Effective GoF Patterns

with C++11 and Boost

Tobias Darm
me

• no books
• no blogs
• no tweets
• no open source
me

- **Dräger** Lübeck
- C++ developer
- Subsystemdesigner
- Trainings/Workshops
Motivation
My Approach
Unfair
Content

• Implementation classic vs. C++11
• Implementation classic vs. Boost
• Application classic vs. C++11/Boost
• Conclusion concerning patterns
CoffeeMachine (classic) Application
CoffeeMachine (C++11/Boost) Application
GoF-Patterns

- Common solution to a reoccurring problem
- GoF-Patterns = Micropatterns
- How many?
GoF-Patterns

• Creational
  – Abstract Factory
  – Builder
  – Factory Method
  – Prototype
  – (Singleton)

• Structural
  – Adapter
  – Bridge
  – Composite
  – Decorator
  – Facade
  – Flyweight
  – Proxy

• Behavioural
  – Chain
  – Command
  – Interpreter
  – Iterator
  – Mediator
  – Memento
  – Observer
  – State
  – Strategy
  – Template
  – Visitor
GoF-Patterns

• Creational
  – Abstract Factory

• Structural
  – Flyweight

• Behavioural
  – Chain
  – Command
  – Observer
  – Strategy
  – State
Strategy

• Capsules a family of algorithms and makes them exchangeable
Strategy
Example: CaffeineBeverage

- 1
- 2
- 3

Effective GoF Patterns - Tobias Darm
Strategy (classic)
Recipe Interface

class Recipe
{
    virtual int amountWaterML() = 0;
    virtual void brew() = 0;
};

class CaffeineBeverage
{
    void prepare()
    {
        boilWater(recipe.amountWaterML());
        recipe.brew();
        pourInCup();
    }
};
class CoffeeRecipe : public Recipe
{
    CoffeeRecipe(int amountWaterML)
    : Recipe()
        , amountWaterML(amountWaterML)
    {}

    virtual void brew() { std::cout << "dripping Coffee through filter\n"; }

    virtual int amountWaterML() { return amountWaterML; }
};

class TeaRecipe : public Recipe
{
    TeaRecipe(int amountWaterML)
    : Recipe()
        , amountWaterML(amountWaterML)
    {}

    virtual void brew() { std::cout << "steeping Tea\n"; }

    virtual int amountWaterML() { return amountWaterML; }
};
Strategy (classic)
Application

CoffeeRecipe coffeeRecipe(150);
TeaRecipe teaRecipe(200);
CaffeineBeverage coffee(coffeeRecipe);
CaffeineBeverage tea(teaRecipe);

typedef vector<CaffeineBeverage*> Beverages;
Beverages beverages;

beverages.push_back(&coffee);
beverages.push_back(&tea);

for(Beverages::iterator it=beverages.begin(); it != beverages.end(); ++it)
{
  (*it)->prepare();
}

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
class CaffeineBeverage
{
    CaffeineBeverage(std::function<int()> amountWaterML, std::function<void()> brew)
        : brew(brew)
          , amountWaterML(amountWaterML)
    {}

    void prepare() const
    {
        boilWater(amountWaterML());
        brew();
        pourInCup();
    }
};

static void brewCoffee() { std::cout << "dripping Coffee through filter\n"; }

static void brewTea() { std::cout << "steeping Tea\n"; }

static int amountWaterML(int mL) { return mL; }
Strategy (C++11)
Application with bind

CaffeineBeverage coffee(
    bind(&Recipes::amountWaterMl, 150), &Recipes::brewCoffee);
CaffeineBeverage tea(
    bind(&Recipes::amountWaterMl, 200), &Recipes::brewTea);

typedef vector<CaffeineBeverage*> Beverages;
Beverages beverages;
beverages.push_back(&coffee);
beverages.push_back(&tea);

for_each(
    begin(beverages), end(beverages),
    bind(&CaffeineBeverage::prepare, placeholders::_1));

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup

Effective GoF Patterns - Tobias Darm
Strategy (C++11)
Application with lambda

```cpp
CaffeineBeverage coffee()
    { return Recipes::amountWaterMl(150); }, &Recipes::brewCoffee;
CaffeineBeverage tea()
    { return Recipes::amountWaterMl(200); }, &Recipes::brewTea);

using Beverages = vector<CaffeineBeverage*>;
Beverages beverages;
beverages.push_back(&coffee);
beverages.push_back(&tea);

for(auto beverage : beverages){ beverage->prepare(); }
```

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
Strategy
classic vs. C++11

• Classic

<table>
<thead>
<tr>
<th>CaffeineBeverage</th>
<th>Recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaffeineBeverage(Recipe)</td>
<td>amountWaterMl():int brew()</td>
</tr>
<tr>
<td>prepare()</td>
<td></td>
</tr>
</tbody>
</table>

TeaRecipe | CoffeeRecipe

• C++11

<table>
<thead>
<tr>
<th>CaffeineBeverage</th>
<th>Recipes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaffeineBeverage( function&lt;int()&gt; amountWaterMl, function&lt;void()&gt; brew)</td>
<td>amountWaterMl(int):int brewCoffee() brewTea()</td>
</tr>
<tr>
<td>prepare()</td>
<td></td>
</tr>
</tbody>
</table>
Chain

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
Chain
Example: Condiments
class Condiment
{
    Condiment(Condiment* next) : next(next) {}

    std::string description()
    {
        if(next) return this->onDescription() + next->description();
        return this->onDescription();
    }

    float price()
    {
        if(next) return this->onPrice() + next->price();
        return this->onPrice();
    }

    virtual std::string onDescription() = 0;
    virtual float onPrice() = 0;
};
class Sugar : public Condiment
{
    Sugar(Condiment* next) : Condiment(next) {}

    virtual std::string onDescription() { return "-Sugar-"; }

    virtual float onPrice() { return 0.07f; }
};

class Milk : public Condiment
{
    Milk(Condiment* next) : Condiment(next) {}

    virtual std::string onDescription() { return "-Milk-"; }

    virtual float onPrice() { return 0.13f; }
};
Chain (classic) Application

Condiment* milk = new Milk();
Condiment* sugarMilk = new Sugar(milk);
Condiment* doubleSugarMilk = new Sugar(sugarMilk);

cout << "Condiments: " << doubleSugarMilk->description() << '
';
cout << "Price: " << doubleSugarMilk->price() << '
';

Condiments: -Sugar--Sugar--Milk-
Price: 0.27
Chain (C++11)
Condiment - Sugar and Milk

```cpp
struct Condiment {
    std::function<std::string()> description;
    std::function<float()> price;
};

class Sugar {
    static std::string description() { return "-Sugar-"; }
    static float price() { return 0.07f; }
};

class Milk {
    static std::string description() { return "-Milk-"; }
    static float price() { return 0.13f; }
};
```

Effective GoF Patterns - Tobias Darm
Chain (C++11)
accu with function

template<typename Res>
static Res accu(std::function<Res()> call, std::function<Res()> next)
{
    if(next) return call() + next();
    return call();
}
Chain (C++11) Application with bind

```cpp
Condiment condiments;
condiments.description = bind(&accu<string>, &Milk::description, condiments.description);
condiments.description = bind(&accu<string>, &Sugar::description, condiments.description);
condiments.description = bind(&accu<string>, &Sugar::description, condiments.description);

condiments.price = bind(&accu<float>, &Milk::price, condiments.price);
condiments.price = bind(&accu<float>, &Sugar::price, condiments.price);
condiments.price = bind(&accu<float>, &Sugar::price, condiments.price);

cout << "Condiments: " << condiments.description() << '\n';
cout << "Price: " << condiments.price() << '\n';
```

```
Condiments: -Sugar--Sugar--Milk-
Price: 0.27
```
Chain (C++11)

accu for lambdas

template<typename Call, typename NextCall>
static auto accu(Call call, NextCall next) -> decltype(call() + next())
{
    if(next) return call() + next();
    return call();
}
Chain (C++11) Application with lambda

Condiment *condiments,*
condiments.description = [=] { return accu(&Milk::description, condiments.description); };
condiments.description = [=] { return accu(&Sugar::description, condiments.description); };
condiments.description = [=] { return accu(&Sugar::description, condiments.description); };

condiments.price = [=] { return accu(&Milk::price, condiments.price); };
condiments.price = [=] { return accu(&Sugar::price, condiments.price); };
condiments.price = [=] { return accu(&Sugar::price, condiments.price); };

cout << "Condiments: " << condiments.description() << '\n';
cout << "Price: " << condiments.price() << '\n';

Condiments: -Sugar--Sugar--Milk-
Price: 0.27
Chain
classic vs. C++11

- **classic**
  - Condiment
    - description()
    - price()
    - onDescription()
    - onPrice()
  - Milk
  - Sugar

- **C++11**
  - Condiment
    - description: function<string>()
    - price: function<float>()
  - Call, NextCall, Result
  - accu(call: Call, nextCall: NextCall): Result
  - Milk
    - description()
    - price()
  - Sugar
    - description()
    - price()
Command

• Encapsulate a request as an object
Command
Example: CoffeeMachine

Orders:
1. Coffee
2. Coffee
3. Tea
Command (classic)
MakeCaffeineBeverage

class Order
{
    virtual void execute() = 0;
};

class MakeCaffeineBeverage : public Order
{
    MakeCaffeineBeverage(CaffeineBeverage& beverage)
        : Order()
        , beverage(beverage)
    {}

    virtual void execute()
    {
        beverage.prepare();
    }
};
```cpp
class CoffeeMachine
{
    typedef std::vector<Order*> OrderQ;
    OrderQ orders;

    CoffeeMachine()
        : orders()
    {
    }

    void request(Order* order)
    {
        orders.push_back(order);
    }

    void start()
    {
        for(CommandQ::iterator it(orders.begin()); it != orders.end(); ++it)
        {
            (*it)->execute();
            delete (*it);
        }
        orders.clear();
    }
};
```
Command (classic) Application

CoffeeMachine coffeeMachine;

coffeeMachine.request(new MakeCaffeineBeverage(coffee));
coffeeMachine.request(new MakeCaffeineBeverage(tea));
coffeeMachine.start();

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
class CoffeeMachine
{
    typedef std::vector<std::function<void()>> OrderQ;

    CoffeeMachine()
        : orders()
    {}

    void request(OrderQ::value_type order)
    {
        orders.push_back(order);
    }

    void start()
    {
        for(auto const& order : orders){ order(); }
        orders.clear();
    }
};
Command (C++11) Application

CoffeeMachine coffeeMachine;

coffeeMachine.request(bind(&CaffeineBeverage::prepare, &coffee));
coffeeMachine.request(bind(&CaffeineBeverage::prepare, &tea));
coffeeMachine.start();

CoffeeMachine coffeeMachine;

coffeeMachine.request([&]{ coffee.prepare(); });
coffeeMachine.request([&]{ tea.prepare(); });
coffeeMachine.start();

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
Command classic vs. C++11

• classic

<table>
<thead>
<tr>
<th>CoffeeMachine</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(Order*)</td>
<td>execute()</td>
</tr>
<tr>
<td>start()</td>
<td>0..n</td>
</tr>
</tbody>
</table>

• C++11

<table>
<thead>
<tr>
<th>CoffeeMachine</th>
<th>CaffeineBeverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(function&lt;void()&gt;)</td>
<td>prepare()</td>
</tr>
<tr>
<td>start()</td>
<td></td>
</tr>
<tr>
<td>cmdQ:vector&lt;function&lt;void()&gt; &gt;</td>
<td></td>
</tr>
</tbody>
</table>
Benefits
Benefits are maintained

- Loose coupling
- Extendable
- Unit testable
Some More Benefits

• Less code
• Less coupling
• Easier to extend
Criticism
Criticism
Debugging

• Do not debug Library code
Criticism
Too Loose

CaffeineBeverage
CaffeineBeverage(
  function<int()> amountWaterMl,
  function<void()> brew
) prepare()

Recipes
amountWaterMl(int):int
brewCoffee()
brewTea()
Criticism

Too Loose

<table>
<thead>
<tr>
<th>coffee</th>
<th>Recipes</th>
</tr>
</thead>
</table>
| "function<void()>
  [brewCoffee]" | |
| "function<int()>
  [amountWaterMI]" | |
| tea | |
| "function<int()>
  [brewTea]" | |

Effective GoF Patterns - Tobias Darm
Criticism
Performance

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Virtual [ms]</th>
<th>Function [ms]</th>
<th>MalteSkarupke [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS 2012 Debug</td>
<td>1917</td>
<td>4195</td>
<td>3262</td>
</tr>
<tr>
<td>VS 2012 Release</td>
<td>800</td>
<td>770</td>
<td>630</td>
</tr>
<tr>
<td>Clang 3.1 –O0 –g3</td>
<td>1864</td>
<td>3161</td>
<td>2513</td>
</tr>
<tr>
<td>Clang 3.1 –O3</td>
<td>564</td>
<td>474</td>
<td>456</td>
</tr>
<tr>
<td>GCC 4.7.2 –O0 –g3</td>
<td>1755</td>
<td>3363</td>
<td>2587</td>
</tr>
<tr>
<td>GCC 4.7.2 –O3</td>
<td>555</td>
<td>466</td>
<td>431</td>
</tr>
</tbody>
</table>

- [http://probablydance.com/2013/01/13/a-faster-implementation-of-stdfunction/](http://probablydance.com/2013/01/13/a-faster-implementation-of-stdfunction/)
Criticism
Size

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Virtual [bytes]</th>
<th>Function [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang 3.3 –O0 –g3</td>
<td>58608</td>
<td>110808</td>
</tr>
<tr>
<td>Clang 3.3 –O3</td>
<td>24044</td>
<td>35096</td>
</tr>
</tbody>
</table>

- http://dl.dropbox.com/u/27990997/compare_functions.cpp
Criticism
Ownership

CoffeeMachine coffeeMachine;
Coffee coffee;

coffeeMachine.request([&](){ coffee.prepare(); });
coffeeMachine.start();

// class CoffeeMachine (C++11)
typedef std::vector<std::function<void()>> OrderQ;

void start()
{
    for(auto const& order : orders){ order(); }
}
Criticism
Ownership

// class CoffeeMachine (classic)
typedef std::vector<Order*> OrderQ;

void start()
{
    for(CommandQ::iterator it(orders.begin()); it != orders.end(); ++it)
    {
        (*it)->execute();
        delete (*it);
    }
}
Criticism

Transfer of Ownership

```cpp
CoffeeMachine coffeeMachine;
std::unique_ptr<

```
```cpp
CoffeeMachine::CoffeeMachine();
```
```cpp
std::unique_ptr<CaffeineBeverage> coffee(new Coffee());

```cpp
// this way?
coffeeMachine.request([&]((){ coffee->prepare() });

```cpp
// or this way?
coffeeMachine.request([c = std::move(coffee)](){ c->prepare() });
```cpp
```cpp
coffeeMachine.start();
```cpp
```


Effective GoF Patterns - Tobias Darm
Patterns and Boost
Observer

• Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Observer (classic)
Example: Finished!
Observer (classic)

CoffeeMachineObserver

class CoffeeMachineObserver
{
    virtual void finished() = 0;
};

class View : public CoffeeMachineObserver
{
    View()
        : Observer()
    {}

    virtual void finished()
    {
        std::cout << "Orders are ready to be served\n";
    }
};
class CoffeeMachine
{
    void addObserver(Observers::value_type o)
    {
        Observers::iterator it = std::find(Observers.begin(), Observers.end(), o);
        if(it == Observers.end()) Observers.push_back(o);
    }

    void removeObserver(Observers::value_type o)
    {
        Observers::iterator it = std::find(Observers.begin(), Observers.end(), o);
        if(it != Observers.end()) Observers.erase(it);
    }

    void notifyObservers()
    {
        for(Observers::iterator it(Observers.begin()); it != Observers.end(); ++it)
        { (*it)->finished(); }
    }

    void start()
    {
        // ... execute all commands
        this->notifyObservers();
    }
};

Effective GoF Patterns - Tobias Darm
Observer (classic) Application

```java
CoffeeMachine coffeeMachine;
View view;

coffeeMachine.addObserver(&view);

coffeeMachine.request(new MakeCaffeineBeverage(coffee));
coffeeMachine.request(new MakeCaffeineBeverage(tea));
coffeeMachine.start();
```

---

*boiling 150ml water*
*dripping Coffee through filter*
*pour in cup*
*boiling 200ml water*
*steeping Tea*
*pour in cup*
*Orders are ready to be served*
Observer
Boost.Signals2

- Managed signals/slots system
- Controlling order of callbacks
- Connection tracking
void hello()
{
    std::cout << "Hello ";
}

struct World
{
    void operator()()
    {
        std::cout << "World";
    }
};

struct CoutChar
{
    CoutChar(char c)
    :
        letter(c)
    {}

    void print()
    {
        std::cout << letter;
    }

    char letter;
};
Signals Introduction
Connect

World world;
CoutChar c("!");
signal<void ()> s;

s.connect(&hello);
s.connect(world);
s.connect(std::bind(&CoutChar::print, c));
s();

Hello World!
Signals Introduction
Order

s.disconnect_all_slots();
s.connect(1, world);
s.connect(0, &hello);
s.connect(2, std::bind(&CoutChar::print, c));
s();

Hello World!
Signals Introduction

Tracking

```cpp
s.disconnect_all_slots();
s.connect(1, world);
s.connect(0, &hello);
{
    std::shared_ptr<CoutChar> c(new CoutChar('!'));
    s.connect(2,
        signal<void>::slot_type(
            &CoutChar::print,
            c.get()).track_foreign(c));

    std::cout << s.num_slots();
}
s();
std::cout << s.num_slots();
```

```
3
Hello World
2
```
More about Boost.Signals2

- Explicit connection management
- Combining multiple return values
- Thread-Safe
- Header-Only
Observer (Signals)

CoffeeMachine

class CoffeeMachine
{
    void start()
    {
        for(auto const& cmd : commands){ cmd(); }
        commands.clear();
        sigFinished();
    }

    signal_type<void(), keywords::mutex_type<dummy_mutex>::type sigFinished;
};

class View
{
    void coffeeMachineFinished()
    {
        std::cout << "Orders are ready to be served\n";
    }
};

Effective GoF Patterns - Tobias Darm
Observer (Signals)
Application

```cpp
CoffeeMachine coffeeMachine;
View view;
coffeeMachine.sigFinished.connect(bind(&View::coffeeMachineFinished, &view));

coffeeMachine.request(bind(&CaffeineBeverage::prepare, &coffee));
coffeeMachine.request(bind(&CaffeineBeverage::prepare, &tea));
coffeeMachine.start();

CoffeeMachine coffeeMachine;
View view;
coffeeMachine.sigFinished.connect([&]{ view.coffeeMachineFinished(); });

coffeeMachine.request([&]{ coffee.prepare(); });
coffeeMachine.request([&]{ tea.prepare(); });
coffeeMachine.start();

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
Orders are ready to be served
```
Observer
classic vs. Signals

• classic

```
- addObserver()
- removeObserver()
- notifyObservers()
```

```
CoffeeMachineObserver
- finished()
```

```
View
```

• Boost

```
CoffeeMachine
- sigFinished:signal<void()>  
```

```
View
- coffeeMachineFinished()
```
Factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes

```
AbstractFactory
create():Product

FactoryA
create():Product

FactoryB
create():Product

Product

ProductA

ProductB
```
Factory (classic)
Example: BeverageFactory
Factory (classic)

CaffeineBeverageFactory

class CaffeineBeverageFactory
{
    virtual CaffeineBeverage* create() = 0;
};

class CoffeeFactory : public CaffeineBeverageFactory
{
    virtual CaffeineBeverage* create()
    {
        return new Coffee();
    }
};

class TeaFactory : public CaffeineBeverageFactory
{
    virtual CaffeineBeverage* create()
    {
        return new Tea();
    }
};
Factory (classic)
BeverageFactory

class BeverageFactory
{
    BeverageFactory()
    : factory()
    {
        factory["Coffee"] = new CoffeeFactory();
        factory["Tea"] = new TeaFactory();
    }

    ~BeverageFactory()
    {
        delete factory["Coffee"];  
        delete factory["Tea"]; 
    } 

    CaffeineBeverage* create(std::string const& beverage)
    {
        return factory[beverage]->create();
    }

    std::map<std::string, CaffeineBeverageFactory*> factory;
};
Factory (classic)
Application

BeverageFactory factory;
CaffeineBeverage* b1 = factory.create("Coffee");
CaffeineBeverage* b2 = factory.create("Tea");

b1->prepareRecipe();
b2->prepareRecipe();
delete b1;
delete b2;

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
Factory
Boost.Functional.Factory

• Lets you encapsulate a new expression as a function object
Introduction
Functional.Factory

boost::factory<T*>(arg1, arg2, arg3);

// same as
new T(arg1, arg2, arg3);

boost::value_factory<T*>(arg1, arg2, arg3);

// same as
T(arg1, arg2, arg3);
Introduction

Functional.Factory

- Forwarding arguments to the constructor
- Member functions to create different kinds of objects
- Factory base class might not be necessary
- Allows use of customized memory management
Factory (Boost)
BeverageFactory

class BeverageFactory
{
    BeverageFactory()
    : factory()
    {
        factory["Coffee"] =
            std::bind(
                boost::factory<CaffeineBeverage*>(()),
                std::function<int ()>(std::bind(&Recipes::amountWaterML, 150)),
                &Recipes::brewCoffee);

        factory["Tea"] =
            std::bind(
                boost::factory<CaffeineBeverage*>(()),
                std::function<int ()>(std::bind(&Recipes::amountWaterML, 200)),
                &Recipes::brewTea);
    }

    std::unique_ptr<CaffeineBeverage> create(std::string const& beverage)
    {
        return std::unique_ptr<CaffeineBeverage>(factory[beverage]());
    }

    std::map<std::string, std::function<CaffeineBeverage*()>> factory;
};
Factory (Lambda)
BeverageFactory

BeverageFactory()
: factory()
{
    factory["Coffee"] = []
    {
        return new CaffeineBeverage(
            []{ return Recipes::amountWaterMl(150); },
            &Recipes::brewCoffee);
    }

    factory["Tea"] = []
    {
        return new CaffeineBeverage(
            [] { return Recipes::amountWaterMl(200); },
            &Recipes::brewTea);
    }
}
Factory (Boost/Lambda)
Application

BeverageFactory factory;

factory.create("Coffee")->prepare();
factory.create("Tea")->prepare();

boiling 150ml water
dripping Coffee through filter
pour in cup
boiling 200ml water
steeping Tea
pour in cup
Factory
classic vs. Boost.Functional.Factory

• Classic

• Boost
Library
Benefits

• Has been done for you
• Just using the library
• Many shortcomings addressed
Library
Benefits

• Development speed up
• GoF-Patterns focused on OO-paradigm
• Use of generic programming model
## Boost and Patterns

<table>
<thead>
<tr>
<th></th>
<th>Function Bind</th>
<th>Phoenix Lambda</th>
<th>Signals</th>
<th>Statechart MSM</th>
<th>Flyweight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Observer</td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>State</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>Proxy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effective GoF Patterns - Tobias Darm
Boost and Patterns
Flyweight
Boost.Flyweight
Example: Colour

class Colour
{
    ~Colour() { --s_counter; }

    Colour() { ++s_counter; }

    Colour(char red, char green, char blue)
    {
        ++s_counter;
    }

    Colour(Colour const& right)
    {
        ++s_counter;
    }

    Colour& operator=(Colour const& right) { ... }

    bool Colour::operator==(Colour const& right) const { ... }
};
Boost.Flyweight
Boost.Hash

```cpp
std::size_t hash_value(Colour const& c)
{
    std::size_t seed = 0;

    boost::hash_combine(seed, c.getBlue());
    boost::hash_combine(seed, c.getGreen());
    boost::hash_combine(seed, c.getRed());

    return seed;
}
```
class Shape
{
    //...
    boost::flyweight<Colour> m_colourFg;
    boost::flyweight<Colour> m_colourBg;
};

Shape s1; // 1
Shape s2; // 1

s1.setBgColour(
    Colour(12, 56, 253)); // 2
s2.setFgColour(
    Colour(12, 56, 253)); // 2
Boost.Flyweight

- Tagging
- Hash- or Set-based Tablelookup
- Tracking policy for Values
- Locking policy for Factory
Boost.MSM
State

• EDSL for transitiontable
• State, Transition, Event
• Submachines, Orthogonal Regions, Pseudostates
• History
• Completion/Anonymous transition
• Internal transitions (Action in State)
MSM
Example: Ping - Pong
MSM
States

```cpp
struct sm_ : public msm::front::state_machine_def<sm_>
{
    struct state_1 : public msm::front::state<>
    {
        template <class Event, class FSM>
        void on_entry(Event const&, FSM& ) {std::cout << "entering: state_1\n"; }

        template <class Event, class FSM>
        void on_exit(Event const&, FSM& ) {std::cout << "leaving: state_1\n"; }
    };

    struct state_2 : public msm::front::state<>
    {
        template <class Event, class FSM>
        void on_entry(Event const&, FSM& ) {std::cout << "entering: state_2\n"; }

        template <class Event, class FSM>
        void on_exit(Event const&, FSM& ) {std::cout << "leaving: state_2\n"; }
    };

    typedef state_1 initial_state;

    // ...
};
```
MSM
Actions

// struct sm_

struct action_ping_received
{
    template <class EVT, class FSM, class SourceState, class TargetState>
    void operator() (EVT const& , FSM& , SourceState& , TargetState& )
    {
        std::cout << "action_ping_received\n";
    }
};

struct action_pong_received
{
    template <class EVT, class FSM, class SourceState, class TargetState>
    void operator() (EVT const& , FSM& , SourceState& , TargetState& )
    {
        std::cout << "action_pong_received\n";
    }
};
namespace event { struct ping {}; struct pong {}; }

// struct sm_ ...

struct transition_table : mpl::vector<
    // Start   Event   Next   Action   Guard
    // +----------+--------+--------+----------------+----------------+
    Row< state_1 , event::ping , state_2 , action_ping_received , none  >,
    Row< state_2 , event::pong , state_1 , action_pong_received , none  >
}>;

template <class FSM, class Event>
void no_transitoin(Event const& e, FSM&, int state)
{
    std::cout << "no transition from state " << state
               << " on event " << typeid(e).name() << '\n';
}
typedef msm::back::state_machine<sm_> sm;

int main()
{
    sm s;
    s.start();

    s.process_event(event::ping());
    s.process_event(event::pong());
    s.process_event(event::ping());
    s.process_event(event::pong());
    s.process_event(event::pong());
    s.stop();
}

entering: sm_
entering: state_1
leaving: state_1
action_ping_received
entering: state_2
leaving: state_2
action_pong_received
entering: state_1
leaving: state_1
action_ping_received
entering: state_2
leaving: state_2
action_pong_received
entering: state_1
no transition from state 0 on event N5event4pongE
leaving: state_1
leaving: sm_
Putting it all together

- Writing a Coffeemachine application
- Putting all the patterns together
Putting it all together
classic

typedef std::vector<CaffeineBeverage*> Beverages;
Beverages beverages;
coffeeMachine.addObserver(&view);
do {
    std::string inBeverage;
    if(!view.askForBeverage(inBeverage)) break;
    beverages.push_back(beverageFactory.create(inBeverage));
    CondimentFactory condimentFactory;
    Condiment* condiments = 0;
    do {
        std::string inCondiment;
        if(!view.askForCondiments(inCondiment)) break;
        condiments = condimentFactory.create(inCondiment, condiments);
    } while(true);
    beverages.back()->condiments(condiments);
} while(true);
Putting it all together

```cpp
if(!beverages.empty())
{
    for(Beverages::iterator it(beverages.begin()); it != beverages.end(); ++it)
    {
        coffeeMachine.request(new MakeCaffeineDrink(**it));
    }
    coffeeMachine.start();
    do
    {
        beverages.back()->description();
        beverages.back()->price();
        delete beverages.back();
        beverages.pop_back();
    } while(!beverages.empty());
}
```
Putting it all together
classic
Putting it all together

C++11

using Beverages = std::vector<std::unique_ptr<CaffeineBeverage>>;
Beverages beverages;
coffeeMachine.getNotifiedOnFinished([&]{ view.coffeeMachineFinished(); });
do {
  std::string inBeverage;
  if(!view.askForBeverage(inBeverage)) break;
  beverages.emplace_back(beverageFactory.create(inBeverage));
  Condiment condiments;
  do {
    CondimentFactory condimentFactory;
    std::string inCondiment;
    if(!view.askForCondiments(inCondiment)) break;
    Condiment condiment = condimentFactory.create(inCondiment);
    condiments.description = [=]{
      return accu(condiment.description, condiments.description); }
    condiments.price = [=]{
      return accu(condiment.price, condiments.price); }
  } while(true);
  beverages.back()->condiments(condiments);
} while(true);
Putting it all together

C++11

```cpp
if(!beverages.empty())
{
    for(auto& beverage : beverages)
    {
        coffeeMachine.request([]{ beverage->prepareReceipt(); });
    }
    coffeeMachine.start();
    for(auto& beverage : beverages)
    {
        beverage->getDescription();
        beverage->getPrice();
    }
}
```
Putting it all together
C++11
Putting it all together (classic)
Dependencies

Effective GoF Patterns - Tobias Darm
Putting it all together (C++11/Boost)
Dependencies

Effective GoF Patterns - Tobias Darm
# Putting it all together

## Analysis

<table>
<thead>
<tr>
<th></th>
<th>classic</th>
<th>C++11</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountDeclClass</td>
<td>25</td>
<td>12</td>
</tr>
<tr>
<td>CountDeclMethodAll</td>
<td>168</td>
<td>58</td>
</tr>
<tr>
<td>CountLineCode</td>
<td>522</td>
<td>278</td>
</tr>
<tr>
<td>CountPath</td>
<td>74</td>
<td>31</td>
</tr>
<tr>
<td>MaxInheritanceTree</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SumCyclomatic</td>
<td>74</td>
<td>31</td>
</tr>
</tbody>
</table>

*Effective GoF Patterns - Tobias Darm*
Conclusion

3 Observations
Patterns are crutches ...

• ... for features that the language does not have
Peter Norvig

16 of the 23 GoF patterns are simpler or even invisible in higher-level languages

http://norvig.com/design-patterns/ppframe.htm
Where is the Pattern?

Strategy

<table>
<thead>
<tr>
<th>CaffeineBeverage</th>
<th>Recipes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaffeineBeverage( function&lt;int()&gt; amountWaterMl, function&lt;void()&gt; brew) prepare()</td>
<td>amountWaterMl(int):int brewCoffee() brewTea()</td>
</tr>
</tbody>
</table>

```cpp
CaffeineBeverage(std::function<int()> amountWaterMl, std::function<void()> brew)
```
condiments.description = [=]{ return accu(condiment.description, condiments.description); };

Effective GoF Patterns - Tobias Darm
Where is the Pattern?
Command

<table>
<thead>
<tr>
<th>CoffeeMachine</th>
<th>CaffeineBeverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(function&lt;void()&gt;)</td>
<td>prepare()</td>
</tr>
<tr>
<td>start()</td>
<td></td>
</tr>
<tr>
<td>cmdQ:vector(function&lt;void()&gt; &gt;</td>
<td></td>
</tr>
</tbody>
</table>

// class CoffeeMachine
typedef std::function<void()> Order;
typedef std::vector<Order> OrderQ;
Jeff Atwood

Design patterns are a form of complexity

Patterns for Communication
Patterns for Communication
Solving Problems

Effective GoF Patterns - Tobias Darm
Patterns for Communication
Solving Problems

Strategy

Effective GoF Patterns - Tobias Darm
Patterns for Communication
Solving Problems

• Classic

<table>
<thead>
<tr>
<th>CaffeineBeverage</th>
<th>Recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaffeineBeverage(Recipe)</td>
<td>amountWaterMl():int</td>
</tr>
<tr>
<td>prepare()</td>
<td>brew()</td>
</tr>
</tbody>
</table>

TeaRecipe  CoffeeRecipe

• C++11

<table>
<thead>
<tr>
<th>CaffeineBeverage</th>
<th>Recipes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaffeineBeverage( function&lt;int()&gt; amountWaterMl, function&lt; void ()&gt; brew )</td>
<td>amountWaterMl(int):int</td>
</tr>
<tr>
<td>prepare()</td>
<td>brewCoffee()</td>
</tr>
<tr>
<td>brewTea()</td>
<td></td>
</tr>
</tbody>
</table>
Effective GoF Patterns - Tobias Darm
Patterns for Communication
Flyweight

Effective GoF Patterns - Tobias Darm
Patterns for Communication Chain

Effective GoF Patterns - Tobias Darm
Patterns for Communication
Observer

Effective GoF Patterns - Tobias Darm
Patterns for Communication
Command
It is not a requirement for a pattern to be visible in a class diagram
2nd Edition – GoF book?
Evolution

- Assembler ➔ if/else, do-while, ...
- C ➔ class, polymorphism, ...
- C++ ➔ Strategy, Command, ...
- C++11 ➔ ...
Pattern Lifecycle

• Discovered
• Published
• Test of time
• Adopted by language/library
• Disappears
PPG

- Pattern
- Preservation
- Group
Books
Thank you for your attention