# CSE 331
# SOFTWARE DESIGN & IMPLEMENTATION
# MIDTERM REVIEW

Autumn 2011

## The other kind of testing…

2

- Actually, it's the same as software testing (mostly)
- By picking effective subdomains, I hope to determine how likely it is that you understand the material – it's inherently sampling, not proof
- In this situation, a single test suite will be executed across 56 different processors

CSE 331 Autumn 2011

## The form of the test: Subject to some (but limited) change

3

- Part I: True/false with a brief justification
  - 5-10 questions
- Two examples from last year
  - *"hashCode can be determined at most once – that is, only when it is first actually requested by a client and then it can be cached."*
  - *"If an immutable object throws an exception, it is never left in an undesirable or indeterminate state."*

**In small groups, spend two minutes discussing these**

CSE 331 Autumn 2011

## Part II: Testing

4

- More than on last year's exam – now about 20% of the total points
  - Don't worry about white-box testing, as we haven't yet covered it
- Likely kinds of questions
  - In what ways is/is not unit testing like <some other kind of testing like system testing or acceptance testing or …>? You will not need a deep understanding of these other kinds of testing
  - Write some black-box tests for a given specification and describe what subdomains they are intended to address

CSE 331 Autumn 2011

## Part III: Specifications

5

- (Not entirely distinct from the next Part on ADTs)
- Likely kinds of questions
  - Infer a *likely* specification (requires/modifies/etc.) from a piece of code
  - Given a specification, provide an implementation that is almost surely *not* what the specification intended
  - ??

CSE 331 Autumn 2011

## Part IV: ADTs

6

- Likely kinds of concepts to be tested
  - Is one ADT specification a true subtype of another ADT specification? Some variant of the following question from last year

**1)** (20 points) For each of the five specifications below, mark a + in the box if specification X (down the left column) is stronger than specification Y (across the top row), a – in the box if specification X is weaker than specification Y, an = in the box if they are equivalent, and a • in the box if they are none of +, - or =. You only need to fill in the empty boxes in the lower-left triangular area.

| | $\alpha < \beta$ | $\alpha \leq \beta$ | $\alpha \leq \beta \mid \alpha < \beta$ | $\alpha \leq \beta$ & $\alpha < \beta$ | $\alpha \neq \beta$ |
|---|---|---|---|---|---|
| $\alpha < \beta$ | = | | | | |
| $\alpha \leq \beta$ | | = | | | |
| $\alpha \leq \beta \mid \alpha < \beta$ | | | = | | |
| $\alpha \leq \beta$ & $\alpha < \beta$ | | | | = | |
| $\alpha \neq \beta$ | | | | | = |

**In small groups, spend two minutes discussing this example question**

## Part IV: ADTs continued

7

- Likely kinds of concepts to be tested
  - What is, or is wrong with, a representation invariant for a given class?
  - Is there representation exposure? How might you fix it?
  - Relationship of representation invariants and abstraction functions

Computers store multi-byte information either in big-endian or little-endian form. Consider an integer stored in two bytes (16 bits total). In a big-endian machine, the first byte represents the first eight bits of the integer and the second byte the second eight bits of the integer. These bytes are reversed on little-endian machines. So, in the example below, on a big-endian machine this would represent the integer $2^{11} + 2^9 + 2^2 + 2^0$ equaling 2048+512+4+1 = 2565. On a little-endian machine, this would represent the integer $2^{10} + 2^8 + 2^3 + 2^1$ equaling 1024+256+8+2 = 1290.

| | First byte | | | | | | | | Second byte | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Big-endian<br>Little-endian | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

1) (10 points) Is interpreting the numeric value of big- vs. little-endian two-byte integers more related to the notion of *abstraction function* or of *representation invariant*. Briefly justify (use a maximum of two sentences).

**In small groups, spend two minutes discussing this example question**

## Part V: Miscellaneous

8

- This part might not be included
- And if included, I'm not yet sure what it will be



## Per lecture: points to focus upon

### But others are fair game still

9

- Lecture 1 – introduction
- Programs (implementation) satisfying specifications
  - It's tricky business
  - It's a many-to-many mapping
- No notion of a "correct" specification
  - Some can surely admit implementations that are highly unlikely to be desired



## Lecture 2 – specifications

10

- The value of specifications in addressing complexity in software
- The dual roles of client and implementer
  - What does the client depend upon?
  - What does the implementer need to provide?
  - Why is a specification useful for this?
- Why not just read code? Just use documentation? Just use Java interfaces? Etc.?
- Javadoc and the 331 extensions to it



## Lecture 3 – testing

11

- Testing is one form of quality assurance for software
- Testing terminology – pass, fail, test case, test suite, …
- General notion of kinds of testing
- Subdomains
- JUnit's role – what can it help you do and not do?



## Lecture 4 – equality

12

- Different notions of equality
- Key underlying properties of (any useful) equality
- Relationship of `equals` and `hashCode`
- Overriding vs. overloading

## Lectures 5 and 6 – ADTs

13

- Motivations for the use of ADTs
- Primary focus on ADT operations rather than representations
  - Different kinds of ADT operations (observers, mutators, etc.) and differences between mutable and immutable ADTs
  - Hide the implementation decisions to allow change
- Abstraction function – what is it, why is it important, how is it used?
- Representation invariant – what is it, why is it important, how is it used?
- The relationship between the AF and RI, the ADT and its implementation (that diagram)
- Representation exposure – what is it, how to eliminate it?



## Lecture 7 – subtyping & subclassing

14

- A way to share behaviors and/or code
- Weaker and stronger specifications – and the relationship to satisfying implementations
- True subtyping vs. Java subtyping – allowing substitutability
- Subtyping is over specifications; subclassing is over implementations – both use similar mechanisms in Java
- Mutability can be useful, but can confuse the issue of true subtyping



## Lecture 8 – modular design principles

15

- Cohesion (why together?) and coupling (how do modules interact?)
- Different kinds of dependences – invokes, names, extends, etc.
- Ways to manage dependences – e.g., Law of Demeter
- Module dependence diagrams (largely to identify coupling)



## Lecture 9 – design style

16

- Long list of good things to do in coding
- Method, field, constructor design
- Naming
- Class design ideals (cohesion, coupling, clarity, etc.)
- Documentation
- Invariants
- `static` vs. non-static; `public` vs. `private`; etc.
- Selecting types
- Independence of views



## Lectures 10-12

17

- Design patterns, basic GUI
- Not a focus of this test – will be fair game on the final