To use visitor pattern on swimmer, etc. create an accept method in waterbody.

To use visitor pattern to calculate entrainment possibility for different junctions:

Create a Junction class, which include node, wb[], index of flow in. wb[] not include boundaries.

GSJunction implement Junction, HOJunction implement Junction, RegularJunction implement Junction;

A junction can accept a salmon or a steelhead or a smelt dispatcher

SalmonDispatcher

SmeltDispatcher

SteelheadDispatcher

ParticleDispatchers: one container for all junctions and dispatchers. It should hold if statements for different type of junctions and dispatchers.

**interface** Dispatcher {

**Waterbody** dispatch(GSJunction gsj);

**Waterbody** dispatch(HOJunction hoj);

**Waterbody** dispatch(AJunction aj);

}

**interface** Junction {

**Waterbody** accept(Dispatcher dp); *// any junction have to provide accept().*

}

**class** GSJunction **implements** Junction {

**private** String name;

pubic final int node = //current internal node number : need to be changed if internal node# is changed;

private Waterbod[] wb;

private int[] inflowWaterbodyIndices;

// add non-physical barrier operations

**public** Junction(String name) {

**this**.name = name;

}

**public** String getName() {

**return** **this**.name;

}

**public** **Waterbody** accept(Dispatcher dp) {

*/\**

*\* accept(Dispatcher) in GSJunction implements*

*\* accept(Dispatcher) in Junction, so the call*

*\* to accept is bound at run time. This can be considered*

*\* the first dispatch. However, the decision to call*

*\* dispatch(GSJunction) (as opposed to dispatch(HOJunction) etc.) can be*

*\* made during compile time since 'this' is known at compile*

*\* time to be a GSJunction. Moreover, each implementation of*

*\* Dispatcher implements the dispatch(GSJunction), which is*

*\* another decision that is made at run time. This can be*

*\* considered the second dispatch.*

*\*/*

return dp.dispatch(**this**);

}

}

**class** HOJunction **implements** Junction {

**public** **Waterbody** accept(Dispatcher dp) {

return dp.dispatch(**this**);

}

}

**class** AJunction **implements** Junction {

**public** **Waterbody** accept(Dispatch dp) {

return dp.dispatch(**this**);

}

}

}

**class** DispatcherForSalmon **implements** Dispatcher {

private Particle \_particle;

public DispatcherForSalmon(Particle particle){}

**public** **Waterbody** dispatch(GSJunction gsj) {

// do the regression model from Russ

}

**public** **Waterbody** dispatch(HOJunction hoj) {

// do the regression model from Rebecca

}

**public** **Waterbody dispatch**(AJunction aj) {

// do Nicky code

}

}

**class** DispatcherForSmelt **implements** Dispatcher {

**public** **Waterbody** dispatch(GSJunction gsj) {

// do something with smelt route selection in GS

}

**public** **Waterbody** dispatch(HOJunction hoj) {

// do something with smelt route selection in head of old river

}

**public** **Waterbody dispatch**(AJunction aj) {

// do things Nicky did in ptm

}

}

**public** **class** DispatcherDemo {

**static** **public** **void** main(String[] args) {

// when hit a node create a junction if @gs create a GSJunction

// run should be specified for salmon or smelt or something else

GSJunction gsj = **new** GSJunction(“GS”, node, …);

If salmon{

wb = gsj.accept(**new** DispatchForSalmon());

}

If smelt{

wb = gsj.accept(**new** DispatchForSmlt());

}

}

Class particle{

……

protected void makeNodeDecision (){

Waterbody next\_wb = NodeDecisionMaker.makeDecision(this);

}

}

Class NodeDecisionMaker{

public Waterbody makeDecision(){}

public

……

}

interface Executor { public void execute(); }

class RobotAction {

String robot;

String action;

Executor executor;

}

then you have some setup:

list.add(new RobotAction("Humanoid", "Forward", new Executor() { public void execute() { humanoid.forward(5) }));

list.add(new RobotAction("Humanoid", "Backward", new Executor() { public void execute() { humanoid.backward(2) }));

list.add(new RobotAction("Snatcher", "Grab", new Executor() { public void execute() { snatcher.grab() }));

then your method becomes:

public void dispatch(String robot, String action) {

for (RobotAction robotAction : list) {

if (robot.equals(robotAction.robot) && action.equals(robotAction.action)) {

robotAction.execute();

}

}

}

interface IActionHandler{

void HandleAction(Action action);

}

class Humanoid: IActionHandler{

void HandleAction(Action action){

switch(action.ActionType){

ActionType.Forward: Forward();

......

}

}

...

}

class Catcher: IActionHandler{

void HandleAction(Action action){

switch(action.ActionType){

ActionType.Grab: Grab();

......

}

}

...

}

class MainActionReceiver{

ReceiceActionRequest(Action action){

GetActioner(action.Actioner).HandleAction(action);

}

IActionHander GetActioner(string actioner){

if (actioner == "Humanoid"){

return humanoidObject;

}

return catcherObject;

}

}

Excuse the semi-C# style - it's what I'm working in today.

If you wanted to avoid the switch statement in the HandleAction functions you could create ActionExecuter classes to actually execute the actions like this:

Interface IExecuter<T>{

bool CanExecute(Action action)

void Execute(T owner, Action action);

}

Then have

class ForwardExecuter<Humanoid>{

bool CanExecute{

return action.ActionType == forward;

}

Execute(Humaniod owner, Action action){

owner.Forward();

}

}

register the available ActionExecuters with the classes and then loop through the in the Handler looking for an Executer that can execute the action and then pass this to the executer.

class Humanoid: IActionHandler{

void HandleAction(Action action){

foreach (IExecuter in executers){

if (executer.CanExecute(action)){

executer.Execute(this, action);

}

}

}

...

}

That's quite possibly overkill for what you are doing, but you then have all your actions and action executers cleanly encapsulated in their own classes.

**class** Car **implements** CarElement {

CarElement[] elements;

**public** Car() {

*//create new Array of elements*

**this**.elements = **new** CarElement[] { **new** Wheel("front left"),

**new** Wheel("front right"), **new** Wheel("back left") ,

**new** Wheel("back right"), **new** Body(), **new** Engine() };

}

**public** **void** accept(CarElementVisitor visitor) {

**for**(CarElement elem : elements) {

elem.accept(visitor);

}

visitor.visit(**this**);

}

**interface** CarElementVisitor {

**void** visit(Wheel wheel);

**void** visit(Engine engine);

**void** visit(Body body);

**void** visit(Car car);

}

**interface** CarElement {

**void** accept(CarElementVisitor visitor); *// CarElements have to provide accept().*

}

**class** Wheel **implements** CarElement {

**private** String name;

**public** Wheel(String name) {

**this**.name = name;

}

**public** String getName() {

**return** **this**.name;

}

**public** **void** accept(CarElementVisitor visitor) {

*/\**

*\* accept(CarElementVisitor) in Wheel implements*

*\* accept(CarElementVisitor) in CarElement, so the call*

*\* to accept is bound at run time. This can be considered*

*\* the first dispatch. However, the decision to call*

*\* visit(Wheel) (as opposed to visit(Engine) etc.) can be*

*\* made during compile time since 'this' is known at compile*

*\* time to be a Wheel. Moreover, each implementation of*

*\* CarElementVisitor implements the visit(Wheel), which is*

*\* another decision that is made at run time. This can be*

*\* considered the second dispatch.*

*\*/*

visitor.visit(**this**);

}

}

**class** Engine **implements** CarElement {

**public** **void** accept(CarElementVisitor visitor) {

visitor.visit(**this**);

}

}

**class** Body **implements** CarElement {

**public** **void** accept(CarElementVisitor visitor) {

visitor.visit(**this**);

}

}

**class** Car **implements** CarElement {

CarElement[] elements;

**public** Car() {

*//create new Array of elements*

**this**.elements = **new** CarElement[] { **new** Wheel("front left"),

**new** Wheel("front right"), **new** Wheel("back left") ,

**new** Wheel("back right"), **new** Body(), **new** Engine() };

}

**public** **void** accept(CarElementVisitor visitor) {

**for**(CarElement elem : elements) {

elem.accept(visitor);

}

visitor.visit(**this**);

}

}

**class** CarElementPrintVisitor **implements** CarElementVisitor {

**public** **void** visit(Wheel wheel) {

System.out.println("Visiting " + wheel.getName() + " wheel");

}

**public** **void** visit(Engine engine) {

System.out.println("Visiting engine");

}

**public** **void** visit(Body body) {

System.out.println("Visiting body");

}

**public** **void** visit(Car car) {

System.out.println("Visiting car");

}

}

**class** CarElementDoVisitor **implements** CarElementVisitor {

**public** **void** visit(Wheel wheel) {

System.out.println("Kicking my " + wheel.getName() + " wheel");

}

**public** **void** visit(Engine engine) {

System.out.println("Starting my engine");

}

**public** **void** visit(Body body) {

System.out.println("Moving my body");

}

**public** **void** visit(Car car) {

System.out.println("Starting my car");

}

}

**public** **class** VisitorDemo {

**static** **public** **void** main(String[] args) {

Car car = **new** Car();

car.accept(**new** CarElementPrintVisitor());

car.accept(**new** CarElementDoVisitor());

}

}

**Note**: A more flexible approach to this pattern is to create a wrapper class implementing the interface defining the accept method. The wrapper contains a reference pointing to the CarElement which could be initialized through the constructor. This approach avoids having to implement an interface on each element. [see article Java Tip 98 article below]

[[edit](http://en.wikipedia.org/w/index.php?title=Visitor_pattern&action=edit&section=6)]**Output**

Visiting front left wheel

Visiting front right wheel

Visiting back left wheel

Visiting back right wheel

Visiting body

Visiting engine

Visiting car

Kicking my front left wheel

Kicking my front right wheel

Kicking my back left wheel

Kicking my back right wheel

Moving my body

Starting my engine

Starting my car