```cpp
// TestStrategy.cpp
//
//  Daniel Duffy
//
// Strategy Design Pattern based on C++ Concepts. Not a single virtual function in
// sight. We give examples of both Stateless and Stateful variants.
//
// The difference between classic OOP and Concepts is:
//
// 1. OOP The relationship class-algorithm 1:N e.g. Shape - {draw, compute, etc.}
// 1. Concepts The relationship class-algorithm N:N
//                   e.g. {Shape, Valve} - {draw, compute, etc.}
//
// Is this the end of GOF patterns? We subsume them as special cases of C++
/  Concepts-based Design Patterns.
//
// Advantages:
//
//            1. No clumps and silos of isolated, unstable and semantically incorrect
//            class hierarchies, each with its own abstract methods.
//            2. No messing around with pointers to base classes, casting; less fragile
//               (aka more robust) code.
//            3. Non-intrusive code maintainability.
//            4. Compile-time checking; run-time performance.
//            5. Closer to the _problem_  GOF is a _solution_ to the _problem_.
//            6. Integration with Domain Architectures.
//            7. Multiparadigm!
//            (8. Harold Kasperink and Daniel Duffy have been applying using (not only
//                but also) GOF patterns on a wide range
//                   of applications since 1992 (patterns are from the 1980s).)
//
// // More in ..
//
// Modern Multiparadigm Software Architectures and Design Patterns
// with Examples and Applications in C++, C# and Python Volume I
// Datasim Press 2023, Daniel J.Duffy and Harold Kasperink.
//
// Volume II Ïnteroperability" (
//
//     Call C++ from Python and vice versa
//     Call C# from Python and vice versa
//     Call C# from native C++and vice versa
//     (foundations, methods and applications)
//
// (C) Datasim Education BV 2023
//

#include <concepts>
#include <type_traits>
#include <string>
#include <iostream>
#include <complex>

// Simplest case: universal draw() and compute() protocols
// for all types (no traditional class hierarchies needed)
template<typename Sender> // typically returns void
       concept IDraw = requires (Sender x) { x.draw(); };
```

```
template<typename Algo> // typically returns a value
	concept ICompute = requires (Algo algo, int v) { algo.compute(v); };

// Combine abstract methods into an INTERFACE
template<typename Sender, typename Algo>
	concept IService = IDraw<Sender> && ICompute<Algo>;

template<IDraw Sender> // == requires IDraw<Sender>
	struct Client
{
	Sender s_; // Member

	Client() {}
	Client(Sender s) { s_(s); }

	// Stateful Strategy
	void doit() { s_.draw(); }

	// Stateless Strategy
	template <ICompute Algorithm> // == requires ICompute<Algorithm>
	int doit2(Algorithm alg, int v) { return alg.compute(v); }
};

// Combining two concept abstract method into a combined interface
template<IDraw Sender, ICompute Algorithm>
			requires IService<Sender, Algorithm>
	struct Client2
{
	Sender s_; // Member

	Client2() {}
	Client2(Sender s) { s_(s); }

	// Stateful Strategy
	void doit() { s_.draw(); }

	// Stateless Strategy
	template <ICompute Algorithm> // requires ICompute<Algorithm>
	int doit2(Algorithm alg, int v) { return alg.compute(v); }
};

struct Shape
{
	void draw() { std::cout << "shape draw \n"; }
	int compute(int n) { return n; }
};

struct Valve
{
	void draw() { std::cout << "Valve draw \n"; }
	int compute(int n) { return 3 * n * n; }
};

struct Empty
{
	Empty() {}
};
```

```
// Concrete algorithms
struct Algorithm
{
	int compute(int n) { return n; }
};

struct Algorithm2
{
	int compute(int n) { return n * n; }
};

int main()
{
	// 001
	Algorithm alg1;
	Algorithm2 alg2;

	Client<Shape> client1;
	client1.doit();
	int n = 2;
	std::cout << "compute shape " << client1.doit2(alg1, n) << '\n';

	Client<Valve> client2;
	client2.doit();
	int m = 2;
	std::cout << "compute valve " << client2.doit2(alg2, m) << '\n';

	// Constraints not met
	//Client<Empty> client3;
	//client3.doit();
	//std::cout << "compute empty " << client3.doit2(alg1, n) << '\n';

}
```