JgChain of Responsibility pattern

To understand the use of the chain of responsibility design pattern, think "chain of command." A request lands in Jane's inbox. Depending on the request and Jane's power or ability, she decides whether or not she can handle the request. If she can't handle the request, she passes it on to the next person in the chain, perhaps a manager. The manager either handles the request, or again passes it on.

The Design Patterns book suggests that chain of responsibility is applicable when more than one potential handler exists for a given request. The actual handler isn't known ahead of time, and the client need not care who handles the request. The pattern also allows for the command chain to change dynamically.

One conceptual example of the chain of responsibility pattern is the Java exception handling mechanism. Either a method handles a method, or it passes it up to the caller to handle.

Workflow-based systems are another area where the chain of responsibility pattern is particularly applicable. Expense reimbursement is one common example of such a workflow system.

An employee typically submits an expense report to his or her manager, looking for approval and subsequent reimbursement for business travel expenses. The manager (if in his or her office) often can approve the report immediately, as long as it doesn't exceed a certain amount and no other special circumstances exist. If the manager cannot approve the expense report, it moves along the chain to the next appropriate person. The next person might be a VP, or a peer manager if the original manager happens to be out of the office. Different rules and powers apply to the VP, and every once in a while, a big shot will have to get involved. In all cases, the person currently holding the expense report knows who the next person in line is.

Listing 1 shows the relevant code for an ExpenseReport class. This class tracks the amount, whether or not the expense involve international travel (a special case), and the person who ultimately handled (approved or rejected) the report.

Listing 2 presents the Approver class, the core of the chain of responsibility pattern. The Approver class represents the entity known in the pattern as the Handler. It's typically an abstract class. In this case, it could be represented as either; I've chosen to implement three specific handler subtypes: Manager, VicePresident, and CEO (see Listing 3). An AutoRejectHandler also exists; approvers of this type reject everything.

The client sends an ExpenseReport object to an Approver using the handle method. Code in the handle method determines whether to send the report on or to approve it. The report is sent on if the approver isn't allowed to handle it (too much money, or it represents international travel and they aren't cleared for that) or if the approver is out of the office.

Each of the Approver subclasses constrains the approval details. Managers can approve up to $5,000, and only some managers can approve international travel. Vice Presidents (VPs) can approve up to $100,000 and all international travel. CEOs can approve everything, unless they're out of the office, in which case the expense report automatically goes to an AutoRejectHandler (too bad for the employee!).

**Listing 1:** ExpenseReport.

public enum State {

initial, approved, rejected

}

public class ExpenseReport {

private int totalDollarAmount;

private boolean isInternationalTravel;

private State state = State.initial;

private Approver handler;

public int getTotalDollarAmount() {

return totalDollarAmount;

}

public void setTotalDollarAmount(int amount) {

totalDollarAmount = amount;

}

public boolean isInternationalTravel() {

return isInternationalTravel;

}

public void setIsInternationalTravel(boolean

isInternationalTravel) {

this.isInternationalTravel = isInternationalTravel;

}

public void reject() {

state = State.rejected;

}

public void approve(Approver approver) {

this.handler = approver;

state = State.approved;

}

public State state() {

return state;

}

**Listing 2 – approver class**

public Approver getHandler() {

return handler;

}

}

public class Approver {

private Approver nextApprover;

private int dollarLimit;

private boolean isOutOfOffice;

private boolean canApproveInternational;

private final String name;

public Approver(String name, boolean canApproveInternational,

int dollarLimit) {

this.name = name;

this.canApproveInternational = canApproveInternational;

this.dollarLimit = dollarLimit;

}

public String getName() {

return name;

}

public void handle(ExpenseReport report) {

if (!canApprove(report) || isOutOfOffice())

nextApprover.handle(report);

else

report.approve(this);

}

private boolean canApprove(ExpenseReport report) {

return report.getTotalDollarAmount() <=

getDollarLimit() &&

(canApproveInternational() ||

!report.isInternationalTravel());

}

public void setNextApprover(Approver approver) {

nextApprover = approver;

}

public boolean isOutOfOffice() {

return isOutOfOffice;

}

public void setOutOfOffice(boolean isOutOfOffice) {

this.isOutOfOffice = isOutOfOffice;

}

public int getDollarLimit() {

return dollarLimit;

}

public boolean canApproveInternational() {

return canApproveInternational;

}

public void setCanApproveInternational(boolean

canApproveInternational) { this.canApproveInternational =

canApproveInternational;

}

}

**Listing 3:** The Approver subtypes.

public class Manager extends Approver {

private static final int APPROVAL\_LIMIT = 5000;

public Manager(String name, boolean canApproveInternational) {

super(name, canApproveInternational, APPROVAL\_LIMIT);

}

}

public class VicePresident extends Approver {

private static final int APPROVAL\_LIMIT = 100000;

public VicePresident(String name) {

super(name, true, APPROVAL\_LIMIT);

}

}

public class CEO extends Approver {

private static final int APPROVAL\_LIMIT = Integer.MAX\_VALUE;

public CEO(String name) {

super(name, true, APPROVAL\_LIMIT);

setNextApprover(new AutoRejectHandler());

}

}

public class AutoRejectHandler extends Approver {

public AutoRejectHandler() {

super("auto-reject", false, 0);

}

@Override

public void handle(ExpenseReport report) {

report.reject();

}

}

Testing chain of Responsibility

import static org.junit.Assert.\*;

import org.junit.\*;

public class ChainTest {

private Manager manager;

private VicePresident vp;

private CEO ceo;

private ExpenseReport report;

private static final int SMALL\_AMOUNT = 100;

private static final int VP\_AMOUNT = 7000;

private static final int CEO\_AMOUNT = 207000;

@Before

public void amountPreconditions() {

assertTrue(SMALL\_AMOUNT <= manager.getDollarLimit());

assertTrue(manager.getDollarLimit() < VP\_AMOUNT &&

VP\_AMOUNT <= vp.getDollarLimit());

assertTrue(vp.getDollarLimit() < CEO\_AMOUNT);

}

@Before

public void createReport() {

report = new ExpenseReport();

}

@Before

public void createDefaultChain() {

manager = new Manager("Joe", false);

vp = new VicePresident("Jane");

ceo = new CEO("Zeus");

manager.setNextApprover(vp);

vp.setNextApprover(ceo);

}

@Test

public void approvedByManager() {

report.setTotalDollarAmount(SMALL\_AMOUNT);

manager.handle(report);

assertApprovedBy(manager);

}

@Test

public void approvedByVPWhenOverManagerLimit() {

report.setTotalDollarAmount(VP\_AMOUNT);

manager.handle(report);

assertApprovedBy(vp);

}

@Test

public void approvedByCEOWhenOverVPLimit() {

report.setTotalDollarAmount(CEO\_AMOUNT);

manager.handle(report);

assertApprovedBy(ceo);

}

@Test

public void approvedIntlByIntlManager() {

report.setTotalDollarAmount(SMALL\_AMOUNT);

report.setIsInternationalTravel(true);

manager.setCanApproveInternational(true);

manager.handle(report);

assertApprovedBy(manager);

}

@Test

public void approvedIntlByVPWhenManagerNotIntl() {

report.setTotalDollarAmount(SMALL\_AMOUNT);

report.setIsInternationalTravel(true);

assertFalse(manager.canApproveInternational());

manager.handle(report);

assertApprovedBy(vp);

}

@Test

public void approvedByVPWhenManagerOutOfOffice() {

report.setTotalDollarAmount(SMALL\_AMOUNT);

manager.setOutOfOffice(true);

manager.handle(report);

assertApprovedBy(vp);

}

@Test

public void rejectedWhenCEOOutOfOffice() {

report.setTotalDollarAmount(CEO\_AMOUNT);

ceo.setOutOfOffice(true);

manager.handle(report);

assertSame(State.rejected, report.state());

}

@Test

public void approvedByPeerWhenManagerOutOfOffice() {

report.setTotalDollarAmount(SMALL\_AMOUNT);

Manager peer = new Manager("Jim", false);

peer.setNextApprover(vp);

manager.setOutOfOffice(true);

manager.setNextApprover(peer);

manager.handle(report);

assertApprovedBy(peer);

}

private void assertApprovedBy(Approver approver) {

assertSame(State.approved, report.state());

assertSame(approver, report.getHandler());

}

}

The client working with the chain of responsibility doesn't know and doesn't really care which approver will finally handle the expense report. In this implementation, the employee submitting the report knows the initial approver—his or her manager. However, there's no reason that the client even needs to know this much information. A factory could produce an appropriate Approver implementation for the client.

