

C++ Software Design @ O'Reilly

3. Design Pattern Cheat Sheet

Klaus Iglberger

March, 3rd-4th, 2025

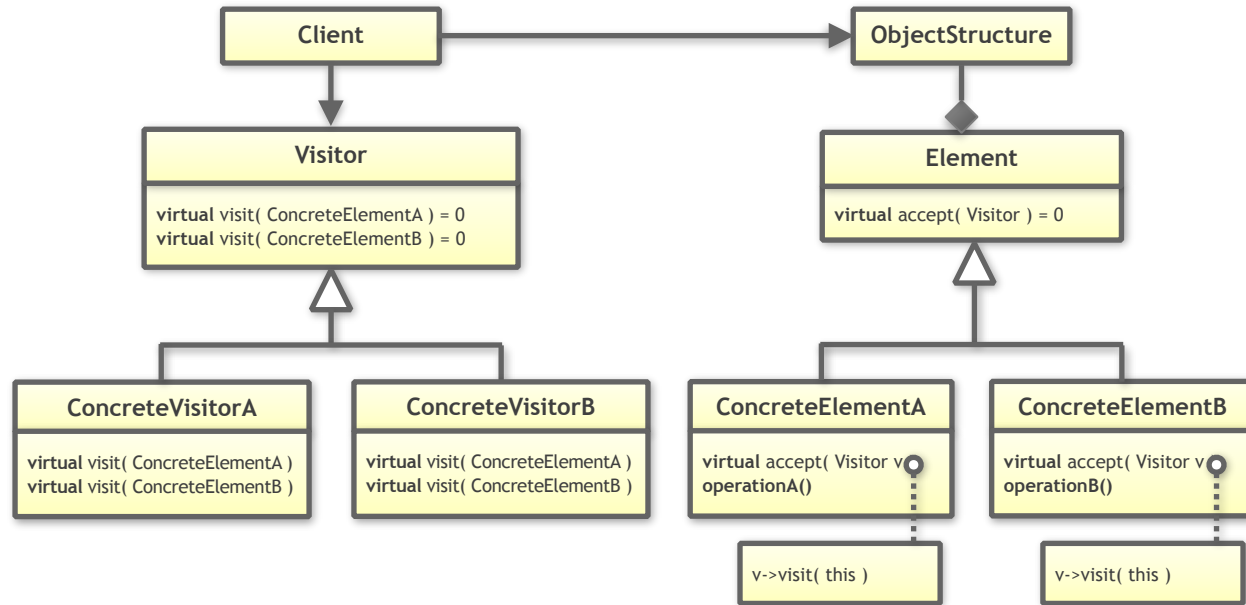
3. Design Pattern Cheat Sheet

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

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

3. Design Pattern Cheat Sheet

Name: Command		Origin: GoF Year: 1994
Intent: Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.		
Structure: <pre>classDiagram class Client class Invoker class Command { <<abstract>> virtual execute() = 0 } class ConcreteCommand { execute() receiver state } class Receiver { action() } Client --> Invoker Client --> Receiver Client --> ConcreteCommand Invoker *--> Command Command < -- ConcreteCommand ConcreteCommand --> Receiver : receiver ConcreteCommand --> Receiver : receiver->action() ConcreteCommand --> state</pre>		
Advantages/Strengths: <ul style="list-style-type: none">💡 Logical decoupling of interface and implementation details💡 Non-intrusive design pattern		Disadvantages/Weaknesses: <ul style="list-style-type: none">💡 —
Relation to other design patterns: <ul style="list-style-type: none">💡 Strategy: Structurally identical to Command, but specifies HOW should be done, instead of WHAT.💡 Bridge: Structurally identical to Command, but used only inside a class to switch between possible implementations.💡 State: Structurally identical to Command, but used only internally to switch behavior based on some input.		
Implementation notes: <ul style="list-style-type: none">💡 Can be implemented by means of <code>std::function</code> (C++11), which enables value semantics.		

3. Design Pattern Cheat Sheet

Name: State		Origin: GoF Year: 1994
Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.		
Structure: <pre>classDiagram class Context { request() state } class State { virtual handle() = 0 } class ConcreteStateA { virtual handle() } class ConcreteStateB { virtual handle() } Context o--> State : state State < -- ConcreteStateA State < -- ConcreteStateB Context ..> State : state->handle()</pre>		
Advantages/Strengths: <ul style="list-style-type: none">Structured representation of state machines		Disadvantages/Weaknesses: <ul style="list-style-type: none">Strong coupling between states and transitionsIntrusive design pattern (i.e. usually requires changes when new states/transitions are introduced).
Relation to other design patterns: <ul style="list-style-type: none">Strategy: Structurally identical to State, but exposed to clients (dependency injection).Command: Structurally identical to State, but used externally to encapsulate operations.Bridge: Structurally identical to State, but used only inside a class to switch between possible implementations.		
Implementation notes: <ul style="list-style-type: none">Can be implemented by means of <code>std::variant</code> (C++17), which enables a similar separation of concerns and value semantics.		

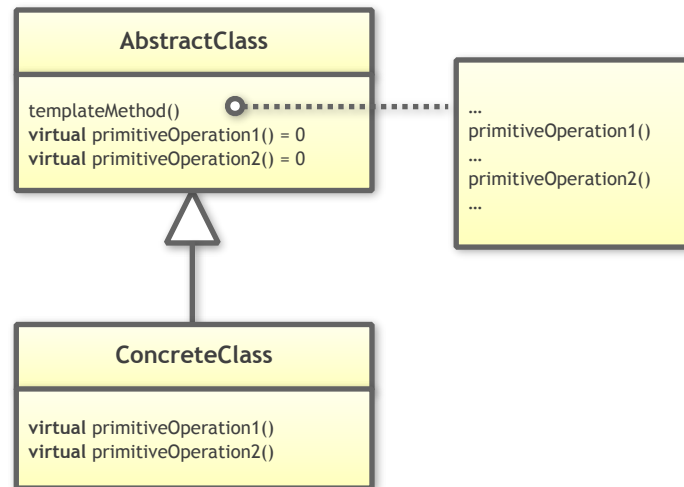
3. Design Pattern Cheat Sheet

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: <pre>classDiagram class Subject { +attach(Observer) +detach(Observer) +notify() } class Observer { +virtual update() } class ConcreteSubject { +getState() +setState() +subjectState } class ConcreteObserver { +virtual update() +observerState } Subject < -- ConcreteSubject Observer < -- ConcreteObserver Subject --> Observer : observers ConcreteObserver --> ConcreteSubject : subject</pre> <p>for (all o in observers) { o->update() }</p> <p>return subjectState</p> <p>observerState = subject->getState()</p>		
Advantages/Strengths: <ul style="list-style-type: none">💡 Separation of interface and implementation details		Disadvantages/Weaknesses: <ul style="list-style-type: none">💡 Intrusive design pattern
Relation to other design patterns: <ul style="list-style-type: none">💡 —		
Implementation notes: <ul style="list-style-type: none">💡 Can be implemented by means of std::function (C++11), which enables value semantics.💡 Can be implemented as push or pull observer		

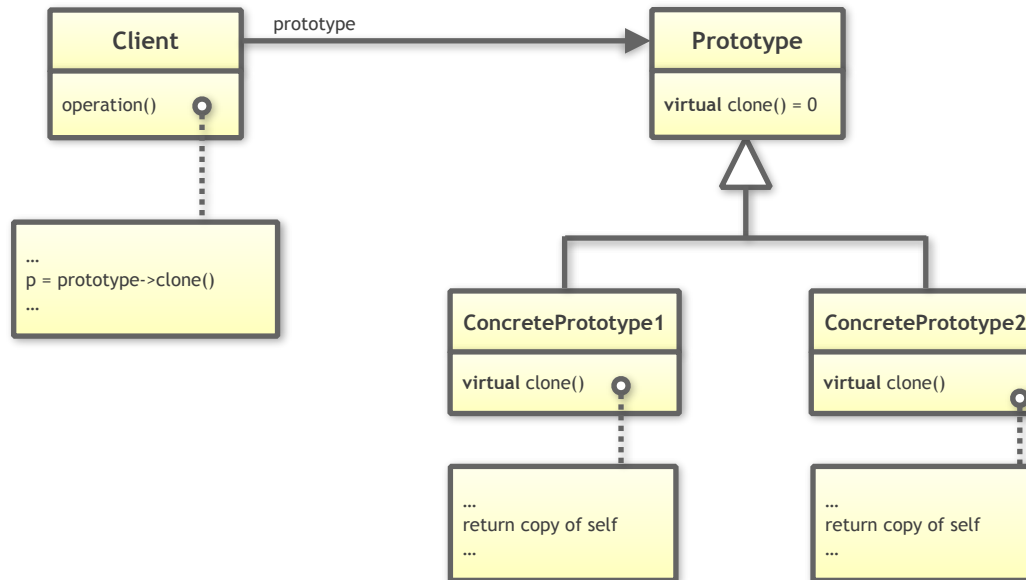
3. Design Pattern Cheat Sheet

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:

💡 —

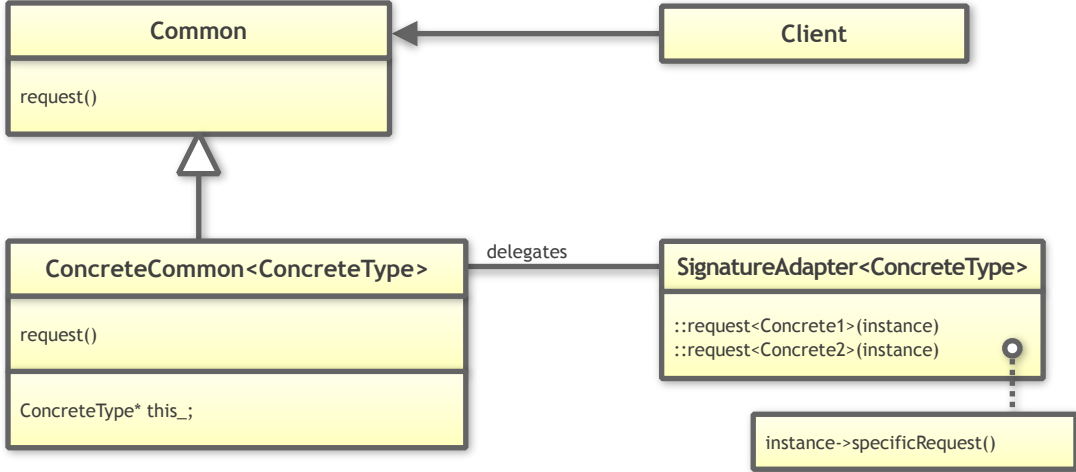
Implementation notes:

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

3. Design Pattern Cheat Sheet

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

3. Design Pattern Cheat Sheet

Name: External Polymorphism	Origin: “External Polymorphism” by Cleeland, Schmidt and Harrison Year: 1996
Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	
Structure:  <pre>graph TD Client --> Common class Common { request() } class ConcreteCommon["ConcreteCommon<ConcreteType>"] { request() ConcreteType* this_ } class SignatureAdapter["SignatureAdapter<ConcreteType>"] { requestConcrete1(instance) requestConcrete2(instance) } class Instance["instance->specificRequest()"] ConcreteCommon -- > Common ConcreteCommon -- delegates --> SignatureAdapter SignatureAdapter -.-> Instance</pre> <p>The diagram illustrates the structure of External Polymorphism. It features a Common interface with a <code>request()</code> method. A Client depends on the Common interface. ConcreteCommon<ConcreteType> is a concrete implementation that inherits from Common and implements the <code>request()</code> method. It holds a reference to a SignatureAdapter<ConcreteType> object, which it delegates the <code>request()</code> call to. The SignatureAdapter class has two methods: <code>request<Concrete1>(instance)</code> and <code>request<Concrete2>(instance)</code>. The <code>request<Concrete2>(instance)</code> method delegates the call to <code>instance->specificRequest()</code>.</p>	
Advantages/Strengths: <ul style="list-style-type: none">Very strong decoupling of types and operationsEasy to add new typesNon-intrusive design pattern	Disadvantages/Weaknesses: <ul style="list-style-type: none">Difficult to introduce new operations
Relation to other design patterns: <ul style="list-style-type: none">Adapter: Adapter is based on an existing inheritance hierarchy, while External Polymorphism introduces a new hierarchy.	
Implementation notes: <ul style="list-style-type: none">Possible optimizations: Small Buffer Optimization (SBO), manual virtual function tables, ...	

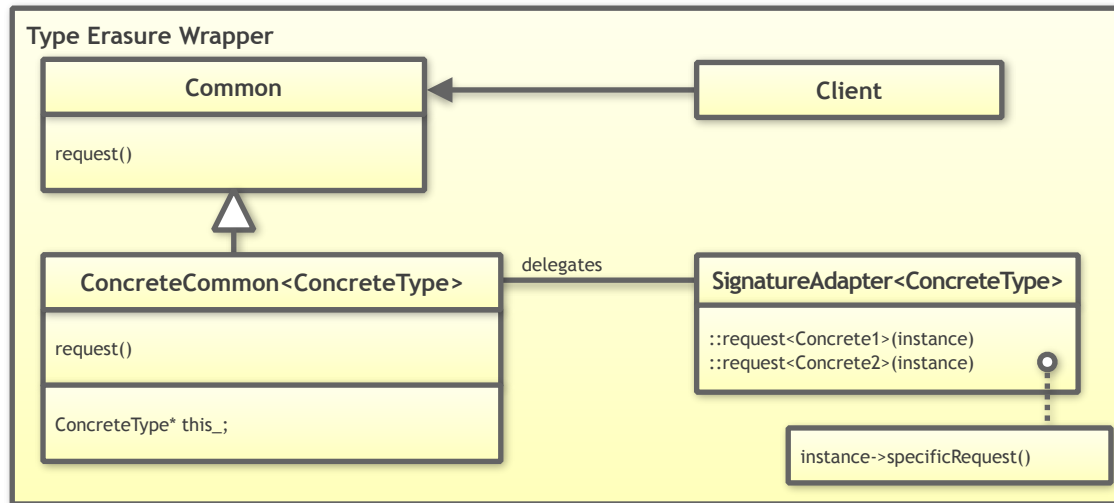
3. Design Pattern Cheat Sheet

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

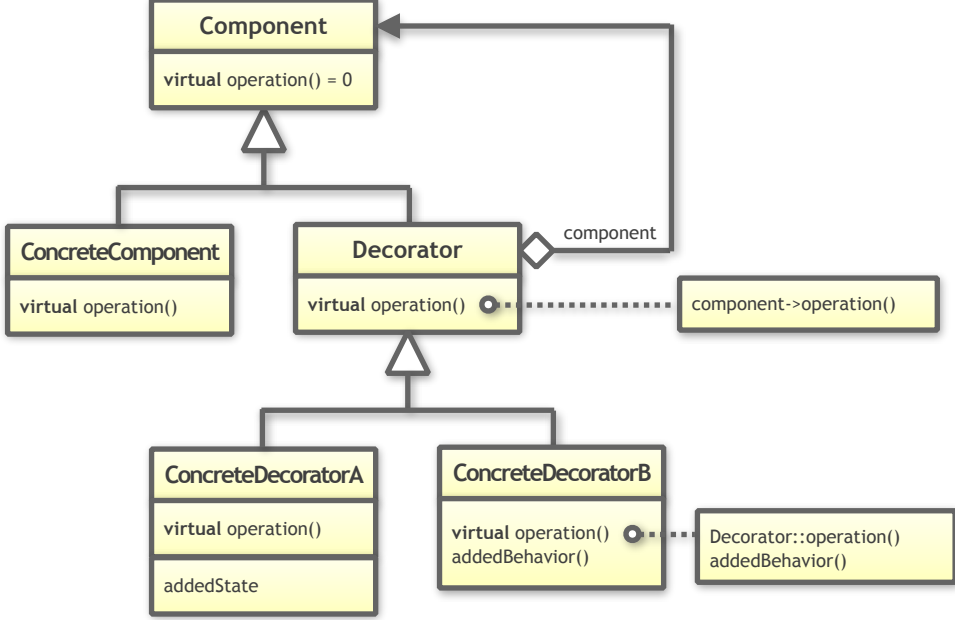
Relation to other design patterns:

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

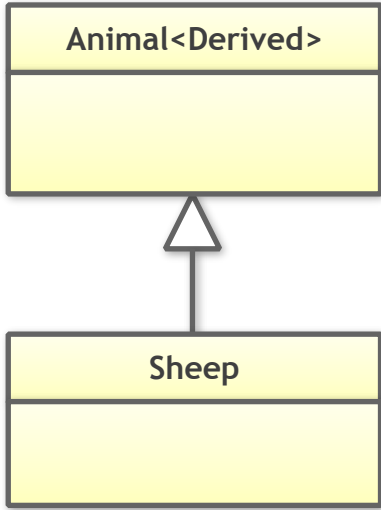
Implementation notes:

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

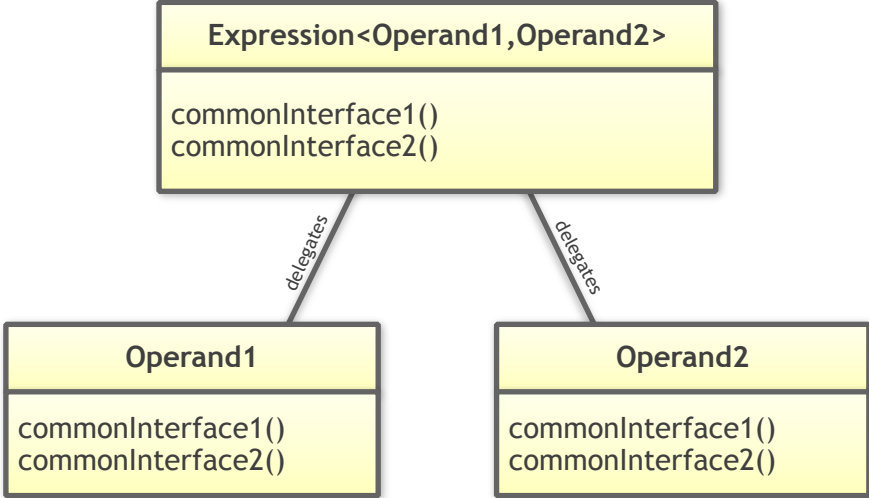
3. Design Pattern Cheat Sheet

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

3. Design Pattern Cheat Sheet

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

3. Design Pattern Cheat Sheet

Name: Expression Templates	Origin: “Expression Templates”, Todd Veldhuizen Year: 1995
Intent: Introduce lazy evaluation for expressions.	
Structure:  <pre>graph TD; Expression["Expression<Operand1, Operand2>"] Operand1["Operand1"] Operand2["Operand2"] Expression -- delegates --> Operand1 Expression -- delegates --> Operand2</pre> <p>The diagram illustrates the structure of Expression Templates. It features three classes: Expression<Operand1, Operand2>, Operand1, and Operand2. Each class is represented by a yellow box with a black border, divided into two sections. The top section contains the class name, and the bottom section contains the methods <code>commonInterface1()</code> and <code>commonInterface2()</code>. The Expression class is at the top, and it has two arrows pointing down to Operand1 and Operand2. Each arrow is labeled "delegates", indicating that the Expression class delegates the calls to <code>commonInterface1()</code> and <code>commonInterface2()</code> to the respective operand objects.</p>	
Advantages/Strengths: <ul style="list-style-type: none">• Suited for maximum performance, no runtime overhead	Disadvantages/Weaknesses: <ul style="list-style-type: none">• Template-heavy
Relation to other design patterns: <ul style="list-style-type: none">• Decorator: Expressions templates are based on the Decorator design pattern.	
Implementation notes: <ul style="list-style-type: none">• Expression Templates is a non-intrusive design pattern.	

3. Design Pattern Cheat Sheet

Name: Adapter		Origin: GoF Year: 1994
Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.		
Structure: <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 --> Shape Shape < -- Line Shape < -- TextShape TextShape --> TextView </pre>		
Advantages/Strengths: <ul style="list-style-type: none"> Non-intrusive design pattern Applicable for both classes and functions 		Disadvantages/Weaknesses: <ul style="list-style-type: none"> —
Relation to other design patterns: <ul style="list-style-type: none"> External Polymorphism: EP creates a new inheritance hierarchy, while Adapter is based on an existing hierarchy. Decorator: Decorator preserves an interface and adds behavior, while Adapter changes an interface and does not add behavior 		
Implementation notes: <ul style="list-style-type: none"> 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>). 		

3. Design Pattern Cheat Sheet

Name: Proxy		Origin: GoF Year: 1994
Intent: Provide a surrogate or placeholder for another object to control access to it.		
Structure: <pre>classDiagram class Graphic { <<abstract>> 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 < -- Image Graphic < -- ImageProxy ImageProxy ..> Image : if(image == 0) ImageProxy ..> Image : image = LoadImage(fileName); ImageProxy ..> Image : image->Draw(); ImageProxy ..> Image : if(image == 0) ImageProxy ..> Image : return extent; ImageProxy ..> Image : else ImageProxy ..> Image : return image->GetExtent();</pre>		
Advantages/Strengths: <ul style="list-style-type: none">💡 Non-intrusive design pattern💡 Usually invisible to the caller.		Disadvantages/Weaknesses: <ul style="list-style-type: none">💡 —
Relation to other design patterns: <ul style="list-style-type: none">💡 Adapter: Proxy is focused on managing access, while Adapter is focused on changing an interface💡 Decorator: Decorators can be combined hierarchically, Proxies cannot		
Implementation notes: <ul style="list-style-type: none">💡 Can appear in a class hierarchy, but also in the context of templates (e.g. <code>std::vector<bool></code> or <code>std::bitset<N>::operator[]</code>)		

3. Design Pattern Cheat Sheet

Name: Factory Method		Origin: GoF Year: 1994
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.		
Structure: <pre>classDiagram class Product { } class ConcreteProduct { } class Creator { virtual factoryMethod() = 0 } class ConcreteCreator { virtual factoryMethod() } Product < -- ConcreteProduct Creator < -- ConcreteCreator ConcreteCreator ..> Product : product = factoryMethod(); ConcreteCreator ..> ConcreteProduct : return new ConcreteProduct();</pre>		
Advantages/Strengths: 💡 Logical decoupling of interface and implementation details		Disadvantages/Weaknesses: 💡 Intrusive (because of dependency injection)
Relation to other design patterns: 💡 Strategy: Factory Method is very similar to Strategy, but focused on creating something (possibly introducing a second abstraction)		
Implementation notes: 💡 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

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

3. Design Pattern Cheat Sheet

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

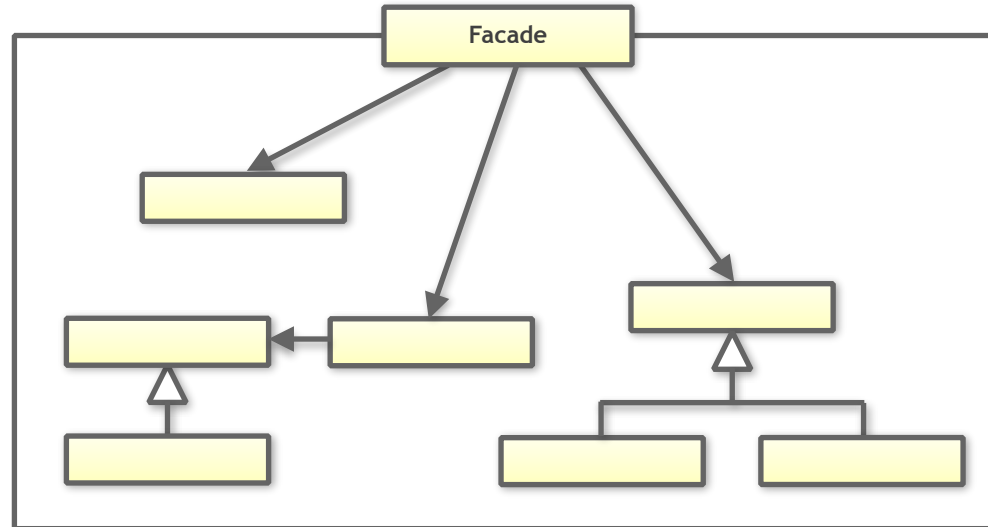
3. Design Pattern Cheat Sheet

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:

-

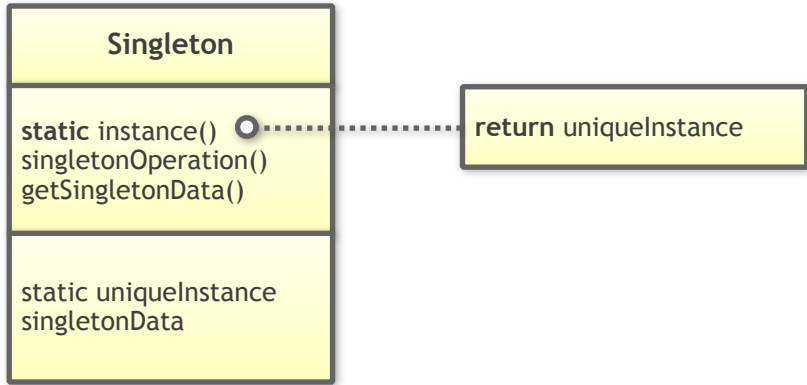
Relation to other design patterns:

-

Implementation notes:

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

3. Design Pattern Cheat Sheet

Name: Singleton		Origin: GoF Year: 1994
Intent: Ensure a class only has one instance, and provide a global point of access to it.		
Structure: 		
Advantages/Strengths: 💡 —		Disadvantages/Weaknesses: 💡 Destroys design/architecture due to lack of dependency management 💡 Provides the characteristics of a global variable/constant
Relation to other design patterns: 💡 —		
Implementation notes: 💡 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

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)