Tutorial:
Functions and Functional Abstraction

Nathaniel Osgood

CMPT 858

2-8-2011

Building the Model Right: Some Principles of Software Engineering

Technical guidelines

- Try to avoid needless complexity
- Use abstraction & encapsulation to simplify reasoning & development
- Name things carefully
- Design & code for transparency & modifiability
- Document & create self-documenting results where possible
- Consider designing for flexibility
- Use defensive programming
- Use type-checking to advantage
  - Subtyping (and sometimes subclassing) to capture commonality
  - For unit checking (where possible)

Process guidelines

- Use peer reviews to review
  - Code
  - Design
  - Tests
- Perform simple tests to verify functionality
- Keep careful track of experiments
- Use tools for version control & documentation & referent.integrity
- Do regular builds & system-wide "smoke" tests
- Integrate with others' work frequently & in small steps
- Use discovery of bugs to find weaknesses in the Q & A process

The Challenges of Complexity

- Complexity of software development is a major barrier to effective delivery of value
- Complexity leads to systems that are late, over budget, and of substandard quality
- Complexity has extensive impact in both human & technical spheres

Why Modularity?

- As a way of managing complexity: Allows decoupling of pieces of the system
  - "*Separation of Concerns*" in comprehension & reasoning
  - Example areas of benefit
    - Code creation
    - Modification
    - Testing
    - Review
    - Staff specialization
  - *Modularity allows 'divide and conquer' strategies to work*
- As a means to reuse

Abstraction: Key to Modularity

- Abstraction is the process of forgetting certain details in order to treat many particular circumstances as the same
- We can distinguish two key types of abstraction
  - *Abstraction by parameterization.* We seek generality by allowing the same mechanism to be adapted to many different contexts by providing it with information on that context
  - *Abstraction by specification.* We ignore the implementation details, and agree to treat as acceptable any implementation that adheres to the specification
  - [Liskov&Guttag 2001]

A Key Motivator for Abstraction: Risk of Change

- Abstraction by specification helps lessen the work required when we need to modify the program
- By choosing our abstractions *carefully*, we can gracefully handle anticipated changes
  - e.g. Choose abstracts that will hide the details of things that we anticipate changing frequently
  - When the changes occur, we only need to modify the implementations of those abstractions

Abstraction by Parameterization

- Major benefit: *Reuse*
  - Common needs identified
  - Elimination of need to separately
    - Develop
    - Test
    - Review
    - Debug
- Diverse forms
  - Functions: Formal parameters
  - Generics/Parameterized types
  - Cross cutting: Aspects (parameterized by pointcuts)

Types of Abstraction in Java

- Functional abstraction: Action performed on data
  - We use functions (in OO, *methods*) to provide some functionality while hiding the implementation details

  We are concentrating on this today
- Interface/Class-based abstraction: State & behaviour
  - We create "interfaces"/"classes" to capture behavioural similarity between sets of objects (e.g. agents)
  - The class provides a contract regarding
    - Nouns & adjectives: The characteristics (properties) of the objects, including state that changes over time
    - Verbs: How the objects do things (*methods*) or have things done to them

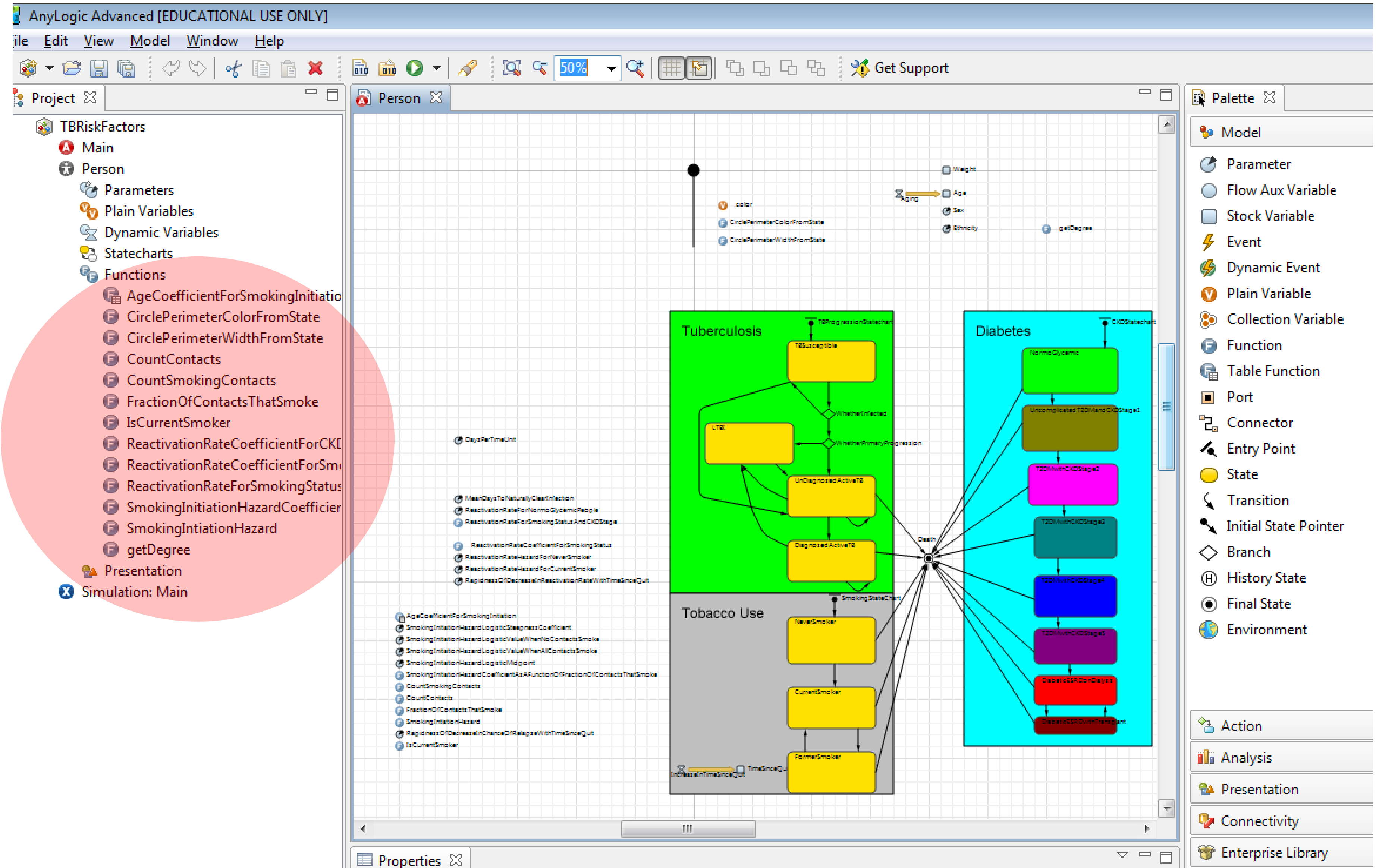
Functional Abstraction

- Functional abstraction provides methods to do some work (*what*) while hiding details of *how* this is done
- A method might
  - Compute a value (hiding the algorithm)
  - Test some condition (hiding all the details of exactly what is considered and how): e.g. ask if a person is susceptible
  - Perform some update on e.g. a person (e.g. infect a person, simulate the change of state resulting from a complex procedure, transmit infection to anther)
  - Return some representation (e.g. a string) of or information about a person in the model

Why Use Functional Abstraction?

- Easier modifiability: Only one place to update
- Transparency : What the code does is clearer
  - Reduced clutter throughout code: Don’t have to look at all the gory details every time want to undertake this task
  - Can communicate intention from clear name
- Easier later reuse
- Reduced complexity lowers risk of programming error

Using Functional Abstraction in AnyLogic



Methods

- Methods are “functions” associated with a class
- Methods can do either or both of
  - Computing values
  - Performing actions
    - Printing items
    - Displaying things
    - Changing the state of items
- Consist of two pieces
  - Header: Says what “types” the method expects as arguments and returns as values, and exceptions that can be thrown
  - Body: Describes the algorithm (code) to do the work (the “implementation”)

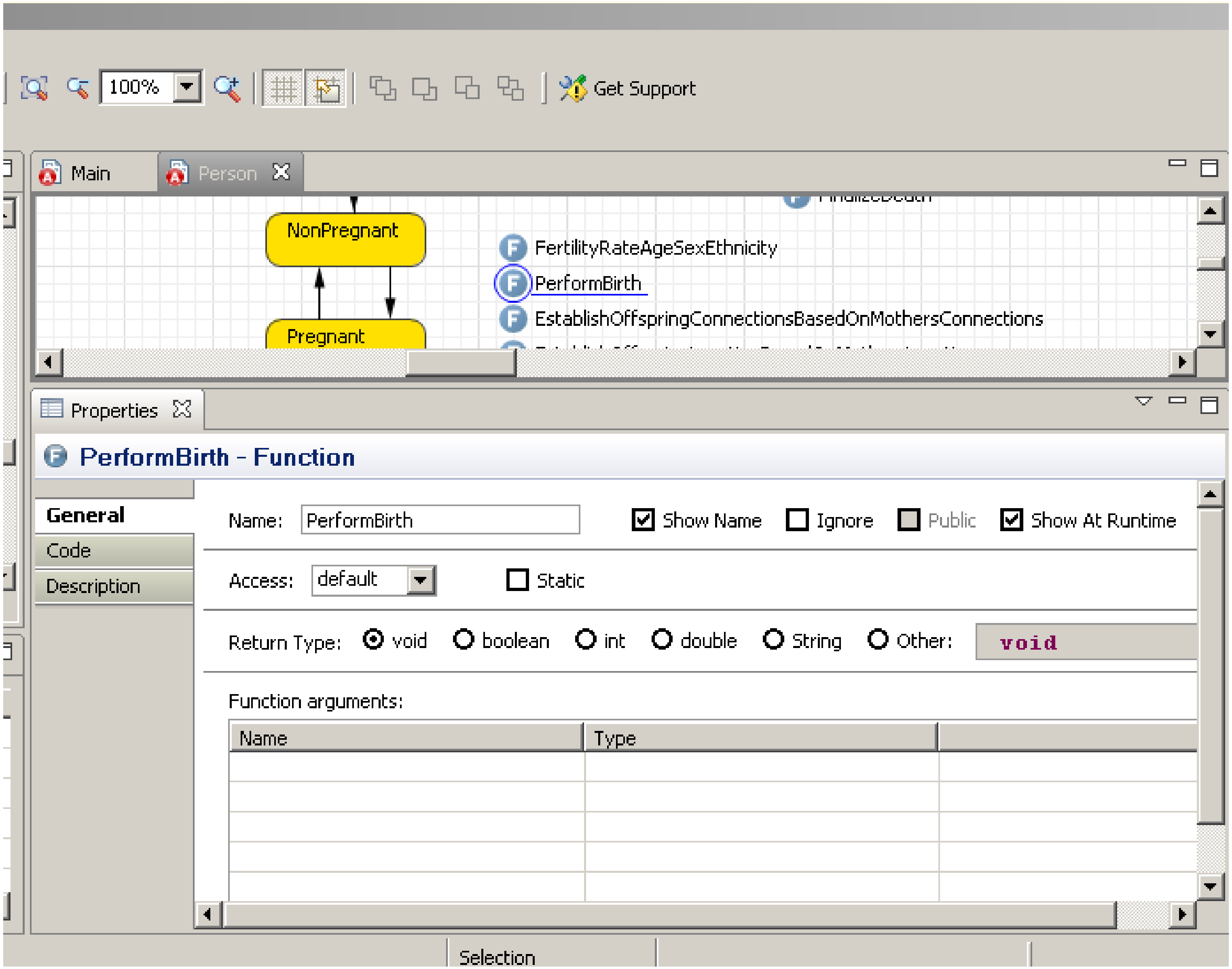
Method Bodies

- Method bodies consist of
  - Variable Declarations
  - Statements
- Statements are “commands” that *do* something (effect some change), for example
  - Change the value of a variable or a field
  - Return a value from the function
  - Call a method
  - Perform another set of statements a set of times
  - Based on some condition, perform one or another set of statements

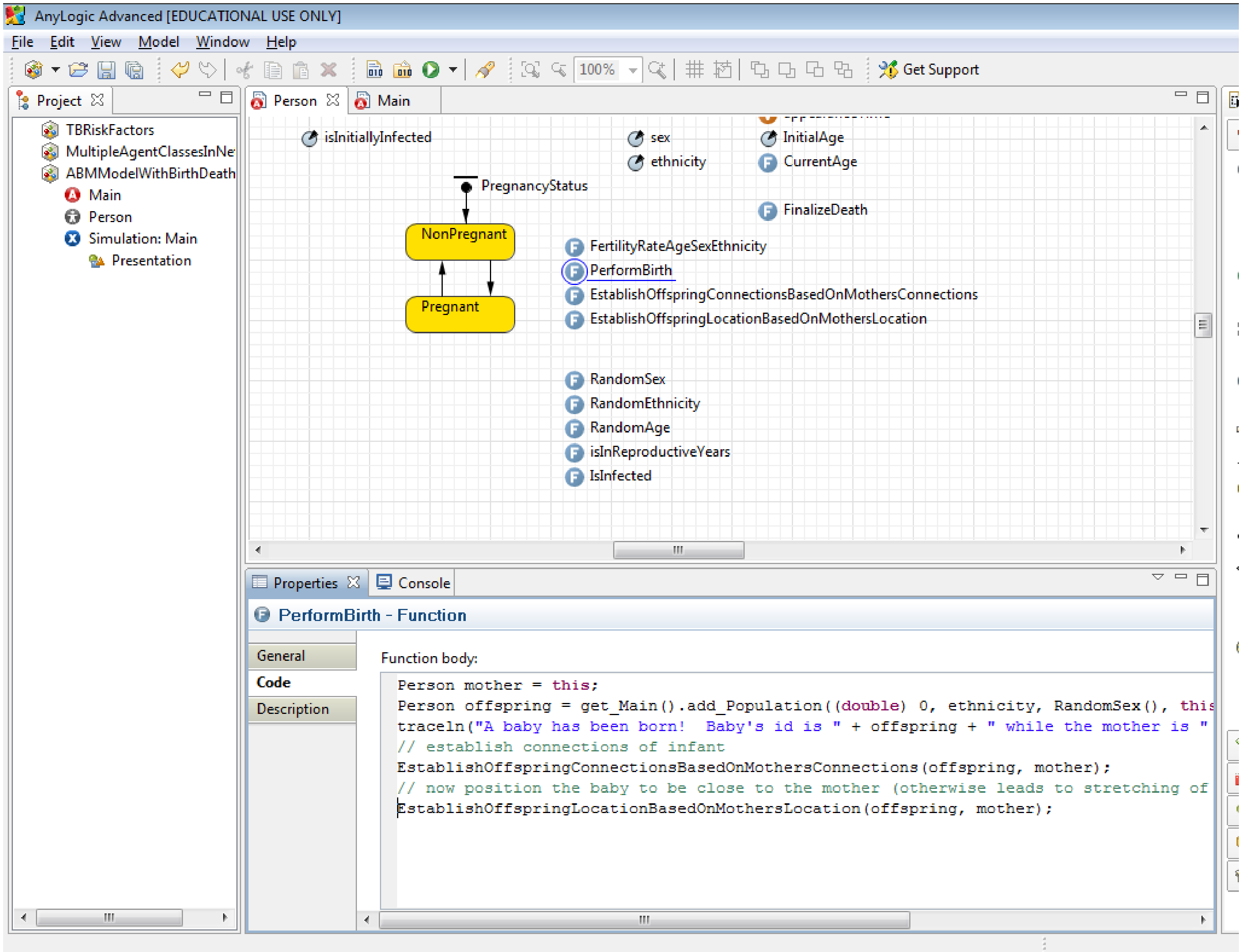
Using Functional Abstraction in AnyLogic: Example Functions

- Functions
  - AgeCoefficientForSmokingInitiation
  - CirclePerimeterColorFromState
  - CirclePerimeterWidthFromState
  - CountContacts
  - CountSmokingContacts
  - FractionOfContactsThatSmoke
  - IsCurrentSmoker
  - ReactivationRateCoefficientForCKDStage
  - ReactivationRateCoefficientForSmokingStatus
  - ReactivationRateForSmokingStatusAndCKDStage
  - SmokingInitiationHazardCoefficientAsAFunctionOfFractionOfContactsThatSmoke
  - SmokingIntiationHazard
  - getDegree

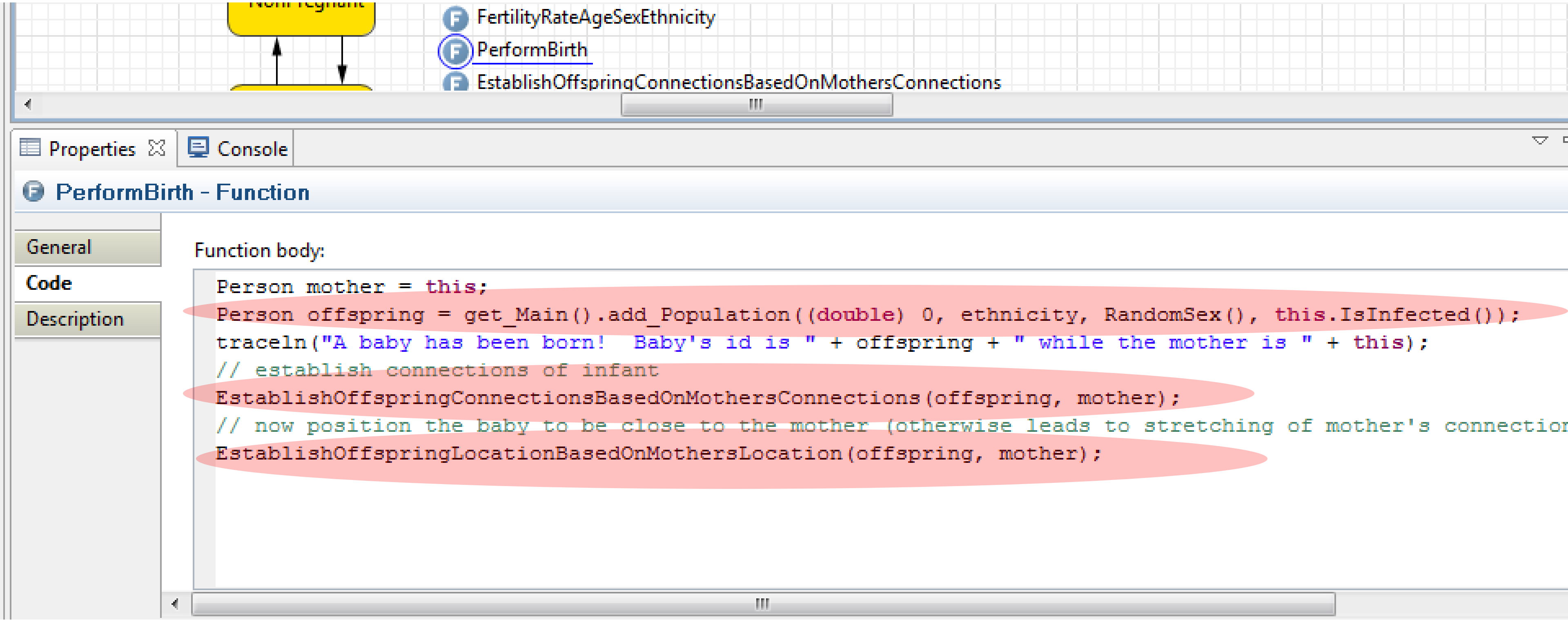
A Function's Definition



Another Example



A Closer Look at the Code…



What is called a "function" in AnyLogic is classically called a "Method"

Parameterization

- We can parameterize functions, so that the values that they yield depends on the values passed to them as “arguments” by callers
  - This allows flexibly: A function can be used somewhat differently in different contexts
  - While parameters may differ, the *behavior of the function* will typically be the same

Examples of Parameterization

- We may build a function that identifies all people who have been smokers for more than *n* years
  - *n* here is a parameter! Different contexts, we might be interested in different *n.*
- We may wish to count the number of people of a certain sex
  - Rather than independently creating separate methods for Males and Females, we may create a method that is called CountPopulationOfSex that takes a parameter that specifies the sex of interest

A Hierarchy of Functional Abstractions

- We build up higher-level functional abstractions out of lower level ones
  - For example
    - The implementation of FractionOfContactsThatSmoke() might make use of CountSmokingContacts() and CountContacts()
    - We might define CountMen() and CountWomen() with implementation of both calling CountPopulationOfSex()
- Particularly powerful functional abstractions are those which are parameterized by functions
  - In object-oriented programming, we generally do this by using *polymorphism* – passing objects that match some interface, but whose implementation of that interface can differ