

CS 3

Introduction to Software Engineering

9: Specifications (II); Validation;
Verification

Last Time

- Specifications
 - Where they come from.
 - How to interpret them.
 - How to write good ones.
- Specificand Set
 - Set of (potential) implementations that (would) satisfy spec.
- Restrictiveness
- Generality
- Specifications define correctness.
 - Of implementations.
 - Of clients.

Clarity

- Specs are written for human readers.
 - Avoid ambiguous phrases.
 - Explain anything potentially confusing and/or support with examples.
 - Redundancy is okay, since it helps finding mistakes.

Redundancy

/* Returns true if and only if this is a subset of S. */

Redundancy

/* Returns true if and only if this is a subset of S. */

What happens if not? Exception? Infinite loop?

Redundancy

/* Returns true if this is a subset of S, false if not. */

Not actually redundant!

Redundancy

/* Returns true if this is a **subset** of S, false if not. */

Does subset mean subset, or *proper* subset?

Redundancy

/* Returns true if this is a **subset** of S, false if not. */

Does subset mean subset, or *proper* subset?

Subset means subset.

But people might get confused anyway.

Redundancy

```
/* Returns true if this is a subset of S, false if not.  
 * That is, returns true if every element of this is  
 * an element of S, and false otherwise.  
 */
```

Better.

Redundancy and Errors

```
/* @param t the temperature in deg. Celsius.  
 * @return true if t is less than 32, false if not.  
 */
```

(Maybe a little suspicious, but could be useful...)

Redundancy and Errors

```
/** Test whether a temperature is below freezing.  
 * @param t the temperature in deg. Celsius.  
 * @return true if t is less than 32, false if not.  
 */
```

Oops!

Redundancy and Errors

```
/** Test whether a temperature is below freezing.  
 * @param t the temperature in deg. Celsius.  
 * @return true if t is less than 32, false if not.  
 */
```

Spec tries to say the same thing two ways, really says two different things.

- Which one accurately describes the code?
- Which one was intended?

Redundancy and Jargon

/* Returns the **mgu** of s and t. */

TermSubst **mgu**(Term s, Term t) { ... }

I bet you have no idea what this is.

Redundancy and Jargon

```
/* Returns the most-general unifier of s  
 * and t. */
```

```
TermSubst mgu(Term s, Term t) { ... }
```

Probably not much better.

Redundancy and Jargon

```
/* Returns the most-general unifier of s and t. That is,  
 * returns theta such that theta(s) = theta(t) and for any phi  
 * where phi(s) = phi(t) there exists a psi such that  
 * phi(s) = psi(theta(s)) and phi(t) = psi(theta(t)).  
 */
```

```
TermSubst mgu(Term s, Term t) { ... }
```

Does that help?

If you were working on this program, it might.

Guidelines

- Start with most concise description.
- Some parts may need clarification:
 - Potentially ambiguous terms (e.g., “subset”)
 - Obscure or application-specific technical terms (e.g., “most-general unifier”)
 - Arbitrary or unmotivated properties (e.g., $t < 32$)

Clarify by offering explanations and/or examples.

- Clear specifications make it easier to:
 - Judge correctness of implementation.
 - Implement clients that use the abstraction appropriately.
 - Notice bugs in the specification.

Validation

- Validation: “a process designed to increase our confidence that a program will function as we intend it to.” (Liskov, p. 221)

Validation

- Validation: “a process designed to increase our confidence that a program will function **as we intend it to.**” (Liskov, p. 221)
- Does it meet its specification?
- Does it satisfy the customer’s needs?
 - If not, may need to respecify.

Validation

- Validation: “a process designed to **increase our confidence** that a program will function as we intend it to.” (Liskov, p. 221)
- Does it meet its specification?
- Does it satisfy the customer’s needs?
 - If not, may need to respecify.
- Perfect guarantees usually not possible.

Validation

- Two main approaches:
 - Verification: Analyze program, specification, try to *prove* correctness.
 - That is, argue program works for *all* possible inputs.
 - If unsuccessful, try to find **counterexample**.
Analyze case that *doesn't* work to track down source of problem.
 - Testing: Run program on several inputs.
 - Construct **test cases** to exercise as much of program as possible.
 - Test individual modules in isolation (**unit testing**).
 - Test groups of modules (**integration testing**).
 - Let customer provide tests (**acceptance testing**).

Formal Verification

- We've seen that types can be like specifications.
- Variable's type tells you what you can expect from the object.
- Types are subject to rules enforced by the compiler.
 - You have to write types that say what you mean.
 - Your functions must live up to their types' promises.
- Spec details in comments express what types can't.
- But have we reached the limit of what we can check with mechanized tools?

Formal Verification

- Can computers help check correctness?
- Idea:
 - Formalize all pre-/postconditions in symbolic logic. (Say, classical first-order predicates.)
 - Formalize program (and maybe language semantics).
 - Ask automated verification tool,
“Does this program satisfy this spec?”
Program answers yes or no (or runs out of time/space).
Maybe provides proof or counterexample.
- This is an active research area.
- Produced some useful (even profitable) tools, but can't replace informal reasoning and testing (yet).

ESC/Java Demo