Basics of Java: Statements

Nathaniel Osgood
MIT 15.879

April 11, 2012

Recall: Methods

- Methods are “functions” associated with a class
- Methods can do either or both of
  - Return a value (doing computation as required)
    - Note that this value could be a reference to a value collection
  - Performing actions
    - Printing items
    - Displaying things
    - Changing the state of items
- Best practices
  - A method should do *one* of the above, not both!
    - Should either be a “query” (return values) or a “command” (perform action)
  - Methods should be *well named* (communicate intention)

Method Elements: 2 Pieces

- Header: Specifies what
  - “Types”
    - Expects as “arguments” (formal parameters – values given to the function)
    - Returned Value
  - “Exceptions” that can be thrown
- Body: Describes the algorithm (code) to do the work (the “implementation”)
- Best Practice: A well-documented (specified) method is has a *contract* that specifies *what* it does
  - we give it parameters with certain characteristics, and it does a certain job for us
  - We don’t have to worry about details of *how* it works
- The name & “header” of the function collectively give much hints as to the contract

Method Bodies & “Statements”

- Method bodies consist of
  - Comments (mostly ignored by “build”)
  - Variable Declarations
  - Statements (most involving “Expressions”)
- We discuss each of these below

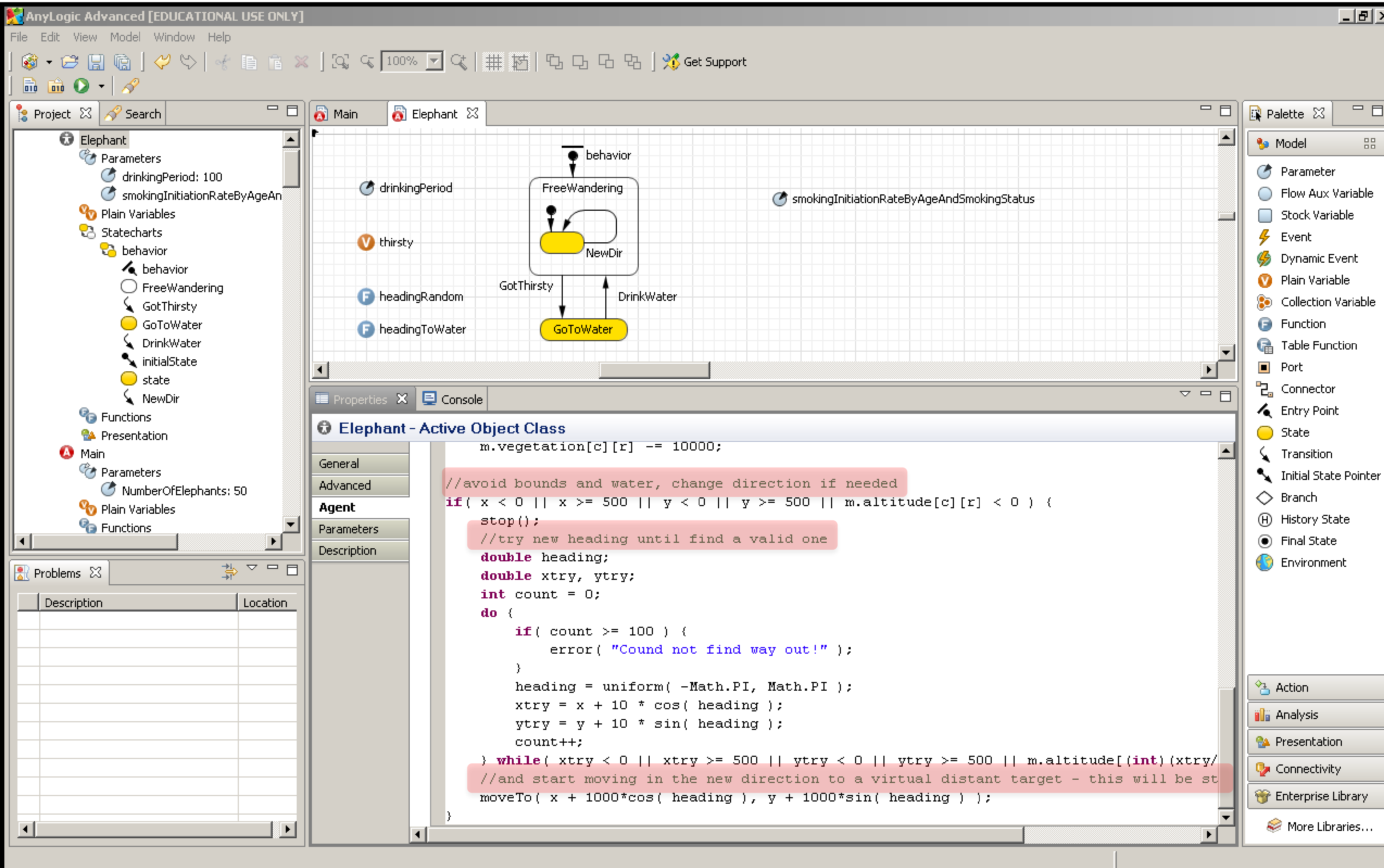
Comments

- Comments in Java are indicated in two different ways
  - Arbitrarily long: Begun with /* and ended with */
    - These can span many lines
  - Within a line: after a //
- Use comments to describe your intentions!

Rerouting Around Barriers (Boundaries & Water)

Poor Style – entire logic, conditions (checks on boundaries, whether water) & rerouting

Logic should all be in separate functions from this & from each other). Remove constants



Method Bodies & “Statements”

- Method bodies consist of
  - √ Comments (mostly ignored by “build”)
  - – Statements (most involving “Expressions”)

Java Statements

- In contrast to Java Expressions (which calculate a value), Java "statements" *do* something – they *effect some change (to "program state")*
- Statements are "commands" that, for example
  - Change the value of a variable or a field (this is an assignment expression)
  - Return a value (computed by an expression!) from the function
  - Call a method (call being in an expression)
  - Perform another sequence of statements a certain number of times (given by an expression) , or until a condition (given by an expression) is true
  - Based on some condition (given by an expression), perform one or another sequence of statements

When AnyLogic seeks *action* code (e.g. as a handler), we can give it a statement (or, typically a sequence of one or more statements).

Common Java Statements

- *if*
- *for*
- *while* or *do-while*
- *Try-Catch-Finally*
- *Throw* (Trigger) exception
- An expression (typically side-effecting – should be terminatede by a ";")
  - Assignment
  - Call to a function
- Composite statement block (multiple statements enclosed in a "{}")

For statements

Note variable declaration. This variable can then be used within the statement itself

- "For" statements "iterate", repeatedly executing some inner statement many times
- Several variants are available

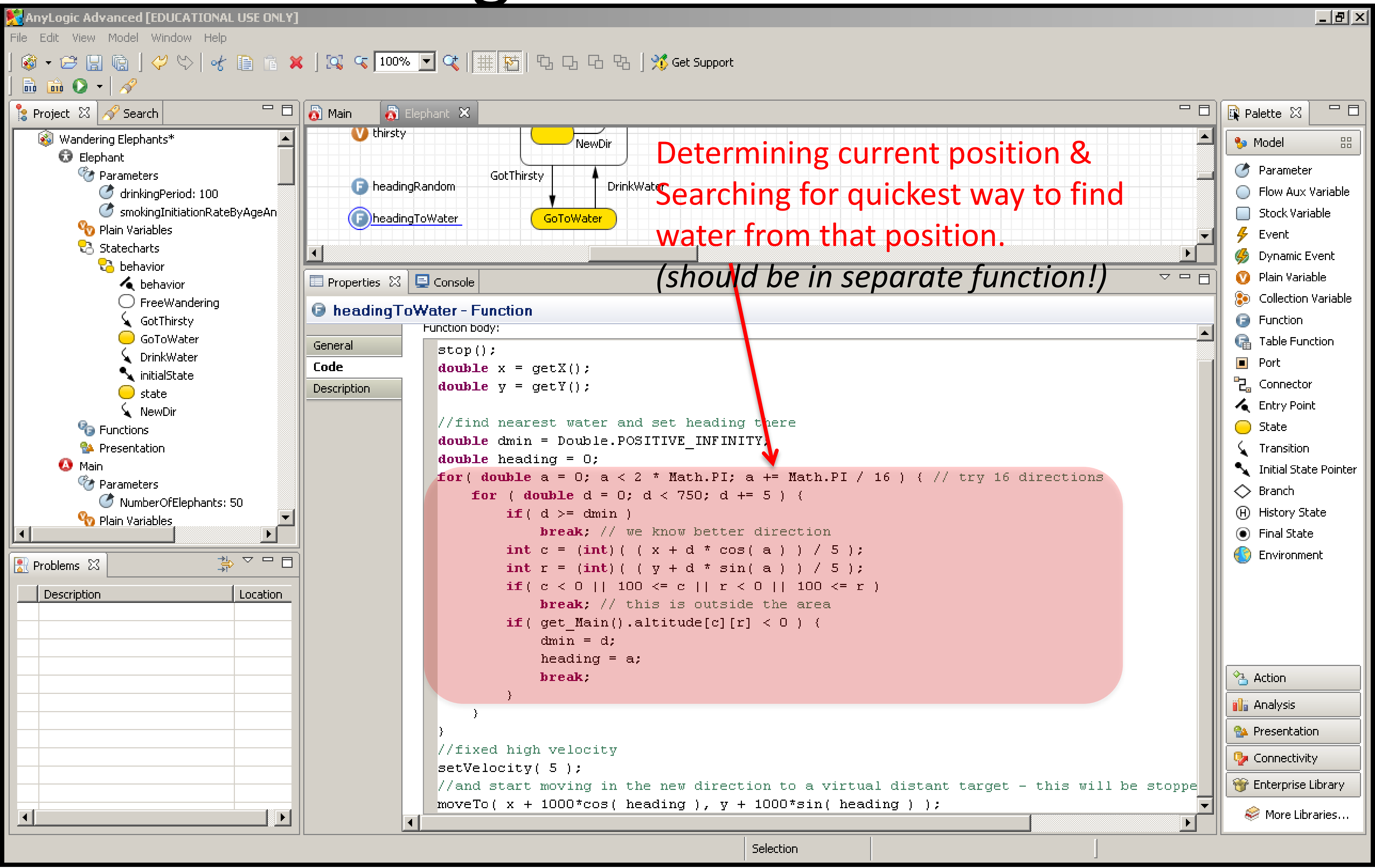
for (int i = 0; i < 100; i++)
    statement

Iterates over all integers from 0 to 99 (inclusive), with i bound to each integer in turn

for (Agent a : *collection*)
    statement

Iterates over all of the agents in Collection (with *a* bound to each element of collection in turn)

Heading Towards Resource



If Statements

- An *if* statement tests a condition expression ("predicate"), and – based on the result – either executes one statement or another (possibly empty) statement

This can be any expression that evaluates to a boolean (true or false) value

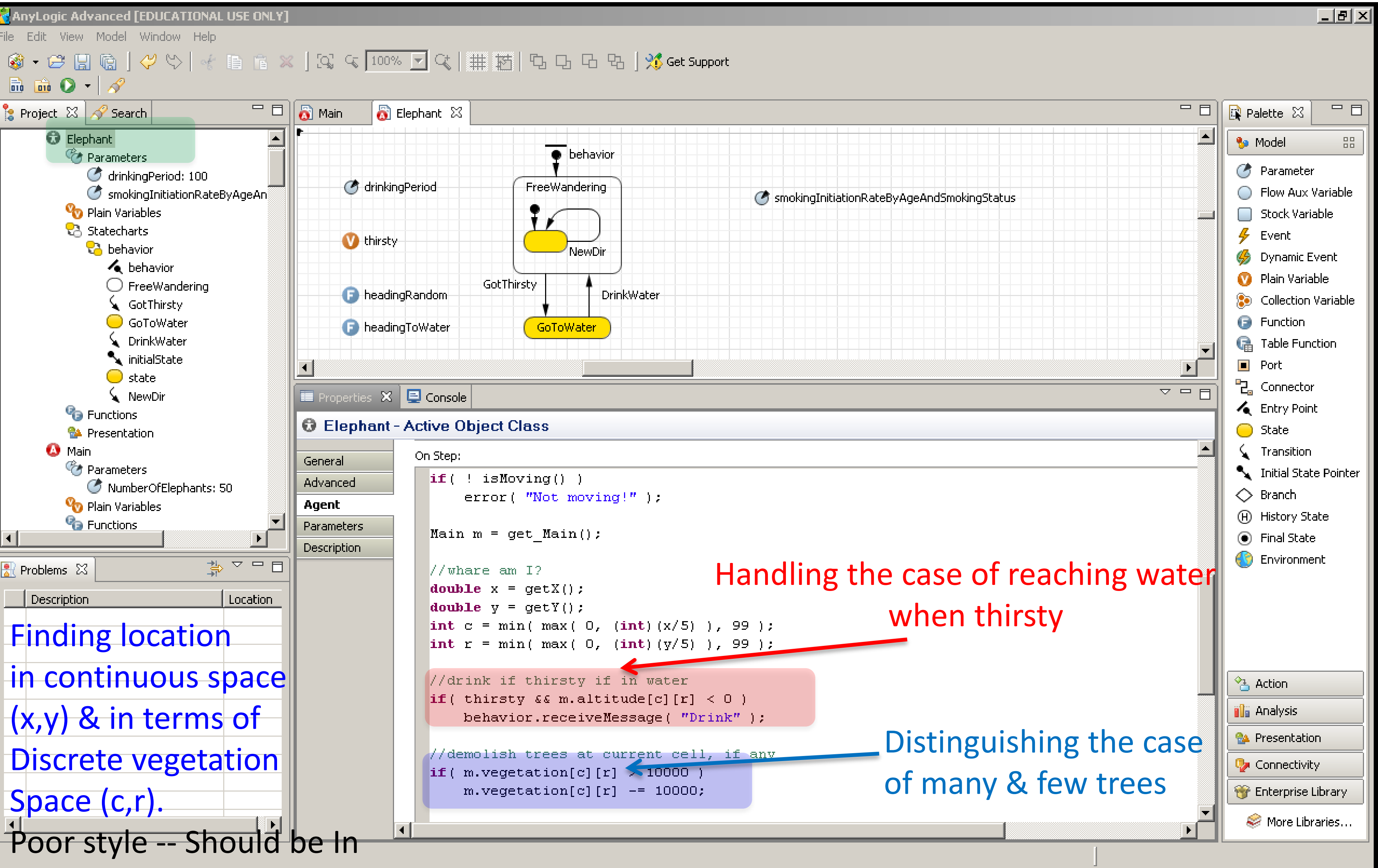
```
if (condition)
      consequent
else
   alternative
```

or

```
if (condition)
   alternative
```

"falls through" to later code if condition is false. This is like having an "empty" (blank) alternative

Handling of Movement Logic



Handling the case of reaching water when thirsty

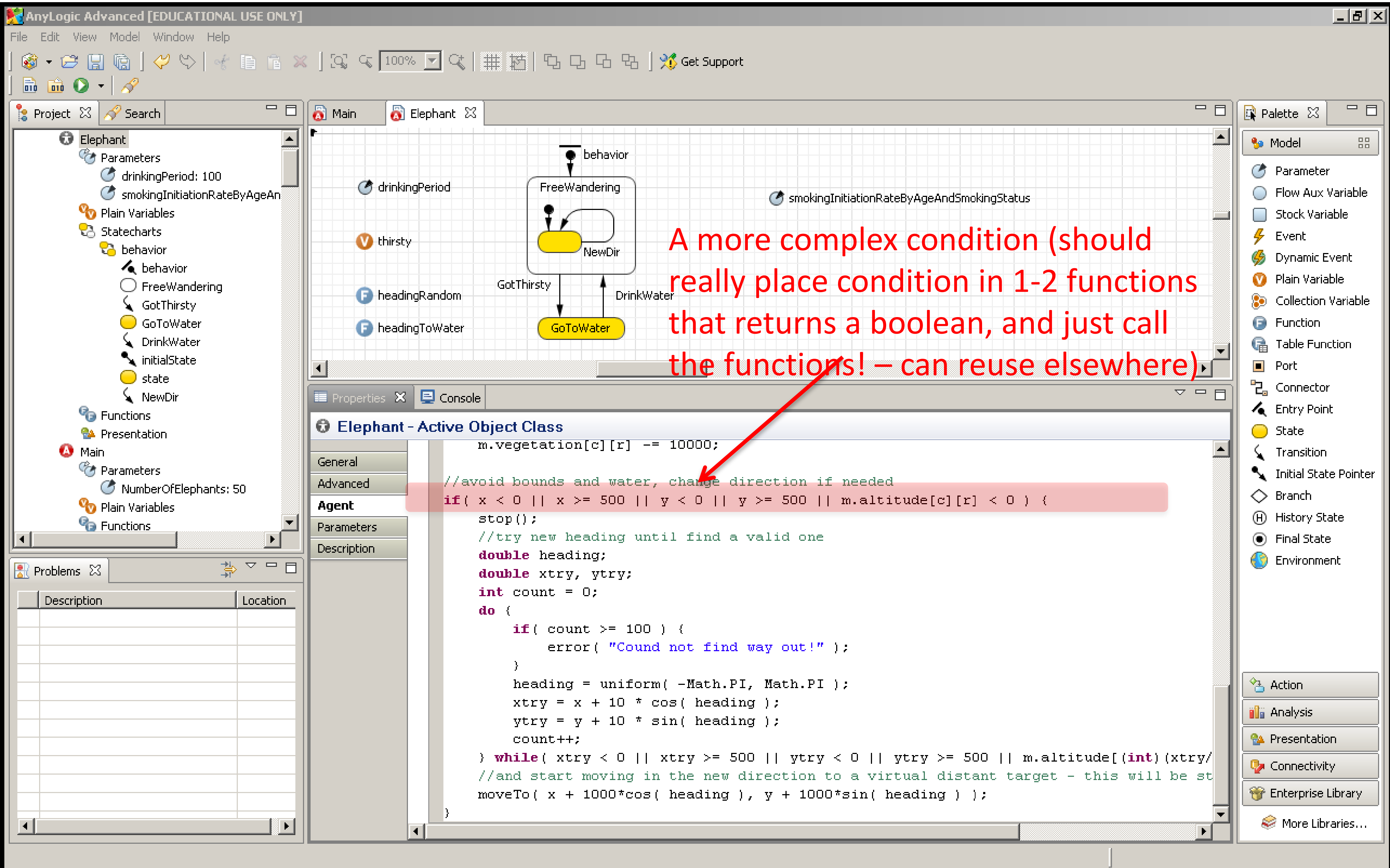
Distinguishing the case of many & few trees

Finding location in continuous space (x,y) & in terms of Discrete vegetation Space (c,r).

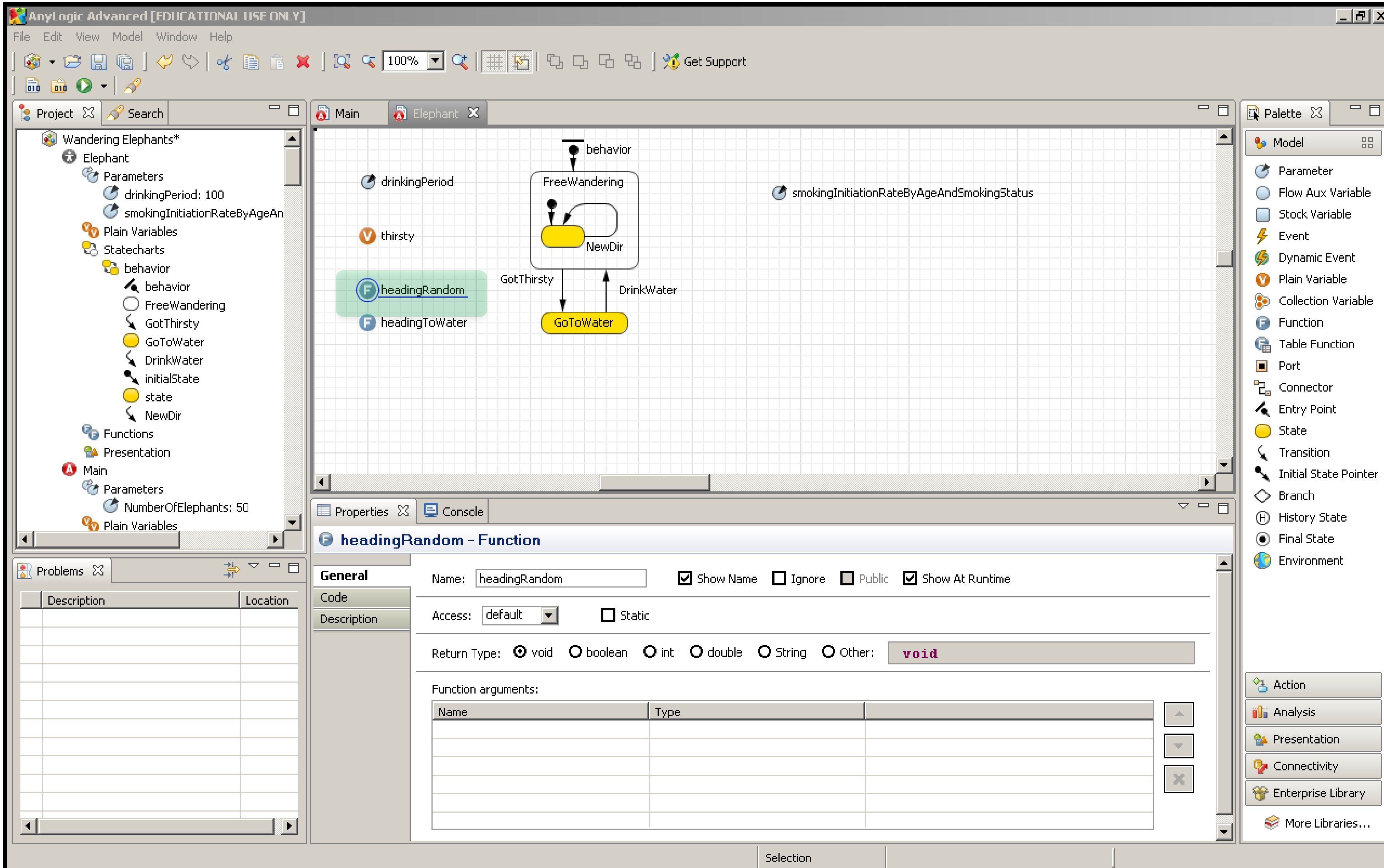
Poor style -- Should be In separate function

Rerouting Around Barriers (Boundaries & Water)

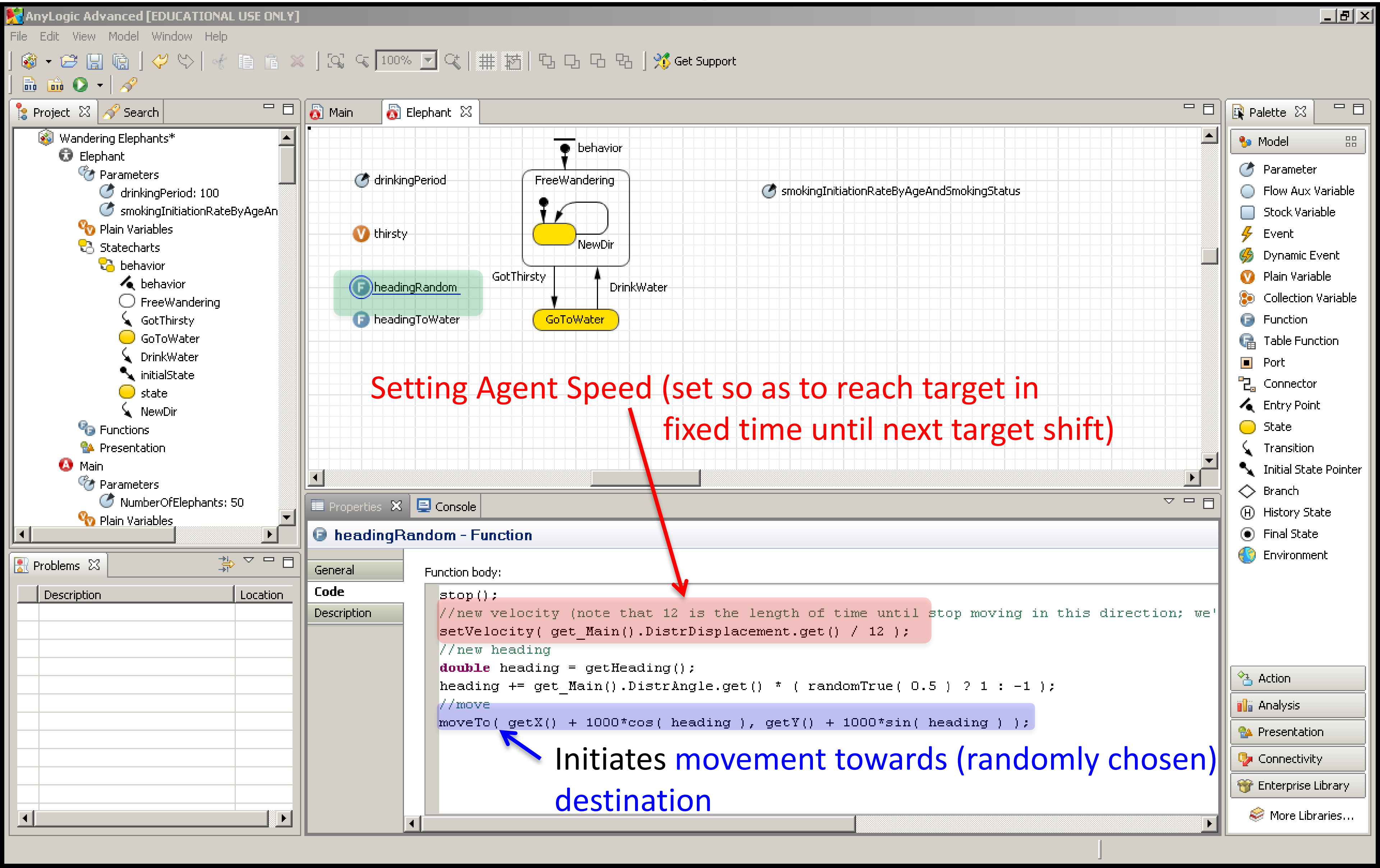
Poor Style – entire logic, conditions (checks on boundaries, whether water) & rerouting Logic should all be in separate functions from this & from each other). Remove constants



New Direction Change Function Info



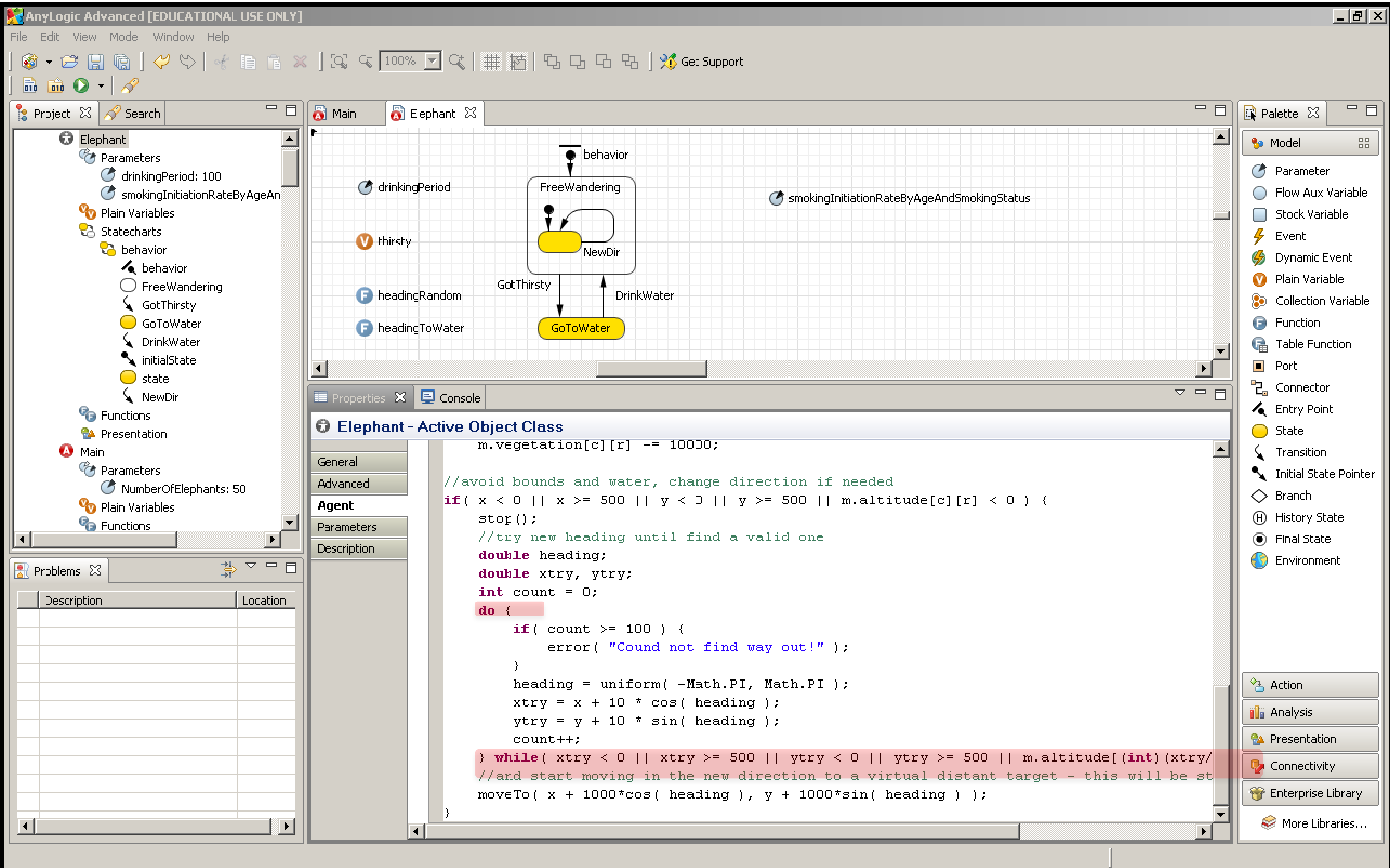
New Direction Change: Function "Body"



“While”/“Do while” loop

- Executes a statement as long as some condition is true
- The classic “while” loop has the test at the beginning
- The “do while” has the test at the end of the loop

While loops

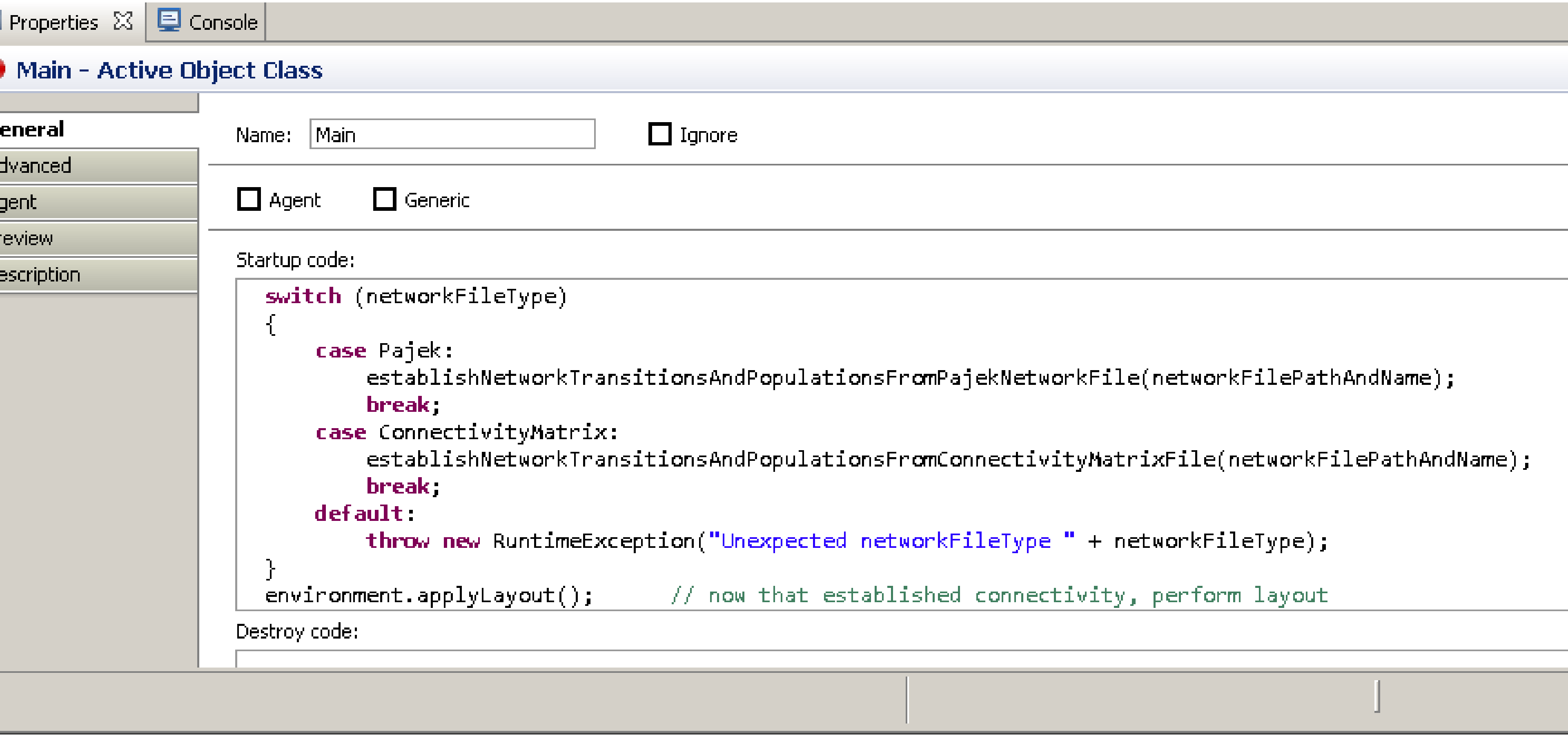


AnyLogic Advanced [EDUCATIONAL USE ONLY]
File Edit View Model Window Help
Get Support
Project
Search
Elephant
Parameters
drinkingPeriod: 100
smokingInitiationRateByAgeAn
Plain Variables
Statecharts
behavior
behavior
FreeWandering
GotThirsty
GoToWater
DrinkWater
initialState
state
NewDir
Functions
Presentation
Main
Parameters
NumberOfElephants: 50
Plain Variables
Functions
Problems
Description
Location
Main
Elephant
behavior
drinkingPeriod
FreeWandering
NewDir
smokingInitiationRateByAgeAndSmokingStatus
thirsty
GotThirsty
DrinkWater
headingRandom
headingToWater
GoToWater
Properties
Console
Elephant - Active Object Class
General
Advanced
Agent
Parameters
Description
m.vegetation[c][r] -= 10000;
//avoid bounds and water, change direction if needed
if(x < 0 || x >= 500 || y < 0 || y >= 500 || m.altitude[c][r] < 0) {
stop();
//try new heading until find a valid one
double heading;
double xtry, ytry;
int count = 0;
do {
if(count >= 100) {
error("Cound not find way out!");
}
heading = uniform(-Math.PI, Math.PI);
xtry = x + 10 * cos(heading);
ytry = y + 10 * sin(heading);
count++;
} while(xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[(int)(xtry/
//and start moving in the new direction to a virtual distant target - this will be st
moveTo(x + 1000*cos(heading), y + 1000*sin(heading));
}
Palette
Model
Parameter
Flow Aux Variable
Stock Variable
Event
Dynamic Event
Plain Variable
Collection Variable
Function
Table Function
Port
Connector
Entry Point
State
Transition
Initial State Pointer
Branch
History State
Final State
Environment
Action
Analysis
Presentation
Connectivity
Enterprise Library
More Libraries...

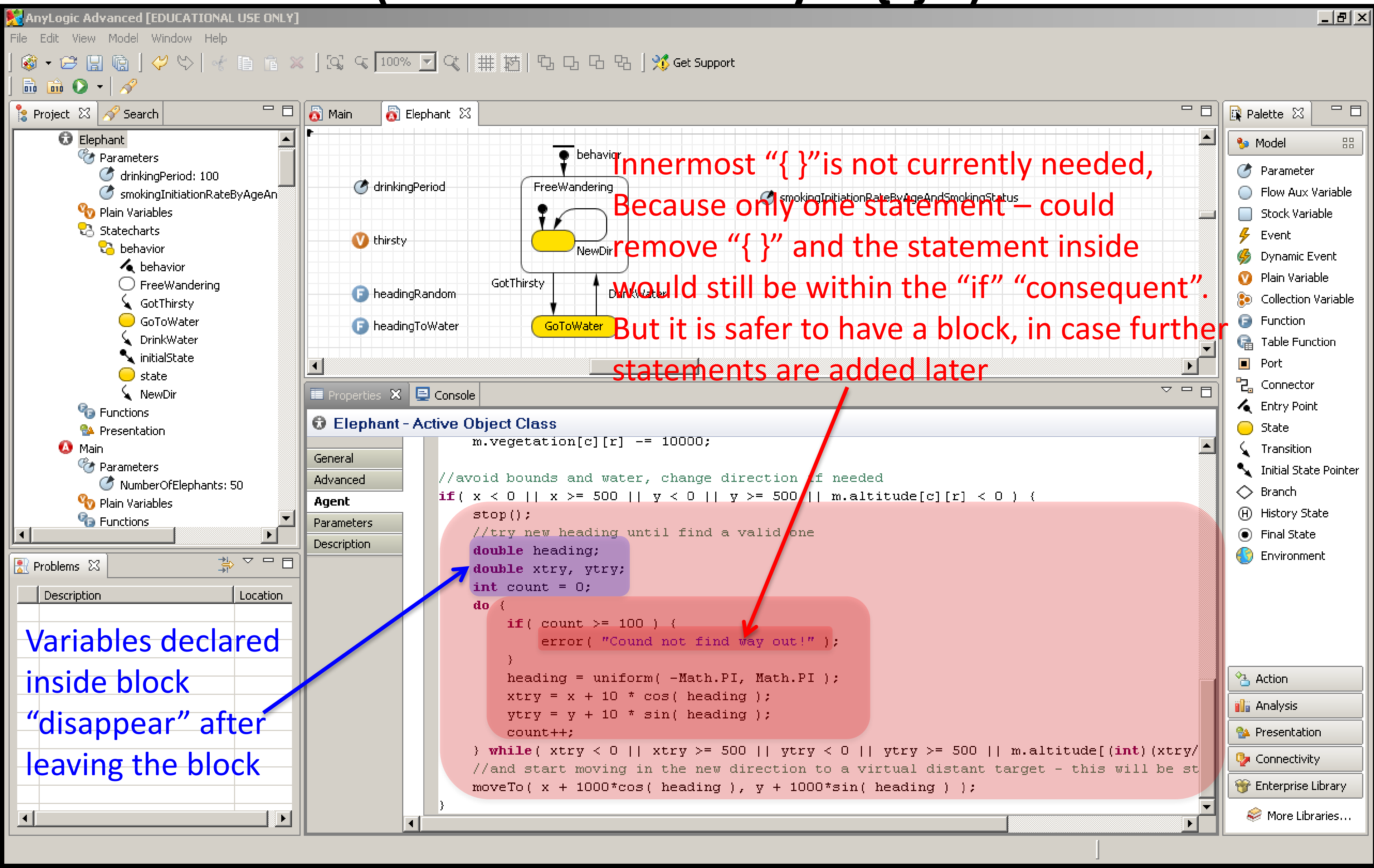
Switch/Case

- A “Switch” statement and its associated “case” clauses are a form of conditional somewhat like a multi-way “if” statement
- Contrast:
  - **If statement**: Is provided with a *boolean* value, and has one clause for the case where this is *true*, and (optionally) another for case where it is *false*
  - **Switch statement:** This is provided with a more general value (int, Enum, char, short, byte, character, in Java 7 a String), and has an arbitrary number of “case” clauses, each to handle different possible concrete values

Example Switch Statement



Composite Statements (“Blocks”) (Delineated by “{ }”)



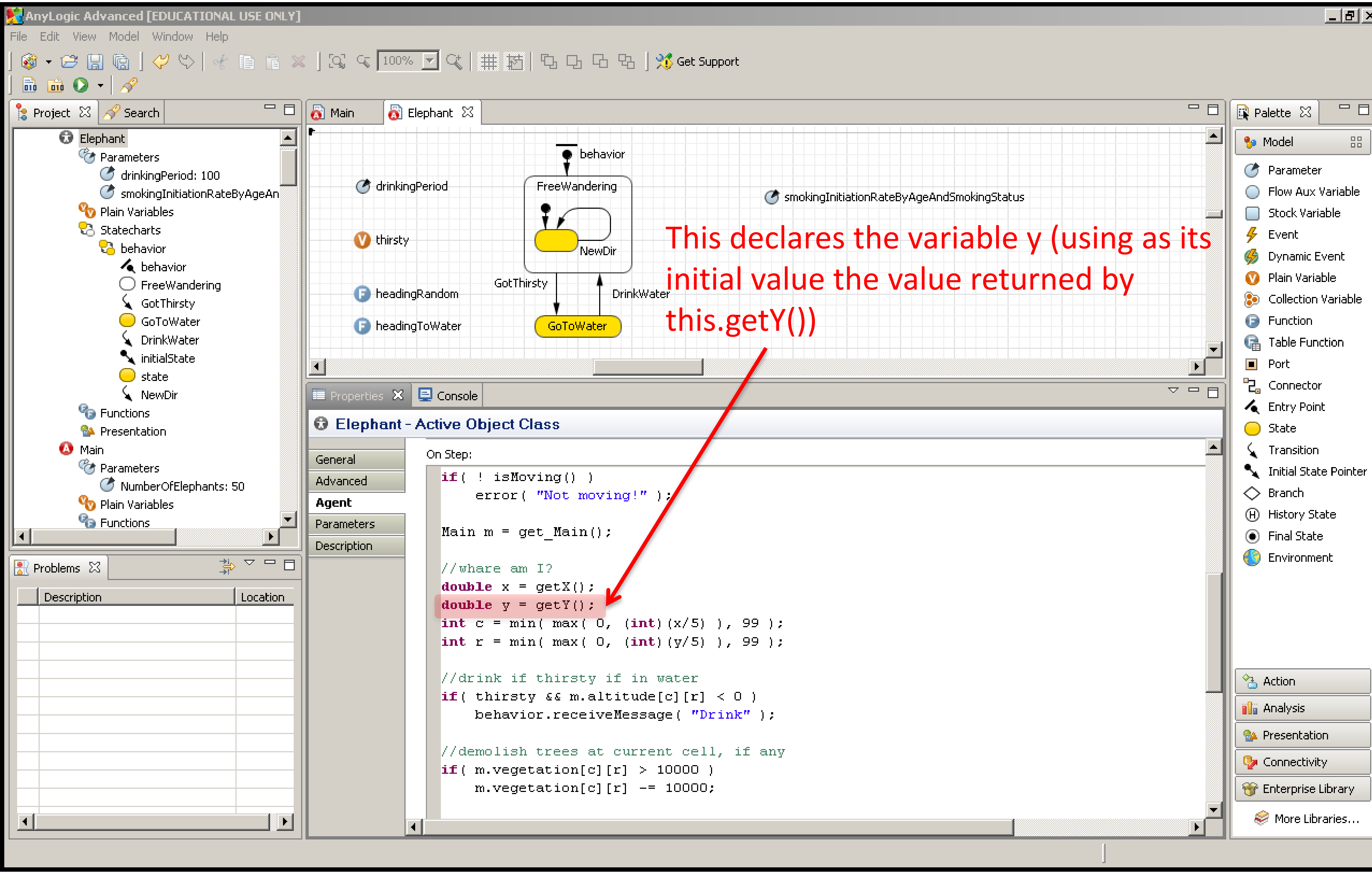
Composite Statements and Variables

- Variables can be declared within a composite statement
- The region of the variable's visibility (i.e. the scope of the variable) is from there to the end of the enclosing statement
- The entire body of a method is a compound statement (hence the "{ }" surrounding it)

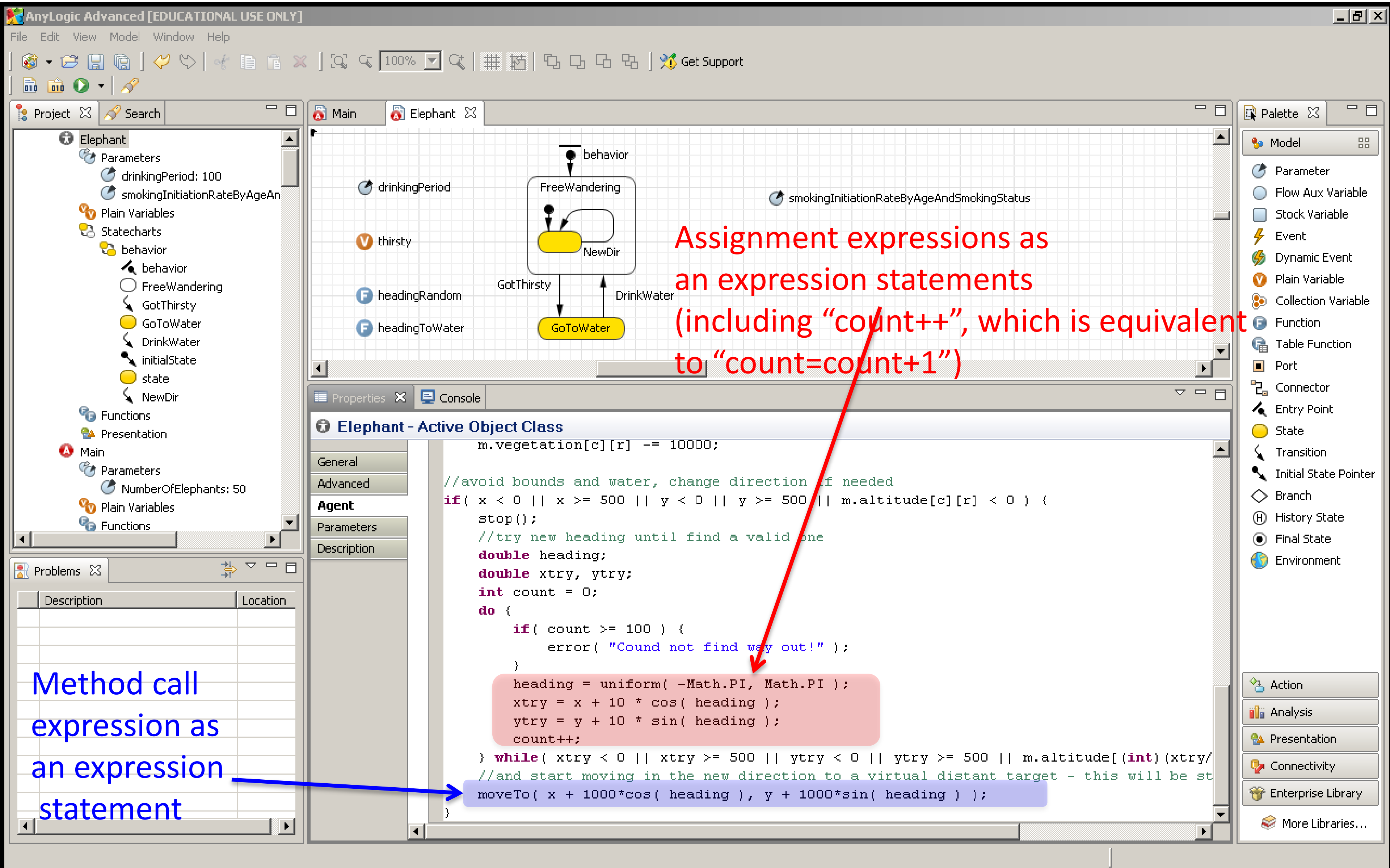
Recall: Variable Declarations

- Variables in Java are associated with "types" and can contain values
- When we "declare" a variable, we indicate its name & type – and possibly an initial value

Variable Declaration Statement



Expression Statements



Exceptions

- Not uncommonly, things may “go wrong” during execution of code
- We frequently want a way to signal that something has gone wrong
  - Stop normal processing of the code
  - Go “up” to a context where we know how to deal with (handle) the error
    - Up is defined in terms of the “call stack” – we wish to return to successive callers until one handles this condition
- To signal such exceptional conditions, java uses *Exceptions*
- Exceptions in Java are *thrown* where they occur & *caught* in “handlers” where we wish to handle them

Try-Catch Statements

```
try
  { try-block }
catch (ExceptionType1 e)
      { catch-block1 }
catch (ExceptionType2 e)
      { catch-block2 }

...

catch (ExceptionTypen e)
      { catch-blockn }
```

Exceptions thrown in this block (a compound statement) that are (most particularly) of this exception type are then handled by running this block

Exceptions thrown in the "try-block" that are of this exception type are then handled by running this block

Example Applications of "Try-Catch"