State-of-the-Art in Java Verification

Joe Kiniry
KindSoftware Research Group
Systems Research Group
Complex Adaptive Systems Laboratory (CASL)
School of Computer Science and Informatics
University College Dublin
Ireland

Acknowledgements

K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, Jim Saxe, Raymie Stata, Cormac Flanagan (while at DEC SRC); David Cok (Kodak); Carl Pully (ACME-Labs); Cees-Bart Breunesse, Arnout Engelen, Christian Haack, Ichiro Hasuo, Engelbert Hubbers, Bart Jacobs, Martijn Oostdijk, Wolter Pieters, Erik Poll, Joachim van den Berg, Martijn Warnier (RUN); Gilles Barthe, Julien Charles, Benjamin Grigore, Marieke Huisman, Clément Hurlin, and Mariela Pavlova, Gustavo Petri (INRIA); Cesare Tinelli, Jeg Hagen, and Alex Fuches (Univ. of Iowa); Aleksey Schubert (Univ. of Warsaw); Michal Moskał (Wroclaw University); David Naumann (Stevens); Patrice Chalin, Perry James, George Karabotos, Frederic Rioux (Concordia University); Torben Amtoft, Anindya Banerjee, John Hatcliff, Venkatesh Prasad Ranganath, Robby, Edwin Rodríguez, Todd Wallenstein (KSU), Yoonsik Cheon, Curtis Clifton, Gary Leaven, Todd Millstein (while at Iowa St.); Claudia Brauchli, Adam Darvas, Werner Dietl, Hermann Lehner, Ovidio Mallo, Peter Müller, Arsenii Rudich (ETHZ); Dermot Cochran, Lorcan Coyle, Steve Neely, Graeme Stevenson (UCD); and my PhD students Fintan Fairmichael, Robin Green, Radu Grigore, Mikoláš Janota, and Alan Morkan and undergraduate students Barry Denby and Conor Gallagher, and Patrick Tierney and the many students in my software engineering courses over the past decade

Dagstuhl in a Nutshell

Verification in Modern Software Development

- verification is not always necessary
- verification is rarely necessary
- your problem domain, team, customer, CEO/CTO matter more than technology
- when your systems have to have high quality, then various verfication techniques can have a role
- modern verification software has dramatically improved over the past ten years and is widely used in certain industries and in many universities

Some Personal Examples

- use in teaching and research in top universities (e.g., Caltech, MIT, CMU)
- but also use in teaching and research in universities not chock-full of geniuses :)
- CAD systems for VLSI
- financial systems on smart cards
- Dutch KOA and Irish Votáil tally systems

Best-practices in (Realistic) Modern Software Engineering

- modern, most-often-used best-practices in writing high-quality software
  - write documentation (at some point)
  - write unit tests (by hand), preferably prior to implementing functionality
  - focus on the code, write simple-but-good code, and refactor often
  - talk to your customer frequently

  - but... most developers are embarrased by, and not confident in, their code

An Example in Java

```
public class Purse {
  int balance;

  public int debit(int amount)
    throws PurseException;
}
```

An Example in Java

```
public class Purse {
  // The balance of this purse.
  int balance;

  // Decrease the balance of this account by
  // “amount”.
  public int debit(int amount)
    throws PurseException;
}
```

An Example in Java

```
public class Purse {
  // The (non-negative) balance of this purse.
  private int balance;

  // Decrease the balance of this account by
  // “amount”.  Return the new balance of the
  // purse.  Throw an exception if something
  // went wrong.
  public int debit(int amount)
    throws PurseException;
}
```

An Example in Java

```
public class Purse {
  // The (non-negative) balance of this purse.
  private int balance;

  /** Decrease the balance of this account by
   * “amount”.  Return the new balance of the
   * purse.  Throw an exception if something
   * went wrong. */
  public int debit(int amount)
    throws PurseException;
}
```

An Example in Java

```
public class Purse {
  // The (non-negative) balance of this purse.
  private int balance;

  /** Decrease the balance of this account.
   * @param amount the non-negative amount
   * of funds of the debit.
   * @return the new balance of the purse.
   * @exception PurseException is thrown if
   * something went wrong.
   */
  public int debit(int amount)
    throws PurseException;
}
```

Unit Tests for Purse

- hand-written unit tests are necessary for explaining, documenting, and testing what happens when:
  - the field balance is non-negative
  - the field balance is negative
  - the amount passed is positive
  - the amount passed is negative
  - the amount passed is zero
  - one or more situations when an exception is thrown

  - this results in about 2 pages of test code

Ambiguity and Evolvability of this Trivial Method

- recall that advocates of modern practices suggest that refactoring happens often
  - but every time you refactor the method you must rewrite all unit tests that mention it
- as the method's purpose and meaning evolves, then so must its documentation
  - but there is very little connection between the English documetation and the method
- and there are still many open questions
  - e.g., what happens to balance when an exception is thrown? can more than the balance of the object be modified? what is the maximum balance possible?

“Gently” Using JML

```
// The (non-negative) balance of this purse.
private int balance;

/** Decrease the balance of this account.
 * @param amount the non-negative amount
 * of funds of the debit.
 * @return the new balance of the purse.
 * @exception PurseException is thrown if
 * something went wrong. */
public int debit(int amount)
  throws PurseException;
```

“Gently” Using JML

```
final int MAX_BALANCE;
/*@ invariant 0 <= balance &
              balance <= MAX_BALANCE; */
private int balance;

/** Decrease the balance of this account.
 * @param amount the non-negative amount
 * of funds of the debit.
 * @return the new balance of the purse.
 * @exception PurseException is thrown if
 * something went wrong. */
public int debit(int amount)
  throws PurseException;
```

“Gently” Using JML

```
final int MAX_BALANCE;
/*@ invariant 0 <= balance &
              balance <= MAX_BALANCE; */
private int balance;

/** Decrease the balance of this account.
 * @return the new balance of the purse.
 * @exception PurseException is thrown if
 * something went wrong. */
//@ requires amount >= 0;
public int debit(int amount)
  throws PurseException;
```

"Gently" Using JML

```
final int MAX_BALANCE;
/*@ invariant 0 <= balance &
              balance <= MAX_BALANCE; */
private int balance;

/** Decrease the balance of this account.
 * @return the new balance of the purse.
 * @exception PurseException is thrown if
 * something went wrong. */
//@ requires amount >= 0;
//@ assignable balance;
public int debit(int amount)
  throws PurseException;
```

“Gently” Using JML

```
final int MAX_BALANCE;
/*@ invariant 0 <= balance &
                balance <= MAX_BALANCE; */
private int balance;

/** Decrease the balance of this account.
 * @exception PurseException is thrown if
 * something went wrong. */
//@ requires amount >= 0;
//@ assignable balance;
//@ ensures balance == \old(balance-amount);
//@ ensures \result == balance;
public int debit(int amount)
  throws PurseException;
```

“Gently” Using JML

```
/** Decrease the balance of this account. */
//@ requires amount >= 0;
//@ assignable balance;
//@ ensures balance == \old(balance-amount);
//@ ensures \result == balance;
//@ signals (PurseException)
//@         balance == \old(balance);
public int debit(int amount)
  throws PurseException;
```

What Does This “Gentle” Use of JML Buy You?

- write less English documentation
- less need to keep English documentation in-sync with code as it evolves
- eliminate defensive programming boilerplate in method implementation
    - parameter validity checking & exceptions
- write no inline assertions
- write and babysit **no** unit tests
- statically check that code fulfills spec automatically with several tools

Demonstration