Exam Review and Persistent Data

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Midterm Friday: first, a review; then, persistent data
- Covers through Wednesday, November 4: equality, not generics
- The week's sessions and deadlines are at the end

The midterm

- **When**: Friday, November 13, in lecture, here in SAV 260
- **How long**: 50 minutes
- **Bring**: your UW ID, ready to be checked
- **One sheet of notes**: letter size, handwritten, both sides
- Otherwise closed notes, and no electronics

The course so far

Your job is to deliver code that you know works.

- **Deliver, to whom?** Clients, who rely on the specification alone
- **Know, how?** Testing, tools such as the compiler, and reasoning
- **Works, meaning what?** It satisfies its specification

The lectures, by those questions

- **Works**: specifications, stronger and weaker (Lecture 2)
- **Know**, by testing: tests and their coverage (Lectures 4, 5)
- **Know**, by reasoning: Hoare triples (Lecture 7)
- **Know**, by reasoning: loop invariants (Lecture 8)
- **Deliver**: abstract data types (Lectures 10, 11)
  - Representation invariants and abstraction functions
- **Deliver**: behavioral subtypes (Lecture 14), equality (Lecture 16)

Practice: the first negative

```
{{ P: i = 0 }}
{{ Inv: I }}
while (i < n && 0 <= a[i]) {
    i = i + 1;
}
{{ Q: 0 <= i <= n and a[0..i] contains no negative values
      and (i = n or a[i] < 0) }}
```

- `n` is the length of `a`; `a[0..i]` is `a`'s first `i` elements
- `P`: the claim before the loop; `b`: its condition; `Q`: the claim after
- Which of the three checks pass, for each candidate `I`?

The first negative: three candidates

Candidates for `I`, lettered as on the Exams page:

B. `0 <= i <= n`

C. `0 <= i <= n and a[0..i] contains only positive values`

E. `0 <= i < n and a[0..i] contains no negative values`

The three checks, from Lecture 8:

(1) `P` implies `I`.

(2) The triple `{{ I and b }} body {{ I }}` is valid.

(3) `I and not b` implies `Q`.

- In checks (2) and (3), assume only what `I` says

The first negative: candidate B

```
{{ P: i = 0 }}
{{ Inv: 0 <= i <= n }}
while (i < n && 0 <= a[i]) {
    i = i + 1;
}
{{ Q: 0 <= i <= n and a[0..i] contains no negative values
      and (i = n or a[i] < 0) }}
```

(1) P implies I: **Y**

(2) {{ I and b }} body {{ I }} is valid: **Y**

(3) I and not b implies Q: **N**

a = {-1, -2}, i = 1: I and not b are true, Q false

The first negative: candidate C

```
{{ P: i = 0 }}
{{ Inv: 0 <= i <= n and a[0..i] contains only positive values }}
while (i < n && 0 <= a[i]) {
    i = i + 1;
}
{{ Q: 0 <= i <= n and a[0..i] contains no negative values
      and (i = n or a[i] < 0) }}
```

(1) P implies I: **Y**

(2) {{ I and b }} body {{ I }} is valid: **N**

a = {0}, i = 0: then a[0..1] contains 0, not positive

(3) I and not b implies Q: **Y**

The first negative: candidate E

```
{{ P: i = 0 }}
{{ Inv: 0 <= i < n and a[0..i] contains no negative values }}
while (i < n && 0 <= a[i]) {
    i = i + 1;
}
{{ Q: 0 <= i <= n and a[0..i] contains no negative values
      and (i = n or a[i] < 0) }}
```

(1) `P` implies `I`: **N**: if `a` is empty, `i < n` is false

(2) `{{ I and b }} body {{ I }}` is valid: **N**

`a = {0}`, `i = 0`: after the body, `i = n`, not below `n`

(3) `I and not b` implies `Q`: **Y**

What an exam question looks like

- A setup, then a grid: one row per item, one box per answer

Candidate `I`	Check (1)	Check (2)	Check (3)
B. `{{ 0 <= i <= n }}`	Y	Y	N

- In each box, write only the answer: here, **Y** or **N**
- Write it clearly, inside its box
  - An answer that could be read two ways can earn 0 for its question
- Do scratch work outside the boxes

Behavioral subtype

Definition: Behavioral subtype

A type B is a **behavioral subtype** of a type A if:
every object that satisfies B's specification also satisfies A's.
In other words: for each method of A, that method exists in B, and
the B version's specification is at least as strong as the A version's.

- At least as strong: precondition no stronger, postcondition no weaker

Practice: counters

```
/** @modifies this
 *  @effects increases the value by exactly 1 */
public void increment() { n++; }
```

```
/** @requires value() < limit
 *  @modifies this
 *  @effects increases the value by exactly 1 */
public void increment()
```

- Is `BoundedCounter` (below) a behavioral subtype of `Counter` (top)?
  - **No**: its precondition is stronger, `value() < limit`

Practice: counters

```
/** @modifies this
 *  @effects increases the value by 1 if below the limit;
 *       otherwise leaves it unchanged */
public void increment() { if (n < limit) n++; }
```

```
/** @requires value() < limit
 *  @modifies this
 *  @effects increases the value by exactly 1 */
public void increment()
```

- Is `SaturatingCounter` (top) a behavioral subtype of `BoundedCounter`?
  - **Yes**: a weaker precondition, and below the limit it adds 1

Practice: counters

```
/** @requires value() < limit
 *  @modifies this
 *  @effects increases the value by exactly 1 */
public void increment()
```

```
/** @modifies this
 *  @effects increases the value by 1 if below the limit;
 *       otherwise leaves it unchanged */
public void increment() { if (n < limit) n++; }
```

- Is `BoundedCounter` (top) a behavioral subtype of `SaturatingCounter`?
  - **No**: its precondition is stronger, as against `Counter`

The book forgets

```
$ curl -d 'name=ada&message=hello' http://localhost:9331/api/messages
["ada: hello"]
$ curl http://localhost:9331/api/messages
[]
```

- Our JavaScript guestbook: post, then stop it (Ctrl+C) and start it again
- The book is a variable, in the program's **memory**: it ends with the run
- A **file** on the disk outlasts the run

Data that outlasts the run

Definition: persistent

Data is **persistent** if it outlasts the run of the program that wrote it.

How long can a mistake in the state last?

- **Batch**: one run; the next run starts clean
- **Interactive**: one session; quitting and starting again clears it
- **Persistent**: as long as the data is kept
- With persistence, your bugs are forever
- The motto: what does "works" mean now, and how will we *know*?

What "works" means now

The promise

When the server responds to a post request,
that message will be saved forever.

- Forever: through every stop, such as an upgrade of the system
- So we save the book to a file, and read it back when the server starts

Loading and saving the book

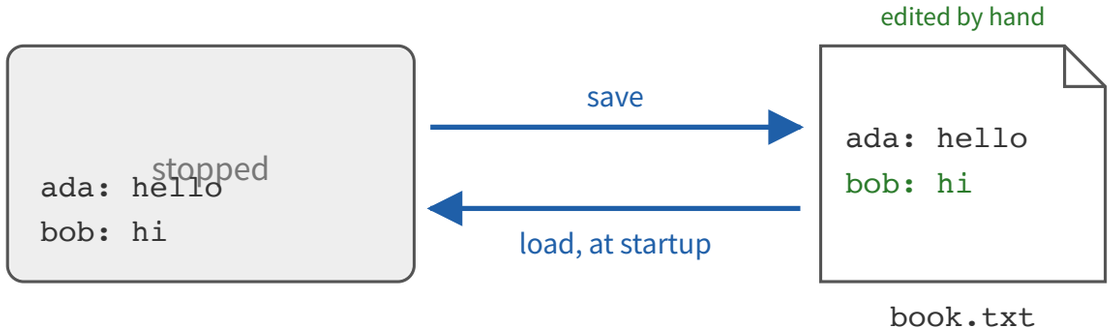
In the constructor, at startup:

```
128   if (Files.exists(file)) {
129       messages.addAll(Files.readAllLines(file, UTF_8));
130   }
```

In save, which add calls on every post:

```
154   try (FileOutputStream out = new FileOutputStream(file.toFile())) {
155       out.write(text().getBytes(UTF_8));
156   }
```

Does it work?



- **Saving**: ada posts, and `save` writes the book to `book.txt`
- **Loading**: stop the server, add a line by hand, start it again

Crashes

- It works. But the promise says "forever": what about a **crash**?
- A **crash**: the program stops between two of its steps
  - A bug, the operating system stopping it, a power loss
- The promise covers only posts that got a response
  - A post that crashed got none: the book may have it or not
- So: save the book first, and only then respond

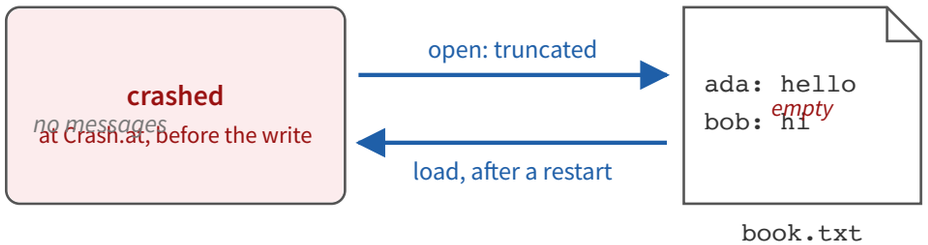
Half sheet, question 1 (3 minutes)

```
154   try (FileOutputStream out = new FileOutputStream(file.toFile())) {
155       out.write(text().getBytes(UTF_8));
156   }
```

- Can a crash in `save` lose a message whose post got a response?
- If so: where is the crash, and which messages are lost?

Injecting a crash, for testing

```
159   try (FileOutputStream out = new FileOutputStream(file.toFile())) {
160       Crash.at("opened");
161       out.write(text().getBytes(UTF_8));
162   }
```



- **Fault injection**: a **crash point**, `Crash.at`, added for testing

A crash at opened

```
$ curl -d 'name=cy&message=hey' http://localhost:9331/api/messages
curl: (52) Empty reply from server
$ curl http://localhost:9331/api/messages
[]
```

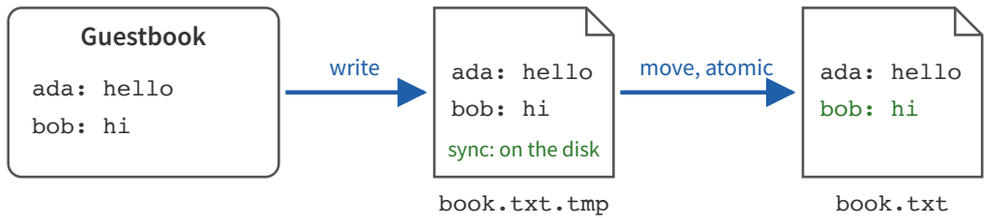
- Cy got no response: allowed
- **After a restart, what does the book hold?**
- Ada's and bob's messages are gone, and both were in the book
  - `book.txt` is empty: truncated on open, crashed before the write

What testing told us

- The file is a new input and a new output: a test chooses it and checks it
- Every normal-case test passes: save, load, restart
- The bug shows only on a crash inside `save`, at a point we chose
- A test of a crash tells us about the one point it crashes at
- **Testing** refutes the promise; it cannot confirm it

Old book or new book

```
159   Path temp = Path.of(file + ".tmp");
160   try (FileOutputStream out = new FileOutputStream(temp.toFile())) {
161       Crash.at("opened");
162       out.write(text().getBytes(UTF_8));
163       out.getFD().sync();
164   }
165   Crash.at("written");
166   Files.move(temp, file, ATOMIC_MOVE);
```



Argue, at every point

The claim, at every point in save

`book.txt` holds a whole book: the old one or the new one.

- **Before the move**: only `book.txt.tmp` changes, so `book.txt` is old
- **At the move**: it swaps old for new in one step
- **After the move**: `book.txt` is the new file, whole
- A crash anywhere leaves a whole book: **reasoning** covers every point

Leaning on the move

- The whole argument rests on one step: the move must be **atomic**
- **Atomic**: before the move or after it, never partway
- `ATOMIC_MOVE` asks the operating system for exactly that
- So the argument is only as good as the system's promise

The crash tests agree

```
$ curl -d 'name=bob&message=hi' http://localhost:9331/api/messages
curl: (52) Empty reply from server
$ curl http://localhost:9331/api/messages
["ada: hello"]
```

- The same at `opened` and at `written`: the old book
- After `written`, `book.txt.tmp` holds ada and bob; `book.txt` holds ada
- The tests pass at two points; the argument covers every point

Half sheet, question 2 (3 minutes)

```
159   Path temp = Path.of(file + ".tmp");
160   try (FileOutputStream out = new FileOutputStream(temp.toFile())) {
161       Crash.at("opened");
162       out.write(text().getBytes(UTF_8));
163       out.getFD().sync();
164   }
165   Crash.at("written");
166   Files.move(temp, file, ATOMIC_MOVE);
```

- Delete `out.getFD().sync()`: every crash test still passes
- Is that call needed? Why or why not?

A crash of the machine

- Delete `out.getFD().sync()`: every run comes out the same
- `halt` crashes the program; the system still writes the bytes later
  - A power loss loses bytes still waiting
- `sync` returns once the bytes are on the disk, before the move
- Reasoning from what `sync` promises puts that call in the code
- So can a test on LazyFS, which drops unsynced bytes on command

LazyFS: Ramos et al., *When Amnesia Strikes*, VLDB 2024

Three ways to know, for persistence

- **Testing**: the file is a new input and output; crash at chosen points
- **Reasoning**: one claim about every state a crash could leave the files in
- **Tools**: ALICE checks every state of the files a crash could leave
- A **database** is a tool too: it keeps the promise for us

ALICE: Pillai et al., *All File Systems Are Not Created Equal*, OSDI 2014

Databases

- Our `save` writes the whole book on every post: a million lines for one
- Write only the new line? That changes the only copy, in place, again
  - The move swaps whole files: it cannot swap in one line
- A **database** keeps data in files, and promises what we promised
  - It changes only what changed, safely: SQLite keeps a journal
- Plus many writers at once, and questions ("every message by ada")
- More in CSE 344

Coming up

- **Tuesday**: section on equality and generics
- **Tuesday**, 4:30 to 6:20pm, CSE2 G01: a mock midterm
- **Wednesday**: no lecture, Veterans Day
- **Thursday**, 4:30 to 6:20pm, CSE2 G01: a midterm review session
- **Friday**: the midterm, in lecture
- The Exams page: try each question before its answers
- Hand in your half sheet at the door