Methods and Functional Abstraction

Nathaniel Osgood
MIT 15.879

May 2, 2012
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 model value
• Complexity leads to systems that are late, over budget, and of substandard quality
• Complexity has extensive impact in both human & technical spheres
• Achieving modularity in a model is key to reducing complexity of model development
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)
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
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 another).
  - Return some representation (e.g. a string) of or information about a person in the model.
Encapsuliation: Accompanies Abstraction

- *Separation of interface from implementation (allowing multiple implementations to satisfy the interface)* facilitates modularity

- Specifications specify expected behavior of anything providing the interface

- Types of benefits
  - *Locality*: Separation of implementation: Ability to build one piece without worrying about or modifying another
    - See earlier examples
  - *Modifiability*: Ability to change one piece of project without breaking other code
  - Some reuse opportunities: Abstract over mechanisms that differ in their details to only use one mechanism: e.g. Shared code using interface based polymorphism
Why Use Functional Abstraction?

• Easier modifiability: Only one place to update

• Transparency: What the code does is clearer
  – Can communicate intention from clear name
  – Reduced clutter throughout code: Don’t have to look at all the gory details of the how every time want to undertake this task

• Easier later reuse

• Reduced complexity lowers risk of programming error
Using Functional Abstraction in AnyLogic
Using Functional Abstraction in AnyLogic: Example Functions

- AgeCoefficientForSmokingInitiation
- CirclePerimeterColorFromState
- CirclePerimeterWidthFromState
- CountContacts
- CountSmokingContacts
- FractionOfContactsThatSmoke
- IsCurrentSmoker
- ReactivationRateCoefficientForCKDStage
- ReactivationRateCoefficientForSmokingStatus
- ReactivationRateForSmokingStatusAndCKDStage
- SmokingInitiationHazardCoefficientAsAFunctionOfFractionOfContactsThatSmoke
- SmokingInitiationHazard
- getDegree
Hands on Model Use Ahead

Load Sample Model: ABMModelWithBirthDeath
(Via “Sample Models” under “Help” Menu)
A Function’s Definition
Another Example
A Closer Look at the Code...

What is called a “function” in AnyLogic is classically called a “Method” in Object Oriented Programming.
Methods

• Methods are “functions” you can call on an object
• 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

• Recall: 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
Method “Parameters” or “Arguments”

- Parameters passed to the method are accessible within the method body.

- These parameters are only available inside the method:
  - Once the method exits, these parameters are no longer available.
  - The fact that we have a parameter named “a” does not change any value of “a” outside the method.
    - If we refer to “a” within the method, we will be referring to the value of the parameter.
    - After we leave the method, “a” will refer to whatever it did before the method call.

- In most (i.e. non-static) “Methods”, “this” is passed implicitly as a parameter – to tell the method on what object we have invoked this method.
Pass by Value & Modifying Parameters

- In Java, changing the value of the parameters does not modify the values of variables passed to this method.
  - Note, however, that if a parameter in the method is a reference to the same object as something outside of method, a change made within method will still be visible after return.

- For example, if we had:
  ```java
  void MyMethod(double a) { a = 5.0; }
  ```
  And we had this use of it:
  ```java
  double b;
  b = 2.0;
  MyMethod(b);
  // b would still be 2.0 here
  ```
Reminder
EstablishOffspringConnectionsBasedOnMothersConnections
Consider How one Method Calls Another

Prior to the method call

After the method call
The “Flow of Control”

```
PerformBirth

EstablishOffspringConnectionsBasedOnMothersConnections

Method call

PerformBirth

```

- "PerformBirth" resumes after the method call
The Call Stack

• Because one function can call another (and so on), and then return, we need a way of keeping track of “where we were” when we resume execution after the return
  – What the values of “local” variables are
  – Where in the program

• The call stack is the structure that performs this function
  – Method variables live on the “call stack”
  – When one method calls another, the new method’s variables are placed in an “activation record” or “stack frame” on the call stack

• You will sometimes encounter the call stack in
  – Error messages
  – The AnyLogic & Eclipse Debuggers
Recall: Code for “PerformBirth”

Prior to the method call:

After the method call:

```java
Person mother = this;
Person offspring = this.getMain().addPopulation(double) 0, ethnicity, RandomSex(), this.IsInfected(), mother);
tracef("A baby has been born! Baby's id is " + offspring + " while the mother is " + this);
// establish connections of infant
EstablishOffspringConnectionsBasedOnMothersConnections(offspring, mother);
// now position the baby to be close to the mother otherwise leads to stretching of mother's connections across field or
EstablishOffspringLocationBasedOnMothersLocation(offspring, mother);
children.add(offspring);
```
A call from
EstablishOffspringConnectionsBasedOnMothersConnections

Call to “connectTo” to add the mother as a connection of the offspring
Call Graph (Static)

- PerformBirth
- EstablishOffspringConnectionsBasedOnMothersConnections
- connectTo
Call **Stack** (Dynamic) for Our Example

Activation record for (this call to) `connectTo`
Activation record for (this call to) `EstablishOffspring`
Activation record for (this call to) `PerformBirth`

These variables happen to be named the same, but the *this* “offspring” doesn’t in any way “know about” or interfere with that “offspring”!
Reinforcing the Previous Points

• Note that because the names and existence of the parameters inside the method doesn’t affect those outside, the preceding would still be true if we had the following:

```c
void MyMethod(double a) { a = 5.0; }
```

And we had this use of it:

```c
double a;
da = 2.0;
MyMethod(a);
traceln(a); // a would still be 2.0 here (it is unaffected by the fact that we happened to temporarily have something called by the same name within “MyMethod”
```
How is this Achieved?

• There is a “call stack”
  – Everytime we make a call to a method, the new parameters get placed “on the stack”

• Suppose we had
  this.foo(1,2)
and suppose we had
  void foo(int a, int b) { this.bar((a+b)*2, b-a); }
Methods

• A method can do either or both of
  – Compute & return a value
  – Perform some action

• Methods come in 2 types
  – By far most common: Methods associated with objects (i.e. with instances of classes)
  – Less common: Static methods
Methods Associated with Objects

• These are by far the most common class of methods

• When we have a method of this sort, it always takes an implicit parameter called “this”
  – This method is always called on an object
    • “this” is a reference to the object on which it is called
  – This parameter is not stated explicitly, but is always passed to the method – even if the method appears to take no “arguments” (parameters)
Examples

p.getConnectionsNumber()
   Within the call to the “getConnectionsNumber()” method, “this” will refer to the same object as does “p” outside

p.getConnectedAgent(0).getName()
   Within the call to the “getConnectedAgent” method, “this” will refer to the same object as does “p” outside
   Within the call to the “getName” method, “this” will refer to the same object as does p.getConnectedAgent(0) outside
Static Methods

• Static methods are associated with a `class`, and not a particular object
  – Syntactically, these look like `Person.nextId()`

• Because they are not associated with a specific object, static methods have no implicit “this” parameter

• These are much closer to our classic notion of a “function” (e.g. `sin(x)`, `sqrt(x)`, `square(x)`)  
  – Like static methods, such classic functions have no object associated with them
Determining if a Method Needs to be Static

• Does the method need to depend on the value of some object, or of some information available through “this”
  – This may not be obvious – e.g. to call “get_Main()”, one needs to have a reference to the current Agent

• Given the same values of the arguments, would the value of the function be identical in any context?
Example “Static” (Non-Object-Specific) Method
A Hierarchy of Functional Abstractions

• We build up higher-level functional abstractions out of lower level ones, e.g.
  • Implementation of PerformBirth calls EstablishOffspringConnectionsBasedOnMothersConnections, which calls connectTo
  • The implementation of FractionOfContactsThatSmoke() might make use of CountSmokingContacts() and CountContacts()
  • We might define CountMen() and CountWomen() with implementation of both just 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
Methods & Exceptions

• Recall: A functions header can be viewed as defining part of a contract that says “if you give me these parameters, I’ll do X for you, and return Y”
  – Normally, extra specifications (via comments or formal guarantees) are needed to make this contract precise

• Because things can go wrong within a method, exceptions may be thrown within it. To be consistent with its stated behavior, the method must either
  – Handle these exceptions itself
  – State explicitly that this sort of exception can be thrown by it (thus delegating handling of it to calling methods)
Recall: 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
Recall: Try-Catch Statements

try
{
  try-block
}
catch (ExceptionType1 e)
{
  catch-block1
}
catch (ExceptionType2 e)
{
  catch-block2
}
...
catch (ExceptionType n 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.
Recall: Output to a File

```java
try {
    FileOutputStream fos = new FileOutputStream("file.tab");
    PrintStream p = new PrintStream(fos);
    p.println(datasetName.toString()); // output tab delim vals
}
catch (Exception e) {
    traceln("Could not write to file.");
}
```

This cleans up after itself in a generic way, but misses the opportunity to elicit much greater information using either the Type of the exception or from the exception object e.
Declaration that a Method Can Throw Exceptions: “Throws” Clauses in Signature

This method “delegates” handling of the exception to its callers, rather than handling them itself.

This method “cleans up after itself” if something goes wrong, and thus doesn’t need a “throws” clause for these exceptions.
Where to Handle Exceptions?

• Often the context of an exception is most clear closer to its source
  – The further one goes up the “call chain” (set of methods calling methods)
    • The less detail one has about where exactly things went wrong
    • The less detail one has for error messages

• Further up, one can more easily abort the overall computation
Shortcomings of AnyLogic Functions (vs. explicitly declared Methods)

• Both primary functions and supporting functions are all visible to others

• They cannot declare that they throw exceptions