
Chain of responsibility



Chain of responsibility

■ Známý jako

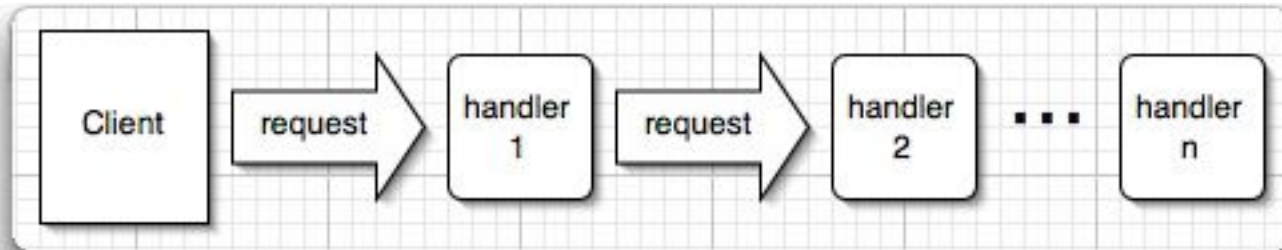
- český překlad – řetěz(ec) odpovědnosti

■ Účel

- umožnit zasílání požadavků (zpráv) neznámým příjemcům
- příjemci tvoří frontu → předávají si zprávu dokud ji někdo nezpracuje
- nechceme dopředu určit, který objekt obsluhuje správu
- umožňuje zrušení vazby mezi odesilatelem a příjemcem zprávy

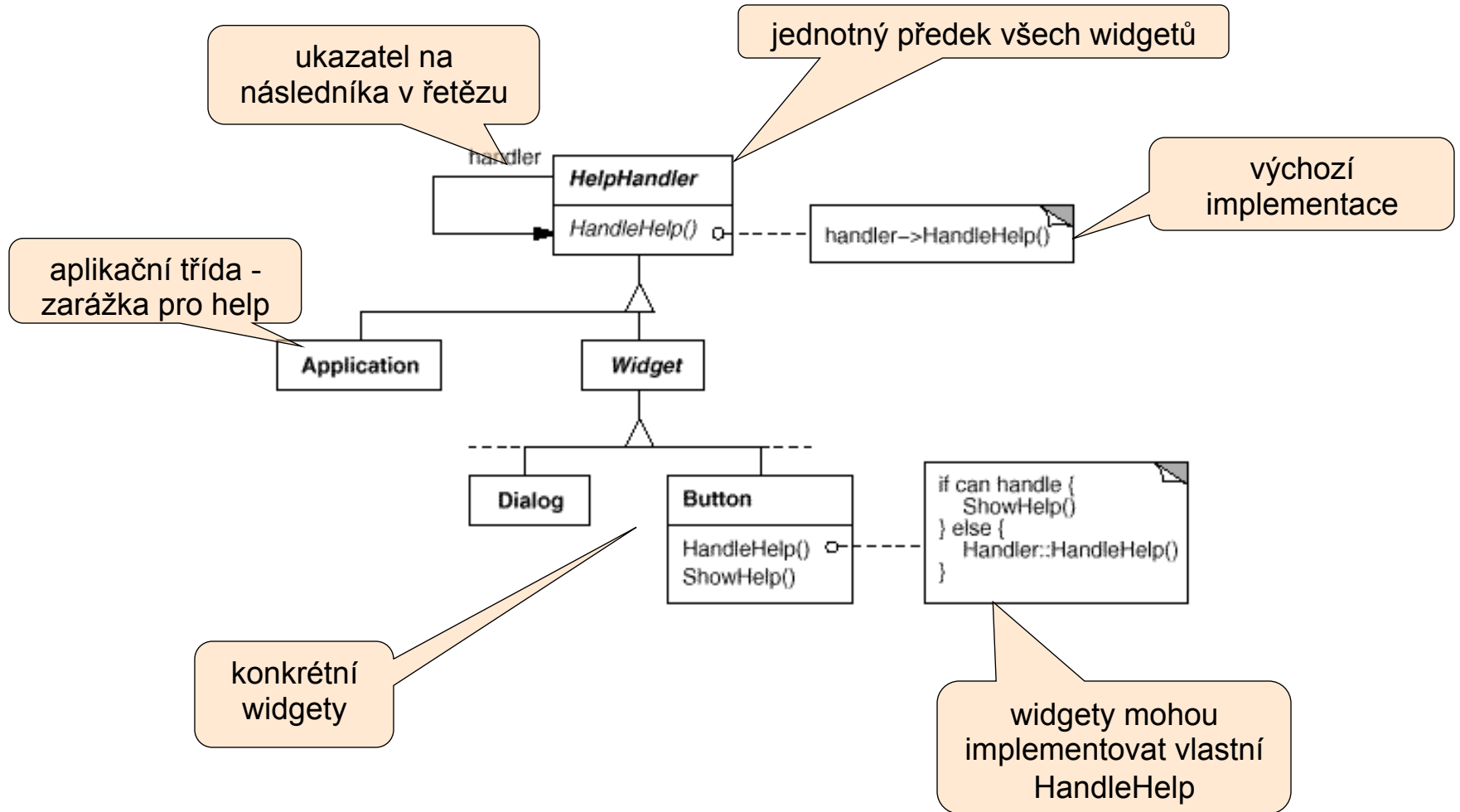
■ Motivace

- přidat k widgetu nápovědu
- chceme reakci v závislosti na kontextu ve kterém se widget nachází





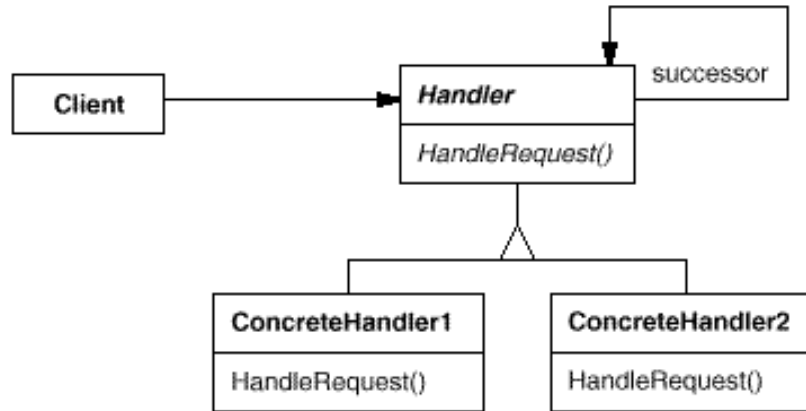
Chain of responsibility – motivace





Chain of responsibility – struktura

■ Struktura



■ Účastníci

■ Handler (HandleHelp)

- definuje rozhraní pro zpracování požadavků
- volitelně implementuje ukazatel na následníka

■ ConcreteHandler (PrintButton, PrintDialog)

- zachytává požadavek, který umí zpracovat
- může přistupovat na svého následníka

■ Client

- iniciuje požadavek předáním na objekt ConcreteHandler zapojený do řetězu



Chain of responsibility – použitelnost, výhody, nevýhody

■ Poutžitelnost

- chceme zaslat zprávu a nevíme kdo ji zpracuje nebo nás to nezajímá
 - důležitý je výsledek
- pokud více než jeden příjemce může přijmout zprávu a není a priori známo který

■ Výhody

- odděluje odesílatele od příjemců
- zjednodušuje odesílatele – neobsahuje reference na všechny možné příjemce
- možnost měnit řetěz dynamicky, za běhu programu

■ Nevýhody

- není zaručeno, že nějaký příjemce zprávu přijme
 - při implementaci ale typicky vytvoříme zarážku
 - zpracuje požadavek na nejobecnější úrovni
 - nebo zahlásí chybu (kulturně)



Chain of responsibility – implementace

■ Implementace

- dva způsoby implementace řetězu
 - definovat nové ukazatele (obvykle v Handleru, možno i v ConcreteHandleru)
 - použít existující ukazatele (Composite)

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

výchozí implementace zpracování požadavku – předej peška dál



Chain of responsibility – příklad

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}
}
```

pomocná metoda, test na existenci nápovědy na konkrétním widgetu

Komentář:

Implementace abstraktního předka všech tříd využívajících kontextovou nápovědu.

HelpHandler definuje ukazatele na následníka v řetězu.

Pro každé téma nápovědy definujeme jednoznačný identifikátor. Jestliže pro daný widget neexistuje nápověda, použije se výchozí hodnota *NO_HELP_TOPIC*.

Použití konkrétních hodnot uvidíme dále.



Chain of responsibility – příklad

■ Abstraktní předek Widgetů

```
class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
    Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
    _parent = w;
}
```

Abstraktní předek všech widgetů. Přidává navíc ukazatel na rodiče, tj. prvek, ve kterém je umístěn.

■ Button

```
class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget operations that Button overrides...
};
```

```
Button::Button (Widget* h, Topic t) :
    Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Tlačítko je vždy součástí jiného widgetu.



Chain of responsibility – příklad

■ Dialog

```
class Dialog : public Widget {
public:
    Dialog(Helper* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Widget operations that Dialog overrides...
    // ...
};

Dialog::Dialog (Helper* h, Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        Helper::HandleHelp();
    }
}
```

Komentář:

Třída Dialog je sama widget, ale navenek existuje samostatně – není součástí jiného widgetu.

Proto v konstruktoru předáváme Helper.



Chain of responsibility – příklad

■ Aplikace

```
class Application : public HelpHandler {  
public:  
    Application(Topic t) : HelpHandler(0, t) { }  
  
    virtual void HandleHelp();  
    // application-specific operations...  
};  
  
void Application::HandleHelp () {  
    // show a list of help topics  
}
```

- Třída **Application** není widget – dědí přímo od **HelpHandler**
- Je-li požadavek na nápovědu propagován až na tuto úroveň, může aplikace zobrazit obecnou nápovědu nebo např. seznam témat nápovědy.



Chain of responsibility – příklad

■ Vytvoření a propojení prvků

```
const Topic PRINT_TOPIC = 1;  
const Topic PAPER_ORIENTATION_TOPIC = 2;  
const Topic APPLICATION_TOPIC = 3;  
  
Application* application = new Application(APPLICATION_TOPIC);  
Dialog* dialog = new Dialog(application, PRINT_TOPIC);  
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

■ Vyvolání nápovědy ze strany klienta

```
button->HandleHelp();
```



Chain of responsibility – příklad II

■ Logování

- Chceme, aby událost mohli zachytit různé typy logů

Abstraktní předek
všech logů

```
abstract class Logger {  
    public static int ERR = 3;  
    public static int NOTICE = 5;  
    public static int DEBUG = 7;  
    protected int mask;  
  
    protected Logger next; // the next element in the chain of responsibility  
    public Logger setNext(Logger l) { next = l; return this; }  
  
    abstract public void message(String msg, int priority);  
}
```

Následující objekt
v chain

Obsluha události –
chybí default obsluha



Chain of responsibility – příklad II

```
class DebugLogger extends Logger{
    public DebugLogger(int mask) { this.mask = mask; }

    public void message(String msg, int priority) {
        if (priority <= mask)
            println("Writing to debug output: "+msg);
        if (next != null) next.message(msg, priority);
    }
}

class EMailLogger extends Logger{
    public EMailLogger(int mask) { this.mask = mask; }

    public void message(String msg, int priority) {
        if (priority <= mask)
            println("Sending via e-mail: "+msg);
        if (next != null) next.message(msg, priority);
    }
}

class StderrLogger extends Logger{
    public StderrLogger(int mask) { this.mask = mask; }

    public void message(String msg, int priority) {
        if (priority <= mask)
            println("Writing to stderr: "+msg);
        if (next != null) next.message(msg, priority);
    }
}
```

Konkrétní potomkové
abstraktní třídy Logger



Chain of responsibility – příklad II

■ Použití

vytvoření chain

```
class ChainOfResponsibilityExample{
    public static void main(String[] args) {
        // building the chain of responsibility
        Logger l = new DebugLogger(Logger.DEBUG).setNext(
            new EMailLogger(Logger.ERR).setNext(
                new StderrLogger(Logger.NOTICE)
            ));

        l.message("Entering function y.", Logger.DEBUG);
        // handled by DebugLogger
        l.message("Step1 completed.", Logger.NOTICE);
        // handled by Debug- and StderrLogger
        l.message("An error has occurred.", Logger.ERR);
        // handled by all three Logger
    }
}
```



Chain of responsibility – reprezentace správ

Jako reprezentovat požadavek klienta?

■ „zadrátované natvrdo“

- omezená pevná množina requestů
- bezpečné

■ Integer alebo stringová konstanta

- Flexibilnejší
- Odesílatel i příjemce se musí shodnout na kodování

■ Object Request

- Všechny requesty se zabalí do objektu se společným předkem
- Accessor function nebo RTTI
- Podtřídy zachytí jen požadavky, které je zajímají, ostatní pošlou dál
- NV Command

```
void ExtendedHandler::HandleRequest (Request* theRequest)
{
    switch (theRequest->GetKind()) {
        case Preview:
            doSomething();
            break;
        default:
            Handler::HandleRequest(theRequest);
    }
}
```



Chain of responsibility – související NV

■ Známé použití

- Grafické toolkity (Java AWT – nevhodné použití, **neujalo se**)
- Okenní systémy
 - běžně používáno pro zpracování událostí jako kliknutí myši, stisk klávesy
- Distribuované systémy
 - v řetězu (do kruhu) je zapojena množina serverů nabízejících určité služby
 - klient předá požadavek libovolnému serveru
 - servery si mezi sebou posílají požadavek, dokud jej některý nezpracuje
- Java Servlet Filter
 - Http request může zpracovat více filtrů



Chain of Responsibility – související NV

■ Související NV

- Command
 - přesně specifikuje příjemce požadavku
- Composite
 - je-li řetěz objektů využívaný součástí rozsáhlejší struktury, je tato struktura obvykle tvořena pomocí Composite vzoru
- Template method
 - využíván k řízení chování jednotlivých objektů ve struktuře
- Decorator
 - Request handler může použít decorator na změnu requestu při předávání