C++ Training

3. Design Pattern Cheat Sheet

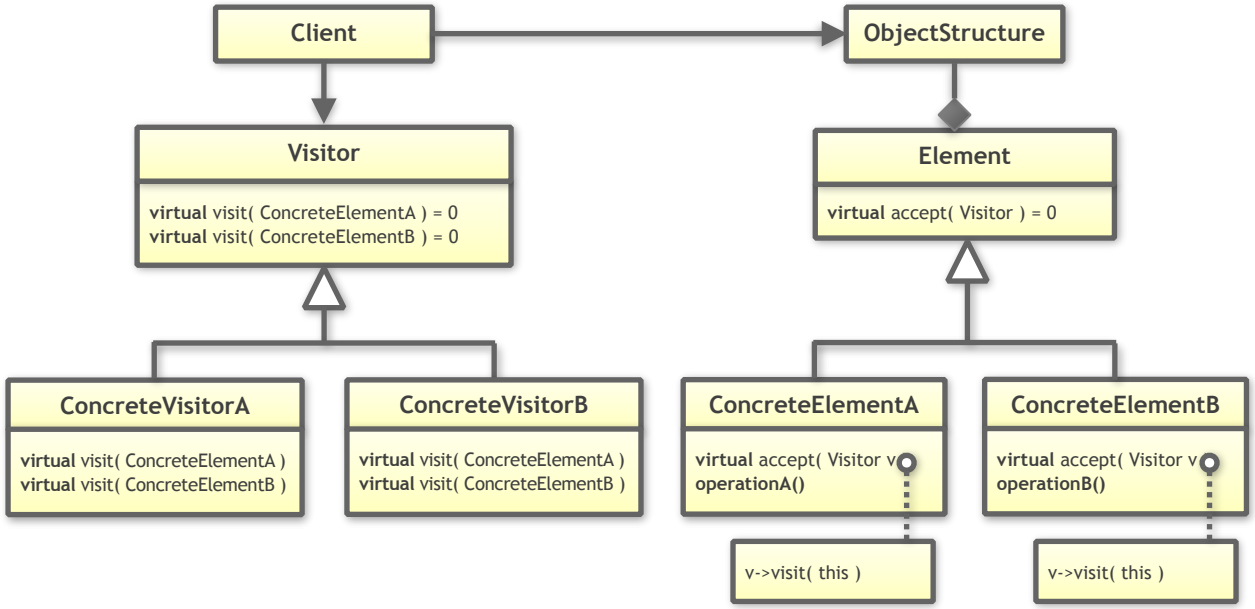
Klaus Iglberger
July, 29th-30th, 2024

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:

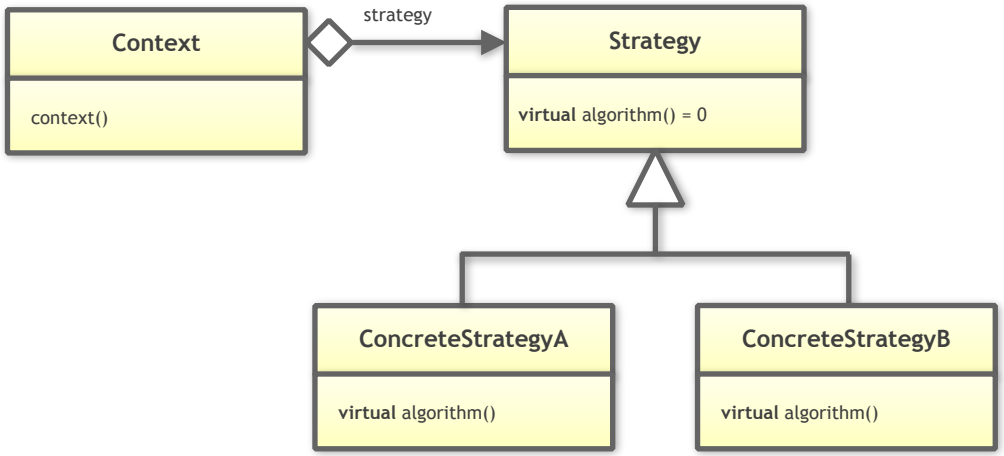
- Often implemented by means of std::variant (C++17).

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:



Advantages/Strengths:

- Logical decoupling of interface and implementation details

Disadvantages/Weaknesses:

- Intrusive (because of dependency injection)
- Proliferation of different abstractions for different strategies

Relation to other design patterns:

- **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:

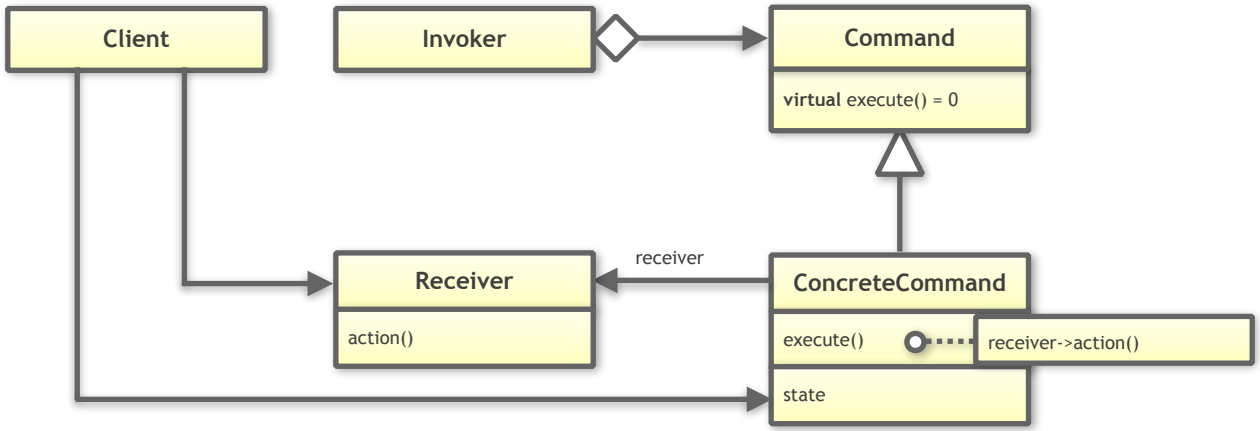
- Can be implemented by means of std::function (C++11), which enables value semantics.

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:



Advantages/Strengths:

- Logical decoupling of interface and implementation details
- Non-intrusive design pattern

Disadvantages/Weaknesses:

- —

Relation to other design patterns:

- **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.

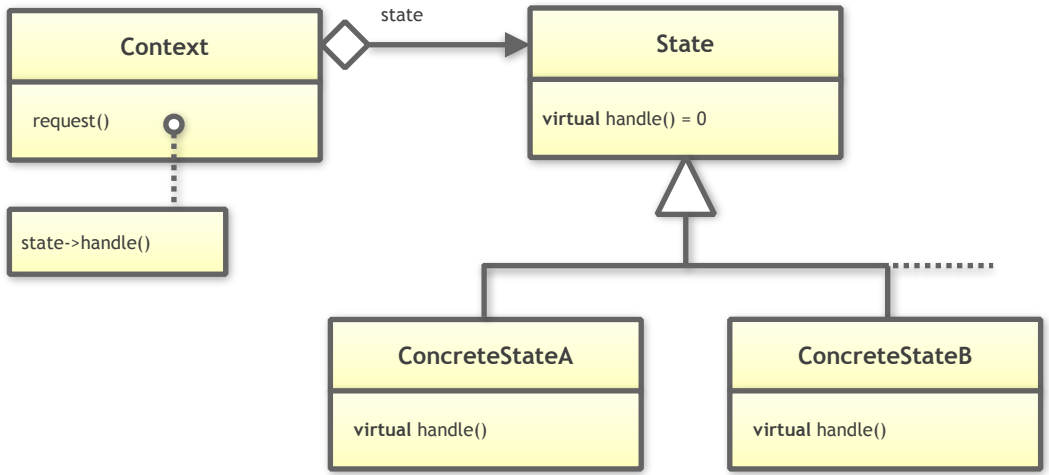
Implementation notes:

- Can be implemented by means of std::function (C++11), which enables value semantics.

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:



Advantages/Strengths:

- Structured representation of state machines

Disadvantages/Weaknesses:

- Strong coupling between states and transitions
- **Intrusive** design pattern (i.e. usually requires changes when new states/transitions are introduced).

Relation to other design patterns:

- **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:

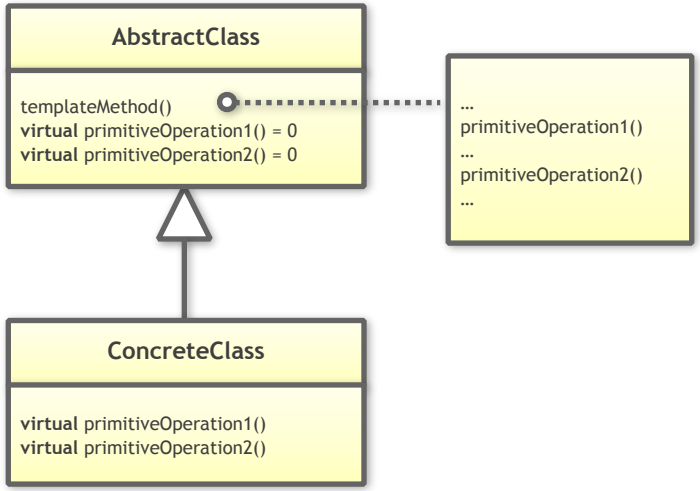
- Can be implemented by means of std::variant (C++17), which enables a similar separation of concerns and value semantics.

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:

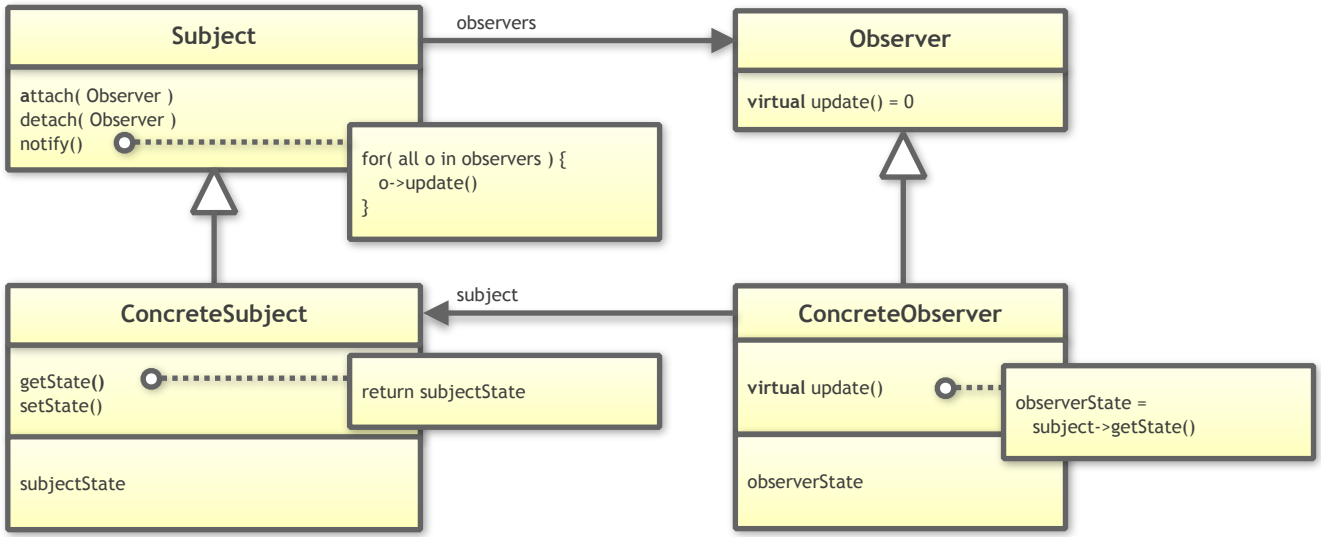
- Template Method is the basis for the Non-Virtual Interface Idiom (NVI).

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:

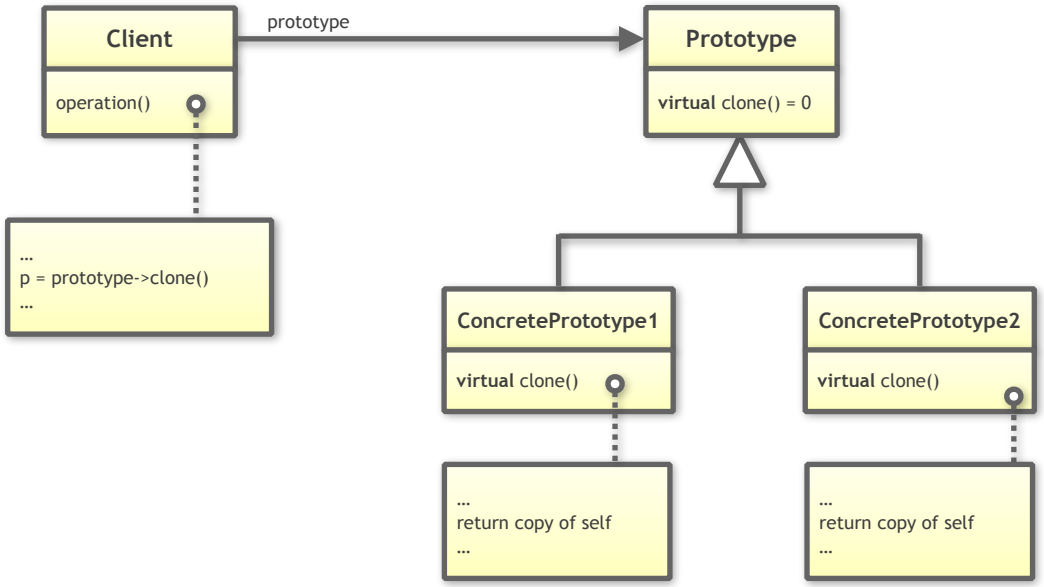
- Can be implemented by means of std::function (C++11), which enables value semantics.
- Can be implemented as push or pull observer

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:

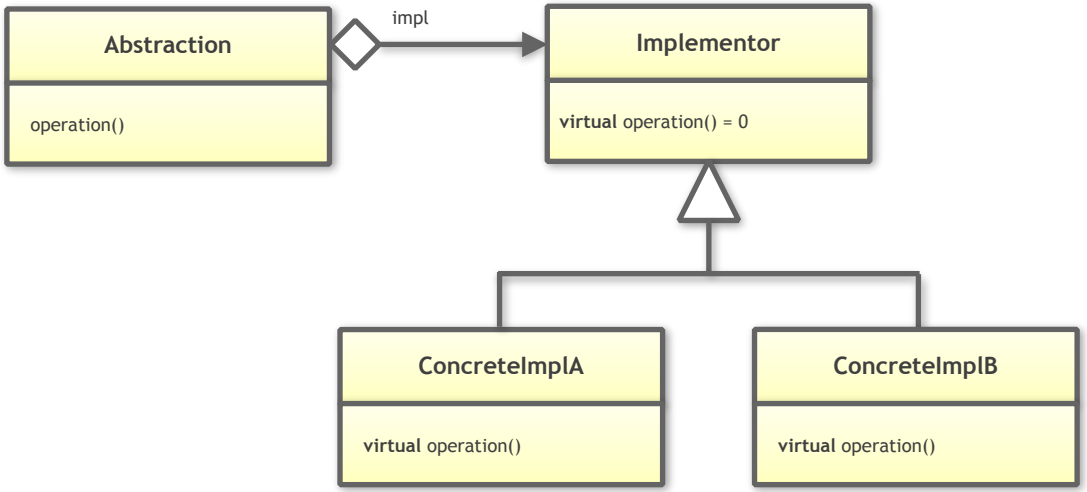
- There is no new (modern) implementation. Therefore the function name `clone()` is very strongly attached to this design pattern.

Name: **Bridge**

Origin: GoF
Year: 1994

Intent: Decouple an abstraction from its implementation so that the two can vary independently.

Structure:



Advantages/Strengths:

- Strong decoupling of physical dependencies

Disadvantages/Weaknesses:

- Introduces a performance penalty

Relation to other design patterns:

- **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:

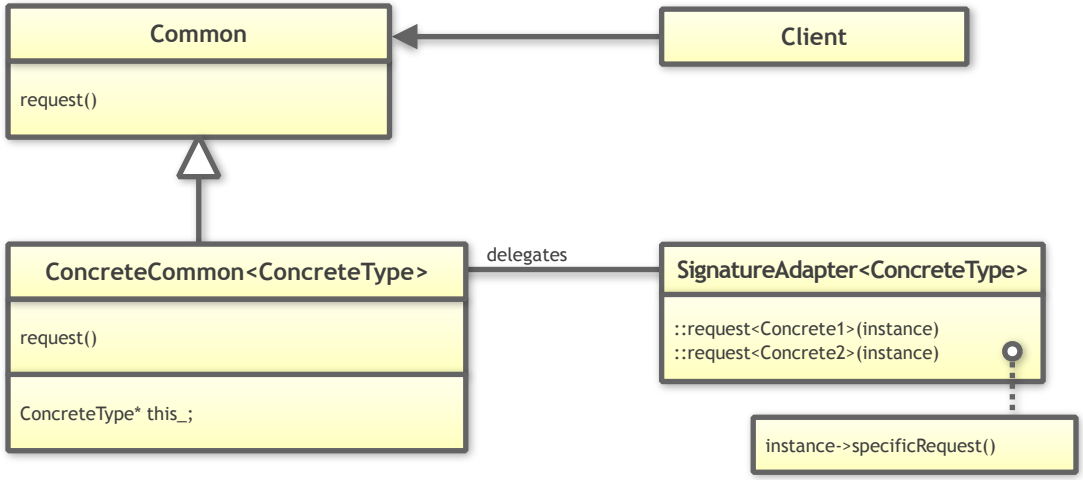
- As an alternative to the dynamic memory, it is possible to implement a Bridge by means of Small-Buffer-Optimization (see Fast Pimpl).
- In case a `std::unique_ptr` is used, the destructor of the `Implementor` class still needs to be defined in the source file.

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:



Advantages/Strengths:

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

Disadvantages/Weaknesses:

- Difficult to introduce new operations

Relation to other design patterns:

- **Adapter**: Adapter is based on an existing inheritance hierarchy, while External Polymorphism introduces a new hierarchy.

Implementation notes:

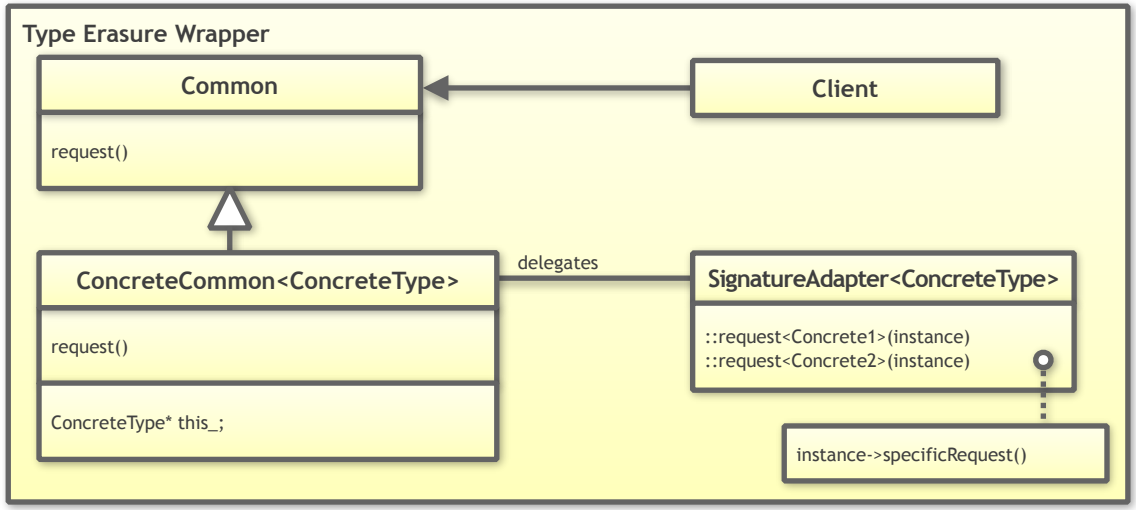
- Possible optimizations: Small Buffer Optimization (SBO), manual virtual function tables, … .

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

Implementation notes:

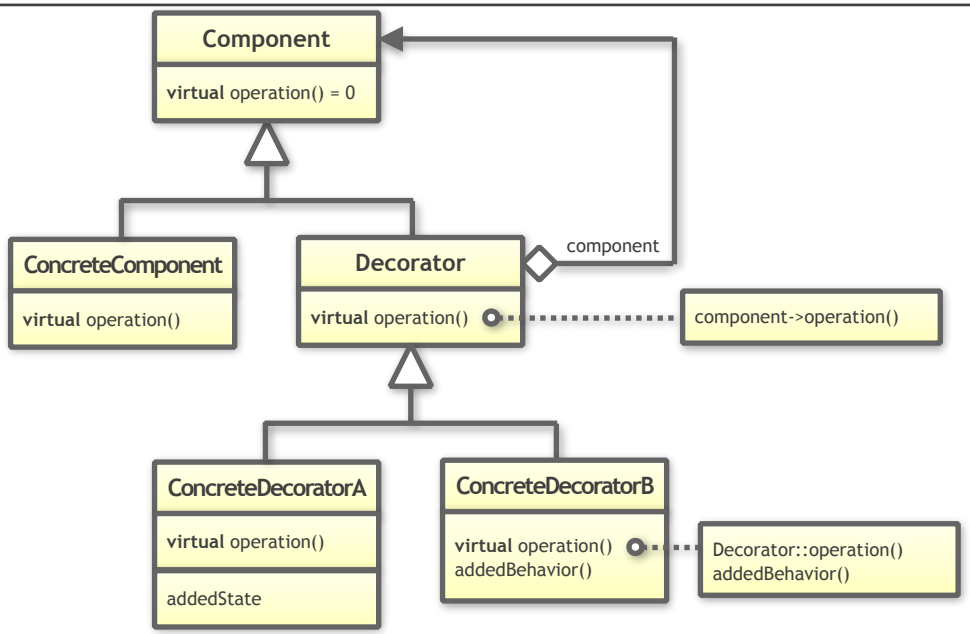
- Possible optimizations: Small Buffer Optimization (SBO), manual virtual function tables, … .
- Type Erasure is a **non-intrusive** design pattern.

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:

- **Non-intrusive** addition of behavior for existing types

Disadvantages/Weaknesses:

- Difficult to introduce new functions

***Relation to other design** patterns:*

- **Strategy**: Strategy is focused on extracting some implementation detail, Decorator is extending the functionality.
- **Adapter**: Adapter changes an interface, Decorator preserves it.

Implementation notes:

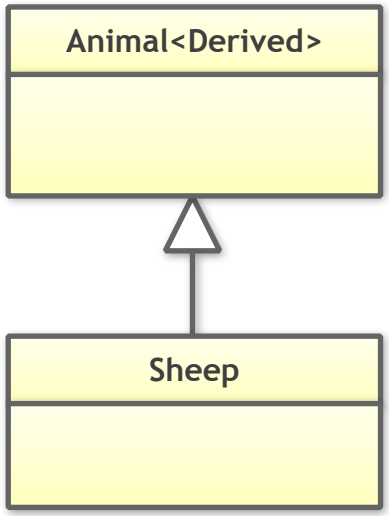
- Can be implemented by means of an inheritance hierarchy, with Type Erasure, or as template.

Name: **CRTP**

Origin: "Curiously Recurring Template Patterns", James Copley
Year: 1995

Intent: Define a compile-time abstraction for a family of related types.

Structure:



Advantages/Strengths:

- Suited for **maximum performance**, no runtime overhead

Disadvantages/Weaknesses:

- Loss of a common base class
- Template-heavy

***Relation to other design** patterns:*

- **Expression Templates**: Combines very well with the intention of Expression Templates.

Implementation notes:

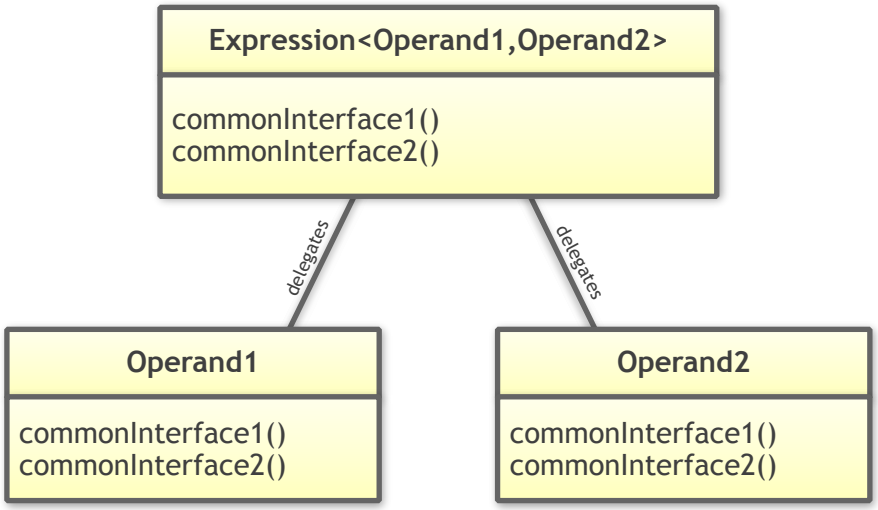
- Can be implemented by means of an inheritance hierarchy, with Type Erasure, or as template.
- CRTP is an **intrusive** design pattern.

Name: **Expression Templates**

Origin: "Expression Templates", Todd Veldhuizen
Year: 1995

Intent: Introduce lazy evaluation for expressions.

Structure:



Advantages/Strengths:

- Suited for **maximum performance**, no runtime overhead

Disadvantages/Weaknesses:

- Template-heavy

***Relation to other design** patterns:*

- **Decorator**: Expressions templates are based on the Decorator design pattern.

Implementation notes:

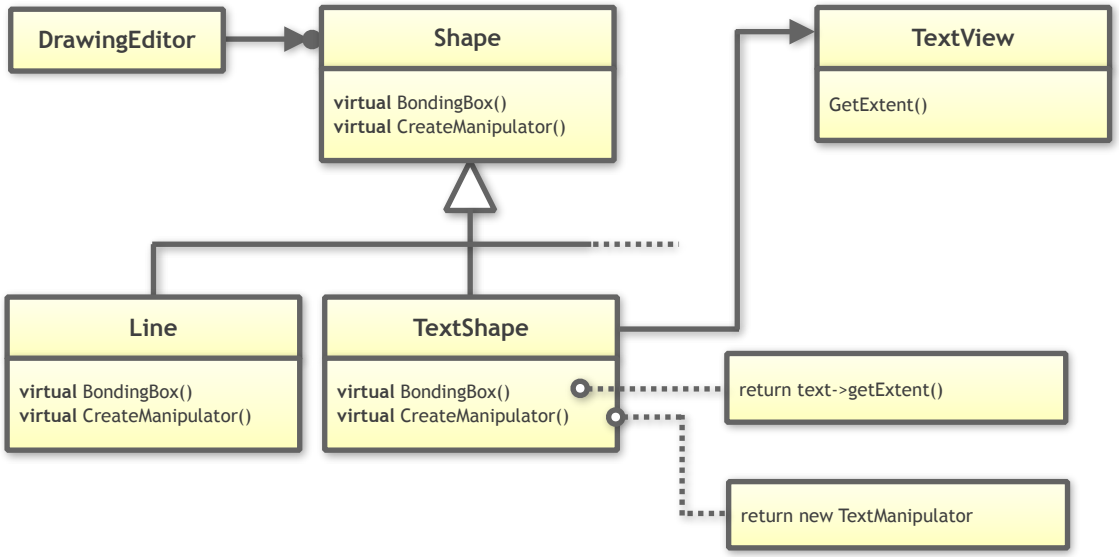
- Expression Templates is a **non-intrusive** design pattern.

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:



Advantages/Strengths:

- **Non-intrusive** design pattern
- Applicable for both classes and functions

Disadvantages/Weaknesses:

- –

Relation to other design patterns:

- **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:

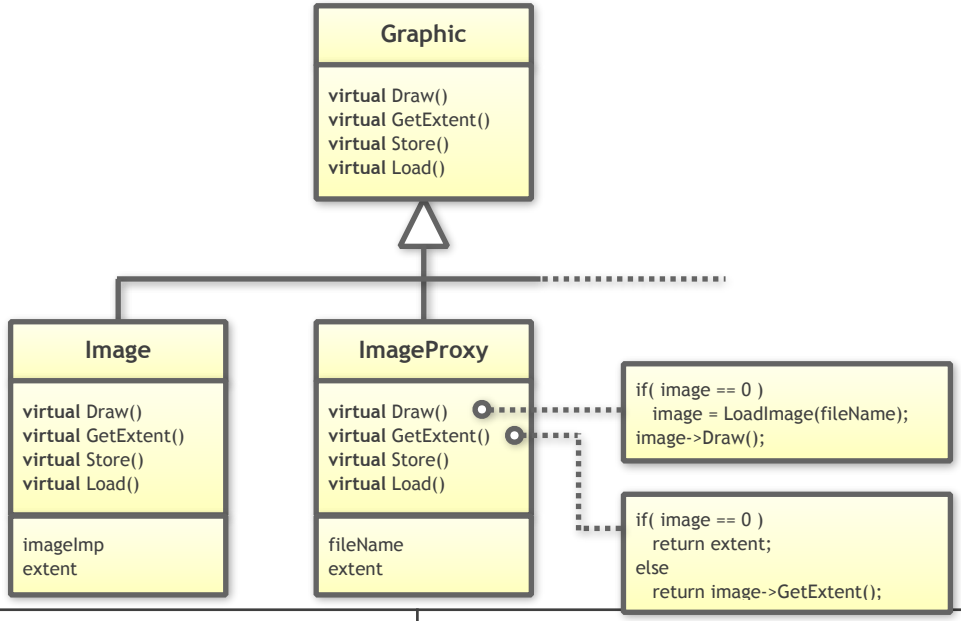
- Next to the classical inheritance-based implementation, adapters can be templates (e.g. `std::stack` and `std::queue`) and simple functions (shims; e.g. `std::begin()`).

Name: **Proxy**

Origin: GoF
Year: 1994

Intent: Provide a surrogate or placeholder for another object to control access to it.

Structure:



Advantages/Strengths:

- **Non-intrusive** design pattern
- Usually invisible to the caller.

Disadvantages/Weaknesses:

- —

***Relation to other design** patterns:*

- **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:

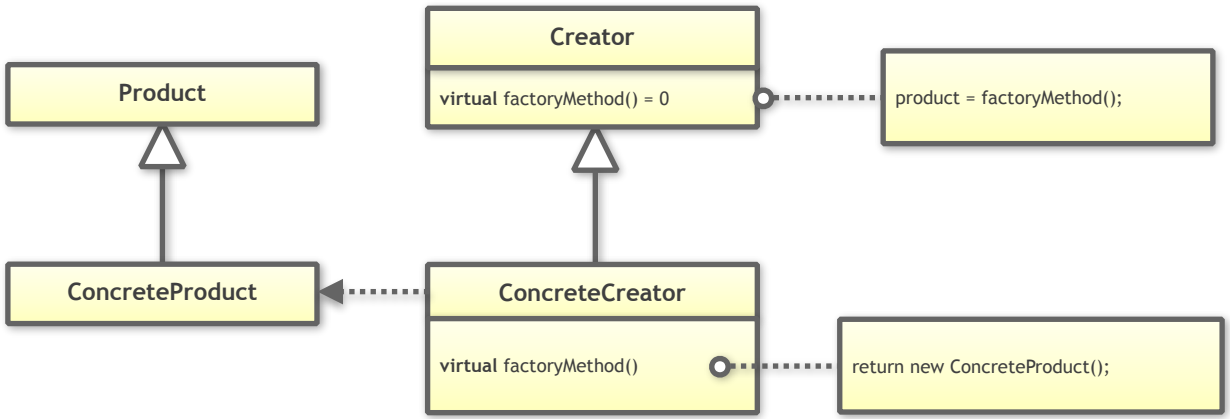
- Can appear in a class hierarchy, but also in the context of templates (e.g. `std::vector<bool>` or `std::bitset<N>::operator[]`)

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:



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)

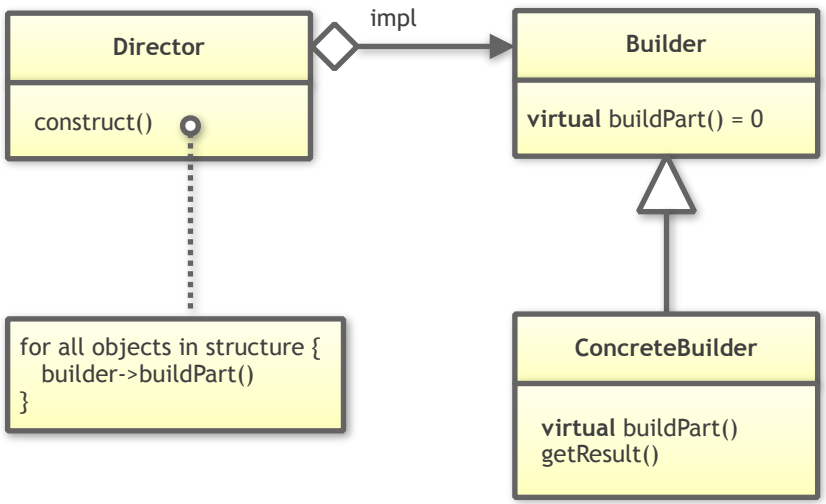
Implementation notes:

- New products should not be returned by raw pointer, but by `std::unique_ptr`, Type Erasure, or `std::variant`.
- A simple function creating something is often called a “factory function”, which has nothing to do with this design pattern.

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:



Advantages/Strengths:

- Logical separation of the steps of a build process

Disadvantages/Weaknesses:

- Usually intrusive (because of dependency injection)

***Relation to other design** patterns:*

- **Factory Method:** Builder is usually composed of several Factory Methods

Implementation notes:

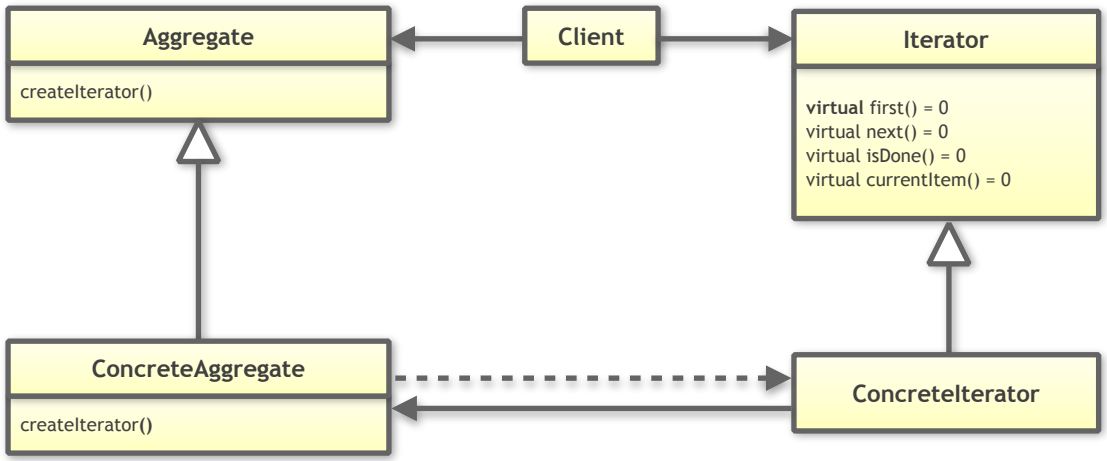
- New products should not be returned by raw pointer, but by `std::unique_ptr`, Type Erasure, or `std::variant`.

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:



Advantages/Strengths:

- Very idiomatic in in C++ due to the STL

Disadvantages/Weaknesses:

- Separation in three steps (increment, compare, access) opens the possibility of access violations

***Relation to other design** patterns:*

- —

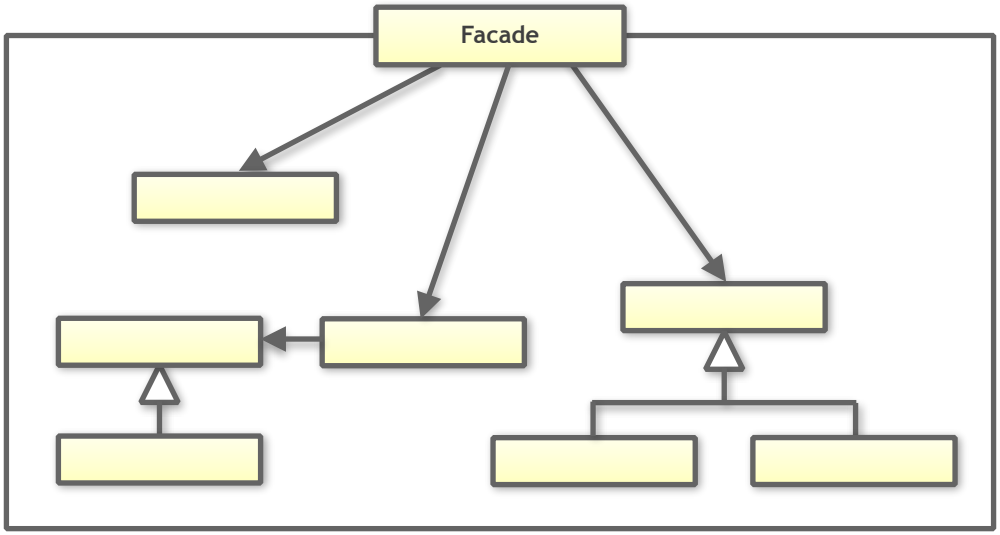
Implementation notes:

- 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.

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:*

- –

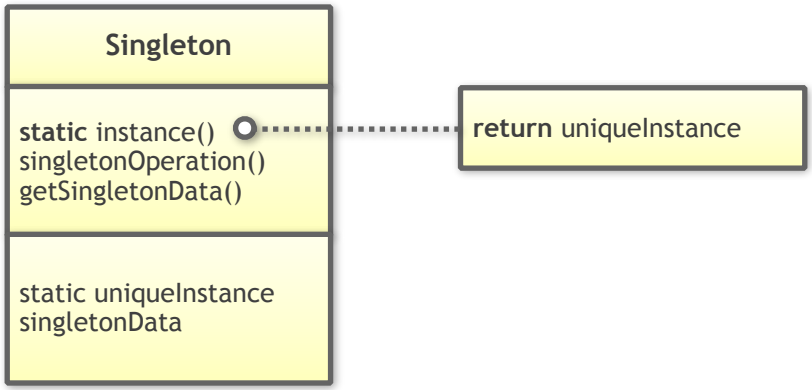
Implementation notes:

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

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

Xing: xing.com/profile/Klaus_Iglberger/cv