

C++ Software Design @ O'Reilly

# 3. Design Pattern Cheat Sheet

---

Klaus Iglberger  
December, 5th-6th, 2024

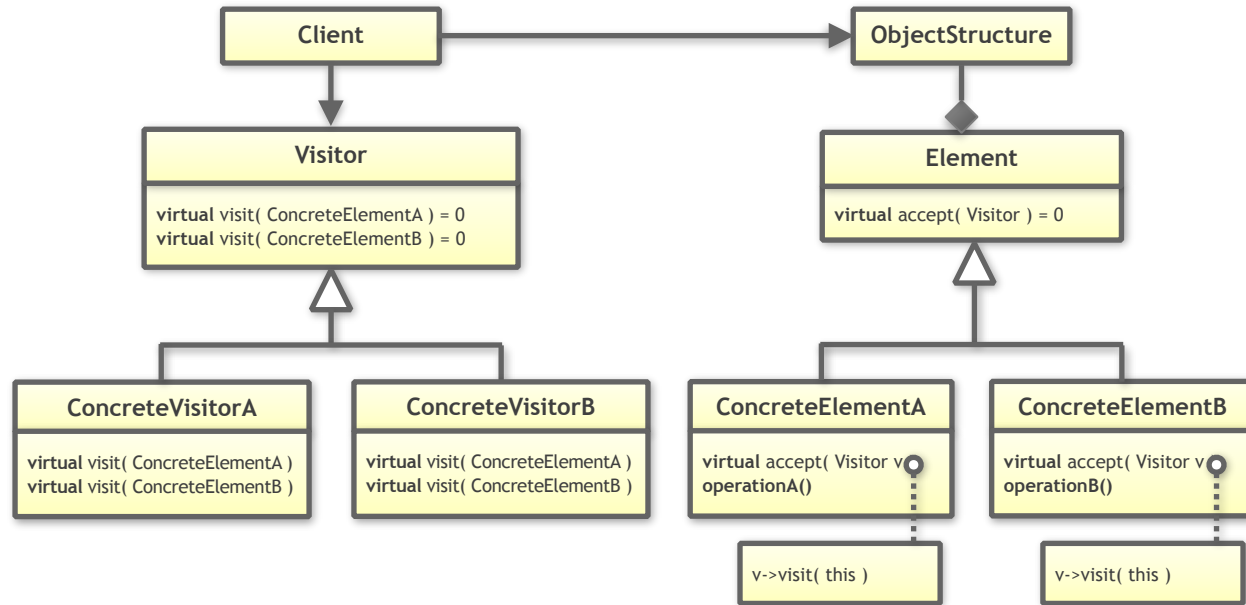
### 3. Design Pattern Cheat Sheet

**Name:** Visitor

**Origin:** GoF  
**Year:** 1994

**Intent:** Represent an operation to be performed of the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Structure:**



**Advantages/Strengths:**

- Easy addition of new operations
- Modern form (std::variant) is non-intrusive

**Disadvantages/Weaknesses:**

- Difficult addition of new types
- Classic form is intrusive (due to the `accept()` function)

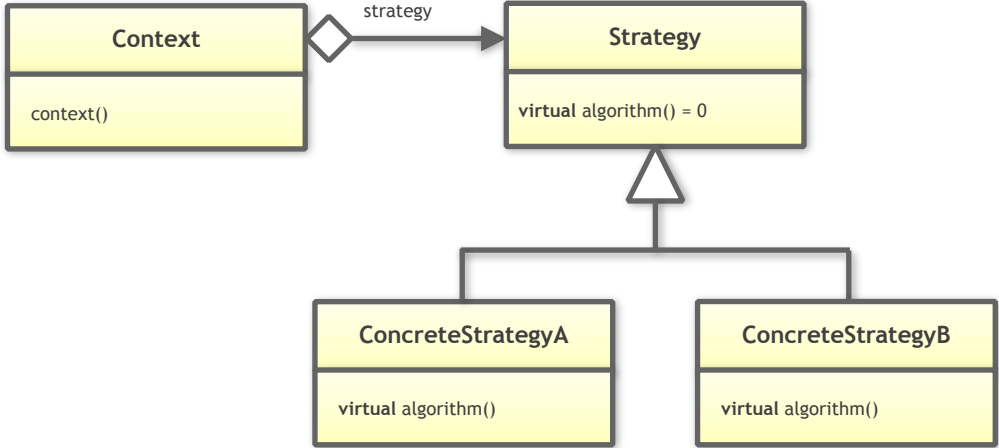
**Relation to other design patterns:**

- External Polymorphism:** Both separate operations from types, but Visitor enables the addition of operations.

**Implementation notes:**

- Often implemented by means of `std::variant` (C++17).

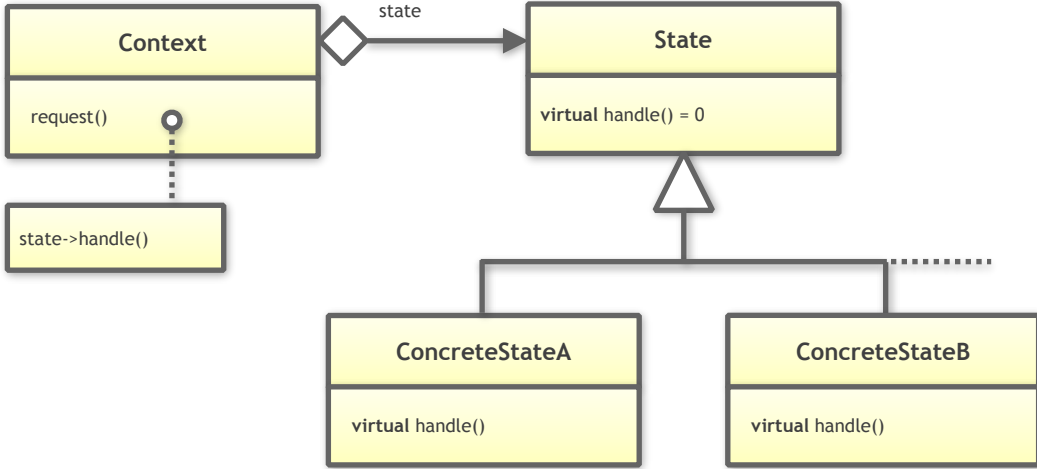
### 3. Design Pattern Cheat Sheet

<b>Name: Strategy</b>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.		
<b>Structure:</b>  <pre>classDiagram     class Context {         context()     }     class Strategy {         &lt;&lt;abstract&gt;&gt;         virtual algorithm() = 0     }     class ConcreteStrategyA {         virtual algorithm()     }     class ConcreteStrategyB {         virtual algorithm()     }     Context o--&gt; Strategy : strategy     Strategy &lt; -- ConcreteStrategyA     Strategy &lt; -- ConcreteStrategyB</pre>		
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Logical decoupling of interface and implementation details</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>Intrusive (because of dependency injection)</li><li>Proliferation of different abstractions for different strategies</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>Command:</b> Structurally identical to Strategy, but specifies WHAT should be done, instead of HOW.</li><li><b>Bridge:</b> Structurally identical to Strategy, but used only inside a class to switch between possible implementations.</li><li><b>State:</b> Structurally identical to Strategy, but used only internally to switch behavior based on some input.</li></ul>		
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>Can be implemented by means of <code>std::function</code> (C++11), which enables value semantics.</li></ul>		

### 3. Design Pattern Cheat Sheet

<b>Name: Command</b>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.		
<b>Structure:</b> 		
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>💡 Logical decoupling of interface and implementation details</li><li>💡 Non-intrusive design pattern</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>💡 —</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li>💡 <b>Strategy:</b> Structurally identical to Command, but specifies HOW should be done, instead of WHAT.</li><li>💡 <b>Bridge:</b> Structurally identical to Command, but used only inside a class to switch between possible implementations.</li><li>💡 <b>State:</b> Structurally identical to Command, but used only internally to switch behavior based on some input.</li></ul>		
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>💡 Can be implemented by means of std::function (C++11), which enables value semantics.</li></ul>		

### 3. Design Pattern Cheat Sheet

<b>Name:</b> State	<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	
<b>Structure:</b> 	
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Structured representation of state machines</li></ul>	<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>Strong coupling between states and transitions</li><li><b>Intrusive</b> design pattern (i.e. usually requires changes when new states/transitions are introduced).</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>Strategy:</b> Structurally identical to State, but exposed to clients (dependency injection).</li><li><b>Command:</b> Structurally identical to State, but used externally to encapsulate operations.</li><li><b>Bridge:</b> Structurally identical to State, but used only inside a class to switch between possible implementations.</li></ul>	
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>Can be implemented by means of <code>std::variant</code> (C++17), which enables a similar separation of concerns and value semantics.</li></ul>	

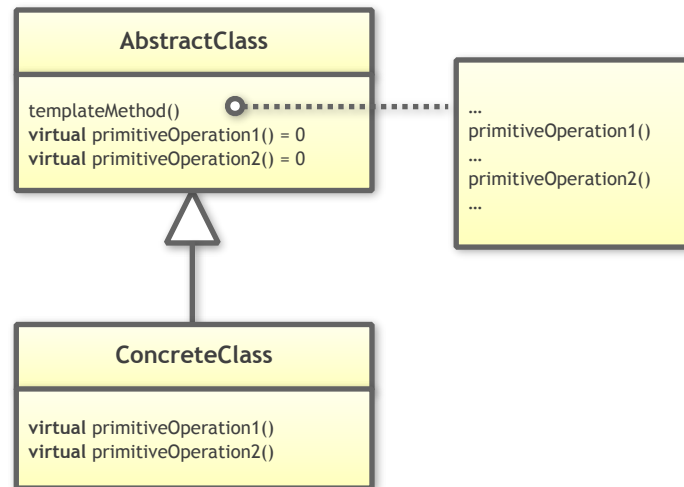
### 3. Design Pattern Cheat Sheet

**Name:** Template Method

**Origin:** GoF  
**Year:** 1994

**Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm, without changing the algorithms structure.

**Structure:**



**Advantages/Strengths:**

- Separation of interface and implementation details

**Disadvantages/Weaknesses:**

- —

**Relation to other design patterns:**

- —

**Implementation notes:**

- Template Method is the basis for the Non-Virtual Interface Idiom (NVI).

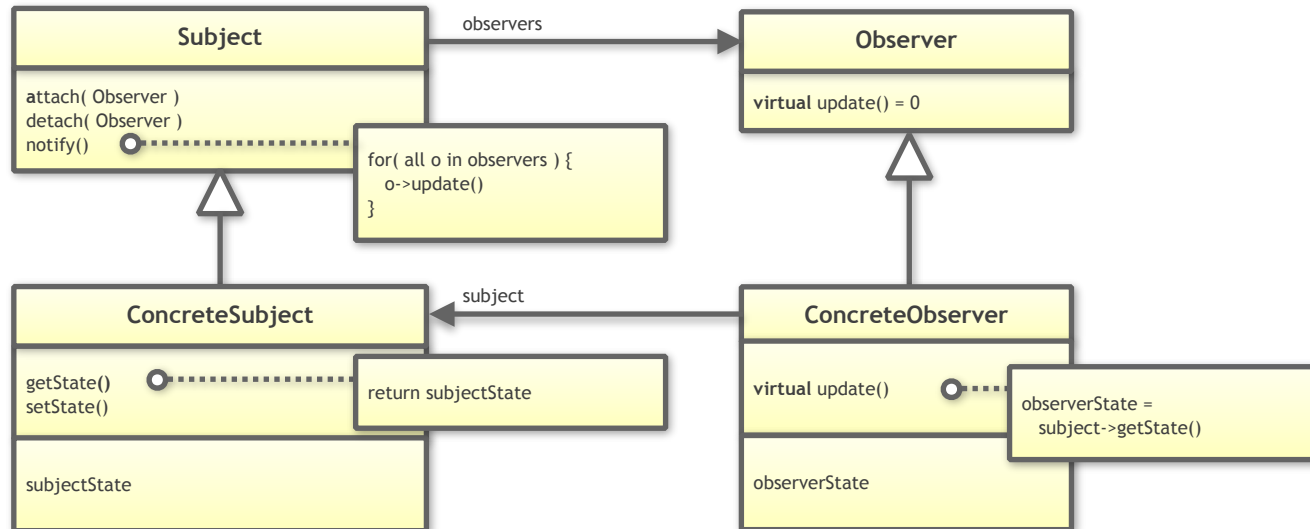
### 3. Design Pattern Cheat Sheet

#### Name: Observer

Origin: GoF  
Year: 1994

**Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

#### Structure:



#### Advantages/Strengths:

- 💡 Separation of interface and implementation details

#### Disadvantages/Weaknesses:

- 💡 Intrusive design pattern

#### Relation to other design patterns:

💡 —

#### Implementation notes:

- 💡 Can be implemented by means of `std::function` (C++11), which enables value semantics.
- 💡 Can be implemented as push or pull observer

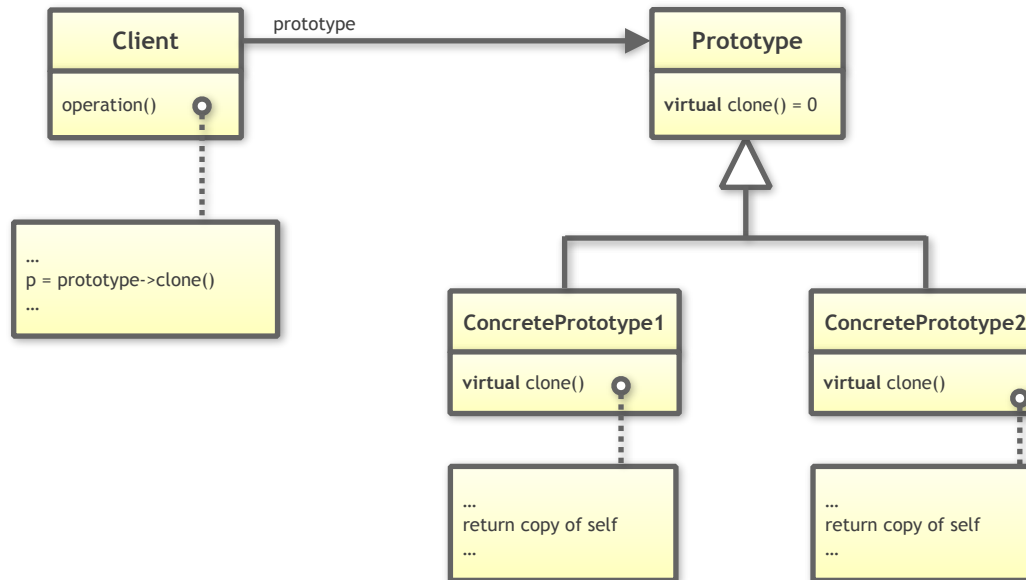
### 3. Design Pattern Cheat Sheet

**Name: Prototype**

*Origin:* GoF  
*Year:* 1994

**Intent:** Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Structure:**



**Advantages/Strengths:**

💡 THE solution for virtual copying (`clone()`) acts like a keyword

**Disadvantages/Weaknesses:**

💡 Only applicable in an **OOP setting** with inheritance hierarchies  
💡 **Intrusive** design pattern

**Relation to other design patterns:**

💡 —

**Implementation notes:**

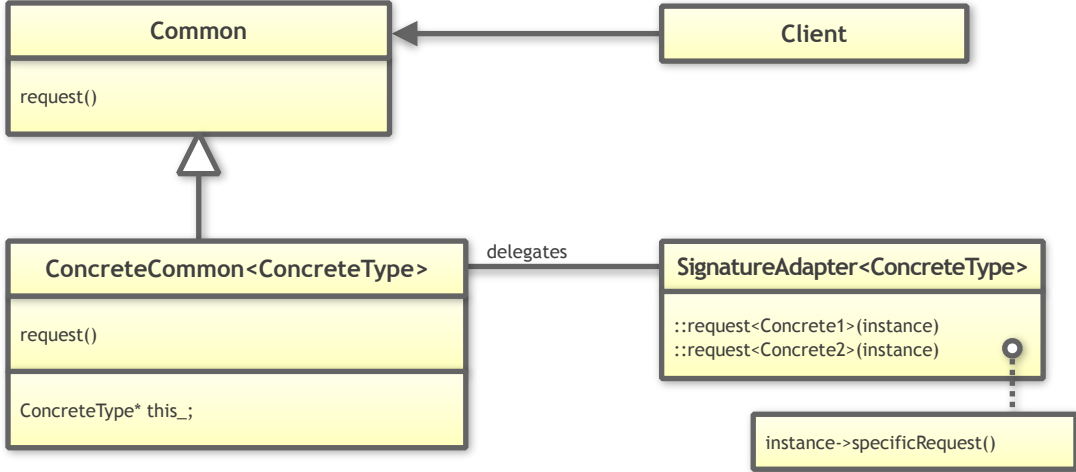
💡 There is no new (modern) implementation. Therefore the function name `clone()` is very strongly attached to this design pattern.



### 3. Design Pattern Cheat Sheet

<b>Name:</b> <i>Bridge</i>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Decouple an abstraction from its implementation so that the two can vary independently.		
<b>Structure:</b> <pre>classDiagram     class Abstraction {         operation()     }     class Implementor {         virtual operation() = 0     }     class ConcreteImplA {         virtual operation()     }     class ConcreteImplB {         virtual operation()     }     Abstraction o--&gt; Implementor : impl     Implementor &lt; -- ConcreteImplA     Implementor &lt; -- ConcreteImplB</pre>		
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>💡 Strong decoupling of physical dependencies</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>💡 Introduces a performance penalty</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li>💡 <b>Strategy:</b> Structurally identical to Bridge, but exposed to clients (dependency injection).</li><li>💡 <b>Command:</b> Structurally identical to Bridge, but used externally to encapsulate operations.</li><li>💡 <b>State:</b> Structurally identical to Bridge, but used only internally to switch behavior based on some input.</li></ul>		
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>💡 As an alternative to the dynamic memory, it is possible to implement a Bridge by means of Small-Buffer-Optimization (see Fast Pimpl).</li><li>💡 In case a <code>std::unique_ptr</code> is used, the destructor of the <code>Implementor</code> class still needs to be defined in the source file.</li></ul>		

### 3. Design Pattern Cheat Sheet

<b>Name:</b> External Polymorphism	<b>Origin:</b> “External Polymorphism” by Cleeland, Schmidt and Harrison <b>Year:</b> 1996
<b>Intent:</b> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	
<b>Structure:</b>  <pre>graph TD     Client --&gt; Common     class Common {         request()     }     class ConcreteCommon["ConcreteCommon&lt;ConcreteType&gt;"] {         request()         ConcreteType* this_     }     class SignatureAdapter["SignatureAdapter&lt;ConcreteType&gt;"] {         request&lt;Concrete1&gt;(instance)         request&lt;Concrete2&gt;(instance)     }     class ConcreteTypeRequest["instance-&gt;specificRequest()"]     ConcreteCommon -- &gt; Common     ConcreteCommon -- delegates --&gt; SignatureAdapter     SignatureAdapter -.-&gt; ConcreteTypeRequest</pre> <p>The diagram illustrates the structure of External Polymorphism. It features a <b>Common</b> interface with a <code>request()</code> method. A <b>Client</b> depends on the <b>Common</b> interface. A <b>ConcreteCommon&lt;ConcreteType&gt;</b> class inherits from <b>Common</b> and implements the <code>request()</code> method. It holds a reference to a <b>ConcreteType</b> object (<code>ConcreteType* this_;</code>). A <b>SignatureAdapter&lt;ConcreteType&gt;</b> class is associated with <b>ConcreteCommon</b> via a <code>delegates</code> relationship. The <b>SignatureAdapter</b> class has two methods: <code>::request&lt;Concrete1&gt;(instance)</code> and <code>::request&lt;Concrete2&gt;(instance)</code>. It holds a reference to a <b>ConcreteType</b> object, which then calls <code>instance-&gt;specificRequest()</code> to fulfill the request.</p>	
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Very strong decoupling of types and operations</li><li>Easy to add new types</li><li>Non-intrusive design pattern</li></ul>	<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>Difficult to introduce new operations</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>Adapter:</b> Adapter is based on an existing inheritance hierarchy, while External Polymorphism introduces a new hierarchy.</li></ul>	
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>Possible optimizations: Small Buffer Optimization (SBO), manual virtual function tables, ...</li></ul>	

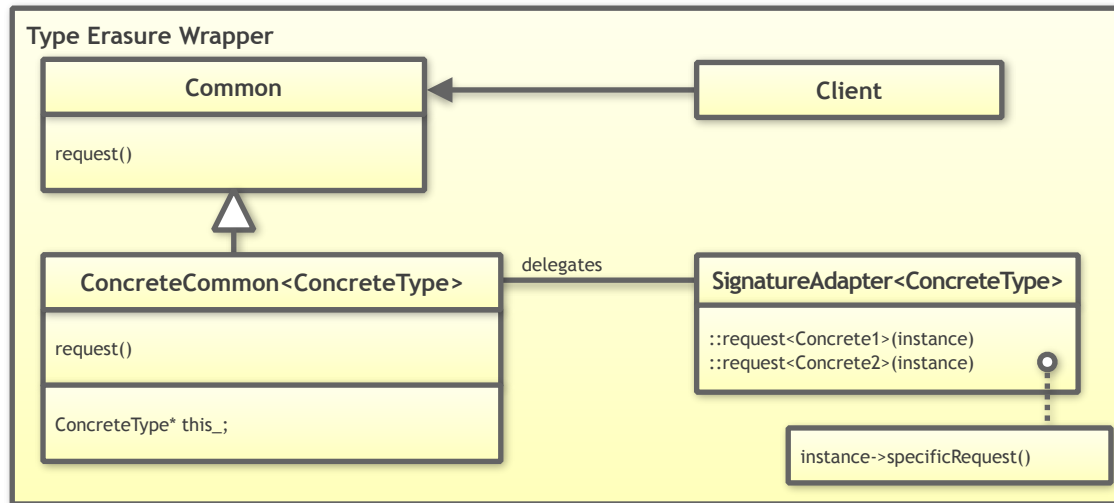
### 3. Design Pattern Cheat Sheet

**Name:** Type Erasure

**Origin:** “Valued Conversions” by Kevin Henney  
**Year:** 2000

**Intent:** Provide a value-based, non-intrusive abstraction for an extendable set of unrelated, potentially non-polymorphic types with the same semantic behavior.

**Structure:**



**Advantages/Strengths:**

- Very strong decoupling of types and operations
- Easy to add new types
- Non-intrusive** design pattern
- Value Semantics**

**Disadvantages/Weaknesses:**

- Difficult to introduce new operations
- Somewhat tricky implementation details

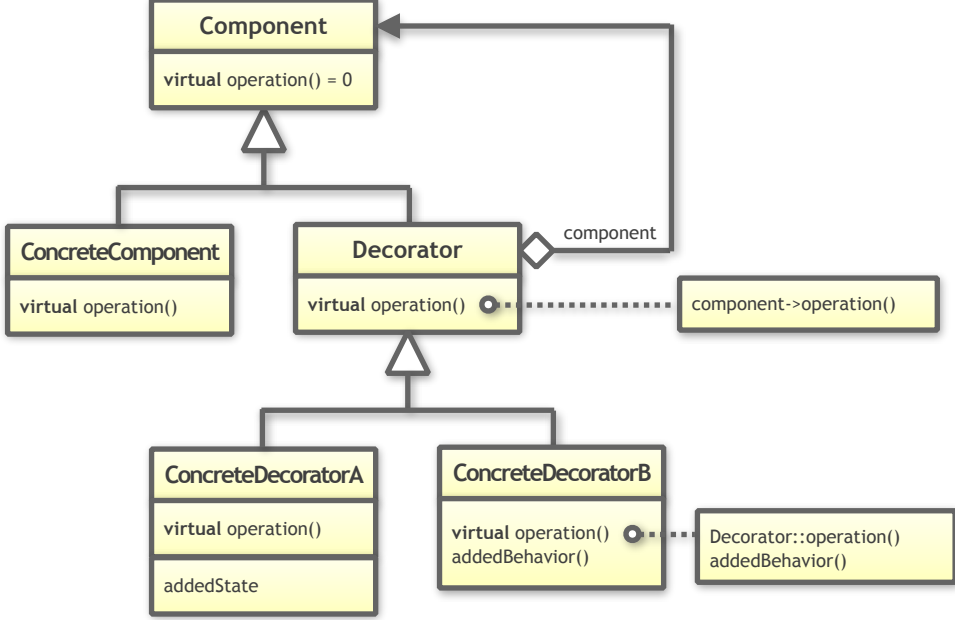
**Relation to other design patterns:**

- External Polymorphism:** Type Erasure is the value semantics based solution of External Polymorphism.
- Bridge:** Type Erasure implements a bridge to the private implementation details

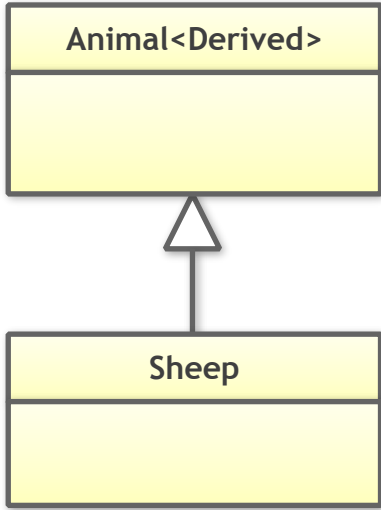
**Implementation notes:**

- Possible optimizations: Small Buffer Optimization (SBO), manual virtual function tables, ...
- Type Erasure is a **non-intrusive** design pattern.

### 3. Design Pattern Cheat Sheet

<b>Name:</b> Decorator	<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.	
<b>Structure:</b> 	
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Non-intrusive addition of behavior for existing types</li></ul>	<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>Difficult to introduce new functions</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>Strategy:</b> Strategy is focused on extracting some implementation detail, Decorator is extending the functionality.</li><li><b>Adapter:</b> Adapter changes an interface, Decorator preserves it.</li></ul>	
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>Can be implemented by means of an inheritance hierarchy, with Type Erasure, or as template.</li></ul>	

### 3. Design Pattern Cheat Sheet

<b>Name:</b> CRTP		<b>Origin:</b> “Curiously Recurring Template Patterns”, James Copley <b>Year:</b> 1995	
<b>Intent:</b> Define a compile-time abstraction for a family of related types.			
<b>Structure:</b> <div><pre>classDiagram     class AnimalDerived["Animal&lt;Derived&gt;"]     class Sheep     Sheep -- &gt; AnimalDerived</pre></div>			
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>💡 Suited for maximum performance, no runtime overhead</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>💡 Loss of a common base class</li><li>💡 Template-heavy</li></ul>	
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li>💡 <b>Expression Templates:</b> Combines very well with the intention of Expression Templates.</li></ul>			
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>💡 Can be implemented by means of an inheritance hierarchy, with Type Erasure, or as template.</li><li>💡 CRTP is an <b>intrusive</b> design pattern.</li></ul>			

### 3. Design Pattern Cheat Sheet

<b>Name:</b> <b>Expression Templates</b>	<b>Origin:</b> “Expression Templates”, Todd Veldhuizen <b>Year:</b> 1995
<b>Intent:</b> Introduce lazy evaluation for expressions.	
<b>Structure:</b> <pre>classDiagram     class Expression["Expression&lt;Operand1, Operand2&gt;"] {         commonInterface1()         commonInterface2()     }     class Operand1 {         commonInterface1()         commonInterface2()     }     class Operand2 {         commonInterface1()         commonInterface2()     }     Expression --&gt; Operand1 : delegates     Expression --&gt; Operand2 : delegates</pre>	
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>💡 Suited for <b>maximum performance</b>, no runtime overhead</li></ul>	<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>💡 Template-heavy</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li>💡 <b>Decorator:</b> Expressions templates are based on the Decorator design pattern.</li></ul>	
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>💡 Expression Templates is a <b>non-intrusive</b> design pattern.</li></ul>	

### 3. Design Pattern Cheat Sheet

<b>Name:</b> Adapter	<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	
<b>Structure:</b> <pre>classDiagram     class DrawingEditor     class Shape {         virtual BondingBox()         virtual CreateManipulator()     }     class Line {         virtual BondingBox()         virtual CreateManipulator()     }     class TextShape {         virtual BondingBox()         virtual CreateManipulator()     }     class TextView {         GetExtent()     }     DrawingEditor --&gt; Shape     Shape &lt; -- Line     Shape &lt; -- TextShape     TextShape --&gt; TextView     TextShape ..&gt; TextView : return text-&gt;getExtent()     TextShape ..&gt; TextManipulator : return new TextManipulator()</pre>	
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Non-intrusive design pattern</li><li>Applicable for both classes and functions</li></ul>	<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>—</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>External Polymorphism:</b> EP creates a new inheritance hierarchy, while Adapter is based on an existing hierarchy.</li><li><b>Decorator:</b> Decorator preserves an interface and adds behavior, while Adapter changes an interface and does not add behavior</li></ul>	
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>Next to the classical inheritance-based implementation, adapters can be templates (e.g. <code>std::stack</code> and <code>std::queue</code>) and simple functions (shims; e.g. <code>std::begin()</code>).</li></ul>	

### 3. Design Pattern Cheat Sheet

<b>Name: Proxy</b>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Provide a surrogate or placeholder for another object to control access to it.		
<b>Structure:</b> <pre>classDiagram     class Graphic {         &lt;&lt;abstract&gt;&gt;         virtual Draw()         virtual GetExtent()         virtual Store()         virtual Load()     }     class Image {         virtual Draw()         virtual GetExtent()         virtual Store()         virtual Load()         imageImp         extent     }     class ImageProxy {         virtual Draw()         virtual GetExtent()         virtual Store()         virtual Load()         fileName         extent     }     Graphic &lt; -- Image     Graphic &lt; -- ImageProxy     ImageProxy ..&gt; Image : association</pre>		
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Non-intrusive design pattern</li><li>Usually invisible to the caller.</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>—</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>Adapter:</b> Proxy is focused on managing access, while Adapter is focused on changing an interface</li><li><b>Decorator:</b> Decorators can be combined hierarchically, Proxies cannot</li></ul>		
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>Can appear in a class hierarchy, but also in the context of templates (e.g. <code>std::vector&lt;bool&gt;</code> or <code>std::bitset&lt;N&gt;::operator[]</code>)</li></ul>		



### 3. Design Pattern Cheat Sheet

<b>Name: Factory Method</b>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.		
<b>Structure:</b> <pre>classDiagram     class Product {     }     class ConcreteProduct {     }     class Creator {         virtual factoryMethod() = 0     }     class ConcreteCreator {         virtual factoryMethod()     }     Product &lt; -- ConcreteProduct     Creator &lt; -- ConcreteCreator     ConcreteCreator ..&gt; Product : product = factoryMethod();     ConcreteCreator ..&gt; ConcreteProduct : return new ConcreteProduct();</pre>		
<b>Advantages/Strengths:</b> 💡 Logical decoupling of interface and implementation details		<b>Disadvantages/Weaknesses:</b> 💡 Intrusive (because of dependency injection)
<b>Relation to other design patterns:</b> 💡 <b>Strategy:</b> Factory Method is very similar to Strategy, but focused on creating something (possibly introducing a second abstraction)		
<b>Implementation notes:</b> 💡 New products should not be returned by raw pointer, but by <code>std::unique_ptr</code> , Type Erasure, or <code>std::variant</code> . 💡 A simple function creating something is often called a “factory function”, which has nothing to do with this design pattern.		

### 3. Design Pattern Cheat Sheet

<b>Name: Builder</b>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Separate the construction of a complex object from its representation so that the same construction process can create different representations.		
<b>Structure:</b> <pre>classDiagram     class Director {         +construct()     }     class Builder {         +virtual buildPart() = 0     }     class ConcreteBuilder {         +virtual buildPart() getResult()     }     Director o-- Builder : impl     Builder &lt; -- ConcreteBuilder     Director ..&gt; : for all objects in structure { builder-&gt;buildPart() }</pre>		
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>Logical separation of the steps of a build process</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>Usually intrusive (because of dependency injection)</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li><b>Factory Method:</b> Builder is usually composed of several Factory Methods</li></ul>		
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>New products should not be returned by raw pointer, but by <code>std::unique_ptr</code>, Type Erasure, or <code>std::variant</code>.</li></ul>		

### 3. Design Pattern Cheat Sheet

<b>Name: Iterator</b>		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.		
<b>Structure:</b> <pre>classDiagram     class Aggregate {         +createliterator()     }     class ConcreteAggregate {         +createliterator()     }     class Iterator {         +virtual first() = 0         +virtual next() = 0         +virtual isDone() = 0         +virtual currentItem() = 0     }     class Concreteliterator {     }     class Client {     }     Aggregate &lt; -- ConcreteAggregate     Iterator &lt; -- Concreteliterator     Client --&gt; Aggregate     Client --&gt; Iterator     ConcreteAggregate ..&gt; Concreteliterator</pre>		
<b>Advantages/Strengths:</b> <ul style="list-style-type: none"><li>💡 Very idiomatic in C++ due to the STL</li></ul>		<b>Disadvantages/Weaknesses:</b> <ul style="list-style-type: none"><li>💡 Separation in three steps (increment, compare, access) opens the possibility of access violations</li></ul>
<b>Relation to other design patterns:</b> <ul style="list-style-type: none"><li>💡 —</li></ul>		
<b>Implementation notes:</b> <ul style="list-style-type: none"><li>💡 In C++, it is very unusual to implement this pattern in the form of an inheritance hierarchy.</li><li>💡 A Type Erasure implementation of Iterator would have to build on the GoF form because of the inequality comparison of iterators.</li></ul>		

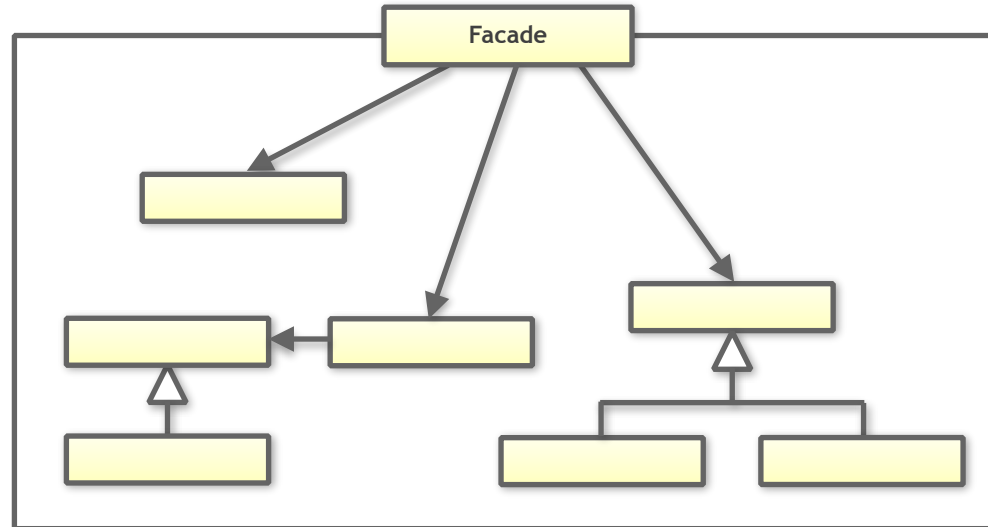
### 3. Design Pattern Cheat Sheet

**Name:** Facade

**Origin:** GoF  
**Year:** 1994

**Intent:** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

**Structure:**



**Advantages/Strengths:**

- Great simplification of complexity
- Non-intrusive design pattern

**Disadvantages/Weaknesses:**

- 

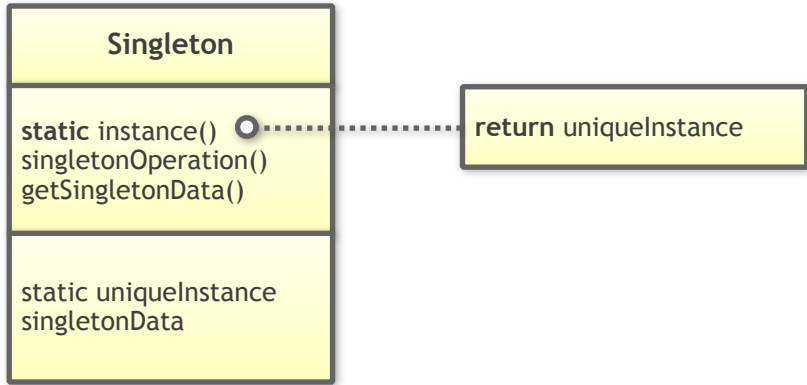
**Relation to other design patterns:**

- 

**Implementation notes:**

- Can be a class (not necessarily a base class) or function.

### 3. Design Pattern Cheat Sheet

<b>Name:</b> Singleton		<b>Origin:</b> GoF <b>Year:</b> 1994
<b>Intent:</b> Ensure a class only has one instance, and provide a global point of access to it.		
<b>Structure:</b>  		
<b>Advantages/Strengths:</b> 💡 —		<b>Disadvantages/Weaknesses:</b> 💡 Destroys design/architecture due to lack of dependency management 💡 Provides the characteristics of a global variable/constant
<b>Relation to other design patterns:</b> 💡 —		
<b>Implementation notes:</b> 💡 Singleton is not a design pattern, as it doesn't provide any abstraction or dependency reduction. It is an implementation pattern.		

email: [klaus.iglberger@gmx.de](mailto:klaus.iglberger@gmx.de)

LinkedIn: [linkedin.com/in/klaus-iglberger](https://www.linkedin.com/in/klaus-iglberger)

Xing: [xing.com/profile/Klaus\\_Iglberger/cv](https://www.xing.com/profile/Klaus_Iglberger/cv)