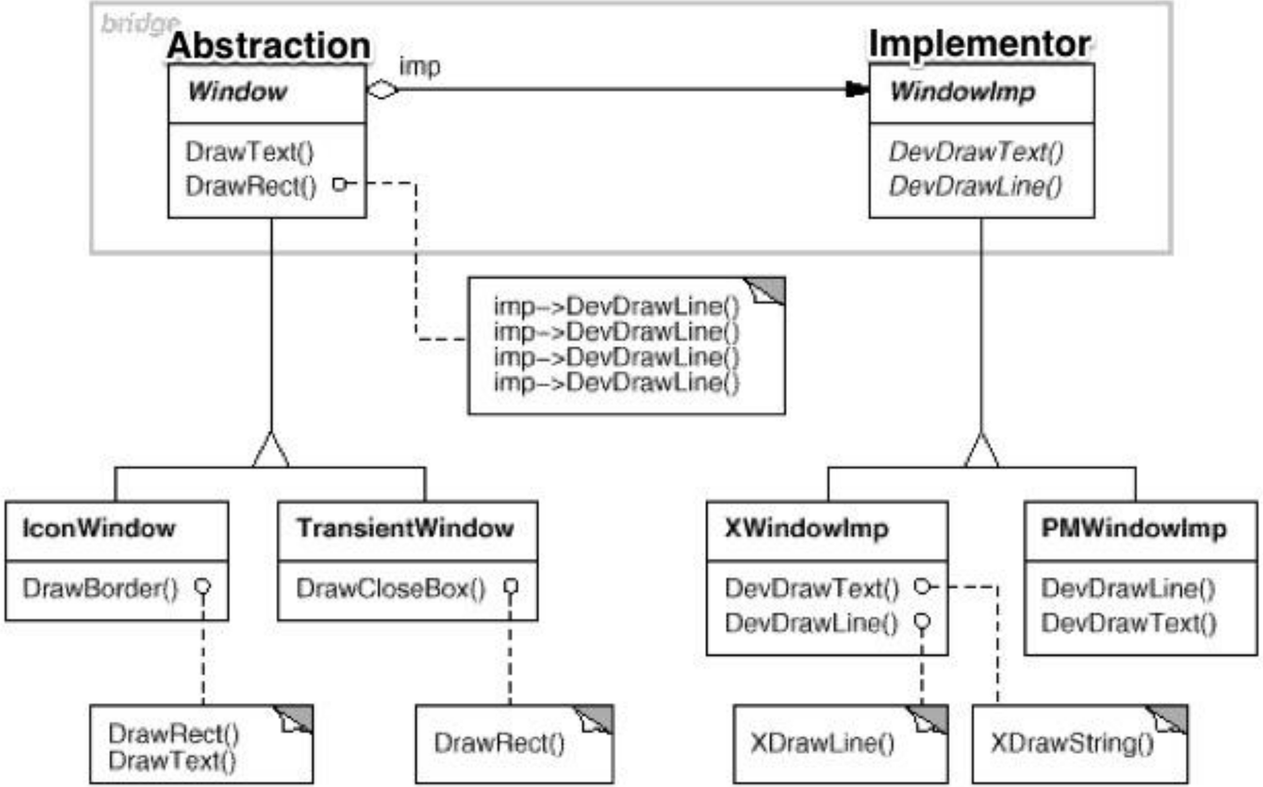
The Problem:

You have a hierarchy with an abstract class and 2 implementations A and B. Then you need another implementation X that also has attributes of A and B, so that you end up with child classes of X called AX and BX.

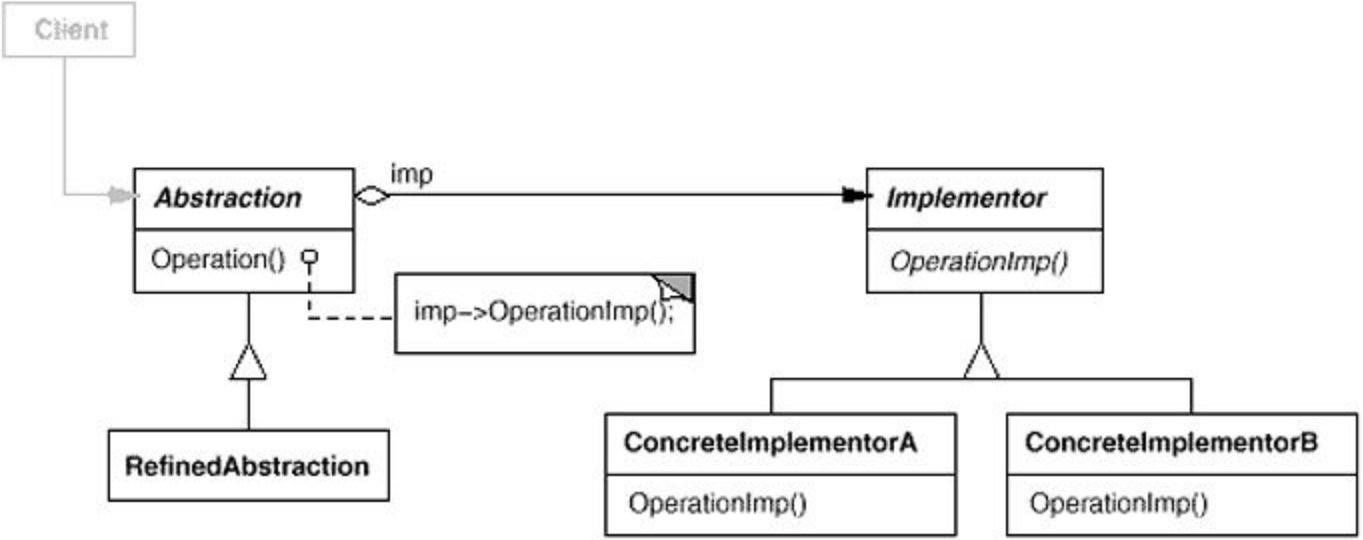
This is rather inconvenient, especially if you design for change, and it is likely that there will be another implementation C, and you need CX respectively. Or even another Y, where you then need a new AY, BY and CY.

Bridge


Window
XWindow
PMWindow
Window
XWindow
PMWindow
IconWindow
XIconWindow
PMIconWindow


bridge
Abstraction
Window
DrawText()
DrawRect()
imp
Implementor
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Bridge


Client
Abstraction
Operation()
imp
imp->OperationImp();
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

Bridge

The Pattern:

1. Create an abstract class `Window` and an abstract class `WindowImp`. Let Window have a member of type `WindowImp`.
2. Create abstract methods in `WindowImp`, e.g. `devDrawLine()`.
3. In `Window`, implement methods, e.g. `drawRect()`, that use the `WindowImp` member to do its work.
4. Implement concrete classes `IconWindow` and `TransientWindow` that extend `Window`. In their methods, use a combination of `Window`’s methods.

Bridge

5. Create concrete classes `XWindowImp` and `PMWindowImp` that extend `WindowImp` and implement all abstract methods of `WindowImp`, e.g. `devDrawLine()`.

Bridge

1. Create an abstract class `Abstraction` and an abstract class `Implementor`. Let `Abstraction` have a member of type `Implementor`.
2. Create abstract methods in `Implementor`.
3. In `Abstraction`, implement methods that use the `Implementor` member to do its work.
4. Implement concrete classes `X` and `Y` that extend `Abstraction`. In their methods, use a combination of `Abstraction`’s methods.
5. Create concrete classes `A` and `B` that extend `Implementor` and implement all abstract methods of `Implementor`.

Bridge

The Pattern:

> “Decouple an abstraction from its implementation so that the two can vary independently.”
>
> ~ Gang of Four

Bridge

Let’s touch some code.

Bridge

Use if…

… both the abstraction and the implementation should be extensible by subclassing.

… you have a proliferation of classes as shown earlier in the first diagram. This sort of hierarchies is also called "nested generalizations".

Bridge

Consequences:

- Decouples interface from implementation.
- Improved extensibility.
- Hides implementation details from clients.

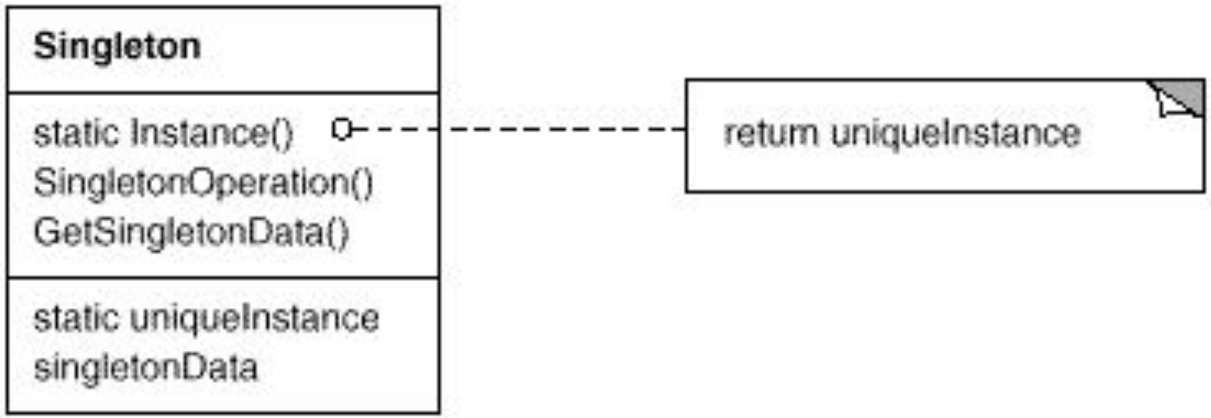
Singleton

Singleton

The Problem:

You have one very large class that holds many other objects, e.g. a database class. You need exactly one instance of it, because more would compromise your application's logic, or would be unnecessary allocation of memory.

Singleton


Singleton
static Instance()
SingletonOperation()
GetSingletonData()
static uniqueInstance
singletonData
return uniqueInstance

Singleton

The Pattern:

1. In the class that you want to make a singleton, create a `private static` member of the same class that you call `instance`.
2. Make the constructor private.
3. Create a `static` method called `getInstance()` that returns your singleton instance. Check if it has been initialized, and if not, create it.

Singleton

Let’s touch some code.

Singleton

Use if…

… there should be exactly one instance of an object.

Singleton

Consequences:

Access control to only instance by singleton class.

No need to hold instances of big object as global variables.

Drawback:

Makes it hard to test a class that uses the singleton instance.

Alternative: Dependency Injection frameworks (e.g. Dagger)

Factory Method

Factory Method

The Problem:

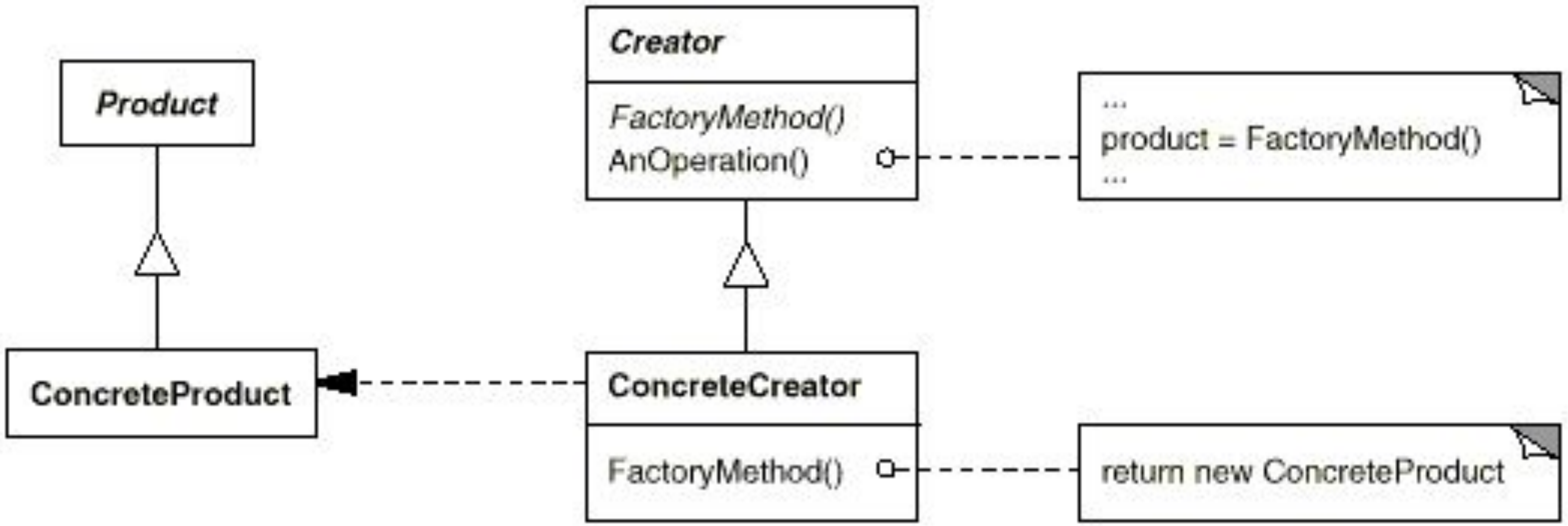
A class hierarchy is changing a lot. You need to create different objects in that hierarchy, but it is variable which class should be instantiated.

You want to separate the parts of your code that will change the most from the rest of your application.

Factory Method

> “Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”
>
> ~ Gang of Four

Factory Method



Factory Method

The Pattern:

1. Create an abstract class `Creator` with an abstract method `create()` that returns the superclass or interface of the type of object you want to create.
2. For each class, create a subclass of `Creator`. Return a concrete object of that class in `create()`.

Factory Method

`Let’s touch some code.`

Factory Method

Use if ...

… a class can't anticipate the class of objects it must create.

… a class wants its subclasses to specify the objects it creates.

Factory Method

Consequences:

Factory methods eliminate the need to bind application-specific classes into your code.

Drawback:

Clients might have to subclass the Creator class just to create a particular ConcreteProduct object.

Commonly overused pattern in Java enterprise applications.

Abstract Factory

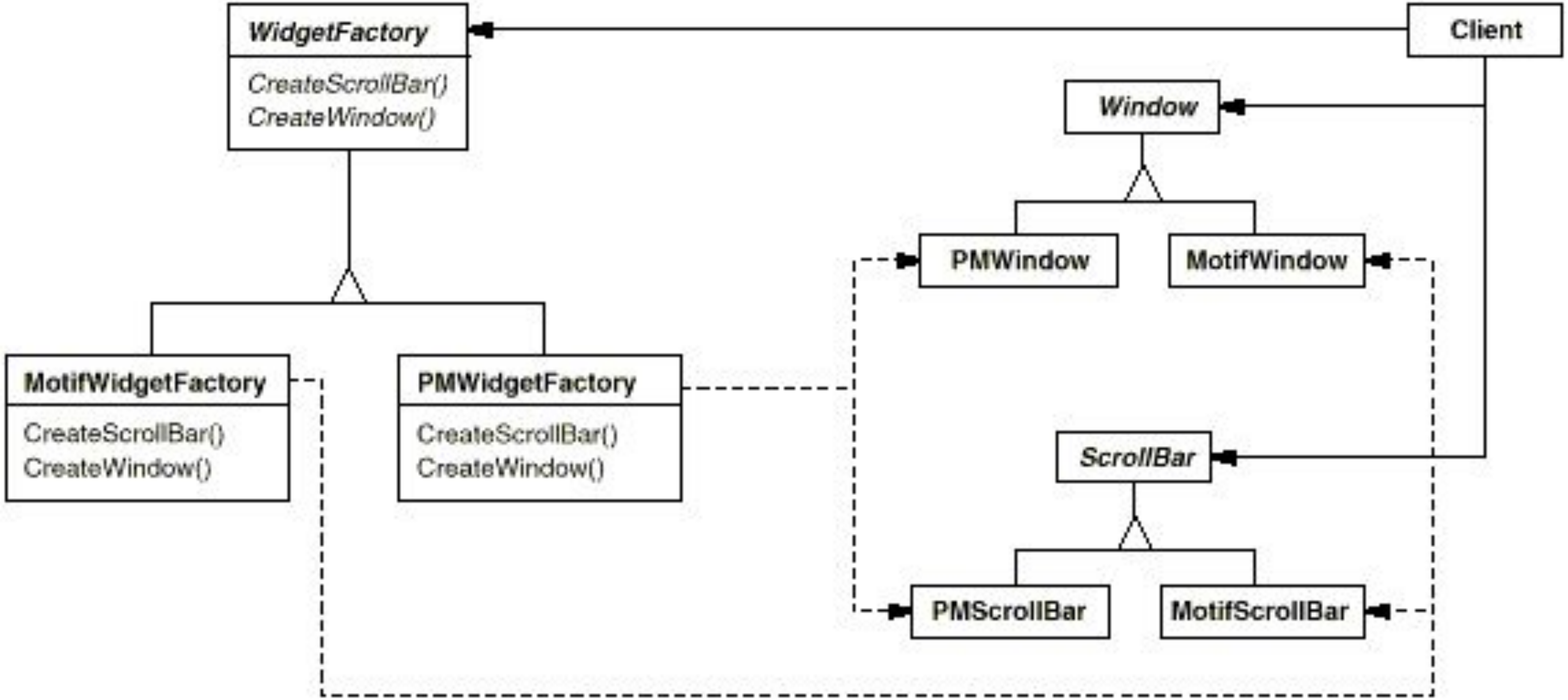
Abstract Factory

The problem:

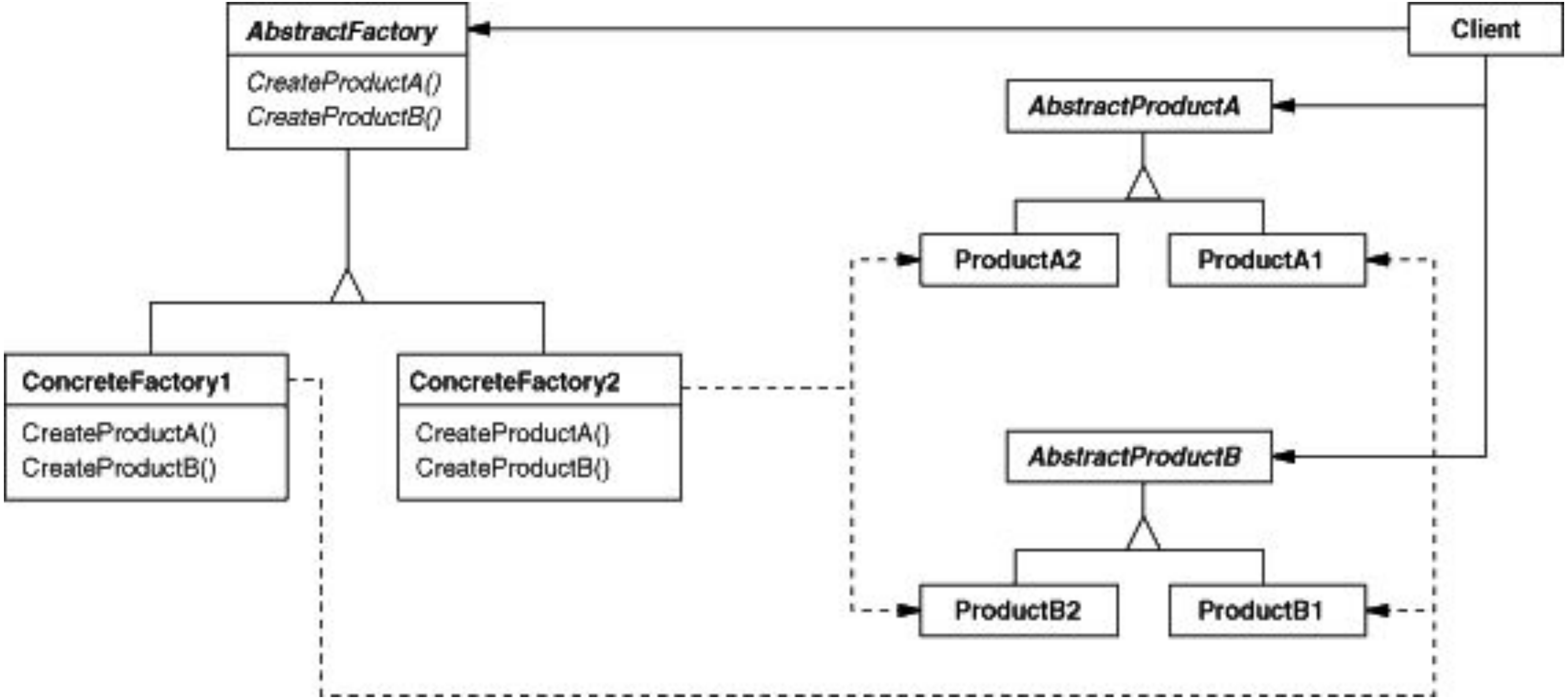
We have parallel hierarchies, and we need to instantiate its objects.

The client doesn’t know which ones to instantiate.

Abstract Factory



Abstract Factory



Abstract Factory

The Pattern:

1. Create an abstract class `AbstractFactory` with an abstract `create()` method for each high-level class of the parallel hierarchies.
2. For each variation (children in the parallel hierarchies), create a factory class that implements `AbstractFactory` and its abstract methods.
3. When you implement the `create()` methods, return an instance of the respective product class.

Abstract Factory

`Let’s touch some code.`

Abstract Factory

Use if ...

… you have parallel class hierarchies or families of classes, and

… you want to separate the creation of instances from the system.

Abstract Factory

Consequences:

It isolates the creation process from the system.

It makes exchanging product families easy.

Drawback:

Introducing new kinds of products is difficult.

Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
Singleton	Composite	Observer
Factory Method	Flyweight	Command
Abstract Factory	Decorator	Mediator
Builder	Proxy	Strategy
Prototype	Adapter	Chain of Responsibility
	Facade	Visitor
	Bridge	Interpreter
		Iterator
		Memento
		State
		Template Method

Thank you!

Hamburg Coding School
www.hamburgcodingschool.com

teresa@hamburgcodingschool.com