Verification-centric Software Development in Java with BON, JML, and ESC/Java2

Joe Kiniry
University College Dublin

Outline

- example projects
- effectively using formal methods
- analysis and design with BON
- assertions and specifications
- contracts and system specifications in BON
- applying BON to Java and JML
- code standards and metrics
- static analysis for software construction
- models in JML

Part 0:
Example Projects

First Year Course: One Dimensional Cellular Automaton Simulator

Cellular Automata

- a fundamental model for computation
- very simple conceptual model
- small set of concepts
- multiple complexity refinements
  - dimensionality
  - cell type

Project Dimensions

- classified as a small-sized project
  - our estimate is <<1,000 LOC
  - ~50 LOC/week/person
- our complete design has 9 classes
- some classes are optional and are only implemented by advanced students

Co-Analysis and Co-Design

- system analysis and design was conducted live, in-class, with first year students
- analysis was captured with BON
  - informal charts only, no tool support
- design was captured with JML

- *students were not told that they were doing formal analysis or design*

Implementation Process

- students implemented the resulting JML-annotated Java using design by contract
- students used Emacs & vi, not Eclipse
- a Makefile was provided that triggered javac, jml, jmlc, escjava2, javadoc, and jmldoc
- no unit testing was performed whatsoever
  - for other, larger projects tests are frequently generated with jml-junit

Results

- ~80% of the teams' programs worked correctly the first time they executed
- one team had an NPE, fixed in an hour after they ran ESC/Java2 for the first time
- another had a mysterious crash, traced and fixed using a debugger in one afternoon
- this process results in a very high-quality Java system that is very nearly "correct-by-construction", accomplished by 1st years

Second Year Course: The C=64 Game "Thrust"

“Thrust”

“Thrust”

The Project:
The C=64 Game "Thrust"

- connection to core computing concepts via discrete event simulation
- a few major components
  - file I/O, GUI and rendering, simulation
- several key algorithms
- looks cool and is fun to play

Project Dimensions

- classified as a medium-sized project
  - our estimate is <<5,000 LOC
  - ~100-125 LOC/week/person
- our (very) complete design has 75 classes
- recall that original game written by one person in a few months in 650X assembler

Project Decomposition

- I/O: keyboard input to start and play game
- GUI: bitmaps (terrain), fonts (scores, fuel), and shapes (spaceship, bullets, stars)
- sound: music and effects
- core data structures: entities (spaceship, factory, bullet, etc.), score and high score
- discrete event simulation: main event loop, animations (barriers, explosions, factory smoke, stars, etc.), physics, collisions

What Ones Mind Wants To Do Now

- How do I open a window?
- How do I make a sound?
- How do I draw a line?
- Will I use arrays?
- Floating point numbers or integers?
- etc.

The Proper Course

- Ignore the problems of programming.
- Forget about Java.
- Step back and take a deep breath.
- Relax.

- Brainstorm about the **idea** of Thrust.

Commercial Software Development: The KOA Tally System

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
- RUN 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
- RUN 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
- coding standard checked 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 would be 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 nearly 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

	File I/O	GUI	Core
classes	8	13	6
methods	154	200	83
NCSS	837	1,599	395
specs	446	172	529
specs:NCSS	1:2	1:10	5:4

Part I: Effectively Using Formal Methods

Software Engineering Processes
incorporating Formal Specification

The Range of 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

Specification in Process

- "Contract the Design"
  - one is given an architecture with no specification, little documentation and one must somehow check the system is correct
- "Design by Contract"
  - one designs and builds a system 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
  - one designs the system by *thinking* and 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 ➡ JML ➡ 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 design by writing a data generator for all interesting types

Part II: Analysis and Design with BON

Two Levels of BON Specifications

- informal charts and diagrams
  - specified primary concepts of system, scenarios of use, primary events
- formal diagrams
  - specifies contracts on type interfaces, method call sequences, architecture structure

Informal BON Charts

- the *static model*
  - system diagrams (informal charts)
  - class dictionary (a dependent chart)
- the *dynamic model*
  - object creation charts
  - scenario charts
  - event charts

Class Dictionary

- lists all primary concepts (classifiers) in the system
  - each class's cluster(s) and description are provided
  - clusters are dependent upon the system and cluster charts
  - description is dependent upon the corresponding class chart
- the MONITORING_SYSTEM class dictionary

Object Creation Charts

- shows what classes create new instances of other classes
- serves as a link between the static and the dynamic models
- only high-level analysis classes are treated
- the MONITORING_SYSTEM <u>creation chart</u>

Creation Chart Example

CREATION	*CONFERENCE_SUPPORT*	**Part:** 1/1
COMMENT List of classes creating objects in the system.	**INDEXING** **created:** 1993-02-18 kw	

Class	**Creates instances of**
CONFERENCE	*PROGRAM_COMMITTEE, TECHNICAL_COMMITTEE, ORGANIZATION_COMMITTEE, TIME_TABLE*
PROGRAM_COMMITTEE	*PROGRAM, PAPER, PAPER_SESSION, PERSON*
TECHNICAL_COMMITTEE	*TUTORIAL, TUTORIAL_SESSION, PERSON*
ORGANIZATION_COMMITTEE	*MAILING, ADDRESS_LABEL, STICKY_FORM, REGISTRATION, PERSON, INVOICE, INVOICE_FORM, ATTENDEE_LIST, LIST_FORM, POSTER_SIGN, POSTER_FORM, EVALUATION_SHEET, EVALUATION_FORM, STATISTICS*
*PRESENTATION**	*STATUS, PERSON*
PAPER	*REVIEW, ACCEPTANCE_LETTER, REJECTION_LETTER, LETTER_FORM, AUTHOR_GUIDELINES*
TUTORIAL	*ACCEPTANCE_LETTER, REJECTION_LETTER, LETTER_FORM*
REGISTRATION	*CONFIRMATION_LETTER, LETTER_FORM, BADGE, BADGE_FORM*

Scenario Charts

- semi-equivalent to UML's use-case diagrams
- a scenario is a type of system usage, user or programmatic
  - focus is on important top-level scenarios that are critical to the system design
  - only natural language is used for the high-level specification

Scenarios

- the description of scenario is used as the documentation for
  - the public interface, and
  - the corresponding unit test suite
- scenarios are refined at the intermediate level of specification into object message passing descriptions

Scenario Chart Example

SCENARIOS	*CONFERENCE_SUPPORT*	**Part:** 1/1
COMMENT Set of representative scenarios to show important types of system behavior.	**INDEXING** **created:** 1993-02-16 kw	

Send out calls and invitations:
Using mailing lists and records of previous conference attendees and speakers, prepare and send out calls for papers and invitations to attend the conference.

Create sessions and chairs:
Partition the conference into sessions of suitable length; allocate session rooms and select a chairperson for each session.

Register paper and start review process:
A paper is registered and three referees are selected; the paper is sent to each referee, and the paper status is recorded.

Accept paper and notify authors:
A submitted paper is selected and an acceptance date is entered; a notification letter is created and sent to the authors.

Assign paper to session:
A session suitable for the paper is selected and the paper is entered in the list of presentations for that session.

Register attendee:
An attendee is registered with his/her address and selected tutorials are recorded.

Print conference attendee list:
All registrations are scanned and a list with attendee names, addresses and affiliations is produced and sent to a printer.

Print badge:
An attendee is selected, and the corresponding badge is printed in appropriate format.

Event Charts

- object interactions are ultimately caused by *external events*
  - external events trigger system execution
- *internal events* are high-level, important triggers within a system
  - typically an external event triggers one or more internal events

Event Identification

- external events connote the external (perhaps public) interface of a system
- internal events connote the private subcomponent interfaces *within* a system
- each event is either ingoing or outgoing
- the MONITORING_SYSTEM external event diagram and internal event diagram

Example External Event Chart

EVENTS	*CONFERENCE_SUPPORT*	**Part:** 1/2
COMMENT Selected external events triggering representative types of behavior.	**INDEXING** **created:** 1993-02-15 kw **revised:** 1993-04-07 kw	

External (incoming)	Involved object types
Request to register a submitted paper	*CONFERENCE, PROGRAM_COMMITTEE, PAPER*
Request to accept a paper	*CONFERENCE, PROGRAM_COMMITTEE, PAPER, STATUS*
Request to assign a paper to a session	*CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, PAPER, PAPER_SESSION*
Selection of a session chairperson	*CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, PAPER_SESSION, PERSON*
Request to register an attendee	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON*
Request to print conference attendee list	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, ATTENDEE_LIST*

Example Internal Event Chart

EVENTS	*CONFERENCE_SUPPORT*	**Part:** 2/2
COMMENT Selected internal events triggering system responses leaving the system.	**INDEXING** **created:** 1993-02-15 kw **revised:** 1993-04-03 kw	

Internal (outgoing)	**Involved object types**
Call for papers is sent	*CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING*
Invitations are sent	*CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING*
A paper is sent to referees	*CONFERENCE, ORGANIZING_COMMITTEE, PAPER, STATUS, REVIEW, PERSON*
An invoice is sent	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, INVOICE, INVOICE_FORM*
Warning issued for exceeding tutorial session capacity	*CONFERENCE, REGISTRATION, TUTORIAL*
An author notification is sent	*CONFERENCE, PROGRAM_COMMITTEE, PERSON, PRINT_OUT*, LETTER_FORM*

Part III: Assertions and Specifications

Assertions

- the **assert** statement is the fundamental construct used to specify the correct behavior of software
- the statement

```
assert S;
```

means

“S **must** be true at **this** point

in the program’s execution”

Assertion Syntax in Java

- **all** modern programming languages have an **assert** statement
- beginning in Java 1.4, **assert** is a keyword
- the syntax of a Java assert statement is

```
assert <boolean>[: <String>]
```

- `boolean` is the predicate that **must** be true
- `String` is an optional message that will be printed if/when the assertion fails

Examples of Assertion Use

```
assert z != 0;
x = y/z;

assert (x > MIN_WIDTH);
my_window.setWidth(x);

assert p(x) : “p failed when x=” + x;
a_method_that_depends_upon_p(x);
```

Assertions vs. Logging

- if an assertion fails, the program **halts**
- thus, assertion failures are **critical** failures
- to assert something that is not critical, then a logging message is appropriate

```
if (Debug.DEBUG && !p(x))
  System.err.println("p("+x+") fails");
a_method_that_depends_upon_p(x);
```

Logging Frameworks

- it is **always** wiser to use a logging framework than to use embedded `printlns`
- if a `println` must be used, guard it with a conditional on a constant boolean
  - setting the guard false eliminates all logging code (saves space and time)
- the premier logging frameworks are `java.util.logging`, log4J, and IDebug

Specifications

- specifications of software range in formality
  - informal - English documentation (e.g., “normal” comments)
  - semi-formal - structured English documentation (e.g., **Javadoc**)
  - formal - annotations and assertions (e.g,. **assert** statements and **contracts**)
- **contracts** are a **key concept** in robust software design and construction

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?
  - amount is bigger than the balance?

Informal Specifications

```
/* Deduct some cash from this account and
   return how much money is left. */

public int debit(int amount)
```

- what happens when:
  - amount is negative?
  - amount is bigger than the balance?
  - is the balanced changed when failure?

Semi-Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  *        <code>amount</code> must be
  *        non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

Semi-Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  *        <code>amount</code> must be
  *        non-negative.
  * @result the balance of this account
  * after the debit successfully occurs.
  */
  public int debit(int amount)
```

- many of the same questions arise even though the documentation is much clearer

Formal Specifications

```
/** Debit this account.
  * @param amount the amount to debit.
  * @result the resulting balance.
  */
/*@ requires amount >= 0;
  @ ensures balance == \old(balance-amount) &&
  @                    \result == balance;
  @*/
  public int debit(int amount)
```

Writing and Calling Methods Incorrectly

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left. */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

Writing and Calling Methods Incorrectly

```
/* Deduct some cash from this account and
   return how much money is left */
  public int debit(int amount) {
    if (amount < 0) throw NDE(amount);
    if (balance < amount)
      throw NBE(balance);
    ...
  }
```

HORRIBLE!

```
try {
  b = debit(a);
  if (b < 0) throw NBE();
} catch (Exception e) {
  System.exit(-1);
}
```

Calling Methods Correctly

```
/*@ requires amount >= 0;
  @ ensures balance == \old(balance-amount) &&
  @                    \result == balance;
  @*/
  public int debit(int amount) {
    ...all conditionals are gone!
    ...
  }

if (debit_amount < 0)
  handle_bad_debit(debit_amount);
else
  resulting_balance = debit(debit_amount);
```

Design by Contract

- capture architectural, class-level decisions early as **constraints**
  - e.g., all `Citizens` have two parents
- realize constraints in software as **invariants**
  - an **invariant** is an assertion that must **always** be true whenever a method is called or exits
- capture contracts at method-level in medium-level design using English
  - realize contracts in code using **requires** and **ensures** statements

An Example Use of Design by Contract

<table>
<tr><td>CLASS</td><td colspan="2">CITIZEN</td><td>Part: 1/1</td></tr>
<tr><td colspan="2">TYPE OF OBJECT
Person born or living in a country</td><td colspan="2">INDEXING
cluster: CIVIL_STATUS
created: 1993-03-15 jmn
revised: 1993-05-12 kw</td></tr>
<tr><td>Queries</td><td colspan="3">Name, Sex, Age, Single, Spouse, Children, Parents,
Impediment to marriage</td></tr>
<tr><td>Commands</td><td colspan="3">Marry. Divorce.</td></tr>
<tr><td>Constraints</td><td colspan="3">Each citizen has two parents.
At most one spouse allowed.
May not marry children or parents or person of same sex.
Spouse s spouse must be this person.
All children, if any, must have this person among their parents.</td></tr>
</table>

Related Class Features

- queries
  - spouse? single?
- command
  - marry! divorce!
- constraints
  - at most one spouse is allowed
  - spouse's spouse must be this person

Class Sketch

```
Citizen my_spouse;
//@ invariant (my_spouse != null) ==>
//@           my_spouse.my_spouse == this;

Citizen spouse() { returns spouse; }
boolean single() { returns spouse == null; }
//@ requires single();
//@ ensures !single() && spouse() == new_spouse;
void marry(Citizen new_spouse)
  { my_spouse = new_spouse; }
//@ requires !single();
//@ ensures single();
void divorce() { my_spouse = null; }
```

Testing with Specifications

- specifications mean that no valid parameter testing is necessary in implementations
  - the precondition is **requiring** the client to fulfill their side of the contract for supplier
- when calling a method that has a specification, checking for errors, return values, etc. is no longer necessary
  - the supplier is **ensuring** (guaranteeing) their side of the contract to client

Unit Testing and Programming with Specs

- ~90% of your method-level unit tests are automatically generated
- ~25% **less code** is written because there is no need to test parameters values nor results of method calls for correctness
- code is not littered with `try/catch` blocks to catch exceptions

Part IV: Contracts and Specifications in BON

BON Assertion Elements

Assertion elements		
Graphical BON	**Textual BON**	**Explanation**
Δ *name* **old** *expr*	**delta** *name* **old** *expr*	Attribute changed Old return value
Result **@** ∅	*Result* *Current* *Void*	Current query result Current object Void reference
+ – * / **^** **//** ****	**+ – * /** **^** **//** ****	Basic numeric operators Power operator Integer division Modulo
= ≠ **<** ≤ **>** ≥	**=** **/=** **<** **<=** **>** **>=**	Equal Not equal Less than Less than or equal Greater than Greater than or equal

BON Assertion Elements

→	**->**	Implies (semi-strict)
↔	**<->**	Equivalent to
¬	**not**	Not
and	**and**	And (semi-strict)
or	**or**	Or (semi-strict)
xor	**xor**	Exclusive or
∃	**exists**	There exists
∀	**for_all**	For all
|	**such_that**	Such that
•	**it_holds**	It holds
∈	**member_of**	Is in set
∉	**not member_of**	Is not in set
: *type*	**:** *type*	Is of type
{ }	{ }	Enumerated set
..	..	Closed range

The Person Class

PERSON

$name, address: VALUE$

$children, parents: LIST\,[PERSON]$

Invariant

$\forall\, c \in children \bullet (\exists\, p \in c.parents \bullet p = \mathbf{@})$

Textual Specification

```
deferred class CITIZEN
  feature name,sex,age: VALUE
  spouse: CITIZEN --Husband or wife
  children, parents: SET[CITIZEN] --Close relatives, if any
  single: BOOLEAN --Is this citizen single?
    ensure Result <-> spouse=Void
  end
  deferred marry --Celebrate the wedding.
    ->sweetheart: CITIZEN
    require sweetheart /= Void and can_marry(sweetheart)
    ensure spouse=sweetheart
  end
  ...
  divorce --Admit mistake.
    require not single
    ensure single and (old spouse).single
  end
  invariant
    single or spouse.spouse=Current;
    parents.count=2;
    for_all c member_of children it_holds
      (exists p member_of c.parents it_holds p=Current)
end --class CITIZEN
```

Example Interface Specifications

```
static_diagram Technical_events
component
    class REVIEW persistent
    feature
        reviewer: PERSON
        score: VALUE
        comments: TEXT
    invariant
        score member_of { 'A' .. 'D' }
    end
```

STATUS

```
class STATUS persistent
feature
    received: DATE
    review_started: DATE
    accepted: DATE
    rejected: DATE
    final_received: DATE
invariant
    received <= review_started;
    review_started <= final_received;
    accepted = Void or rejected = Void
end
```

PAPER

```
class PAPER persistent
inherit
    PRESENTATION
feature
    copyright_transferred: BOOLEAN
    reviews: SET [REVIEW]
    final_score: VALUE
    award_best_paper
    transfer_copyright
        require
            status.accepted /= Void
        ensure
            copyright_transferred
        end
    effective accept
    effective reject
end
```

PRESENTATION

```
deferred class PRESENTATION
feature
    code: VALUE
    title: VALUE
    authors: SET [PERSON]
    status: STATUS
    speakers: SET [PERSON]
    deferred accept
        ensure  status.accepted /= Void  end
    deferred reject
        ensure  status.rejected /= Void  end
invariant
    for_all p, q: PRESENTATION such_that
        p /= q it_holds p.code /= q.code and
          p.title /= q.title
end
```

TUTORIAL

```
class TUTORIAL persistent
inherit PRESENTATION
feature
    capacity: VALUE
    attendee_count: VALUE
    prerequisite_level: VALUE
    track: VALUE
    duration: DURATION
    effective accept
    effective reject
end
```

SESSION

```
class SESSION
feature
    chair: PERSON
    code: VALUE
    track: VALUE
    start, end: DATE
    conference_room: VALUE
invariant start < end
end
```

TUTORIAL_SESSION

```
class TUTORIAL_SESSION persistent
inherit
	SESSION
feature
	lecture: TUTORIAL
invariant
	lecture.status.accepted /= Void
end
```

PAPER_SESSION

```
class PAPER_SESSION persistent
inherit SESSION
feature
    presentations: SET [PAPER]
invariant
    for_all p member_of presentations it_holds
        p.status.accepted /= Void
end
```

BON Tools

- EiffelStudio
- The BON Visio Templates
- BON-CASE
- The BON Tool Suite
- Class Skeletons, Javadoc, and JML
- The BONc Tool (new!)

Part V: Applying BON to Java and JML

Using Code Skeletons for BON and DBC

- rather than using a specification language, one can use a programming language for analysis and design
- code skeletons are used to sketch out concepts and define class interfaces
- language-specific tools are used to annotate higher-level ideas and lower-level contracts

Java Tools

- structured Javadoc comments are used to annotate classes and features
- the Java Modeling Language (JML) is used to annotate the Java with formal models and contracts
- the JML tool suite and ESC/Java2 are used to runtime check contracts, unit test, and statically check code against specifications

Our Running Example

- we will use the CITIZEN/NOBLEPERSON examples from the BON book
- each chart is written as a Javadoc-annotated class skeleton
- each interface specification is written as a JML-annotated class skeleton
- the implementation is written in Java

Informal Charts: CITIZEN

CLASS	*CITIZEN*	**Part:** 1/1
TYPE OF OBJECT Person born or living in a country	**INDEXING** **cluster:** *CIVIL_STATUS* **created:** 1993-03-15 jmn **revised:** 1993-05-12 kw	
Queries	Name, Sex, Age, Single, Spouse, Children, Parents, Impediment to marriage	
Commands	Marry. Divorce.	
Constraints	Each citizen has two parents. At most one spouse allowed. May not marry children or parents or person of same sex. Spouse's spouse must be this person. All children, if any, must have this person among their parents.	

Informal Charts in Java: Citizen

```
/**
 * Person born or living in a country.
 *
 * @created 1993-03-15 jmn
 * @revised 1993-05-12 kw
 *
 */
package civil_status;

class Citizen {
  /** @bon Name? */
  ...
  /** @bon Marry. */
  ...
  /** @bon Each citizen has two
parents. */
}
```

Informal Charts: NOBLEPERSON

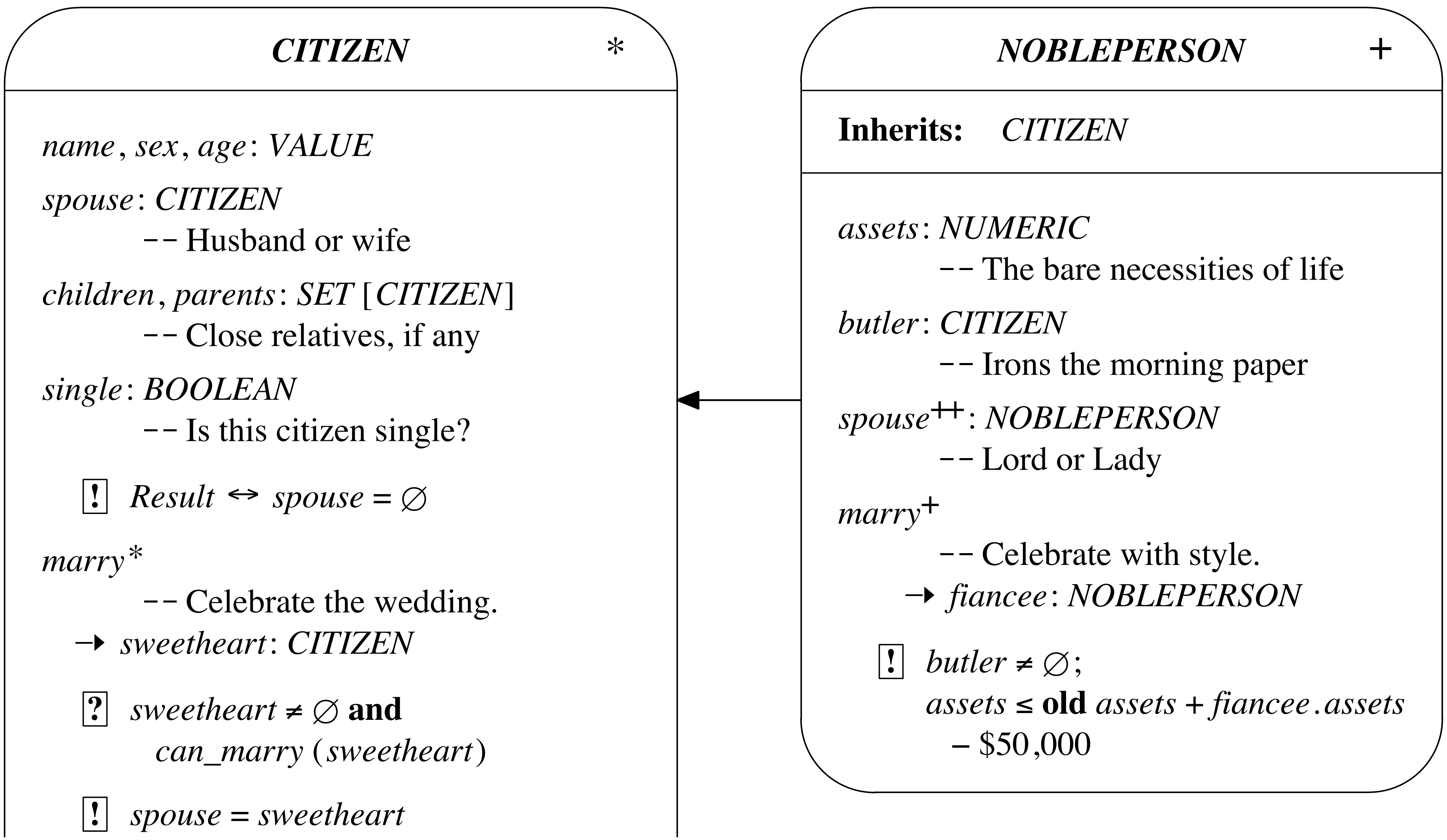
<table>
<tr><td>CLASS</td><td colspan="2">NOBLEPERSON</td><td>Part: 1/1</td></tr>
<tr><td colspan="2">TYPE OF OBJECT
Person of noble rank</td><td colspan="2">INDEXING
cluster: CIVIL_STATUS
created: 1993-03-15 jmn
revised: 1993-05-12 kw, 1993-12-10 kw</td></tr>
<tr><td>Inherits from</td><td colspan="3">CITIZEN</td></tr>
<tr><td>Queries</td><td colspan="3">Assets, Butler</td></tr>
<tr><td>Constraints</td><td colspan="3">Enough property for independence.
Can only marry other noble person.
Wedding celebrated with style.
Married nobility share their assets and must have a butler.</td></tr>
</table>

Informal Charts in Java: Nobleperson

```
/**
 * Person of noble rank.
 *
 * @created 1993-03-15 jmn
 * @revised 1993-05-12 kw, 1993-12-10 kw
 */
package civil_status;

class Nobleperson extends Citizen {
  /** @bon Assets? */
   ...
  /** @bon Enough property for independence. */
}
```

Formal Specification: Graphical BON



Formal Specification: Graphical BON

can_marry: *BOOLEAN*

-- No legal hindrance?

→ *other*: *CITIZEN*

? $other \neq \varnothing$

! $Result \rightarrow (single$ **and** $other.single$
and $other \notin children$
and $other \notin parents$
and $sex \neq other.sex)$

divorce

-- Admit mistake.

? $\neg\, single$

! *single* **and** (**old** *spouse*).*single*

Invariant

single **or** *spouse*.*spouse* = **@**;
$parents.count = 2$;
$\forall\, c \in children \bullet (\exists\, p \in c.parents \bullet p =$ **@** $)$

Formal Specification in BON: CITIZEN

```
deferred class CITIZEN
feature
    name, sex, age: VALUE
    spouse: CITIZEN                          -- Husband or wife
    children, parents: SET [CITIZEN]         -- Close relatives, if any
    single: BOOLEAN                          -- Is this citizen single?
        ensure
            Result <-> spouse = Void
        end
    deferred marry                           -- Celebrate the wedding.
        -> sweetheart: CITIZEN
        require
            sweetheart /= Void and can_marry (sweetheart)
        ensure
            spouse = sweetheart
        end
```

Formal Specification in JML: Citizen

```
abstract class Citizen {
  private Value name,sex,age;
  /** Husband or wife */
  private Citizen spouse;
  /** Close relatives, if any */
  private Set[Citizen] children, parents;
  /** Is this citizen single? */
  //@ invariant single <==> spouse == null;
  private boolean single;

  /** Celebrate the wedding. */
  //@ requires sweetheart != null;
  //@ requires can_marry(sweetheart);
  //@ ensures spouse == sweetheart;
  abstract void marry(Citizen sweetheart);
  ...
```

Formal Specification in BON: CITIZEN

```
can_marry: BOOLEAN                      -- No legal hindrance?
    -> other: CITIZEN
    require
        other /= Void
    ensure
        Result -> (single and other.single
            and other not member_of children
            and other not member_of parents
            and sex /= other.sex)
    end

divorce                                 -- Admit mistake.
    require
        not single
    ensure
        single and (old spouse).single
    end
```

Formal Specification in JML: Citizen

```
/** No legal hinderance? */
/*@ requires other != null;
  @ ensures \result <==> (single &
  @                        other.single &
  @                        !children.has(other) &
  @                        !parents.has(other) &
  @                        sex != other.sex);
  @*/
abstract boolean can_marry(Citizen other);

/** Admit mistake. */
/*@ requires !single;
  @ ensures single & \old(spouse.single);
  @*/
abstract void divorce();
```

Formal Invariant in BON and JML

invariant
    single **or** *spouse.spouse* = *Current*;
    parents.count = 2;
    for_all *c* **member_of** *children* **it_holds**
      (**exists** *p* **member_of** *c.parents* **it_holds** *p* = *Current*)
end -- class *CITIZEN*

```
/*@ invariant single | spouse.spouse == this; */
/*@ invariant parents.count == 2; */
/*@ invariant (\forall Citizen c; children.has(c);
  @           (\exists Citizen p; parents.has(p);
  @                                 p == this;)); */
```

Formal Spec in BON: NOBLEPERSON

```
effective class NOBLEPERSON
inherit
    CITIZEN
feature
    assets: NUMERIC                                  -- The bare necessities of life
    butler: CITIZEN                                  -- Irons the morning paper
    redefined spouse: NOBLEPERSON                    -- Lord or Lady
    effective marry                                  -- Celebrate with style.
        -> fiancee: NOBLEPERSON
        ensure
            butler /= Void;
            assets <= old assets + fiancee.assets – $50,000
        end
end -- class NOBLEPERSON
```

Formal Specification in JML: Nobleperson

```
class Nobleperson extends Citizen {
  /** The bare necessities of life. */
  Numeric assets;
  /** Irons the morning paper. */
  Citizen butler;
  /** Lord or Lady */
  //@ invariant \typeof(spouse) == \type(Nobleperson);

  /** Celebrate with style. */
  //@ ensures butler != null;
  //@ ensures assets <= \old(assets + fiancee.assets - 50000);
  void marry(Nobleperson fiancee) {
    //@ assert false;
  }
}
```

Part VI: Code Standards and Metrics

Code Standards

- the “look and feel” of development artifacts
- includes program code, docs, scripts, etc.
- primary focus is on improving team *communication* and *comprehension*
- team members focus their attention and spend time on *important* things—*not* code formatting or trivial design decisions
- helps with merging and maintenance
- standard are automatically <u>checked</u>

Structural Standards

- small-scale structure
  - code indentation
  - block placement
  - identifier naming
  - method ordering
- large-scale structure
  - package and module structuring
  - design patterns and anti-patterns

Example Use of Standard

```
class Citizen
{
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  Citizen my_spouse = null;
  //@ invariant (my_spouse != null) ==>
  //@           my_spouse.my_spouse == this;

  /** Constructs a new Citizen object who is single. */
  //@ ensures single();
  Citizen() {
    my_spouse = null;
  }
  ...
```

Some Basic Rules of Good Programming

- simple (even trivial!) constructors
- focus on data abstraction
  - appropriate levels of visibility
  - work from tight (private) to loose (public)
- short method signatures
- no globals and few static or class variables
- avoid concurrency at all costs

The KindSoftware Coding Standard

- the "gold standard" of coding standards
- used in dozens of companies and groups around the world
  - e.g,. influenced coding standard at Sun
- written as generic rules with specific application to Java and Eiffel
- http://kind.ucd.ie/documents/whitepapers/code_standards/

Metrics

- provide *quantitative* (but "fuzzy") analysis of software artifacts
- generated numbers mean *absolutely nothing* in almost all cases
  - *they are only valuable in a relative context*
- dozens (hundreds?) of metrics have been invented but *very* few are seriously used
- usually *the worst* metrics are the ones heard about most often (e.g., KLOC)

Standard Metrics

- lines of code (LOC, KLOC, MLOC)
  - effectively means "count the semicolons," *not* the curly braces
  - counts *real* statements, declarations, etc.
- lines of comments/docs (LOD, KLOD, etc.)
  - counts lines of *real* comments
  - count clauses or measure *information complexity* of documentation

Standard Non-Trivial Metrics

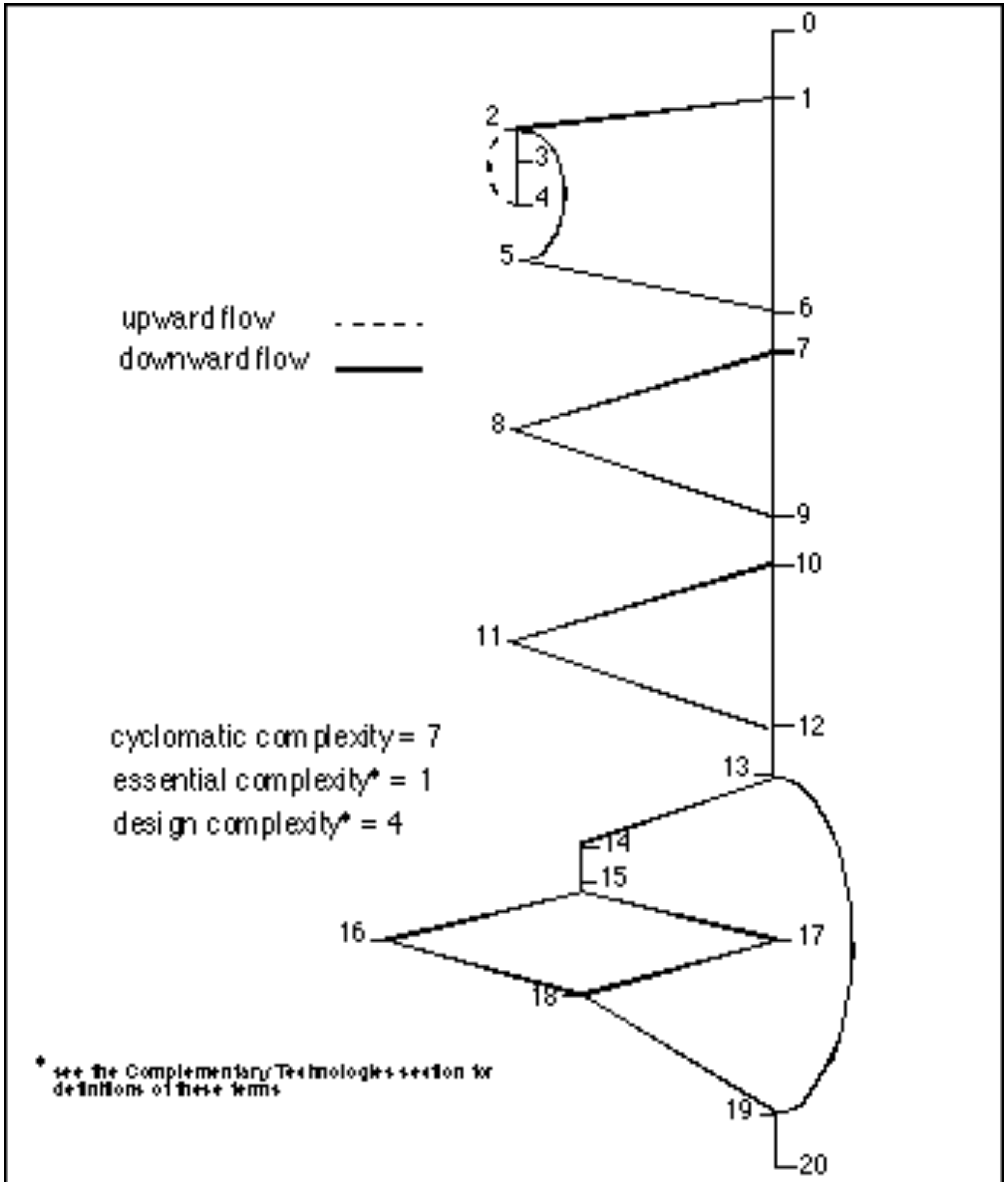
- cyclomatic code complexity
  - roughly counts the number of execution paths through code
  - $CC = E - N + 2p$, where

    E = the number of edges of the graph

    N = the number of nodes of the graph

    p = the number of connected components

CC Example



0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
upward flow
downward flow
cyclomatic complexity = 7
essential complexity* = 1
design complexity* = 4
* see the Complementary Technologies section for definitions of these terms

CC Evaluation

Cyclomatic Complexity	Risk Evaluation for Expert Programmers
1-10	a simple program, low risk
11-20	more complex, moderate risk
21-50	complex, high risk
>50	untestable, very high risk

Other Popular Metrics

Complexity Measure	Primary Measure of
Halstead	Algorithmic complexity, measured by counting operators and operands
Henry and Kafura	Coupling between modules (parameters, global variables, calls)
Bowles	Module and system complexity; coupling via parameters and global variables
Troy and Zweben	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier	Modularity of the structure chart

Doc and Spec Coverage

- documentation coverage
  - ensure all modules, methods, and attributes are documented appropriately
    - i.e., *no* Javadoc warnings whatsoever
- specification coverage—at least one...
  - invariant per attribute/field
  - precondition per method parameter
  - postcondition per method
  - assertion per branch in body

Unit Testing
Code Coverage

- desire that tests exercise all execution paths in your code
  - every branch, try/catch, switch case, etc.
- tools exist that measure code coverage while the program runs its unit tests
  - 100% coverage is ideal but rarely met
  - 80-90% coverage is realistic with effort

Popular Java Code Coverage Tools

- <u>Emma</u> - scalable bytecode instrumentation
  - included with Eclipse installed on server
- <u>Quilt</u> - extended classloader; optimized for JUnit, Ant, and Maven
- <u>Hansel</u> - extended classloader
- <u>Gretel</u> - bytecode recompilation
- <u>GroboUtils</u> - extended classloader

Simple Assessment of Software Quality

- ensure assessment in all programming-related assignments is *directly* coupled with these three forms of simple (sometimes static) checking
- system's code, docs, and specs *must* conform to the provided coding standard and metrics and coverage guidelines
- concrete guidelines are built-in to the environment and/or provided

Part VII: Static Analysis for Software Construction

Static Analysis

- static and dynamic are duals
- dynamic analysis means examining an artifact as it changes
  - e.g., watch a program as it executes
- static analysis means examining an artifact when it does not change, *in the context of its meaning and purpose*

Common Kinds of Static Analysis

- *typechecking*
- source code programming standards
- documentation standards
- metrics guidelines
- unit test coverage guidelines
- null pointer analysis
- checking for good programming idioms/ patterns and poor use of anti-patterns

Code Standard Example

Code Standard Example

```
class Citizen {
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  Citizen my_spouse;

  /** Returns a new citizen who is single. */
  Citizen();
  ...
```

Code Standard Example

```
class Citizen
{
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  Citizen my_spouse;

  /** Returns a new citizen who is single. */
  Citizen();
  ...
```

Code Standard Example

Documentation Example

```
/** The spouse of this Citizen; if null, this citizen
    is single. */
Citizen my_spouse;

/** Returns a new citizen who is single. */
Citizen();

/** @return this citizen’s name. */
String name();

/** Sets this citizen’s age.
 *  @param new_age the new age of this citizen.
 */
void age(byte new_age);
...
```

Specification Example

```
class Citizen
{
  /** The spouse of this Citizen; if null, this citizen
      is single. */
  /*@ nullable @*/ Citizen my_spouse = null;
  //@ invariant (my_spouse != null) ==>
  //@           my_spouse.my_spouse == this;

  /** Returns a new citizen who is single. */
  //@ ensures single();
  Citizen() {
    my_spouse = null;
  }
  ...
```

Trivial Static Checking

- lexical analysis only
- scan/lex source code
- typically keep only a small amount of contextual information
- check each construct on the fly
  - e.g., pattern match on strings

Syntactic Static Analysis

- scan and parse (parts of) a program
- generate AST for structures of interest
- walk over AST, pattern matching on interesting structures
- analyze each match for properties of interest, usually with a simple algorithm
- report results to user

Semantic Static Analysis

- scan, parse, and generate AST as before
- transform AST into an intermediate representation amendable to analysis
  - e.g., reduced language, guarded command language, static single assignment form
- analyze this representation semantically, generate verification conditions that logically express properties of interest
- give VCs to a theorem prover for checking
- interpret prover response for programmer

Some Static Checkers Included in PVE

- CheckStyle - source and docs style checker
- Metrics - source-based metrics analysis
- PMD - source-based good/bad patterns
- FindBugs - bytecode-based patterns
- EclEmma - unit test code coverage
- ESC/Java2 - common programming errors

Grading with Checkers

- project's are partially graded based upon how well documentation, specifications, and code pass static checkers
- essentially, always try to ensure that there are no errors or warnings
  - code conforms to specified style
  - metrics guidelines are followed
  - no PMD or FindBugs markers
  - no typechecking errors from JML checker
  - no warnings from ESC/Java2

Part VIII: Models are the 'M' in JML

Using ADT Models in
Formal Specification with JML

Models, not Modeling

- the ‘M’ in JML is not the same as the ‘M’ in UML, even if both use the term ‘model’
- JML models are mathematical abstractions
  - UML models are pretty pictures
- JML models are used to specify abstract behavior independent of implementation
- an implementation realizes a model and is verified as fulfilling the model

Standard Models

- standard mathematical models include:
  - bag, list, map, pair, relation, sequence, set
  - variants exist for values and objects
- standard Java models include:
  - Byte, Char, Double, Float, Integer, Long, Short, String, Type
  - Collection, Comparable, Enumeration, Iterator

Mathematical Models

- each model is realized by one Java class
  - see the package org.jmlspecs.models
- all methods of all models are functional
- each model has a full specification
  - spec is in OO/ADT style
  - algebraic equational axiomatic spec
- NB no models have been verified yet!

Java Models

- all core classes have models
- some of these models are quite simple (e.g., Byte, Char, Integer, and String)
- others are quite complicated

  (e.g., Double and Float)

Using Models

- models are used by declaring model fields
- one can also declare model methods
- in specifications, models are used in lieu of concrete fields when at all possible
- in implementations, models are bound to implementations with a represents clause
  - representations can be concrete fields or abstract pure method invocations

Example Models: JMLString

```
public /*@ pure @*/ class JMLString
  implements JMLComparable {

    /** The contents of this object. */
    //@ public model String theString;
    //@ public invariant theString != null;

    protected String str_;
    //@              in theString;
    //@ protected represents theString <- str_;

    //@ protected invariant str_ != null;
```

Example Models: JMLInteger

```
public /*@ pure @*/ class JMLInteger
  implements JMLComparable {

    /** The integer value of this object. */
    //@ public model int theInt;

    //@ public constraint theInt == \old(theInt);

    private int intValue;
    //@         in theInt;
    //@ private represents theInt <- intValue;
```

JMLInteger's remainderBy()

```
/**
 * Return a new object containing the remainder of
 * this object's integer value divided by that of
 * the given argument.
 */
/*@  public normal_behavior
  @    requires i2 != null && !i2.equals(new JMLInteger(0));
  @    ensures \result != null
  @         && \result.theInt == theInt % i2.theInt;
  @*/
public /*@ non_null @*/
    JMLInteger remainderBy(/*@ non_null @*/ JMLInteger i2) {
    //@ assume i2.intValue != 0;
    return new JMLInteger(intValue % i2.intValue);
}
```

Issues with Models

- awkward to use
  - all operators are functional and are methods, thus an unfamiliar prefix-notation is necessary
  - all mathematical models are parameterized on a type, but since Java <=1.5 has no parameterized classes, casting is frequent
- execution speed with jmlrac is very slow
  - particularly true of mathematical models

Verifying with Models

- models with built-in types and functional representations work in ESC/Java2
- small models with richer types and functional representations sometimes work
  - primarily complexity issue with Simplify
- medium to large models with richer types do not work at all
  - currently revising core specifications to match ESC/Java2's current capabilities