

Effective Patterns

with C++11 and Boost

me

- no books
- no blogs
- no tweets
- no open source

me

- **Dräger** Lübeck
- C++ developer
- Subsystemdesigner
- Trainings/Workshops



Motivation

My Approach

Unfair

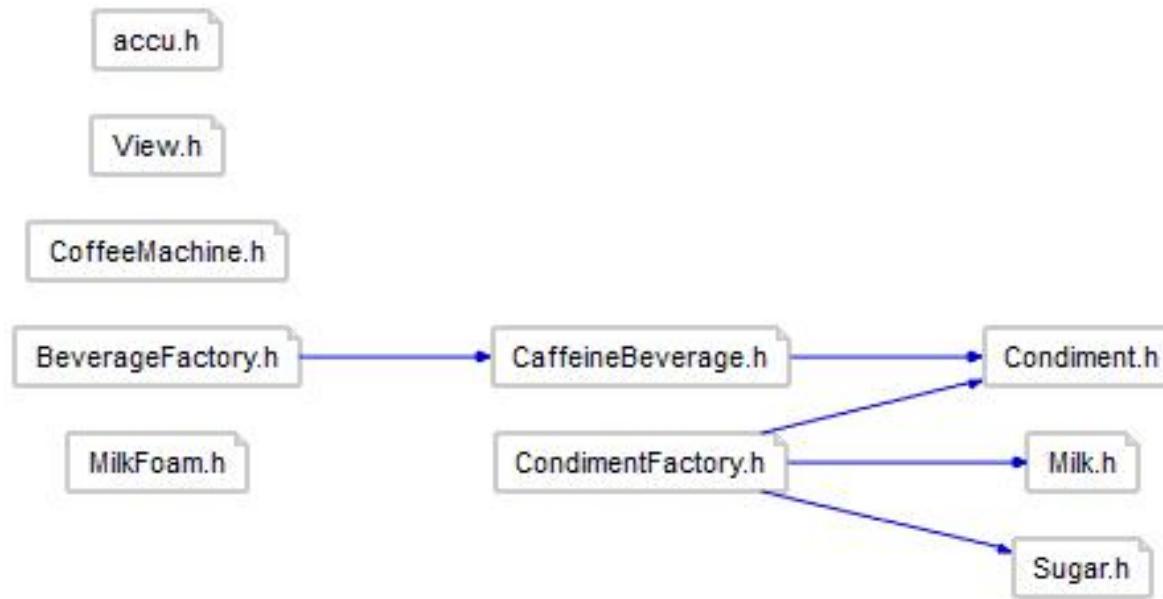
Content

- Implementation classic vs. C++11
- Implementation classic vs. Boost
- Application classic vs. C++11/Boost
- Conclusion concerning patterns

CoffeeMachine (classic) Application

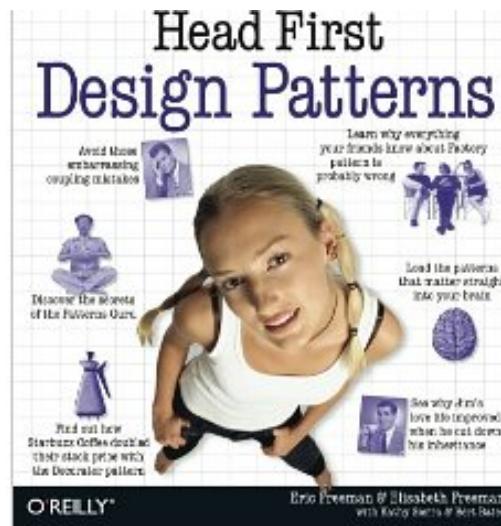
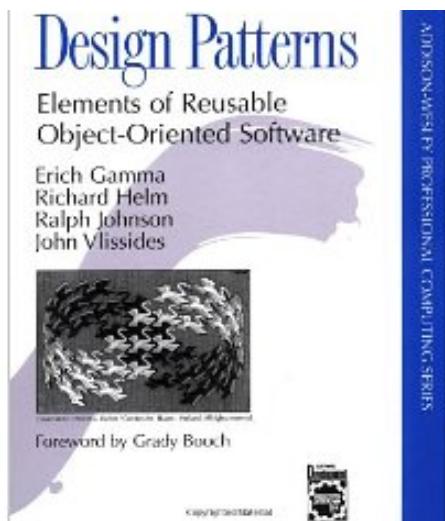


CoffeeMachine (C++11/Boost) Application



GoF-Patterns

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



GoF-Patterns

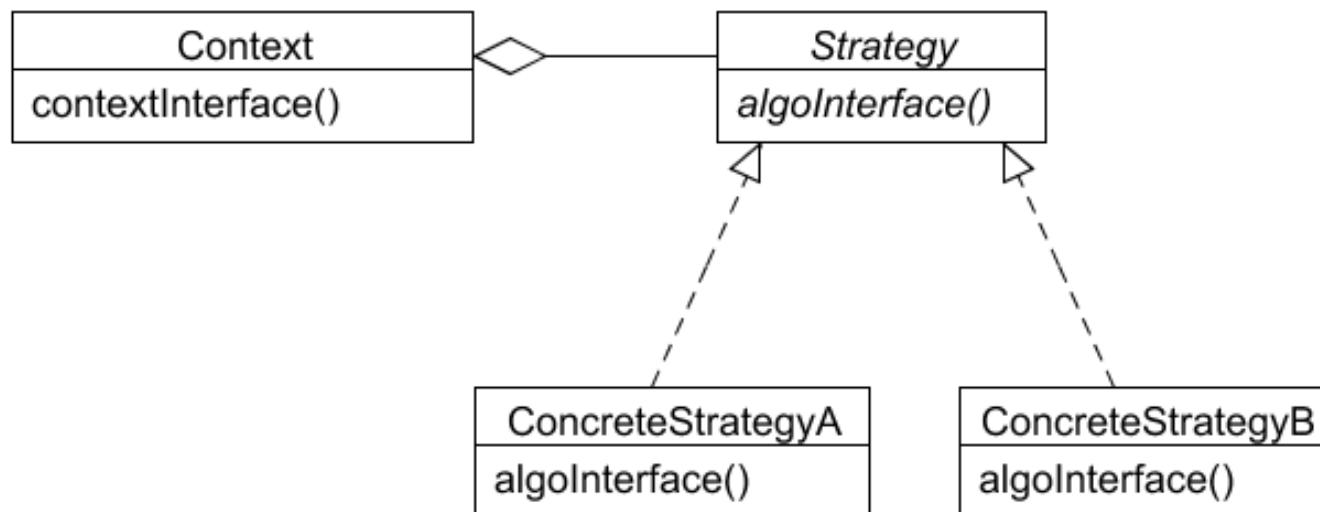
- Creational
 - Abstract Factory
 - Builder
 - Factory Method
 - Prototype
 - (Singleton)
- Structural
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Facade
 - Flyweight
- Behavioural
 - Chain
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - Observer
 - State
 - Strategy
 - Template
 - 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



Strategy (classic) Recipe Interface

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

class CaffeineBeverage
{
    void prepare()
    {
        boilWater(recipe.amountWaterMl());
        recipe.brew();
        pourInCup();
    }
};
```

Strategy (classic) Concrete Recipes

```
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
```

Strategy (C++11)

CaffeineBeverage/Recipes

```
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*

Strategy (C++11)

Application with lambda

```
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(); }
```

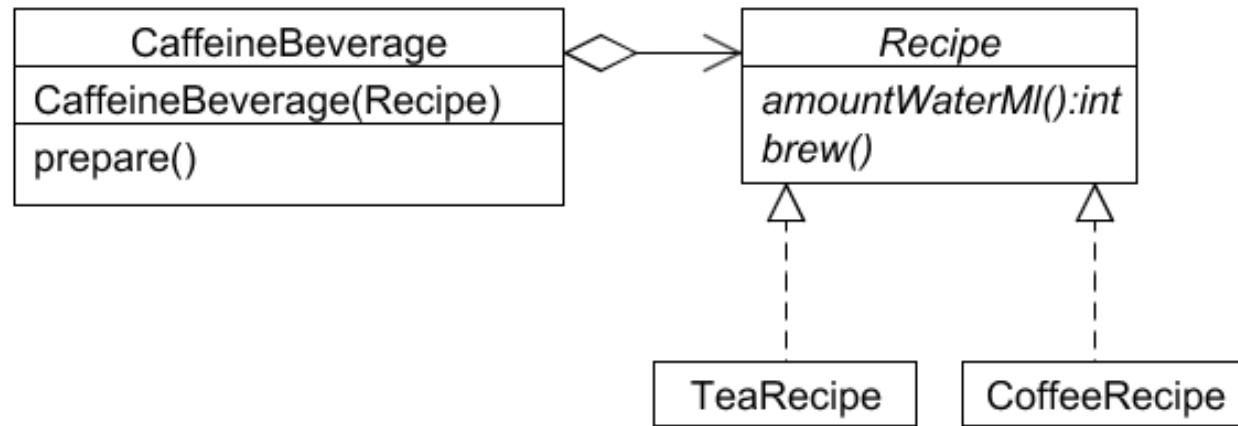
*boiling 150ml water
dripping Coffee through filter
pour in cup*

*boiling 200ml water
steeping Tea
pour in cup*

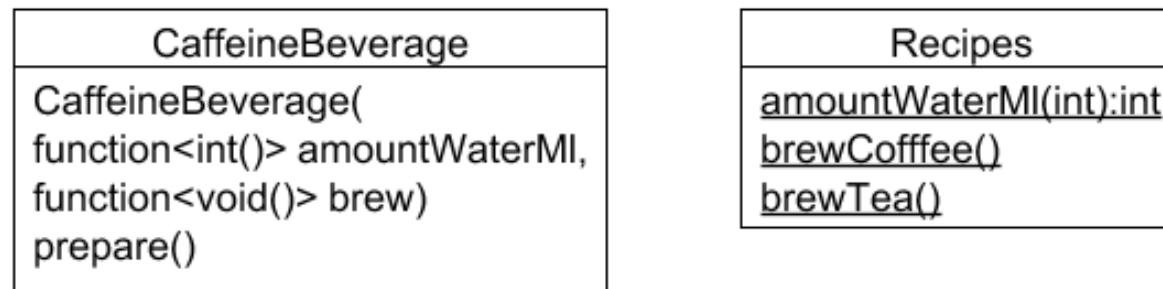
Strategy

classic vs. C++11

- Classic

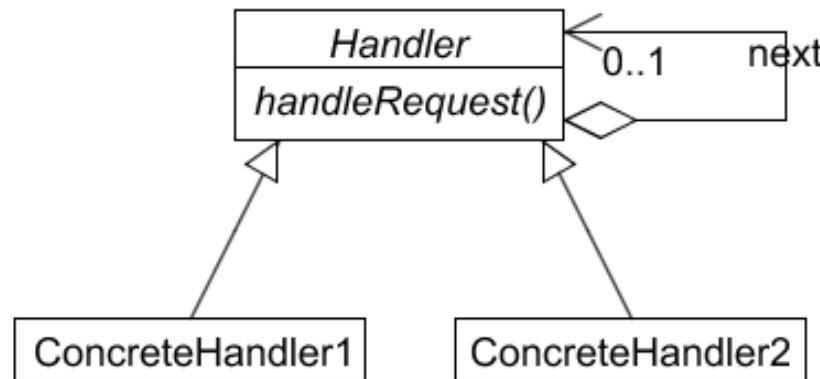


- C++11



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



Chain (classic) Condiment

```
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;
};
```

Chain (classic) Sugar and Milk

```
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() << '\n';
cout << "Price: " << doubleSugarMilk->price() << '\n';
```

Condiments: -Sugar--Sugar--Milk-
Price: 0.27

Chain (C++11)

Condiment - Sugar and Milk

```
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; }
};
```

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

```
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); };

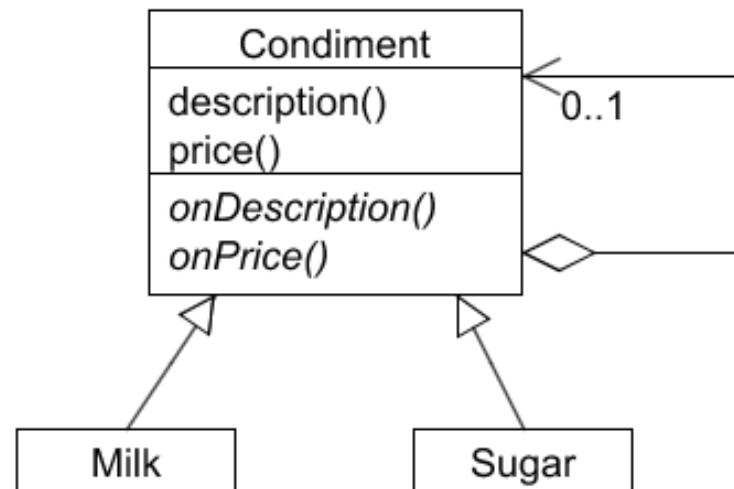
cout << "Condiments: " << condiments.description() << '\n';
cout << "Price: " << condiments.price() << '\n';
```

Condiments: -Sugar--Sugar--Milk-
Price: 0.27

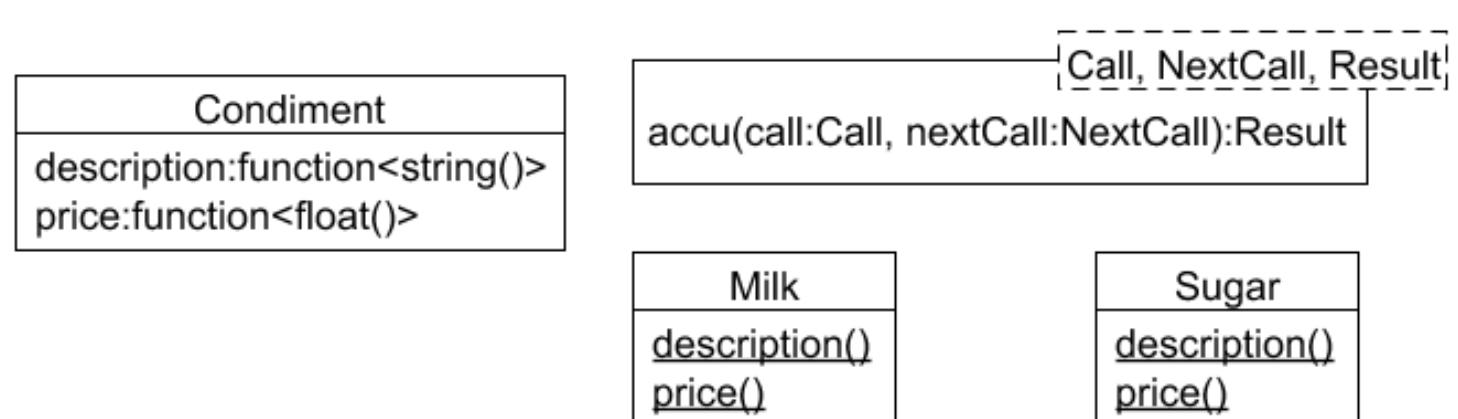
Chain

classic vs. C++11

- classic

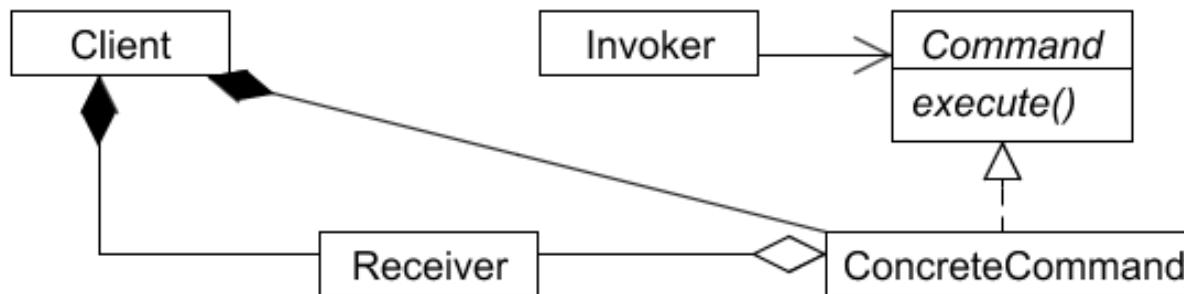


- C++11



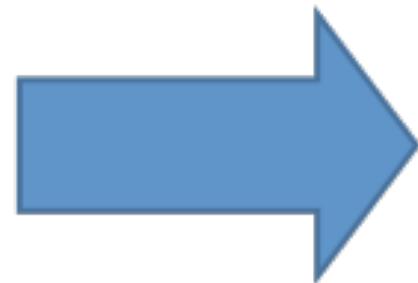
Command

- Encapsulate a request as an object



Command

Example: CoffeeMachine



Command (classic)

MakeCaffeineBeverage

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

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

    virtual void execute()
    {
        beverage.prepare();
    }
};
```

Command (classic)

CoffeeMachine

```
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*

Command (C++11)

CoffeeMachine

```
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

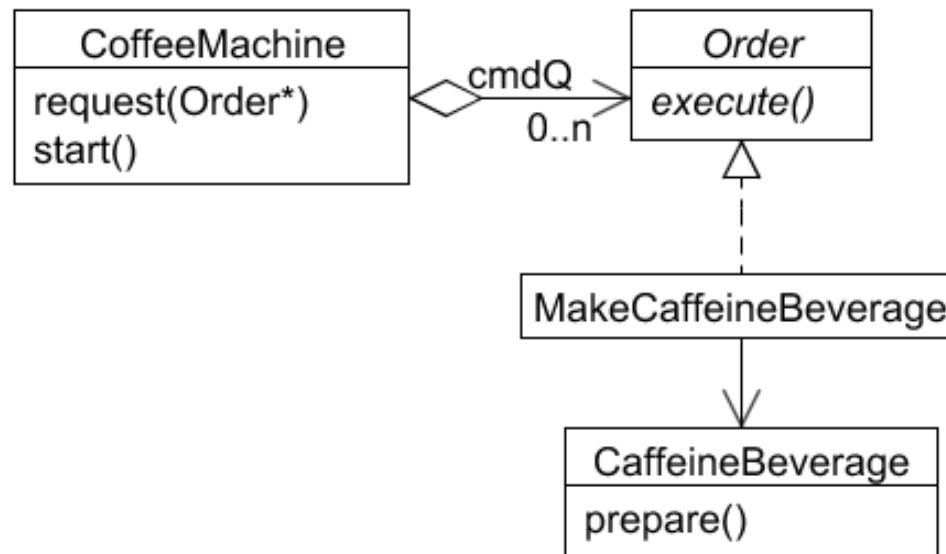
steeping Tea

pour in cup

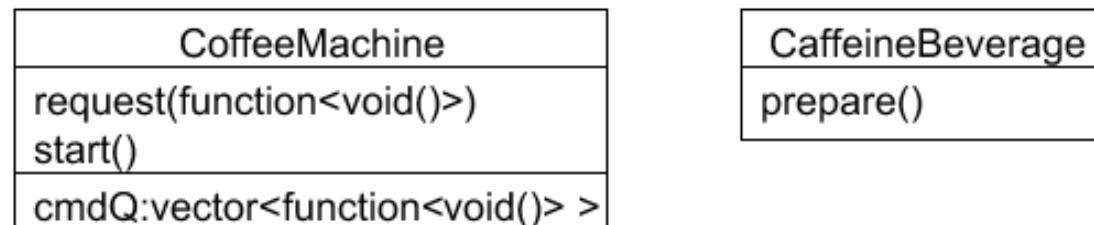
Command

classic vs. C++11

- classic



- C++11



Benefits

Benefits are maintained

- Loose coupling
- Extendable
- Unit testable

Some More Benefits

- Less code
- Less coupling
- Easier to extend

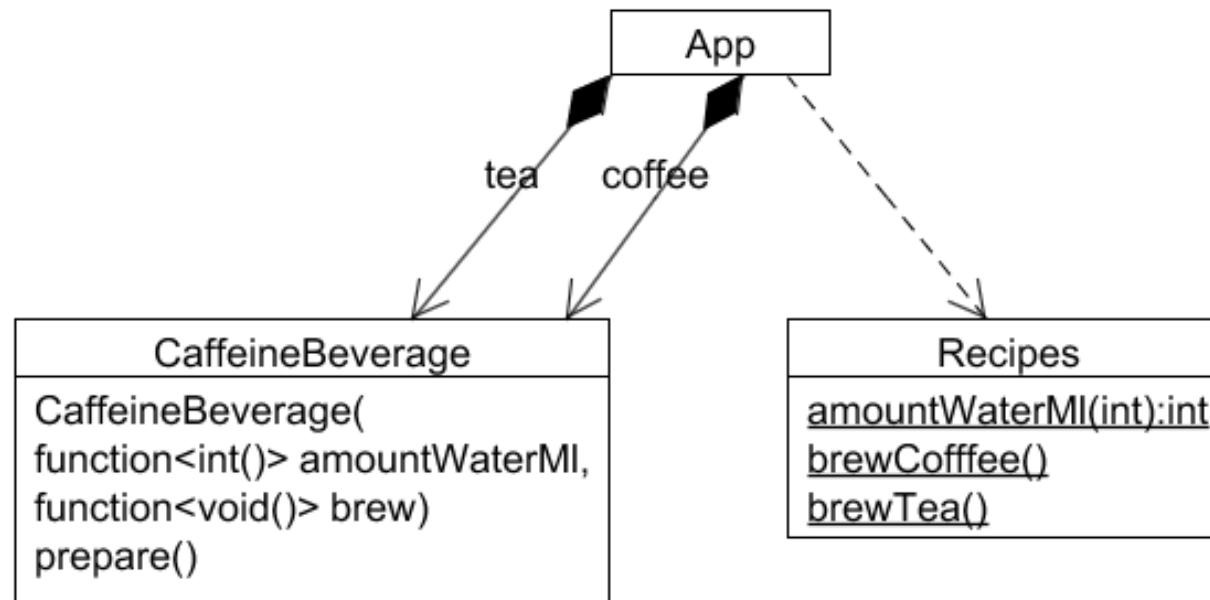
Criticism

Criticism Debugging

- Do not debug Library code

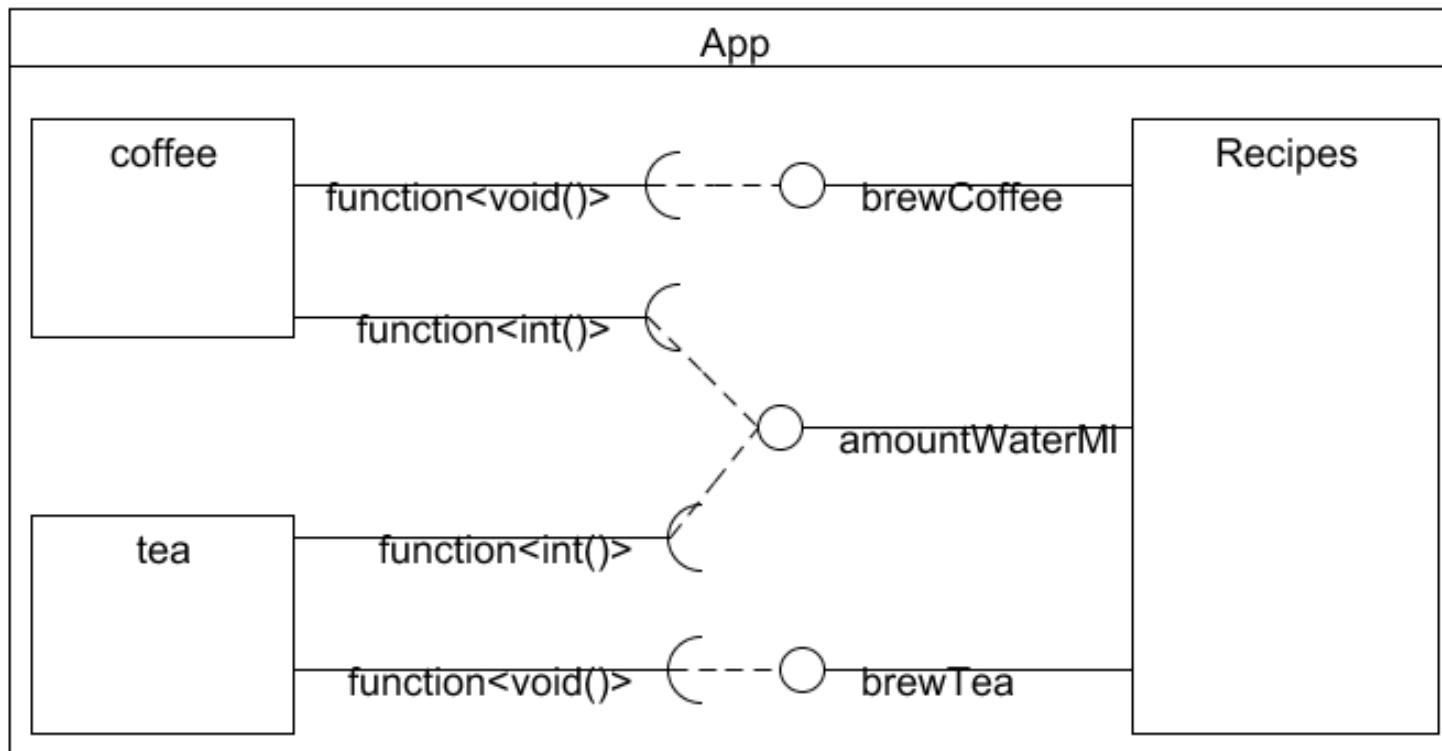
Criticism

Too Loose



Criticism

Too Loose



Criticism Performance

Compiler	Virtual [ms]	Function [ms]	MalteSkarupke[m s]
VS 2012 Debug	1917	4195	3262
VS 2012 Release	800	770	630
Clang 3.1 -O0 -g3	1864	3161	2513
Clang 3.1 -O3	564	474	456
GCC 4.7.2 -O0 -g3	1755	3363	2587
GCC 4.7.2 -O3	555	466	431

- <http://probablydance.com/2013/01/13/a-faster-implementation-of-stdfunction/>

Criticism Size

Compiler	Virtual [bytes]	Function [bytes]
Clang 3.3 -O0 -g3	58608	110808
Clang 3.3 -O3	24044	35096

- 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

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

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

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

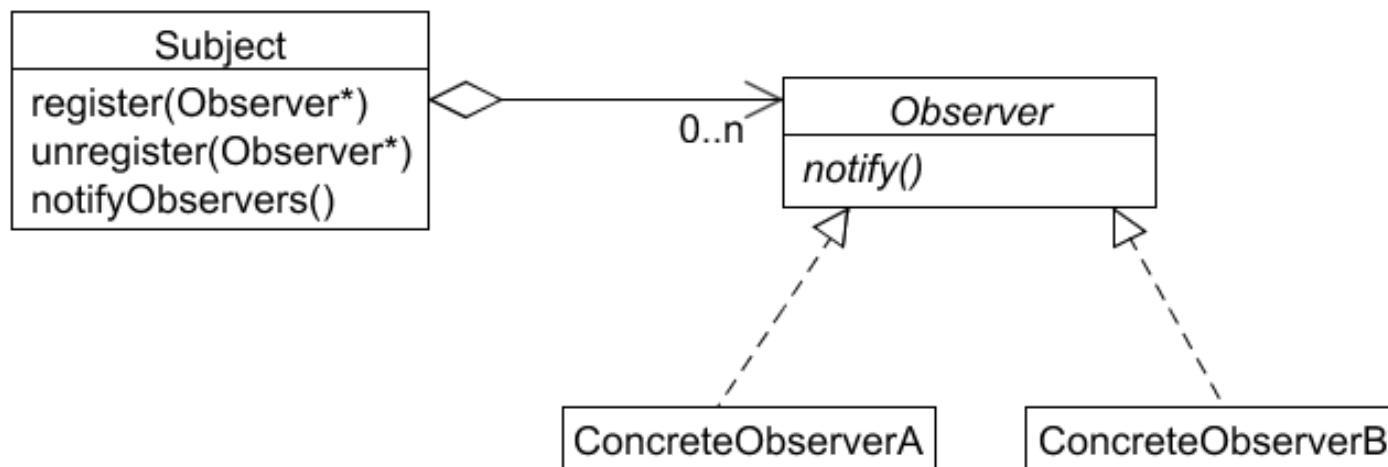
coffeeMachine.start();
```

- <http://isocpp.org/files/papers/n3610.html>

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";
    }
};
```

Observer (classic)

CoffeeMachine

```
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();
    }
};
```

Observer (classic) Application

```
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

Signals Introduction Slots

```
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

```
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";
    }
};
```

Observer (Signals) Application

```
CoffeeMachine coffeeMachine;  
View view;  
coffeeMachine.sigFinsihed.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

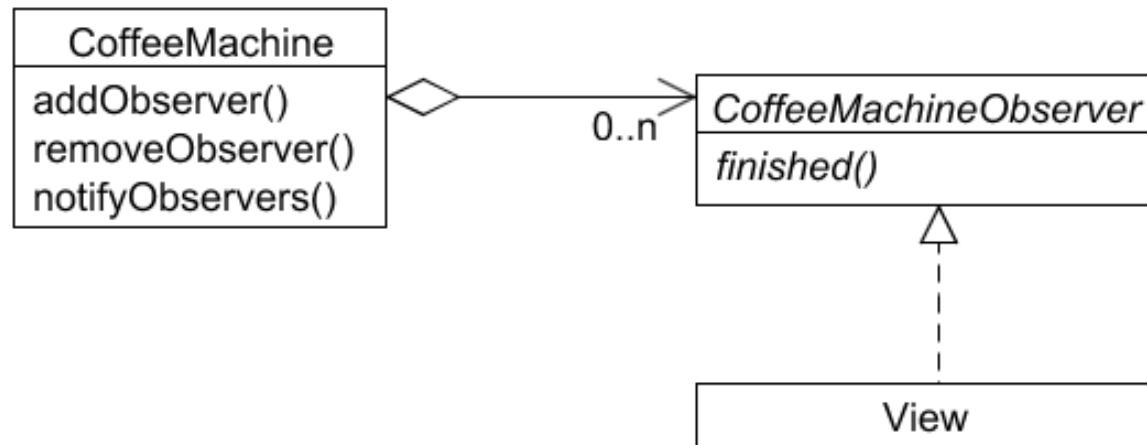
pour in cup

Orders are ready to be served

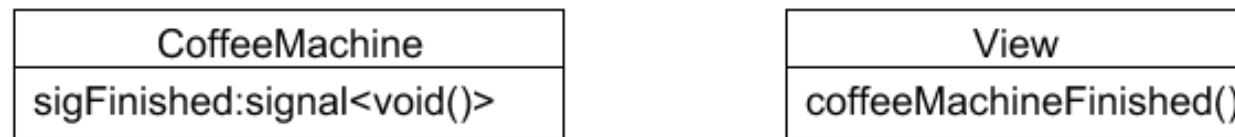
Observer

classic vs. Signals

- classic

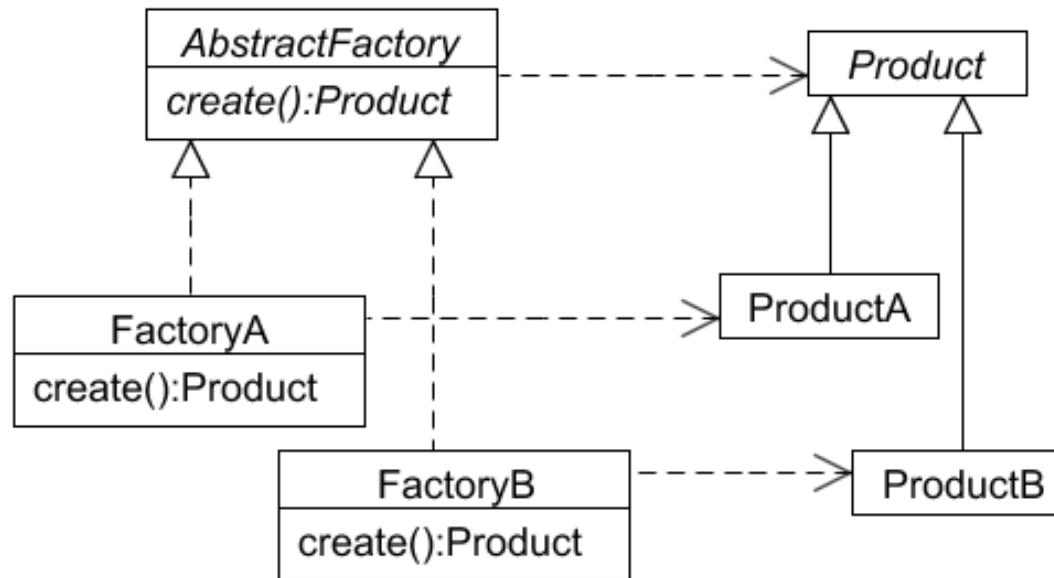


- Boost



Factory

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



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->prepareReceipe();  
b2->prepareReceipe();  
  
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

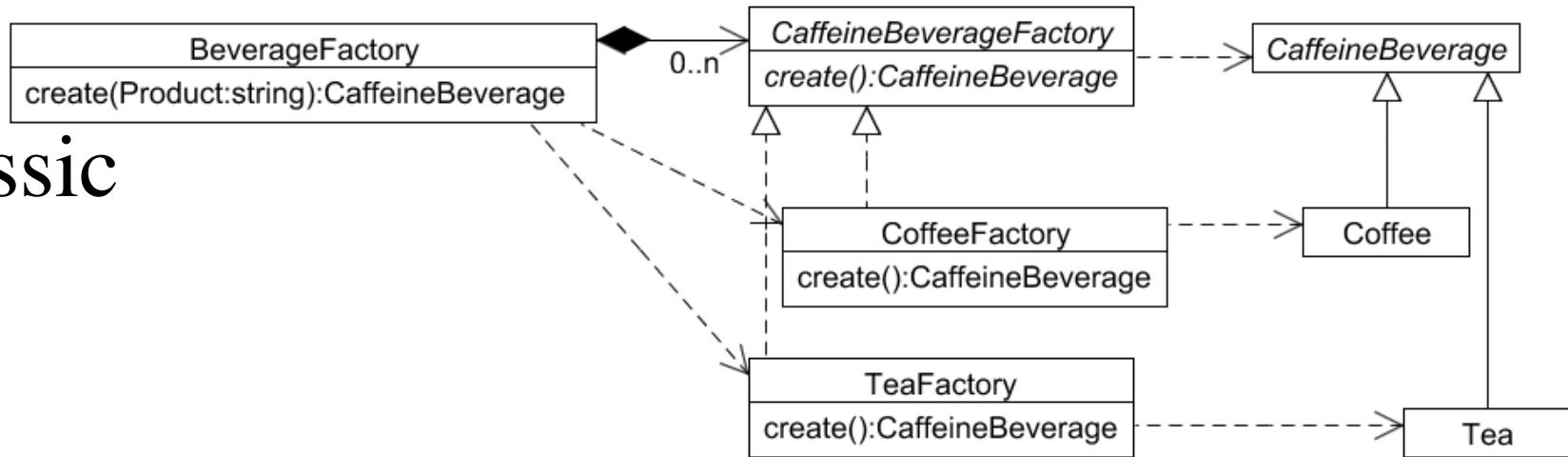
```
BeverageFactory factory;  
  
factory.create("Coffee")->prepare();  
factory.create("Tea")->prepare();
```

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

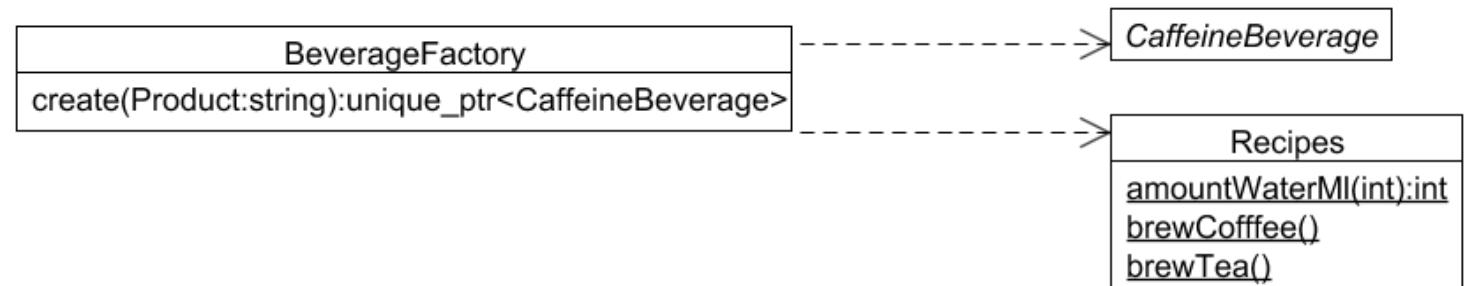
Factory

classic vs. Boost.Functional.Factory

- Classic



- Boost



Library Benefits

- Has been done for you
- Just using the library
- Many shortcomings addressed

Library Benefits

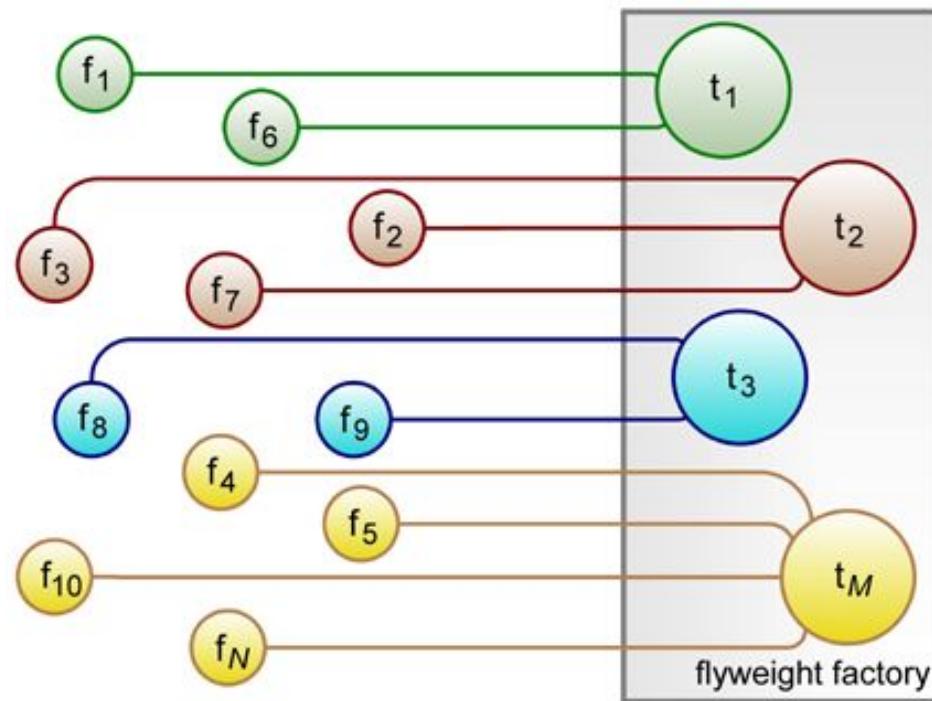
- Development speed up
- GoF-Patterns focused on OO-paradigm
- Use of generic programming model

Boost and Patterns

	Function Bind	Phoenix Lambda	Signals	Statechart MSM	Flyweight
Chain					
Command					
Observer					
Strategy					
State					
Proxy					
Prototype					

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

```
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;
}
```

Boost.Flyweight

Boost.Hash

```
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
```

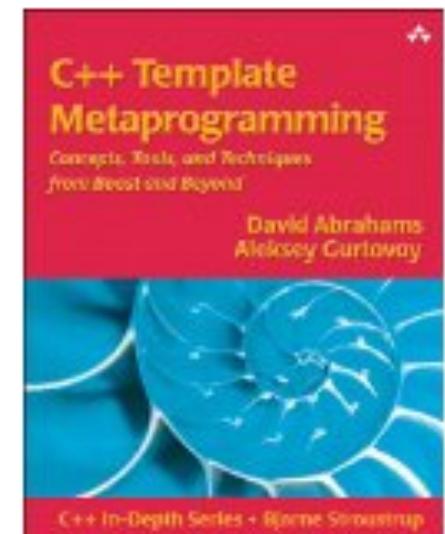
Boost.Flyweight

- Tagging
- Hash- or Set-based Tablelookup
- Trackingpolicy for Values
- Lockingpolicy for Factory

Boost.MSM

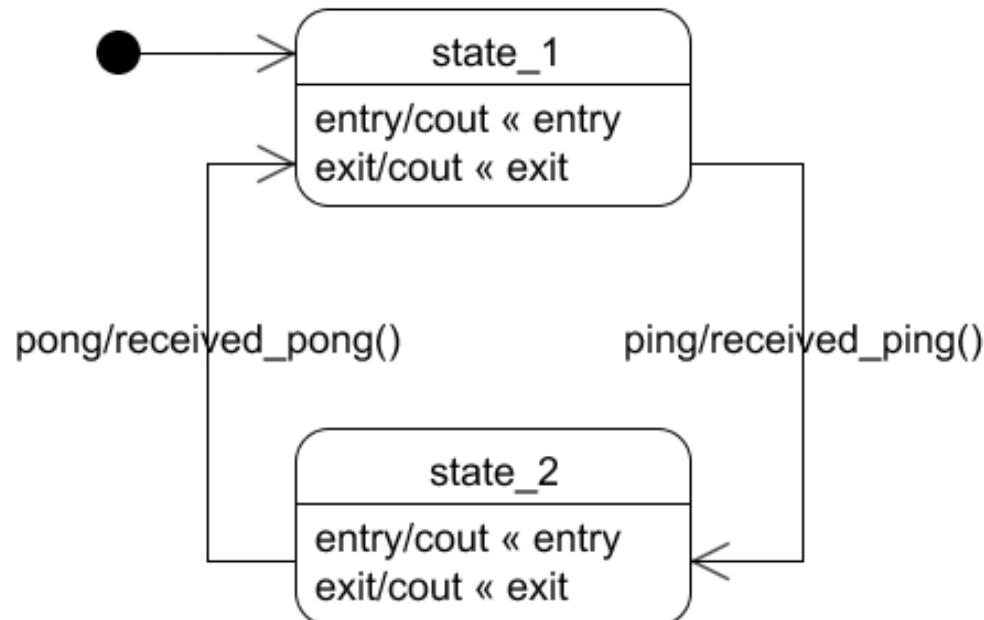
State

- 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

```
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";
    }
};
```

MSM

Transitiontable

```
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_transition(Event const& e, FSM&, int state)
{
    std::cout << "no transition from state " << state
                << " on event " << typeid(e).name() << '\n';
}
```

MSM

Application

```
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 classic

```
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);
```

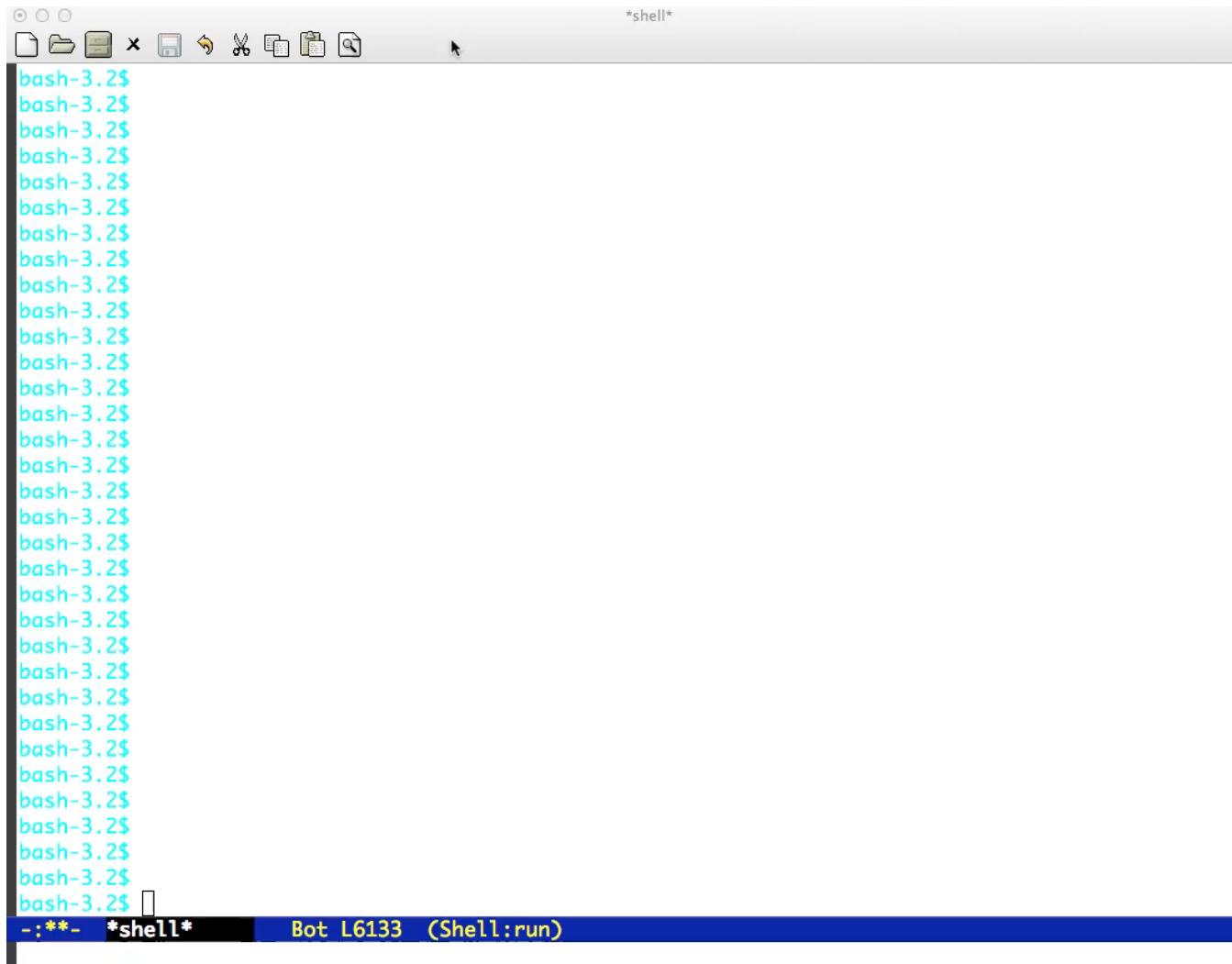
Putting it all together

C++11

```
if(!beverages.empty())
{
    for(auto& beverage : beverages)
    {
        coffeeMachine.request([&]{ beverage->prepareReceipe(); });
    }
    coffeeMachine.start();
    for(auto& beverage : beverages)
    {
        beverage->description();
        beverage->price();
    }
}
```

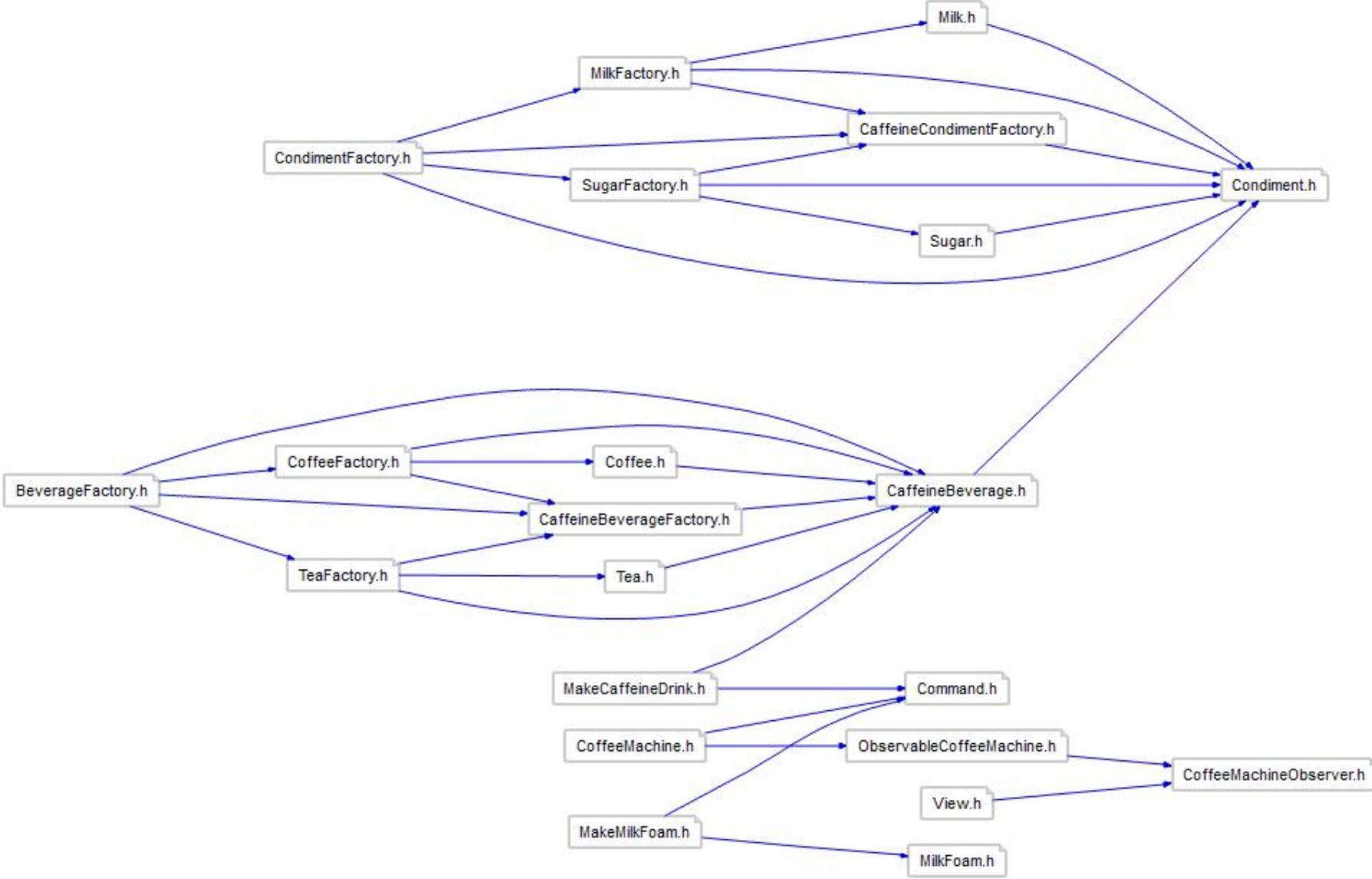
Putting it all together

C++11



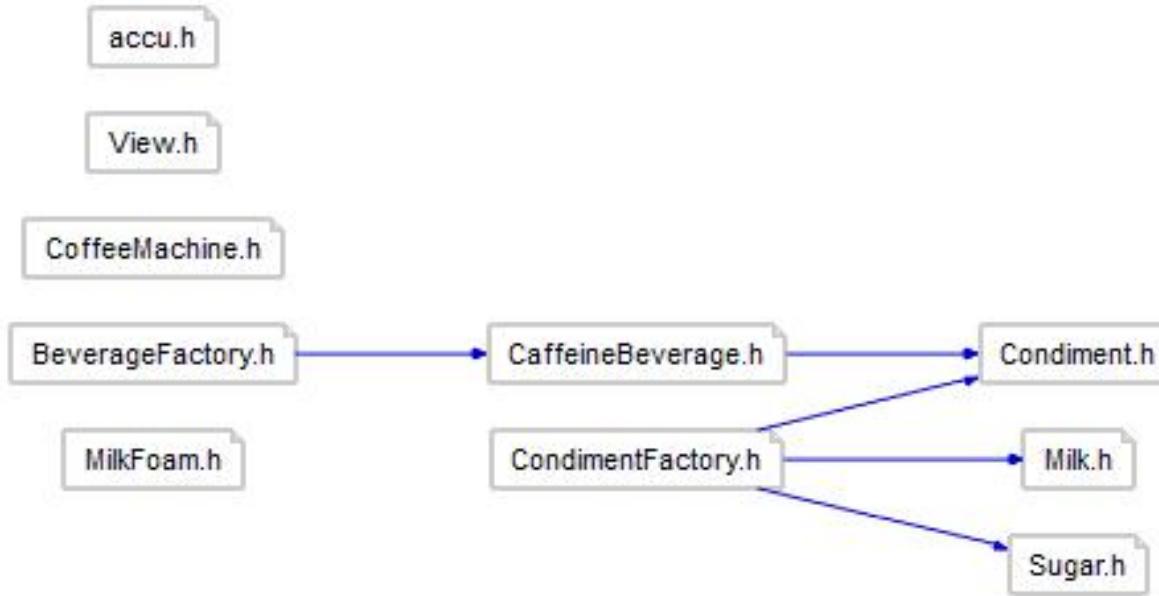
The image shows a screenshot of a terminal window titled "*shell*". The window has a toolbar at the top with icons for file operations like Open, Save, and Cut. The main area of the terminal is filled with numerous identical lines of text, each consisting of the string "bash-3.2\$" followed by a small square cursor icon. At the bottom of the terminal window, there is a dark blue status bar. On the left side of the status bar, it says "-:**- *shell*". To the right of the status bar, the text "Bot L6133 (Shell:run)" is displayed in white. The overall appearance is that of a standard Linux terminal interface.

Putting it all together (classic) Dependencies



Putting it all together (C++11/Boost)

Dependencies



Putting it all together

Analysis

	classic	C++11
CountDeclClass	25	12
CountDeclMethodAll	168	58
CountLineCode	522	278
CountPath	74	31
MaxInheritanceTree	1	0
SumCyclomatic	74	31

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?

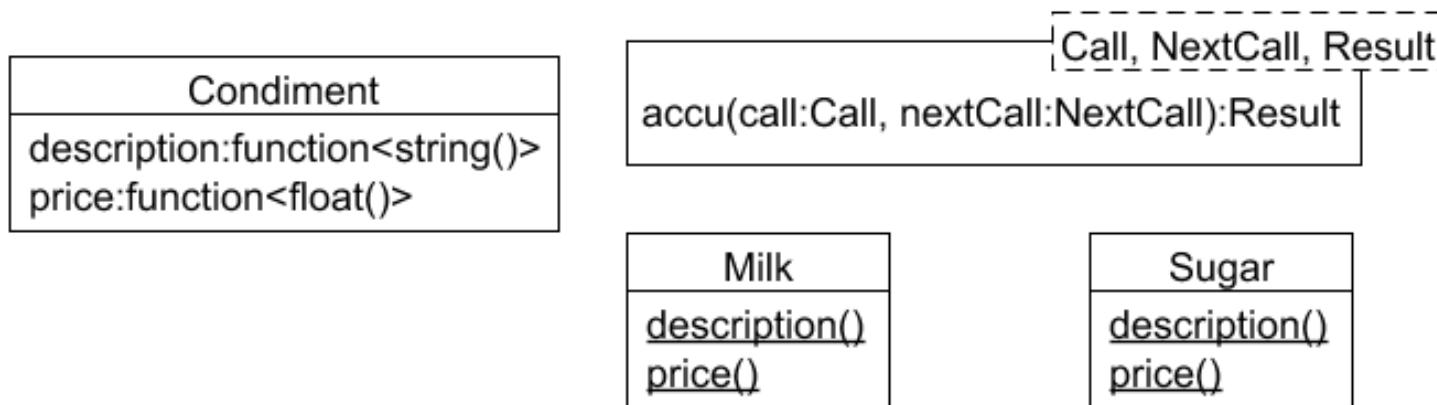
Strategy

CaffeineBeverage	Recipes
CaffeineBeverage(function<int()> amountWaterMI, function<void()> brew) prepare()	<u>amountWaterMI(int):int</u> <u>brewCofffee()</u> <u>brewTea()</u>

CaffeineBeverage(`std::function<int()> amountWaterMI`, `std::function<void()> brew`)

Where is the Pattern?

Chain



```
condiments.description = [=]{ return accu(condiment.description, condiments.description); };
```

Where is the Pattern? Command

CoffeeMachine	CaffeineBeverage
request(function<void()>)	prepare()
start()	

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

Jeff Atwood

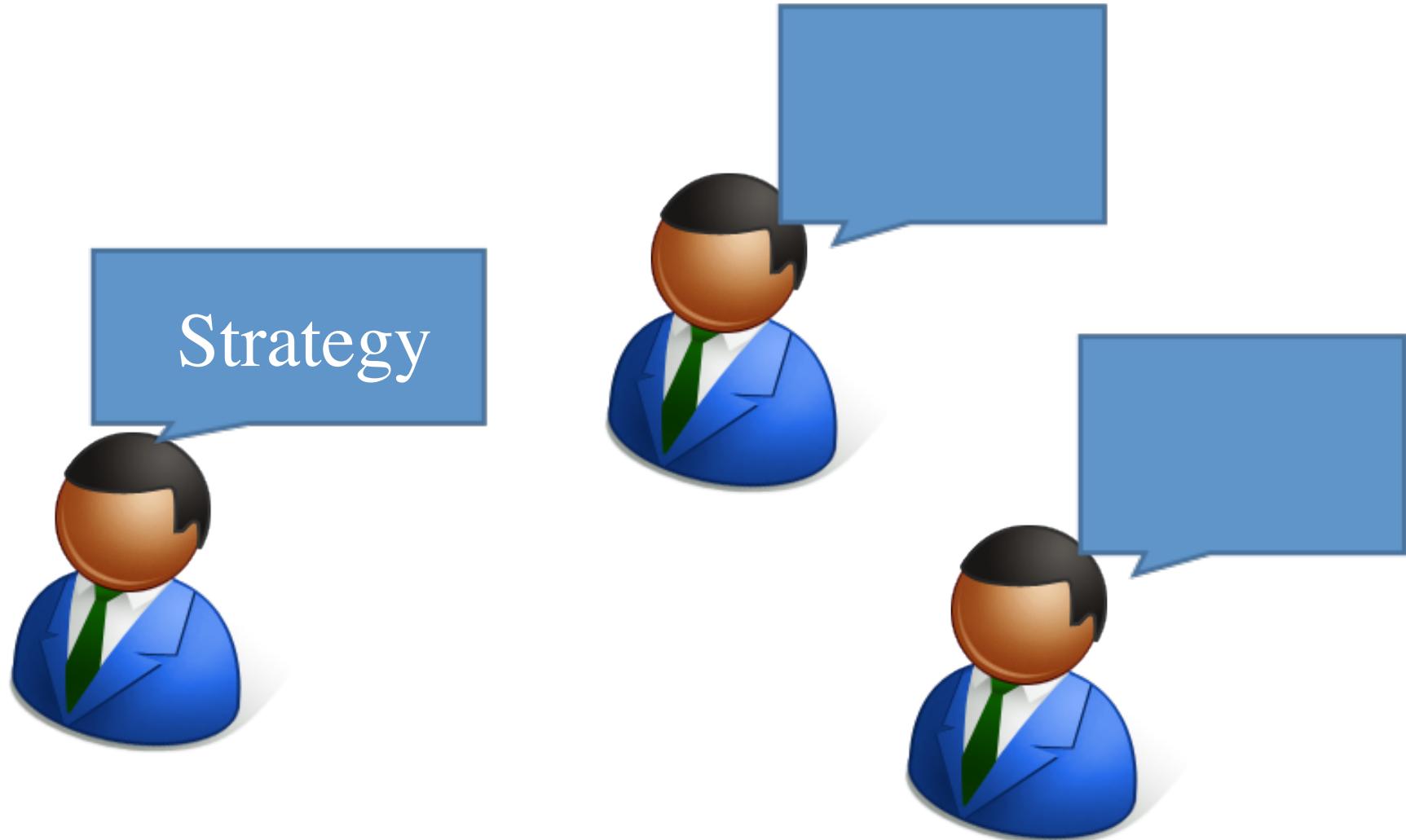
Design patterns are a form of complexity

<http://www.codinghorror.com/blog/2007/07/rethinking-design-patterns.html>

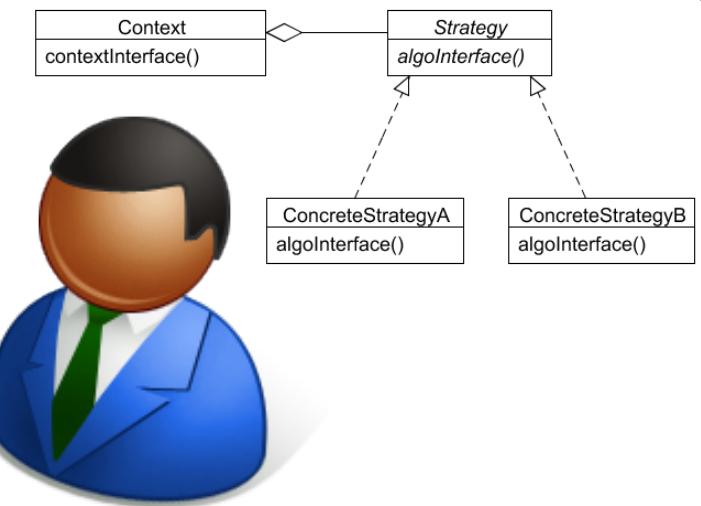
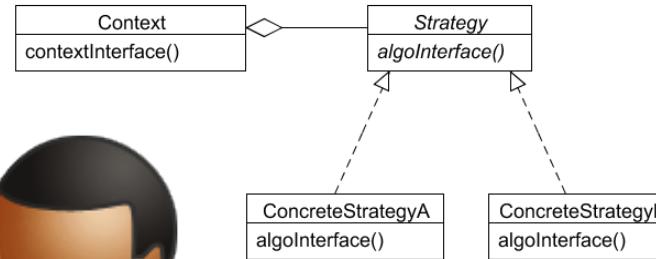
Patterns for Communication

Patterns for Communication

Solving Problems

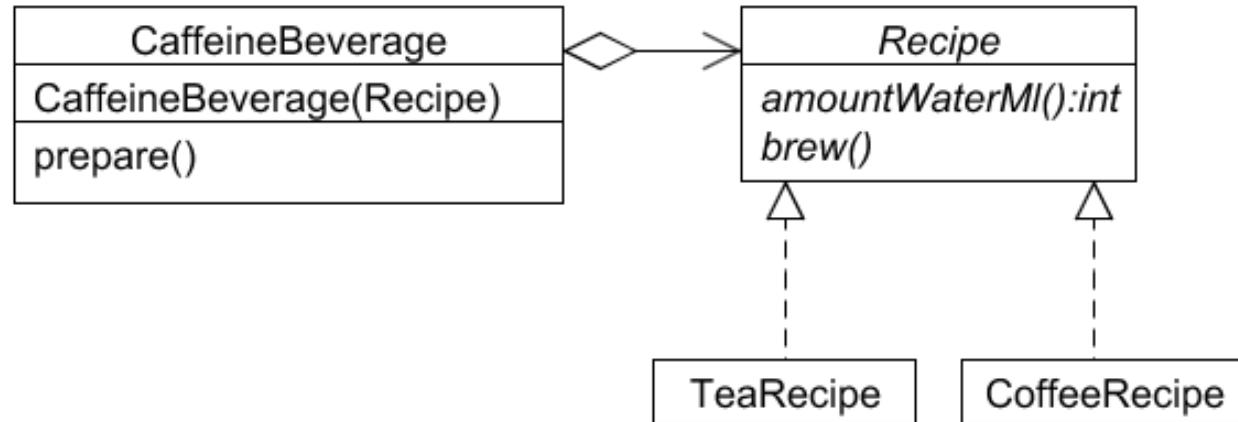


Patterns for Communication Solving Problems

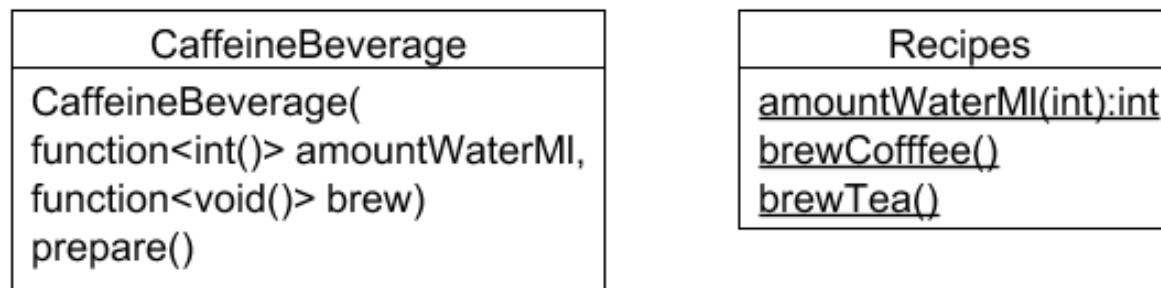


Patterns for Communication Solving Problems

- Classic

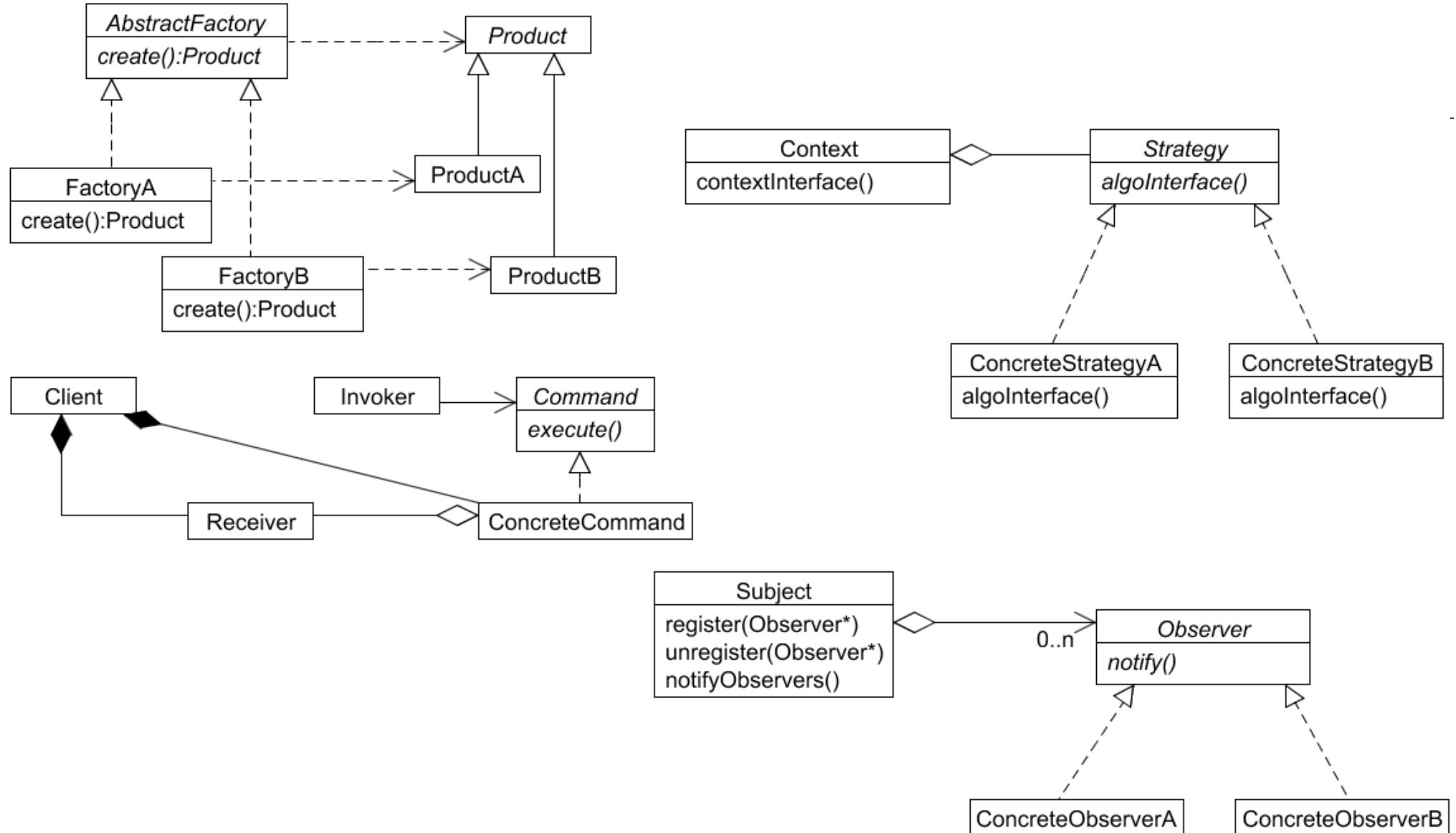


- C++11



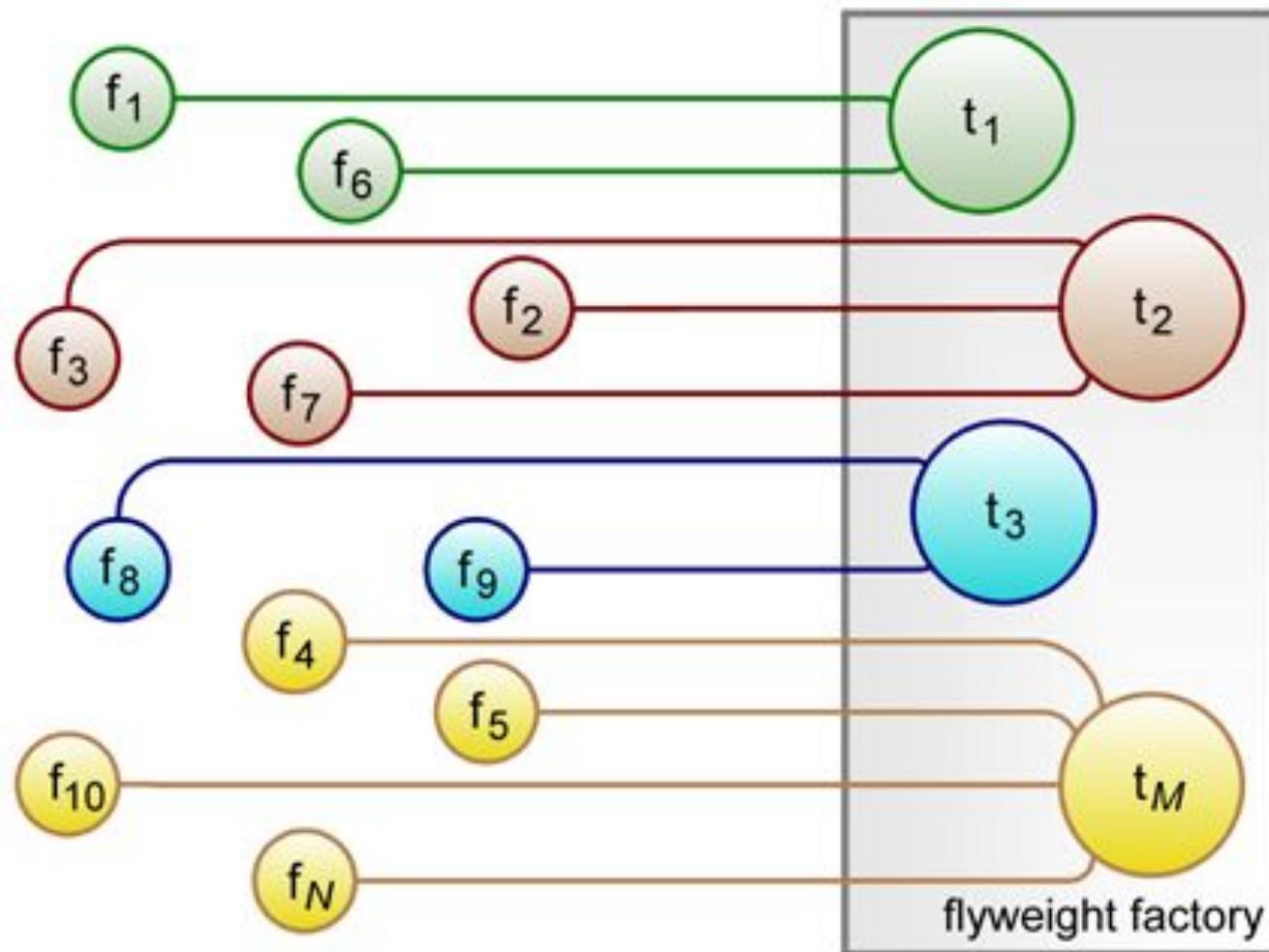
Patterns for Communication

Implementation is a detail

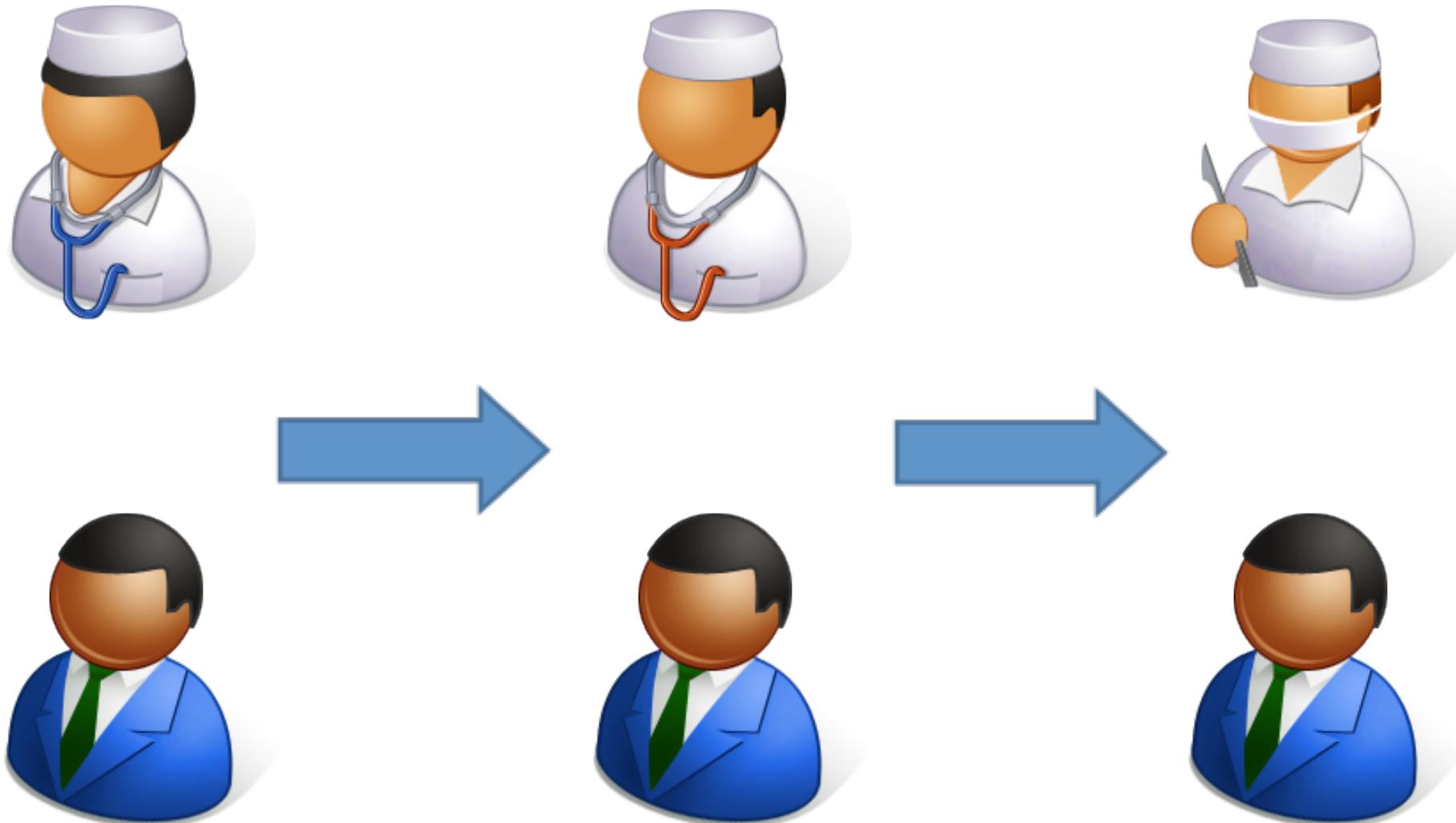


Patterns for Communication

Flyweight

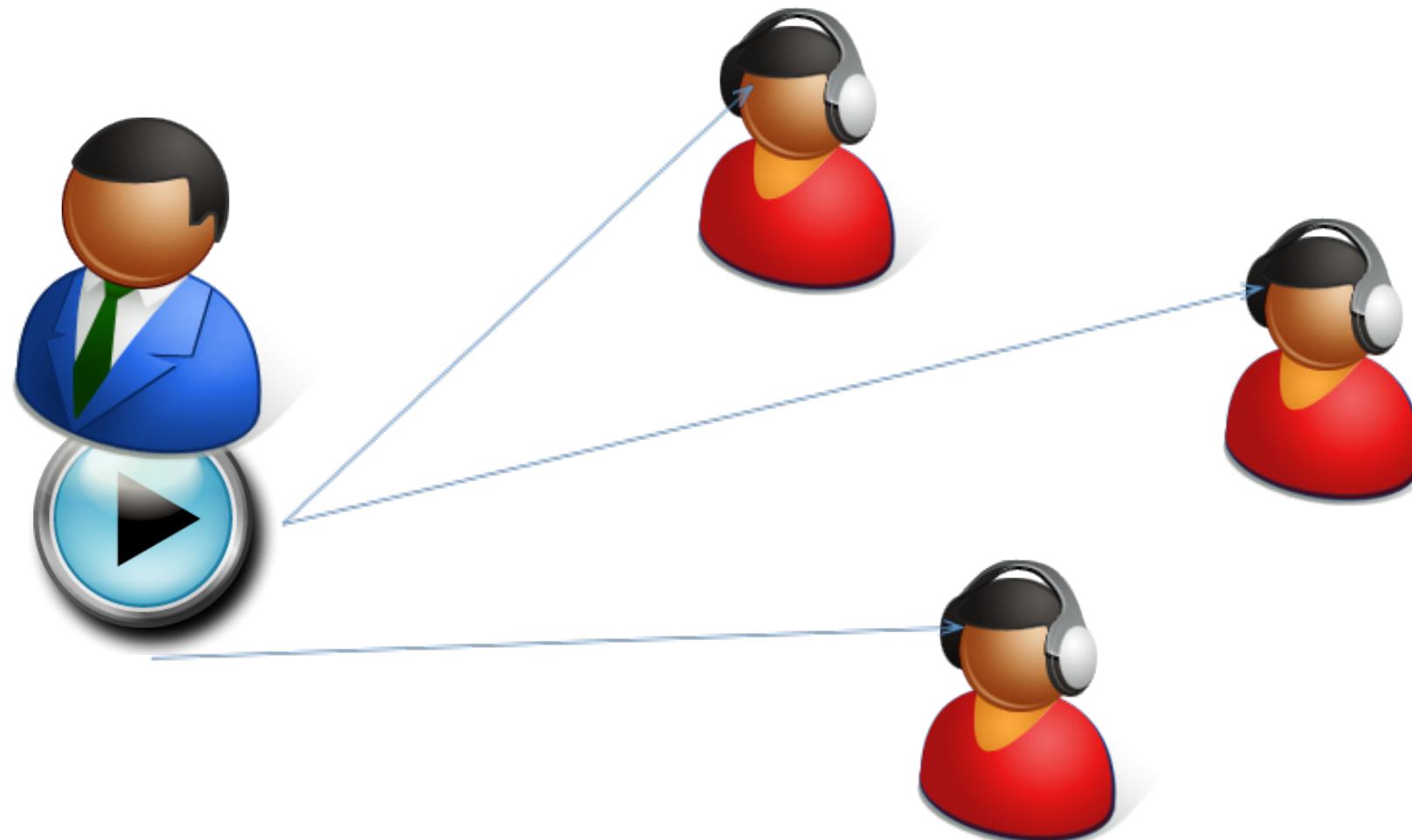


Patterns for Communication Chain



Patterns for Communication

Observer



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 → ...

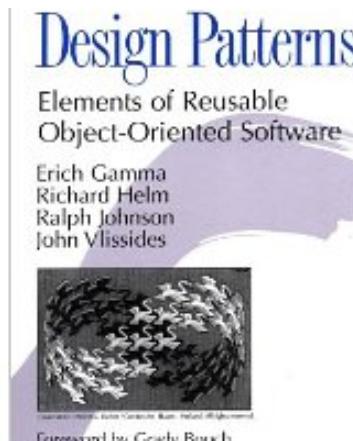
Pattern Lifecycle

- Discovered
- Published
- Test of time
- Adopted by language/library
- Disappears

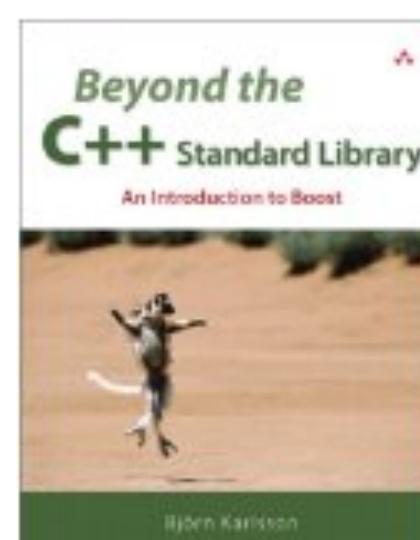
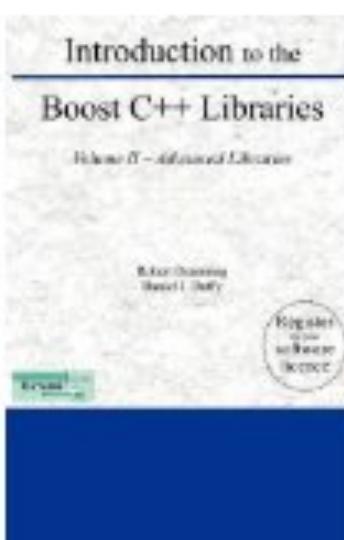
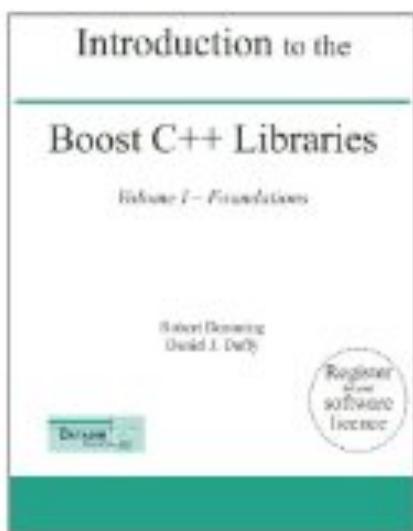
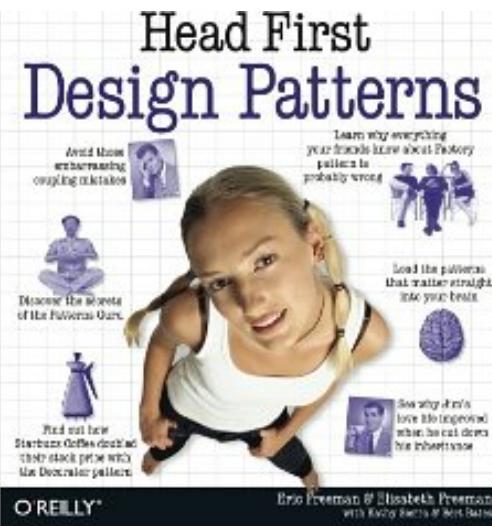
PPG

- Pattern
- Preservation
- Group

Books



ADONIS/WESLEY PROFESSIONAL COMPUTING SERIES



Thank you for your attention