CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
SUBTYPING AND SUBCLASSING

Autumn 2011

Very quick 331 recap

2

- Procedural specification and implementations that satisfy these specifications
  - For specification **S** and program **P**, `P satisfies S iff`
    - Every behavior of **P** is permitted by **S**
    - "The behavior of **P** is a subset of **S**"
- Abstract data type specification and implementations that satisfy such specifications – more complicated, but the same idea
- These are approaches for defining, reasoning about, testing and implementing software that satisfy specific expectations

UW CSE331 Autumn 2011

Similarity

3

- Sometimes it is valuable to take advantage of existing specifications and/or implementations to develop a *similar* piece of software
- That is, we'd like to develop a similar artifact (specification or implementation) not entirely from scratch, but rather as a delta from the original
  - **A' = A + ΔA'**
- Describing the differences and sharing the similarities can simplify development, increase confidence in the properties of the artifact, help in understanding the problem space, etc.

UW CSE331 Autumn 2011

Similarity in the world

4

- Philosophers including Plato, Aristotle, Hegel and others have discussed this for millennia – often in the context of equality/identity
- In what way are two chairs similar? How does a child recognize a (new kind of) chair?
- Why are platypi mammals even though they lay eggs instead of bearing live offspring?
- Should we classify species using taxonomies (like Linnaeus) or phylogenetics (like DNA)?

UW CSE331 Autumn 2011

Similarity in software development

5

- The field has many ways to exploit this notion of similarity – examples include
  - Procedures with parameters – share the algorithm, differ in the data
  - Object-oriented subclassing
  - Object-oriented subtyping
  - Monads in functional programming
  - And many more…
- Just like similarity is confusing in the world, it can be confusing – but very valuable – in software development

These are related but distinct; and the distinctions are often confusing and confused

UW CSE331 Autumn 2011

Substitutability

6

- The notion of satisfiability considered when an implementation met the expectations of a specification
- *Substitutability* will be the key issue in subtyping – can one specification (and its satisfying implementation) be substituted for another specification (and its satisfying implementation)?

UW CSE331 Autumn 2011

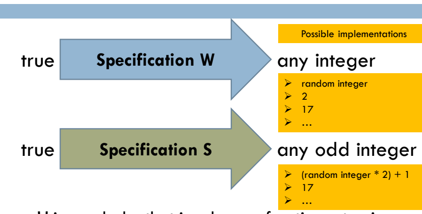
Comparing specifications

7

- A core notion underlying substitutability is the notion of comparing two specifications
- Specification: a stronger specification (**S**) can always be substituted for a weaker specification (**W**)
  - The stronger spec **S** is defined over a (possibly proper) superset of **W**'s inputs and returns a (possibly proper) subset of **W**'s outputs – as **S** includes all of **W**'s behaviors, it will work wherever **W** works
- Implementation: A procedure (**P**) satisfying a stronger specification (**S**) can be used anywhere that a weaker specification (**W**) is required
  - **P** satisfies **S** and **S** works wherever **W** works, so **P** also satisfies **W**

Example: weaker/stronger

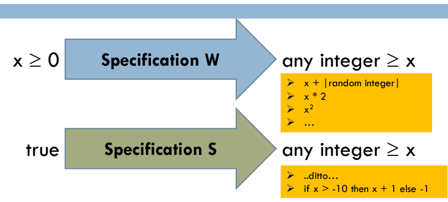
8



- Wherever **W** is needed – that is, where a function returning any integer will suffice – **S** will work because it returns an integer as **W** promises
- **W** cannot substitute for **S**, because of the expectation that **S** produces an odd integer, which **W** might not do

Example: weaker/stronger

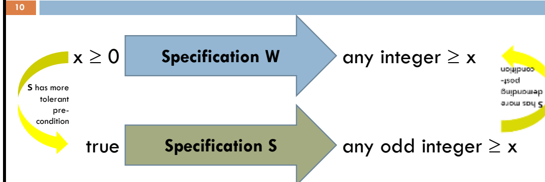
9



- A client depending on **W** can depend on **S**, because whenever **W**'s precondition is satisfied, so is **S**'s precondition

Example: weaker/stronger

10

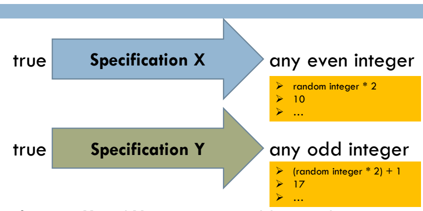


- Stronger specifications are
  - More tolerant on the inputs
  - But more demanding on the outputs
- Weaker specifications are
  - More demanding on the inputs
  - But more tolerant on the outputs

Do not mistake strong/weak as good/bad or as bad/good

Example: incomparable

11



- The specifications **X** and **Y** are incomparable – neither is stronger or weaker than the other one
- A client of either cannot substitute the other and still work in general

Said another way…

12

- A stronger specification is
  - **harder to satisfy** (implement) because it promises more – that is, its `effects` clause is harder to satisfy and/or there are fewer objects in `modifies` clause – but
  - **easier to use** (more guarantees) by the client – that is, the `requires` clause is easier to satisfy
- A weaker specification is
  - **easier to satisfy** (more implementations satisfy it) because it promises less – that is, the `effects` clause is easier to satisfy and/or there are more objects in `modifies` clause – but
  - **harder to use** (makes fewer guarantees) because it asks more of the client – that is, the `requires` clause is harder to satisfy

What about subtyping?

- Subtyping uses substitutability to express the "is-a" relationship
  - A circle is-a shape; a rhombus is-a shape
  - A platypus is-a mammal; a mammal is-a vertebrate animal
  - A `java.math.BigInteger` is-a `java.lang.Number` is-a `java.lang.Object`
- When a programmer declares **B** to be a *subtype* of **A** that it means "every object that satisfies the specification of **B** also satisfies the specification of **A**"
  - Sometimes we call this a *true subtype* relationship – see next slide

Be careful!!!!!

- We are still talking about specifications, not implementations!
  - **`java.math.BigInteger`** might share absolutely positively no code at all with **`java.lang.Object`**
- Java subtypes/subclasses are not necessarily true subtypes
  - No type system, including Java's, can determine the behavioral properties that would be needed to ensure this – the details are beyond the scope of 331
  - Java subtypes that are not true subtypes are confusing at best and dangerous at worst

Subclassing

- Subclassing uses inheritance to share code – take advantage of the similarity of parts of the implementation – enables incremental changes to classes
- Every Java subclass is a Java subtype but is *not necessarily a true subtype*
- Checking for true subtypes requires full specifications (and deeper checking, again beyond the scope of type systems)

Java subtypes

- Java types are defined by classes, interfaces, and primitives
- B is Java subtype of A if there is a declared relationship (B extends A, B implements A)
- Compiler checks that, for each corresponding method
  - same argument types
  - compatible result types
  - no additional declared exceptions
- Again: *not* the same as checking for a true subtype! No semantic behavior is considered

Adding functionality

- Suppose we run a web store with a class for **`Products`** …

```
class Product {
  private String title, description;
  private float price;
  public float getPrice() { return price; }
  public float getTax() { return getPrice() * 0.05f; }
  // ...
}
```

- … and we decide we want another class for **`Products`** that are on sale

We could cut-and-paste


```
class SaleProduct {
   private String title, description;
   private float price;
   private float factor;
   public float getPrice() { return price*factor; }
   public float getTax() { return getPrice() * 0.05f;}
    //...
}
```

- Good idea? Bad idea? Why?

Inheritance allows small extensions

20

- The code for the extension is in some sense comparable in size to the extension
- It's much better to do this

```
class SaleProduct extends Product {
  private float factor;
  public float getPrice() {
    return super.getPrice()*factor;
  }
  //...
}
```


Benefits of subclassing & inheritance

21

- Don't repeat unchanged fields and methods
  - Simpler maintenance of implementation: just fix bugs once
  - Clients who understand the superclass specification need only study novel parts of subclass
- Modularity: can ignore private fields and methods of superclass (if properly defined)
  - Differences are not buried under mass of similarities
- Ability to substitute new implementations
  - Clients need not change their code to use new subclasses

Subclassing can be misused

22

- Poor planning leads to muddled inheritance hierarchy
  - Relationships may not match untutored intuition
- If subclass is tightly coupled with superclass
  - Can depend on implementation details of superclass
  - Changes in superclass can break subclass ("fragile base class")
- Subtyping is the source of most benefits of subclassing
  - Just because you want to inherit an implementation does not mean you want to inherit a type – and vice versa!

Every square is a rectangle

23

```
interface Rectangle {
  // effects: fits shape to given size
  //    this_post.width = w, this_post.height = h
  void setSize(int w, int h);
}
```

Which is the best option for `Square.setSize()`?

`interface Square implements Rectangle {…}`

1.
```
// requires: w = h
// effects: fits shape to given size
void setSize(int w, int h);
```
2.
```
// effects: sets all edges to given size
void setSize(int edgeLength);
```
3.
```
// effects:  sets this.width and this.height to w
void setSize(int w, int h);
```
4.
```
// effects: fits shape to given size
// throws BadSizeException if w != h
void setSize(int w, int h) throws BadSizeException;
```


Square and rectangle are unrelated

24

- `Square` is not a true subtype of `Rectangle`
  - Rectangles are expected to have a width and height that can be changed independently
  - Squares violate that expectation, could surprise client
- `Rectangle` is not a true subtype of `Square`
  - Squares are expected to have equal widths and heights
  - Rectangles violate that expectation, could surprise client
- Inheritance isn't always intuitive – it does encourage clear thinking and prevents errors
  - Possible solution might be to make them incomparable (perhaps as siblings under a common parent)
  - Why isn't the elementary school "every square is a rectangle" true when we think about them as true subtypes?

(im)mutability!

Substitution principle: redux

25

- Constraints on methods
  - For each method in a supertype, the subtype must have a corresponding overriding method
  - Also may introduce new methods
- Each overriding method must
  - Ask nothing extra of client ("weaker precondition")
    - `requires` clause is at most as strict as in the supertype's method
  - Guarantee at least as much ("stronger postcondition")
    - `effects` clause is at least as strict as in the supertype method
    - No new entries in `modifies` clause

Substitution: specification weakening

26

- Method inputs
  - Argument types may be replaced with supertypes ("contravariance")
  - This doesn't place any extra demand on the client.
    - Java forbids any change
- Method results
  - Result type may be replaced with a subtype ("covariance")
    - This doesn't violate any expectation of the client
  - No new exceptions (for values in the domain)
  - Existing exceptions can be replaced with subtypes
    - This doesn't violate any expectation of the client

Substitution exercise

Small groups: 2-3 minutes

27

- Suppose we have a method which, when given one product, recommends another
  `Product recommend(Product ref);`
- Which of these are possible forms of method in a true subtype?
  - `Product recommend(SaleProduct ref);` ❑ bad
  - `SaleProduct recommend(Product ref);` ❑ OK
  - `Product recommend(Object ref);` ❑ OK (Java overloading)
  - `Product recommend(Product ref) throws NoSaleException;` ❑ bad
- Same kind of reasoning for exception subtyping and for **`modifies`** clause

Interfaces and abstract classes

28

- Provide interfaces for your functionality
  - Lets client write code to satisfy interfaces rather than to satisfy concrete classes
  - Allows different implementations later
  - Facilitates composition, wrapper classes – we'll see more of this over the term
- Consider providing helper/template abstract classes
  - Can minimize number of methods that new implementation must provide
  - Makes writing new implementations much easier
  - Using them is optional, so they don't limit freedom to create radically different implementations

Why interfaces instead of classes

29

- Java design decisions
  - A class has exactly one superclass
  - A class may implement multiple interfaces
  - An interface may extend multiple interfaces
- Observation
  - multiple superclasses are difficult to use and to implement
  - multiple interfaces, single superclass gets most of the benefit

Concrete, abstract, or interface?

30

- Telephone: $10 landline, speakerphone, cellphone, Skype, VOIP phone
- TV: CRT, Plasma, LCD
- Table: dining table, desk, coffee table
- Coffee: espresso, frappuccino, decaf, Iced coffee
- Computer: laptop, desktop, server, smart phone
- CPU: x86, AMD64, PowerPC
- Professor: Ernst, Notkin, Stepp, Perkins

Depends on the similarity

31

- …that one wants to benefit from
- The specification of the related objects?
- The implementation of the related objects – or parts thereof?
- Not all similarity is similar
- So thinking about the kind of similarity you want to exploit in software development will drive many design decisions
  - Better to do this consciously than subconsciously

Next steps

- Assignment 2: part B due Friday 11:59PM
- Assignment 3: out on Friday – how to handle pairs?
- Lectures: F (modular design), M (design patterns)
- Upcoming: Friday 10/28, in class midterm – open book, open note, closed neighbor, closed electronic devices