DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

[CS 225] Advanced C/C++

Lecture 6: Dynamic dispatch

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Message dispatch
  - Static dispatch
  - Single dispatch
  - Double dispatch
  - Multiple dispatch
- Design patterns
  - Visitor design pattern
  - Template method design pattern
- Virtualizing non-member functions
- Be careful with virtual function calls!

Message dispatch

In a traditional OOP perspective, calling a member function `obj.DoSomething(args)` can be thought of as sending a *message* `DoSomething` to an *object* `obj`, and including some parameters `args`.

To ***dispatch*** means to establish where a message should be sent; in other words to select an object on which the right member function will be called, based on all parameters (including the object itself).

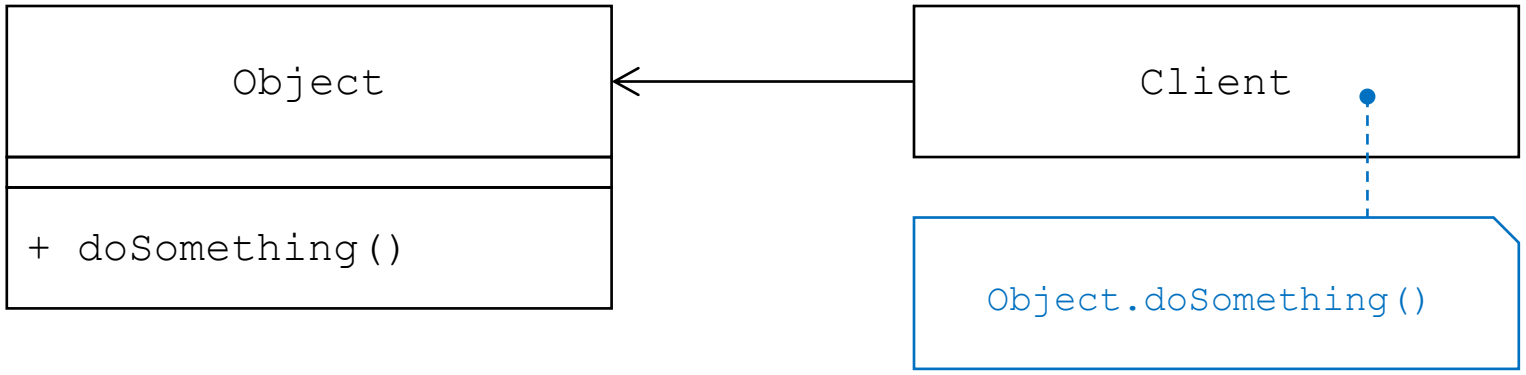
Message dispatch

Static dispatch means calling a member function that is not polymorphic or that is statically polymorphic.

Member functions for static dispatch are selected in the compile-time for given objects.

Message dispatch

Static dispatch



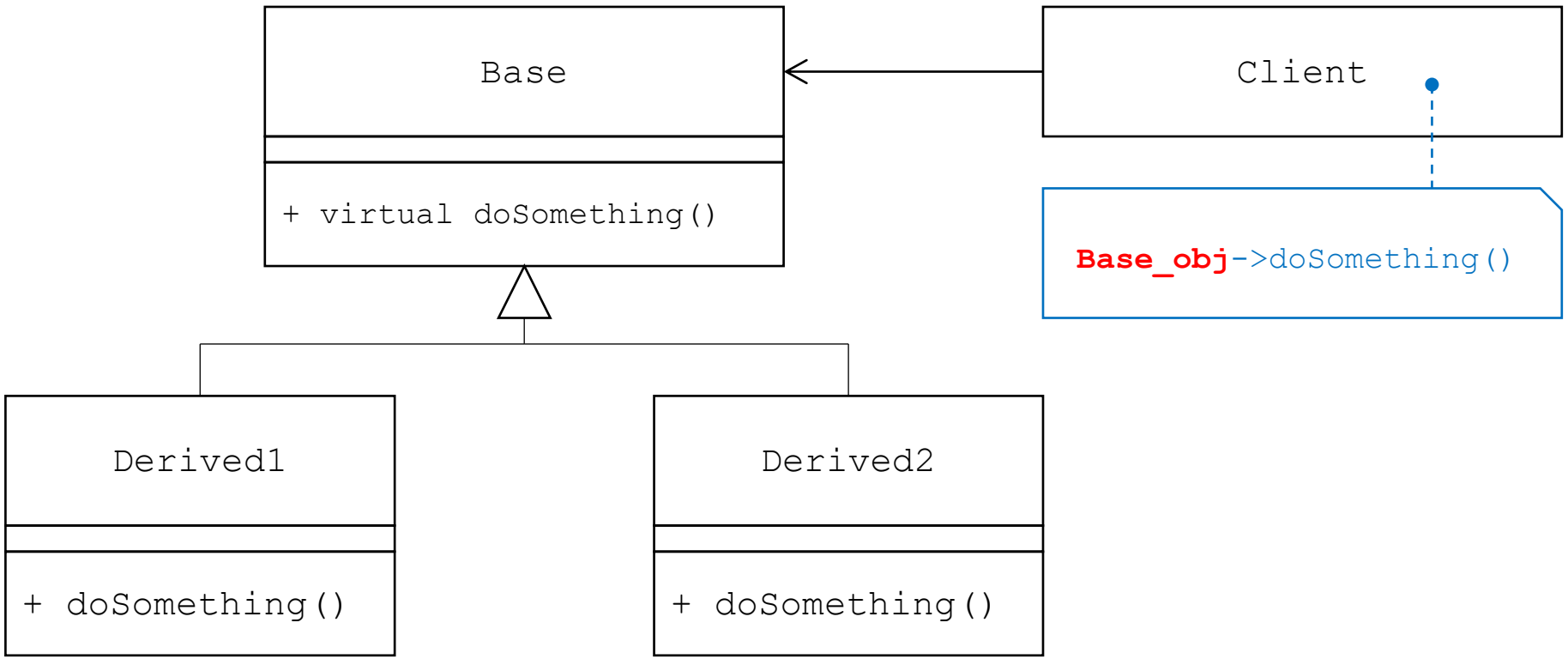
Message dispatch

Single dispatch means calling a member function that is dynamically polymorphic in regard to `this`.

Single dispatch functionality is supported in C++ as *virtual* member functions; `this` through its *vptr* leads to a *vtable* with functions for to this type of objects.

Message dispatch

Single dispatch



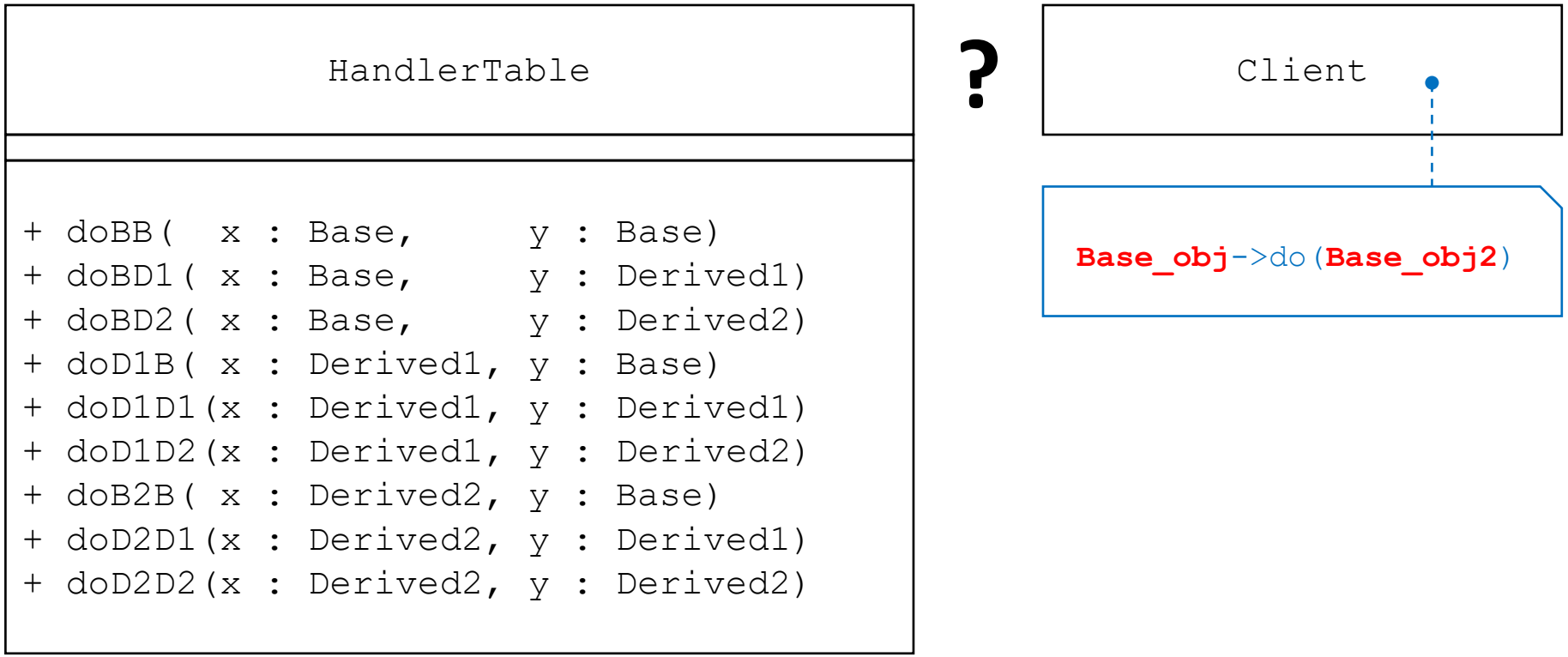
Message dispatch

Double dispatch means calling a member function that is dynamically polymorphic in regard to `this` **and** a call parameter (often of the same base type).

Double dispatch functionality is not supported in C++ directly, but there are a few ways to organize it.

Message dispatch

Double dispatch



Message dispatch

Possible implementations of **double dispatch** in C++:

- Include a ladder of *if*s inside `virtual void do()`

```
void Base::do(Base& other) {
  const type_info& otherType = typeid(other);
  if (otherType == typeid(Base))
    handlerTable->doBB(this, other);
  else if (otherType == typeid(Derived1))
    handlerTable->doBD1(this, other);
  else if (otherType == typeid(Derived2))
    handlerTable->doBD2(this, other);
  else
    throw "Not supported!";
}
// Override in derived classes!
```

This implementation has a big design issue: each class needs to know about every other class.

Message dispatch

Possible implementations of **double dispatch** in C++:

- Include two `virtual` calls

```
struct Base : ICommon {
  virtual void do(ICommon& other);
  virtual void do(Base& other);
  virtual void do(Derived1& other);
  virtual void do(Derived2& other);
};
// Call to this function uses dynamic polymorphism for both parameters.
void Base::do(ICommon& other) {
  // Call to this function uses dynamic polymorphism for other,
  // but static polymorphism for *this (its type is Base)
  other.do(*this);
}
void Base::do(Base& other) {
  handlerTable->doBB(*this, other);
}
// Override in derived classes!
```

This implementation has the same design issue: each class needs to know about every other class.

Message dispatch

Possible implementations of **double dispatch** in C++:

- Use a map to emulate own virtual method table:

```
// Populate a map with function pointers!

// Calculate the compound key based on both objects:
std::string key = Base_obj->key() + " " + Base_obj2->key();

// Call a function based on the key:
MyTable[key](Base_obj, Base_obj2);
```

This implementation has a different issue: each handler must have the same type, and thus accept both parameters by their base class pointer. Individual handlers must inside perform dynamic cast before running the rest of their code.

Message dispatch

Possible implementations of **double dispatch** in C++:

- Use a two-dimensional array to emulate a virtual method table.
- Use the visitor design pattern.

Message dispatch

Multiple dispatch means calling a member function that is dynamically polymorphic in regard to `this` **and** any number of parameters.

Multiple dispatch functionality is not supported in C++ directly (considered for future implementations), it can be implemented similarly to double dispatch.

Double dispatch is a special case of multiple dispatch.

Design patterns

A design pattern is a general, language-independent, reusable solution to a common problem in software design. It is a universally recognized best practice template for organizing class and object interactions.

Visitor design pattern

Pattern name

Visitor

Problem

Offer double dispatch for an object and a parameter.

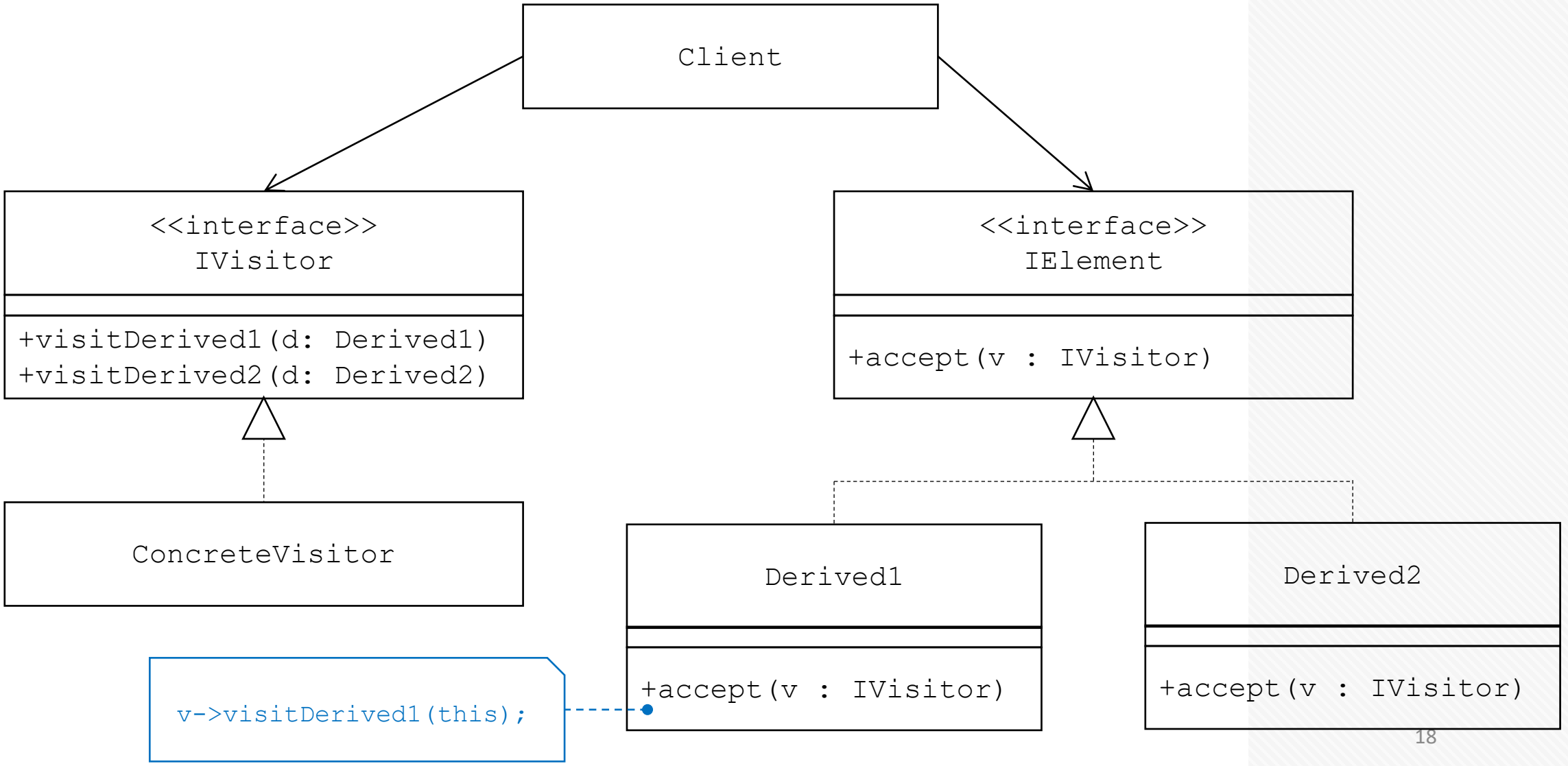
Use a parameter to represent an operation to be performed on the object, even if the operation is not a member function of a class it operates on.

Visitor design pattern

Solution

- Define interface for a desired operation is a separate class (referred here to as a *visitor*) and overload it for different concrete class parameters.
- Implement the operations in concrete classes derived from a visitor.
- Add a member function to the interface of element classes to accept a *visitor.*
- Implement a member function accepting a visitor in concrete classes derived from an element.

Visitor design pattern



Visitor design pattern

Consequences

- Adding operations to classes is possible without modifying classes.
- Related behavior of multiple types is gathered in one class.
- Adding more concrete element classes is difficult as each visitor class must be updated.
- Visiting objects of unrelated classes is possible as long as these classes can accept a visitor.

Template method design pattern

Pattern name

Template method
Non-virtual interface idiom

Problem

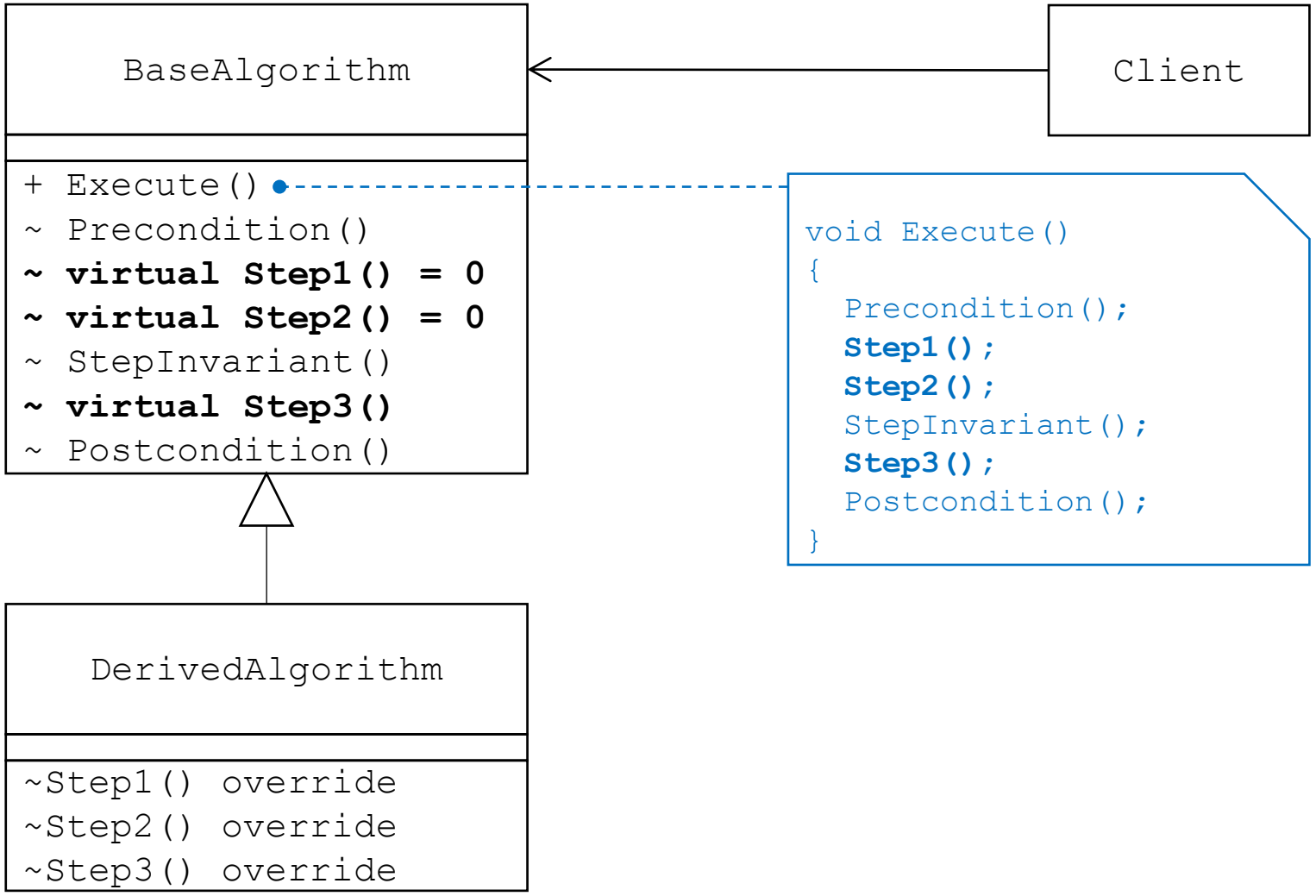
Develop an algorithm with a member function that executes it; derived classes can override individual steps of the algorithm, but not their ordering.

Template method design pattern

Solution

- Define a class for the algorithm that exposes a member function (*method*) that calls member functions of individual steps in a proper sequence; this method should not be `virtual`.
- Define individual steps are member functions; if they can be overridden, make them `virtual`. If they can only be invoked as a part of the algorithm, adjust their access modifiers (`protected` or `private`).

Template method design pattern



Template method design pattern

Consequences

- Separates member functions that are:
  - Not virtual and cannot be overridden (`StepInvariant()`),
  - Pure virtual and must be overridden (`Step1()`),
  - Virtual and may be overridden (`Step3()`).
- A derived class that overrides a function can still call a function's original implementation using a scope to extend it.
- A base class can use virtual functions implemented in a derived class as extension *hooks*.

Template method design pattern

C++ implementation insights

- Access modifiers
  - If an operation step can be invoked independently from the algorithm, make it `public`.
  - If an operation step can be invoked independently only by derived classes, make it `protected`.
  - If an operation step can only be executed as a part of the algorithm, make it `private`.

Template method design pattern

C++ implementation insights

- Access modifiers
  - If an operation step can be invoked independently from the algorithm, make it `public`.
  - If an operation step can be invoked independently only by derived classes, make it `protected`.
  - If an operation step can only be executed as a part of the algorithm, make it `private`.
  - As `private` member functions exist in an inherited virtual table, they can still be overridden (even if not called) by a derived class.

Template method design pattern

C++ implementation insights

- Pure virtual member functions
  - Pure virtual functions must be overridden.
  - A class that has at least one pure virtual function is abstract, and it cannot be instantiated.
  - Steps that must be overridden can optionally provide a default implementation that must be used explicitly.

Template method design pattern

C++ implementation insights

- Pure virtual member functions
  - Pure virtual functions must be overridden,
  - A class that has at least one pure virtual function is abstract, and it cannot be instantiated,
  - Steps that must be overridden can optionally provide a default implementation that must be used explicitly.
  - In C++, pure virtual member functions can have definitions, but they must be provided outside of a class' definition.

Virtualizing non-member functions

Support for dynamic dispatch on non-member functions could allow us to differentiate behavior based on call parameters, i.e. in overloaded global operators.

Non-member functions cannot be `virtual`; they use static dispatch.

To *virtualize* a non-member function, use it to delegate a call to a `virtual` member function in a class hierarchy.

Virtualizing non-member functions

```
#include <iostream>

struct PrinterBase {
  virtual void print(std::ostream& stream) const {
    stream << "Hello world from PrinterBase!";
  }
};
struct Printer : public PrinterBase {
  void print(std::ostream& stream) const override {
    stream << "Hello world from Printer!";
  }
};

std::ostream& operator<<(std::ostream& stream, const PrinterBase& printer) {
  printer.print(stream);
  return stream;
}

int main() {
  PrinterBase printerBase;
  Printer printer;

  std::cout << printerBase << std::endl;
  std::cout << printer << std::endl;
}
```

Be careful with virtual function calls!

Do not invoke any `virtual` functions in constructors or a destructor.

If the class is a base class and a `virtual` function has been overridden in a derived class, a derived class will not be able to handle a dynamic dispatch because:

- *(Constructors)* It has not been initialized yet.
- *(Destructors)* It has already been destroyed.