Best Practices:
Overview & Technically-Related

Nathaniel Osgood
MIT 15.879

May 16, 2012
Two Important Concerns

• Building the right model
  – Level of depth & breadth: Where to stop in disaggregation?
  – Division between endogenous, exogenous & ignored

• Building the model right
  – For agent-based modeling, this involves consideration of software engineering principles
Building the Model Right: Some Principles of Software Engineering

**Technical guidelines**
- Eschew speculative complexity
- Use abstraction & encapsulation to simplify reasoning & development
- Name things carefully
- Design & code for transparency & modifiability
- Use configurable logging
- 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
  - Preliminary design/Code/Tests
- Where possible, perform simple tests to verify functionality
- Preserve successive distinct model versions
- Keep careful track of experiments
- Use tools for version control & documentation & referent.integrity
- Integrate with others’ work frequently & in small steps
- Use discovery of bugs to identify process weaknesses
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
Avoiding Debugging

• Defensive Programming
• Offensive Programming
Offensive Programming: Try to Get Broken Program to Fail Early, Hard

- Asserts: Actually quit the program
- Fill memory allocated with illegal values
- Fill object w/illegal data just before deletion
- Set buffers at end of heap, so that overwrites likely trigger page fault
- Setting default values to be illegal in enums
- We will talk about Assertions & Error Handling later this week
What is an “Assertion”?

• An “Assertion” is a “sanity check” during program execution (model simulation) to confirm that one’s assumptions hold true

• This helps identify
  – Mistaken understanding (on our or others’ part)
  – Logic errors
  – Inconsistencies in reasoning
Assertion Goal: Fail Early!

• Alert programmer to misplaced assumptions as early as possible

• Benefits
  – Documents assumptions
  – Reduces likelihood that error will slip through
    • Helps discourage “lazy” handling of only common case
    • Forces developer to deal explicitly with bug before continuing
  – Reduces debugging time
  – Helps improve thoroughness of tests
Assertions Regarding Coordinates

```c
// x is changing most quickly: elements on a given row are close together
assert(IsLegalCoordinate(x,y));
return y * ctXCells + x;
```
Confirming that Something Has Been Computed Before it is Used

```javascript
assertLocal(rgIsCurrentMovesFromNeighborSpecified.get(iPartner));
return(rgCurrentMovesFromNeighbor.get(iPartner));
```
Checking Assumption Regarding Computation

```java
public void testInitialAppointment()
{
    Person p = s_main.selectRandomPerson();
    Date appointmentTime = scheduleBook.ScheduleAppointmentForPersonAfterDate(p, scheduleOpeningDateTime);
    assertEquals(appointmentTime, scheduleOpeningDateTime); // appointment should not be at the same date.
}

// make sure should be able to fit 4 appts per timeslot
public void testMultipleAppointmentInInitialTimeslot()
{

    int iRegularGrid = (dYofNeighbor - 1) * 3 + (dXofNeighbor - 1); // regular grid should be (iRegularGrid - 4); // neighbor shouldn't be ourselves.
    if (iRegularGrid > 4)
        return (iRegularGrid - 1);
    else
        return (iRegularGrid);
}
```
Avoid Side Effects in Assertions

• Because assertions may be completely removed from the program, it is unsafe to rely on side effects occurring in them

```
assert ++i < max;
```

Enabling Assertions in AnyLogic
Enabling Assertions in Java

• 2 ways

  – Usual: Via java runtime command line
    -enableassertions/-ea[descriptor]
  • e.g.
    -enableassertions:com.acme.Plotter
    -enableassertions:com.acme...
    -disableassertions/-da[descriptor]

  – Less common: via reflection (ClassLoader)
    public void setDefaultAssertionStatus(boolean enabled)
    public void setPackageAssertionStatus(String packageName, boolean enabled)
    public void setClassAssertionStatus(String className, boolean enabled)
Defensive Programming

- Naming conventions
- Formatting
- Separate
  - Commands (side effects)
  - Queries (pure)
- Avoid manifest constants
- Consolidate condition checks in methods or objects ("specification" pattern)
- Minimize variable lifetime & span between references

- Check return values, value legality
- Always handle all cases (even illegal)
- Always put in { } after if
- Beware empty catch blocks
- Use finally blocks
- Don’t reuse temporary variables
- Initialize vars, member data as they are declared or in constructor
- Use pseudocode programming process
Other suggestions

• Strive for transparent code
  – Use variable name conventions
  – Consistent formatting
• Strive for higher abstraction level
  – Spot commonality & place into a separate function or class
  – Encapsulate repetitive actions in methods
  – Move whole & partial conditionals to methods
  – Consider putting body of loop in a method
• Create diverse well-named small functions
• Use enumerations
Bad Smells (Many from McConnell, Code Complete 2.0)

- Duplicate code
- Long routine
- Deep/long if/loops
- Inconsistent interface abstraction
- Lots of special cases
- Poor cohesion
- Too many parameters
- Single update yields changes to many places
- Keep on creating ad-hoc data structures/classes
- Global variables
- Primitive types

- Need to update multiple inheritance hierarchies
- Subclasses not really subtypes
- Related items spread among multiple classes
- Method deals more with other classes than its own
- Need to know implementation of other class
- Unclear name
- Setup & takedown code around call
Style & Convention

• Naming Conventions
• Commenting
• Metadata (e.g. Javadocs)
• Indentation
• Module Naming
• Construct placement
• Compiler Pragma & Mechanisms
Naming Conventions

• Naming conventions are a powerful tool

• Benefits
  – Reduce risk of errors
  – Easier understanding of others’ code
  – Easier understanding of code in future
  – Lower risk of name clashes
  – Easier search for desired item (e.g. method/variable/class)
Java Naming Conventions

• Distinguish Typographic & Grammatical

• Packages
  – Short lowercase alphabetics (digits rare)
  – Start with organization internet domain name (e.g. ca.usask)

• Classes/interfaces
  – First word of each capitalized (TagHasher)
  – Avoid all but most common abbreviations
  – Generally nouns/noun phrase
  – Interfaces sometimes adjective
Java Naming Conventions 2

• Method & Fields
  – Same as classes but first letter lowercase
  – Const static fields all uppercase, “_” as separ.
  – “Action” methods named with verb
  – “is” for booleans
  – Query: noun/noun phrase or verb w/”get” prefix
  – Converters: “toX”, primitiveValue

• Local variables
  – Same as members but can be short, context-dependent
Booleans

• Base name should give clear sense of condition in question
• Use common convention to indicate boolean
  – “f” prefix (e.g. fOpen)
  – is prefix (e.g. isOpen)
  – “?” suffix (e.g. open? – legal scheme)
• Avoid negation in names (e.g. isNotOpen)
Suggestions

• Use consistent abbreviation conventions
• Provide translation table at top of method to clearly describe purpose of each variable
• Avoid similar names
• Be careful of similar letters
• Avoid overloading predefined names (even if syntactically & semantically allowed)
• Avoid throwaway names for “temporary” vars
• Strive for clarity
Use Modifiers

• Use “final” (including for parameters in Java) to prevent side-effects
  – This is exposed through the Anylogic interface
  – Examples
    • Prevent modification to this in method
    • Prevent assignment to parameter

• Declaring variables as static can prevent needless memory use
Output to the Console

• System.err.println(String)
  – System.err.println("Sent cure message to person [" + associatedPerson + "]");
• traceIn(String)
Use in AnyLogic

```java
public void Cure()
{
    associatedPerson.send("Cured!", associatedPerson);
    associatedPerson.deliver("Cured!", associatedPerson);
    System.err.println("Sent cure message to person [" + associatedPerson + "]");
}

@Override
public String toString()
{
    return "Entity for agent " + associatedPerson;
}

/**
 * This number is here for model snapshot storing purpose
 * It needs to be changed when this class gets changed
 */
private static final long serialVersionUID = 1L;
```
Internals of AnyLogic files: XML