Effectively Using JML
Software Engineering Processes incorporating Formal Specification

Joseph Kiniry
Software Engineering
Processes

- old-school processes
  - CRC and state-chart based
- heavyweight processes
  - all up-front design, use UML or similar
- lightweight processes
  - unit test-centric (XP), design on-the-fly
- custom processes
  - use a process that works for you
**Effective JML**

- Effectively using JML means effectively using JML **tools**

- Development process of **project** (macro-scale) is realized by **daily** development process (micro-scale)

- Rich **tool** support must be supported by rich **process** support

  - Code standards and organization support
Facets of Critical Software Engineering

- requires a *rich environment* that synthesizes all primary facets
  - code standards
  - version and configuration management
  - automated build system
  - unit tests

- requires *developer investment* in learning, applying, and understanding the method
Non-technical Facets

• requires *social adoption*

  • internal tensions caused by mandated changes in process can cause a development team to self-destruct

• requires *institutional support*

  • an understanding of the time, resources, and potential results of development with formal methods
“Contract the Design”

- you are given an architecture with no specification, little documentation and you must somehow check the system is correct

“Design by Contract”

- you are designing and building a system yourself, relying upon existing components and frameworks
Contract the Design

- A body of code exists and must be annotated
  - The architecture is typically ill-specified
  - The code is typically poorly documented
  - The number and quality of unit tests is typically very poor
  - The goal of annotation is typically unclear
Goals of Contract the Design

- improve understanding of architecture with high-level specifications
- improve quality of subsystems with medium-level specifications
- realize and test against critical design constraints using specification-driven code and architecture evaluation
- evaluate system quality through rigorous testing or verification of key subsystems
A Process Outline for Contract the Design

- directly translate high-level architectural constraints into invariants
  - key constraints on data models, custom data structures, and legal requirements
- express medium-level design decisions with invariants and pre-conditions
- use JML models only where appropriate
- generate unit tests for all key data values
Design by Contract

※ writing specifications first is difficult but very rewarding in the long-run

※ you *design* the system by writing *contracts*

※ a refinement-centric process akin to early instruction in Dijkstra/Hoare approach

※ ESC/Java2 works well for checking the consistency of formal designs

※ resisting the urge to write code is *hard*
Goals of Design by Contract

- work out application design by writing contracts rather than code
- express design at multiple levels
  - BON/UML $\Rightarrow$ JML $\Rightarrow$ JML w/ privacy
- refine design by refining contracts
- write code *once* when architecture is stable
A Process Outline for Design by Contract

- outline architecture by realizing *classifiers* with *classes*
- capture system constraints with invariants
- use JML models only where appropriate
- focus on preconditions over postconditions
- develop test suite for your design by writing a data generator for your types
Case Study: KOA Tally System

- Dutch government decided to make *remote voting* available in 2004 to expatriates
  - remote voting is voting by *telephone* or via the *Internet*
- a consulting firm LogicaCMG designed, developed, tested, and deployed system
- KUN participated in review of system
KOA Tally System: Background

- A primary recommendation of review was that a 3rd party should re-implement a critical part of the system from scratch.
- Government opened up bid on independent implementation of counting/tally component.
- KUN group bid on contract and won.
  - Key factor in bid was proposed use of formal methods (JML) in application development.
KOA Architecture

- three main components, each the responsibility of one developer
  - file and data I/O (E. Hubbers)
  - GUI (M. Oostdijk)
  - core data structures and counting algorithm (J. Kiniry)
- most of specification and verification effort was focused in the core subsystem
Code Standards

- lightweight code standards for this effort
- basic rules about identifier naming, documentation, annotation, and spacing
- each developer had his own idiom
- avoid enforcement or tool use that causes merge conflicts
- code standard enforcement with checkstyle
- http://checkstyle.sourceforge.net/
Version and Config Management

- version management via CVS
  - policies on commits and merges
    - code must build and specs must be right
    - rules are developer-enforced (not triggers)
- configuration management via Make, a single class of constants, and runtime switches
  - with more time Java properties and bundles are typically used as well
Automated Build System

- GNU make based build system
- works on all operating systems
- single developer responsible for build architecture and major upkeep
- major targets include:
  - normal build, jmlc build, unit test generation and execution, verification, documentation generation, style checking
Unit Testing

- one developer responsible for unit test architecture and major upkeep
- each developer responsible for identifying key values of their data types
- unit test only core classes, not GUI or I/O
- automatically generate ~8,000 tests
- ensure 100% coverage for core
- *complements* verification effort
Verification

- attempt to verify only core classes
- focus effort on opportunities for greatest impact and lowest risk
- results of verification with ESC/Java2.0a7
  - 47% of core methods check with ESC/Java2
  - 10% fail due to Simplify issues
  - 31% of postconditions do not verify due to completeness problems
  - 12% fail due to invariant issues
## Application Summary

<table>
<thead>
<tr>
<th></th>
<th>File I/O</th>
<th>GUI</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>8</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>methods</td>
<td>154</td>
<td>200</td>
<td>83</td>
</tr>
<tr>
<td>NCSS</td>
<td>837</td>
<td>1,599</td>
<td>395</td>
</tr>
<tr>
<td>specs</td>
<td>446</td>
<td>172</td>
<td>529</td>
</tr>
<tr>
<td>specs: NCSS</td>
<td>1:2</td>
<td>1:10</td>
<td>5:4</td>
</tr>
</tbody>
</table>