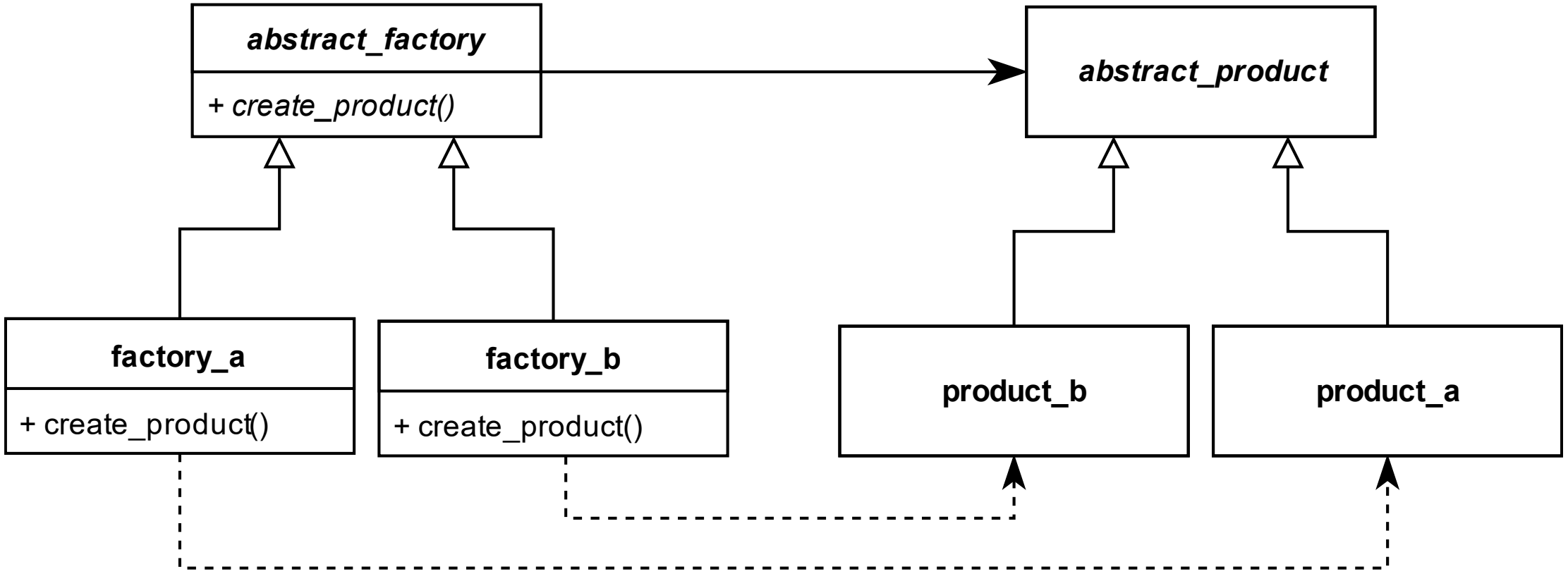
Design Patterns in C++ Done Right

Miro Knejp

MUC++ 2018-03-29

Abstract Factory



```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

struct abstract_factory
{
  virtual auto create_product() -> std::unique_ptr<abstract_product> = 0;
};

struct factory_a : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};

struct factory_b : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

struct abstract_factory
{
  virtual auto create_product() -> std::unique_ptr<abstract_product> = 0;
};

struct factory_a : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};

struct factory_b : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};

std::unique_ptr<abstract_factory> product_factory;
```

```
void install_factory_a()
{
  product_factory = std::make_unique<factory_a>();
}

void install_factory_b()
{
  product_factory = std::make_unique<factory_b>();
}

install_factory_a();

auto product = product_factory->create_product();
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

struct abstract_factory
{
  virtual auto create_product() -> std::unique_ptr<abstract_product> = 0;
};

struct factory_a : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};

struct factory_b : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};

std::unique_ptr<abstract_factory> product_factory;
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

struct abstract_factory
{
  virtual auto create_product() -> std::unique_ptr<abstract_product> = 0;
  virtual ~abstract_factory() = 0;
protected:
  abstract_factory(abstract_factory const&) = default;
  abstract_factory(abstract_factory&&) = default;
  ...
};

struct factory_a : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};

struct factory_b : abstract_factory
{
  auto create_product() -> std::unique_ptr<abstract_product> override;
};
```

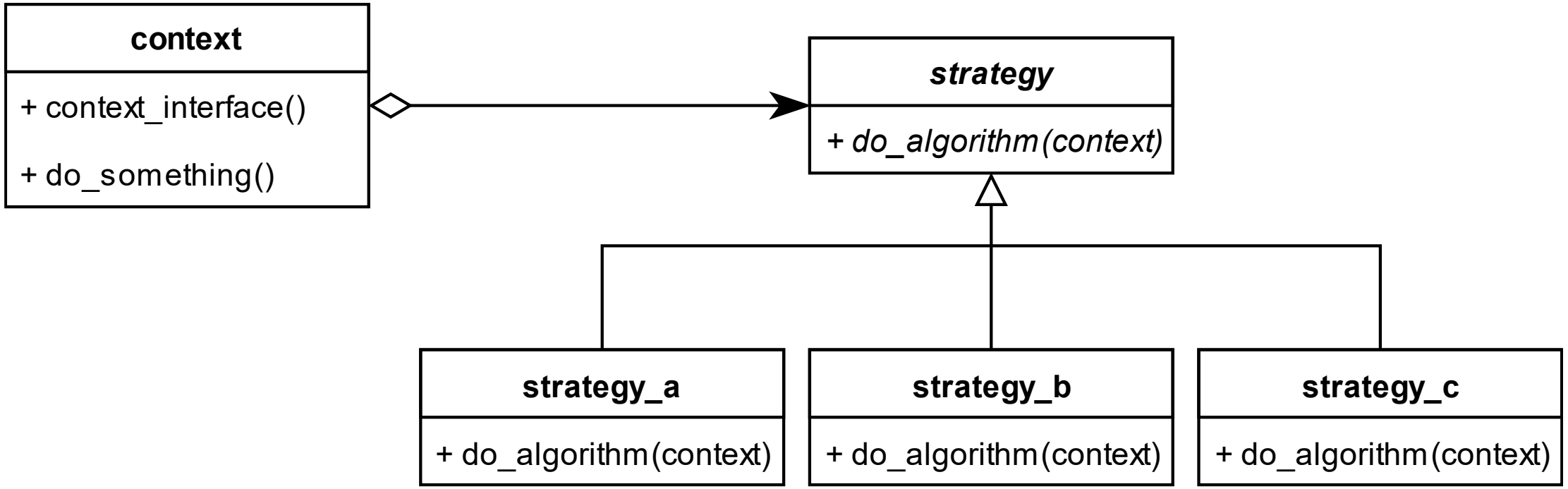
```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

auto (*create_product)() -> std::unique_ptr<abstract_product>;

auto create_product_a() -> std::unique_ptr<abstract_product>;
auto create_product_b() -> std::unique_ptr<abstract_product>;

void install_factory_a()
{
  create_product = &create_product_a;
}

void install_factory_b()
{
  create_product = &create_product_b;
}

install_factory_a();
auto product = create_product();
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

auto (*create_product)() -> std::unique_ptr<abstract_product>;

void install_factory_a()
{
  create_product = [] () { return ...; };
}

void install_factory_b()
{
  create_product = [] () { return ...; };
}

install_factory_a();
auto product = create_product();
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

auto (*create_product)() -> std::unique_ptr<abstract_product>;

void install_factory_a(int x)
{
  create_product = ???
}

void install_factory_b(float y)
{
  create_product = ???
}

install_factory_a(20180329);
auto product = create_product();
```

```
struct abstract_product { ... };
struct product_a : abstract_product { ... };
struct product_b : abstract_product { ... };

std::function<auto() -> std::unique_ptr<abstract_product>> create_product;

void install_factory_a(int x)
{
  create_product = [x] () { return ...; };
}

void install_factory_b(float y)
{
  create_product = [y] () { return ...; };
}

install_factory_a(20180329);
auto product = create_product();
```

Strategy


context
+ context_interface()
+ do_something()
strategy
+ do_algorithm(context)
strategy_a
+ do_algorithm(context)
strategy_b
+ do_algorithm(context)
strategy_c
+ do_algorithm(context)

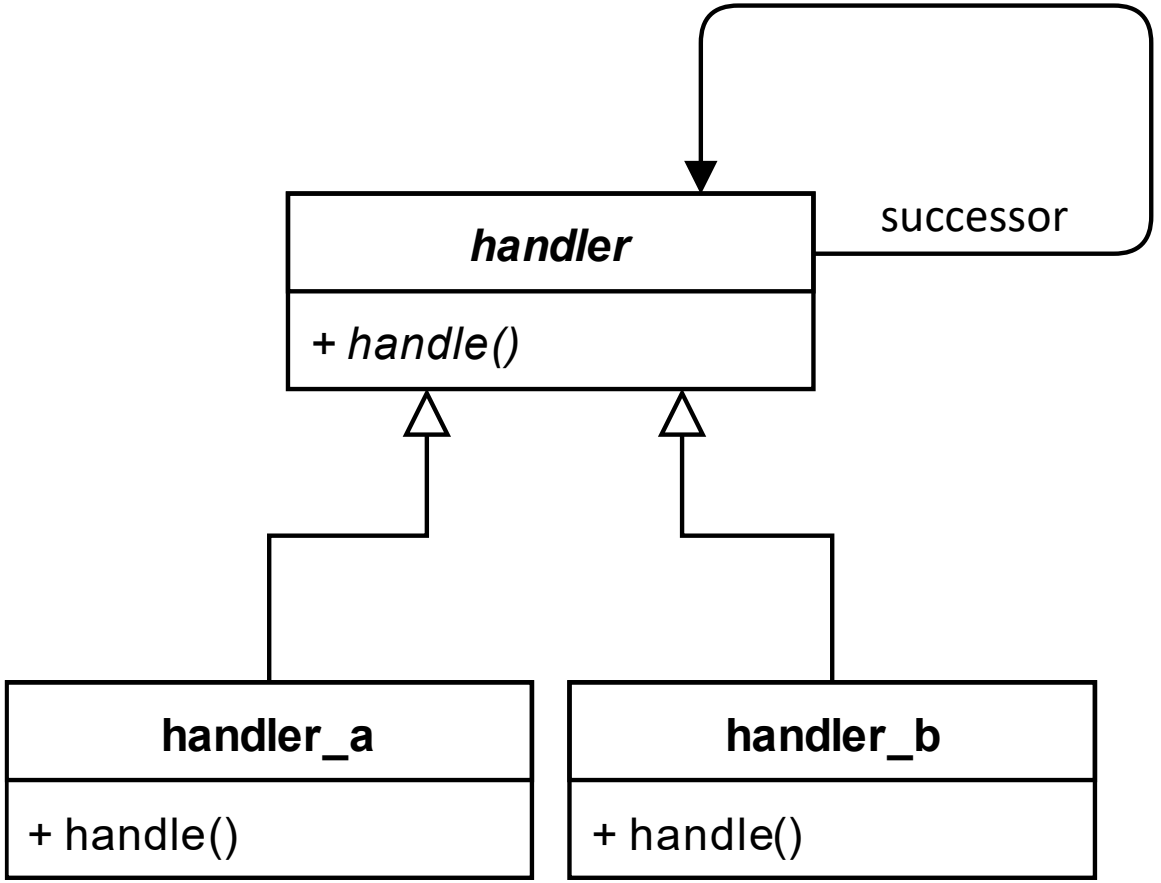
```
struct strategy
{
  virtual auto do_algorithm(context& ctx, int a, float b) -> int = 0;
};

struct context
{
  context(std::unique_ptr<strategy> strategy, ...);
  auto do_something(int a, float b)
  {
    // stuff
    _strategy->do_algorithm(*this, a, b);
    // more stuff
  }
  // context_interface()
private:
  std::unique_ptr<strategy> _strategy;
  ...
};

auto the_algorithm = context(std::make_unique<strategy_b>(...), ...);
```

```
struct context
{
  context(std::function<auto(context&, int, float) -> int> strategy, ...);
  auto do_something(int a, float b) -> int
  {
    // stuff
    _strategy(*this, a, b);
    // more stuff
  }
  // context_interface()

private:
  std::function<auto(context&, int, float) -> int> _strategy;
  ...
};

auto the_algorithm = context([...] (int a, float b) { ... }, ...);
```

```
struct context
{
  context(...);
  // context_interface()
private:
  ...
};

auto do_something = [ctx = context(...), strategy = ...] (int a, float b) mutable
{
  // stuff
  strategy(ctx, a, b);
  // more stuff
};
```

```
struct context
{
  context(...);
  // context_interface()
private:
  ...
};

std::function<auto(int, float) -> int> do_something =
  [ctx = context(...), strategy = ...] (int a, float b) mutable
  {
    // stuff
    strategy(ctx, a, b);
    // more stuff
  };
```

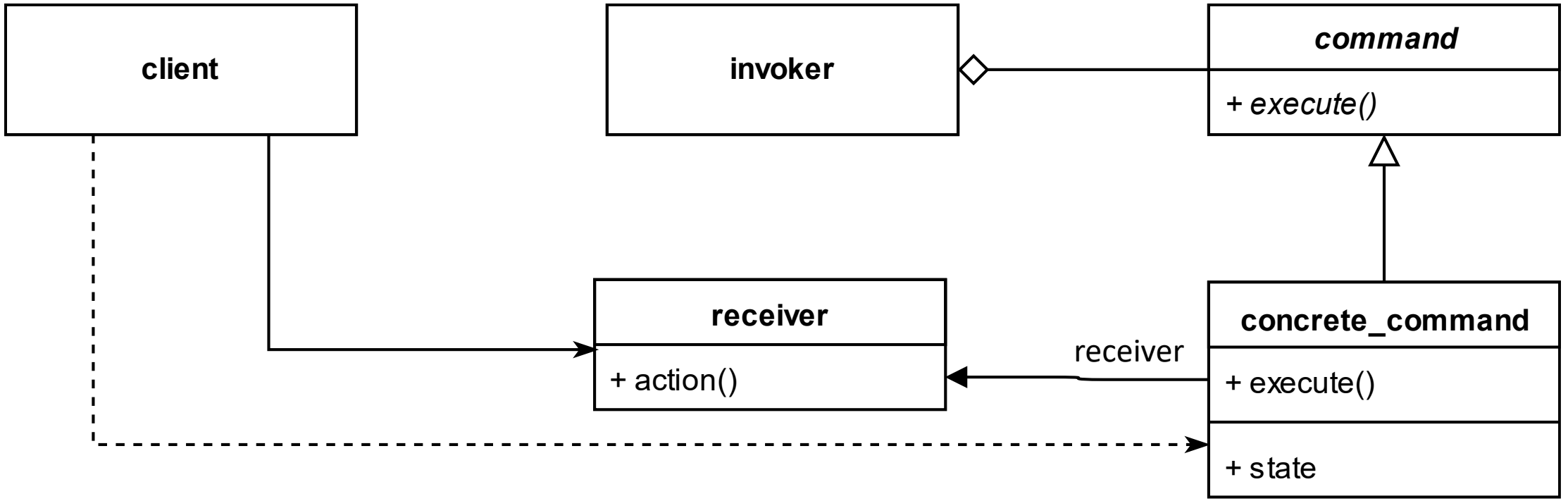
Chain of Responsibility



```
struct handler
{
  handler(std::function<auto() -> void> successor, ...);

  void handle()
  {
    // do my stuff
    if (_successor)
      _successor();
  }

private:
  std::function<auto() -> void> _successor;
  ...
};
```

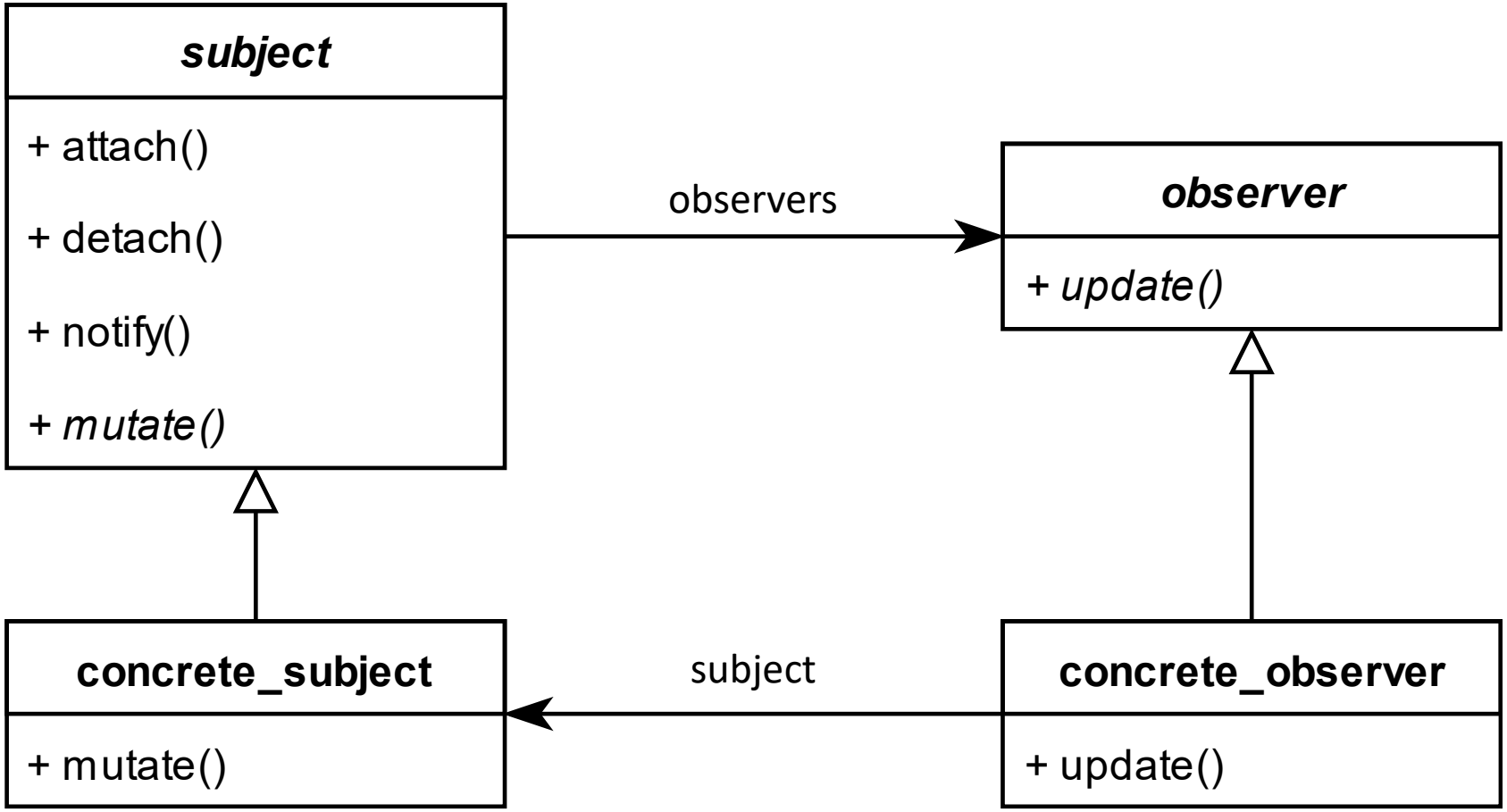
```
auto handlers = std::vector<std::function<auto() -> void>>();

handlers.push_back([] { ... });
handlers.push_back(handler1);
handlers.push_back(f);

for (auto& handler : handlers)
  handler();
```

Command


client
invoker
command
+ execute()
receiver
+ action()
receiver
concrete_command
+ execute()
+ state

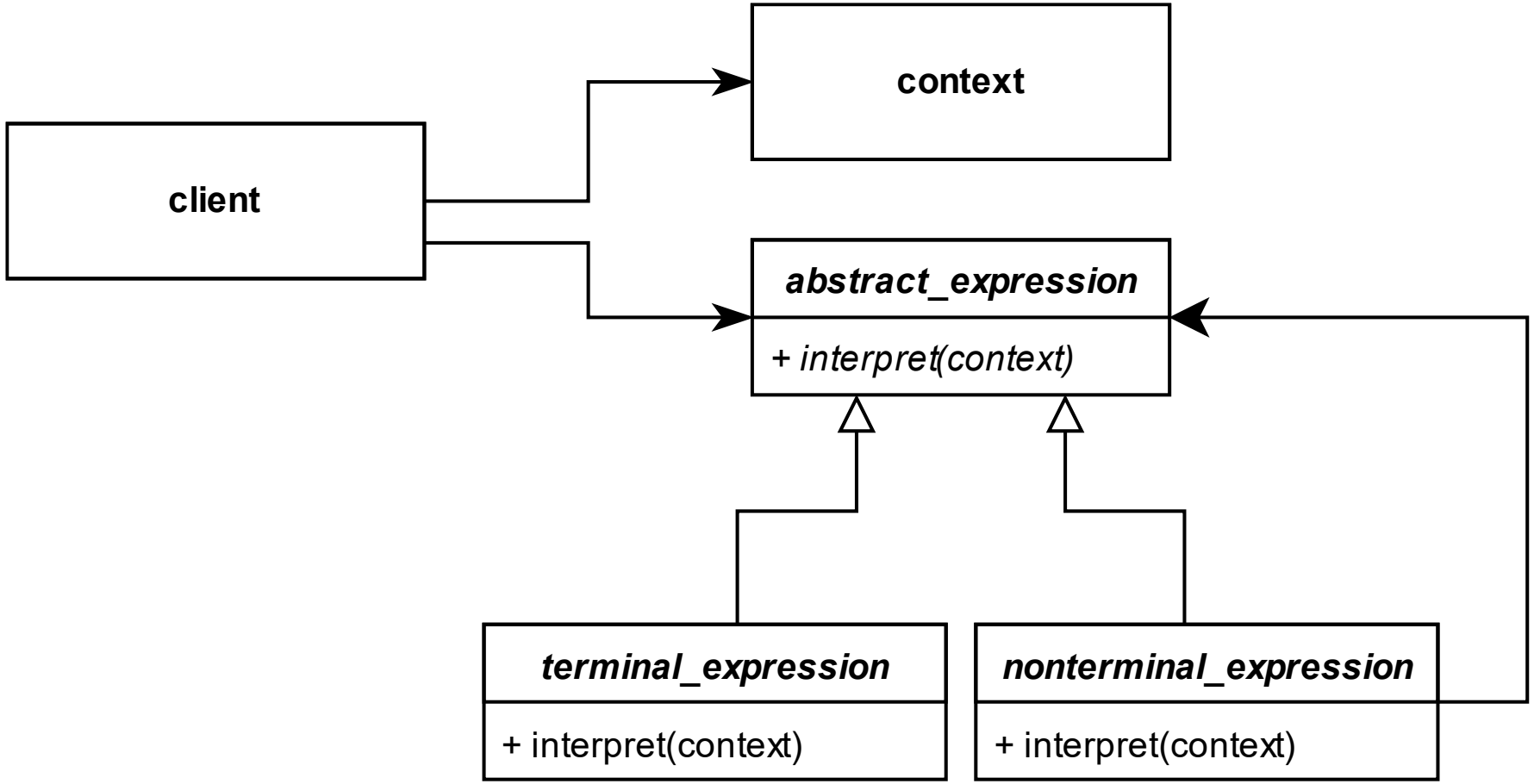
```
std::function<auto() -> void>
              +
          [] () {}
```

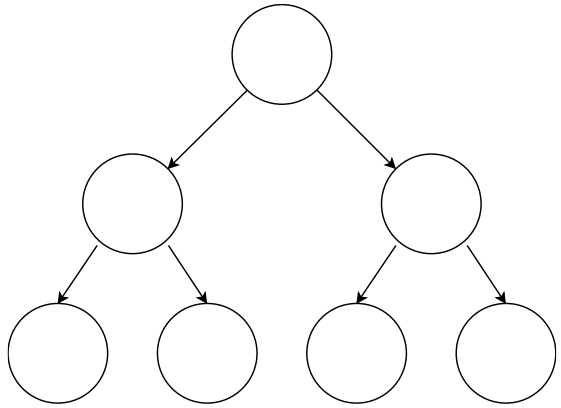
Observer


subject
+ attach()
+ detach()
+ notify()
+ mutate()
observers
observer
+ update()
concrete_subject
+ mutate()
subject
concrete_observer
+ update()

```
std::vector<std::function<auto() -> void>>
                    +
                [] () {}
```

Interpreter





```
std::function<auto(context&) -> void>
                +
            [] () {}
```

`std::function` is the most versatile design pattern.

`std::function` is the most versatile design pattern.

```
struct factory
{
  factory(std::function<auto() -> product> f) : _f(std::move(f)) { }
  auto operator()() const { return _f(); }
private:
  std::function<auto() -> product> _f;
};
```

`std::function` is the most versatile design pattern.

@mknejp 

https://cpplang.now.sh