

BEFORE I ACCEPT THE SOFTWARE YOU WRITE UNDER CONTRACT, TELL ME WHAT DEVELOPMENT METHODOLOGY YOU USE.

WE HOLD VILLAGE MEETINGS TO BOAST OF OUR SKILLS AND CURSE THE DEVIL-SPAWNED END-USERS.

SOMETIMES WE JUGGLE.

AT THE LAST MINUTE WE SLAM OUT SOME CODE AND GO ROLLER SKATING.

I WOULD FIND THIS HUMOROUS IF NOT FOR THE PEG ON MY BACK.

DESIGNING, CODING, AND DOCUMENTING

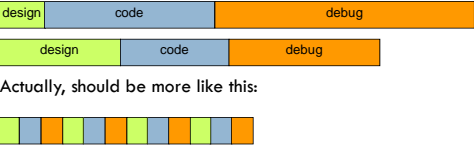
Lecture 16
CS2110 – Spring 2013

Designing and Writing a Program


- Don't sit down at the terminal immediately and start hacking
- Design stage – **THINK** first
 - about the data you are working with
 - about the operations you will perform on it
 - about data structures you will use to represent it
 - about how to structure all the parts of your program so as to achieve abstraction and encapsulation
- Coding stage – code in small bits
 - test as you go
 - understand preconditions and postconditions
 - insert sanity checks (assert statements in Java are good)
 - worry about corner cases
- Use Java API to advantage

The Design-Code-Debug Cycle

- Design is faster than debugging (and more fun)
 - extra time spent designing reduces coding and debugging
- Which is better?



Actually, should be more like this:



Divide and Conquer!

- Break program into manageable parts that can be implemented, tested in isolation
- Define interfaces for parts to talk to each other – develop **contracts** (preconditions, postconditions)
- Make sure contracts are obeyed
 - Clients use interfaces correctly
 - Implementers implement interfaces correctly (test!)

Key: good interface documentation

Pair Programming

- Work in pairs
- Pilot/copilot
 - pilot codes, copilot watches and makes suggestions
 - pilot must convince copilot that code works
 - take turns
- Or: work independently on different parts after deciding on an interface
 - frequent design review
 - each programmer must convince the other
 - reduces debugging time
- Test everything

Documentation is Code

- Comments (esp. specifications) are as important as the code itself
 - determine successful use of code
 - determine whether code can be maintained
 - creation/maintenance = 1/10
- Documentation belongs in code or as close as possible
 - Code evolves, documentation drifts away
 - Put specs in comments next to code when possible
 - Separate documentation? Code should link to it.
- Avoid useless comments
 - `x = x + 1; //add one to x -- Yuck!`
 - Need to document algorithm? Write a paragraph at the top.
 - Or break method into smaller, clearer pieces.

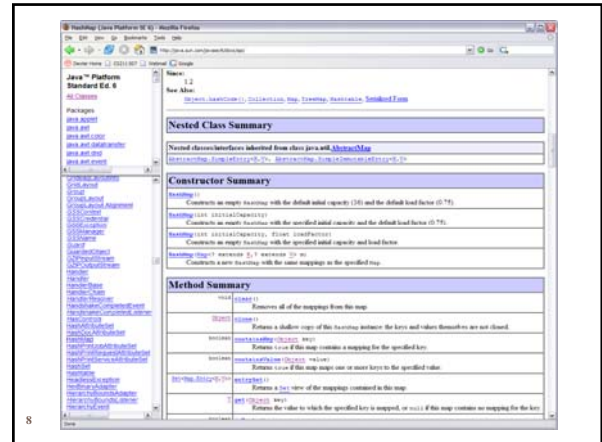
Javadoc

- An important Java documentation tool

```

    graph LR
      A[Java source code (many files)] -- javadoc --> B[Linked HTML web pages]
  
```

- Extracts documentation from classes, interfaces
 - Requires properly formatted comments
- Produces browsable, hyperlinked HTML web pages



How Javadoc is Produced

```

/** indicates Javadoc comment
 * Constructs an empty <tt>HashMap</tt> with the specified initial
 * capacity and the default load factor (0.75).
 * Javadoc keywords
 * @param initialCapacity the initial capacity.
 * @throws IllegalArgumentException if the initial capacity is negative.
 */
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

/** can include HTML
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}
  
```

Some Useful Javadoc Tags

- @return** *description*
 - Use to describe the return value of the method, if any
 - E.g., `@return the sum of the two intervals`
- @param** *parameter-name description*
 - Describes the parameters of the method
 - E.g., `@param i the other interval`
- @author** *name*
- @deprecated** *reason*
- @see** *package.class#member*
- {@code** *expression*
 - Puts expression in code font

Developing and Documenting an ADT

- Write an overview – purpose of the ADT
- Decide on a set of supported operations
- Write a specification for each operation

1. Writing an ADT Overview

- Example abstraction: a closed interval $[a,b]$ on the real number line
 - $[a,b] = \{ x \mid a \leq x \leq b \}$
- Example overview:


```

/**
 * An interval represents a closed interval [a,b]
 * on the real number line.
 */
      
```

 - Javadoc comment
 - Abstract description of the ADT's values

2. Identify the Operations

13

- Enough operations for needed tasks
- Avoid unnecessary operations – keep it simple!
 - ▣ Don't include operations that client (without access to internals of class) can implement

3. Writing Method Specifications

14

- Include
 - ▣ Signature: types of method arguments, return type
 - ▣ Description of what the method does (abstractly)
- Good description (definitional)


```

                /** Add two intervals. The sum of two intervals is
                 * a set of values containing all possible sums of
                 * two values, one from each of the two intervals.
                 */
                public Interval plus(Interval i);
            
```
- Bad description (operational)


```

                /** Return a new Interval with lower bound a+i.a,
                 * upper bound b+i.b.
                 */
                public Interval plus(Interval i);
            
```

Not abstract, might as well read the code...

3. Writing Specifications (cont'd)

15

- Attach before methods of class or interface

```

        /** Add two intervals. The sum of two intervals is
         * a set of values containing all possible sums of
         * two values, one from each of the two intervals.
         *
         * @param i the other interval
         * @return the sum of the two intervals
         */
    
```

Method overview
 Method description
 Additional tagged clauses

Know Your Audience

16

- Code and specs have a target audience
 - ▣ the programmers who will maintain and use it
- Code and specs should be written
 - ▣ With enough documented detail so they can understand it
 - ▣ While avoiding spelling out the obvious
- Try it out on the audience when possible
 - ▣ design reviews before coding
 - ▣ code reviews

Consistency

17

A foolish consistency is the hobgoblin of little minds – Emerson

- Pick a consistent coding style, stick with it
 - ▣ Make your code understandable by "little minds"
- Teams should set common style
- Match style when editing someone else's code
 - ▣ Not just syntax, also design style

Simplicity

18

- *The present letter is a very long one, simply because I had no time to make it shorter. –Blaise Pascal*
- *Be brief. –Strunk & White*
- Applies to programming... simple code is
 - ▣ Easier and quicker to understand
 - ▣ More likely to be correct
- Good code is simple, short, and clear
 - ▣ Save complex algorithms, data structures for where they are needed
 - ▣ Always reread code (and writing) to see if it can be made shorter, simpler, clearer

Choosing Names

19

- Don't try to document with variable names
 - ▣ Longer is not necessarily better

```
int searchForElement(
    int[] array_of_elements_to_search,
    int element_to_look_for);
```

```
int search(int[] a, int x);
```

- Names should be short but suggestive
- Local variable names should be short

Avoid Copy-and-Paste

20

- Biggest single source of program errors
 - ▣ Bug fixes never reach all the copies
 - ▣ Think twice before using edit copy-and-paste function



- Abstract instead of copying!
 - ▣ Write many calls to a single function rather than copying the same block of code around

But sometimes you have no choice

21

- Example: SWING or SWT GUI code
 - ▣ Realistically, you simply have to use cut-and-paste!
- In such situations, do try to understand what you copied and "make it your own"
 - ▣ They wrote it first
 - ▣ But now you've adopted it and will love it and care for it... maybe even rewrite it...

Design vs Programming by Example

22

- Programming by example:
 - ▣ copy code that does something like what you want
 - ▣ hack it until it works
- Problems:
 - ▣ inherit bugs in code
 - ▣ don't understand code fully
 - ▣ usually inherit unwanted functionality
 - ▣ code is a bolted-together hodge-podge
- Alternative: design
 - ▣ understand exactly why your code works
 - ▣ reuse abstractions, not code templates

Avoid Premature Optimization

23

- Temptations to avoid
 - ▣ Copying code to avoid overhead of abstraction mechanisms
 - ▣ Using more complex algorithms & data structures unnecessarily
 - ▣ Violating abstraction barriers
- Result:
 - ▣ Less simple and clear
 - ▣ Performance gains often negligible
- Avoid trying to accelerate performance until
 - ▣ You have the program designed and working
 - ▣ You know that simplicity needs to be sacrificed
 - ▣ You know where simplicity needs to be sacrificed

Avoid Duplication

24

- Duplication in source code creates an implicit constraint to maintain, a quick path to failure
 - ▣ Duplicating code fragments (by copying)
 - ▣ Duplicating specs in classes and in interfaces
 - ▣ Duplicating specifications in code and in external documents
 - ▣ Duplicating same information on many web pages
- Solutions:
 - ▣ Named abstractions (e.g., declaring functions)
 - ▣ Indirection (linking pointers)
 - ▣ Generate duplicate information from source (e.g., Javadoc)
- *If you must duplicate:*
 - ▣ Make duplicates link to each other so can find all clones

Maintain State in One Place

25

- Often state is duplicated for efficiency
- But difficult to maintain consistency
- **Atomicity** is the issue
 - if the system crashes while in the middle of an update, it may be left in an inconsistent state
 - difficult to recover

Error Handling

26

- It is usually an afterthought — it shouldn't be
- User errors vs program errors — there is a difference, and they should be handled differently
- Insert lots of “sanity checks” — the Java `assert` statement is good way to do this
- Avoid meaningless messages

Avoid Meaningless Messages

27



Design Patterns

28

- Introduced in 1994 by Gamma, Helm, Johnson, Vlissides (the “Gang of Four”)
- Identified 23 classic software design patterns in OO programming
- More than 1/2 million copies sold in 14 languages

Design Patterns

29

- **Abstract Factory** groups object factories that have a common theme.
- **Builder** constructs complex objects by separating construction and representation.
- **Factory Method** creates objects without specifying the exact class to create.
- **Prototype** creates objects by cloning an existing object.
- **Singleton** restricts object creation for a class to only one instance.
- **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- **Composite** composes one-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behaviour in an existing method of an object.
- **Facade** provides a simplified interface to a large body of code.
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Design Patterns

30

- **Chain of responsibility** delegates commands to a chain of processing objects.
- **Command** creates objects which encapsulate actions and parameters.
- **Interpreter** implements a specialized language.
- **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento** provides the ability to restore an object to its previous state (undo).
- **Observer** is a publish/subscribe pattern that allows a number of observer objects to see an event.
- **State** allows an object to alter its behavior when its internal state changes.
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Design Patterns

31

- Chain of responsibility delegates commands to a chain of processing objects.
- Command creates objects which encapsulate actions and parameters.
- Interpreter implements a specialized language.
- Iterator accesses the elements of an object sequentially without exposing its underlying representation.
- Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- Memento provides the ability to restore an object to its previous state (undo).
- **Observer** is a publish/subscribe pattern that allows a number of observer objects to see an event.
- State allows an object to alter its behavior when its internal state changes.
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
- Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Observer Pattern

32

- **Observable**
 - changes from time to time
 - is aware of Observers, other entities that want to be informed when it changes
 - but may not know (or care) what or how many Observers there are
- **Observer**
 - interested in the Observable
 - want to be informed when the Observable changes

Observer Pattern

33

- **Issues**
 - does the Observable push information, or does the Observer pull it? (e.g., email vs newsgroup)
 - whose responsibility is it to check for changes?
 - publish/subscribe paradigm

Observer Pattern

34

```

public interface Observer<E> {
    void update(E event);
}

public class Observable<E> {
    private Set<Observer<E>> observers = new HashSet<Observer<E>>();
    boolean changed;

    void addObserver(Observer<E> obs) {
        observers.add(obs);
    }

    void removeObserver(Observer<E> obs) {
        observers.remove(obs);
    }

    void notifyObservers(E event) {
        if (!changed) return;
        changed = false;
        for (Observer<E> obs : observers) {
            obs.update(event);
        }
    }
}
    
```

Visitor Pattern

35

- A data structure provides a generic way to iterate over the structure and do something at each element
- The visitor is an implementation of interface methods that are called at each element
- The visited data structure doesn't know (or care) what the visitor is doing
- There could be many visitors, all doing different things

Visitor Pattern

36

```

public interface Visitor<T> {
    void visitPre(T datum);
    void visitIn(T datum);
    void visitPost(T datum);
}

public class TreeNode<T> {
    TreeNode<T> left;
    TreeNode<T> right;
    T datum;

    TreeNode(TreeNode<T> l, TreeNode<T> r, T d) {
        left = l;
        right = r;
        datum = d;
    }

    void traverse(Visitor<T> v) {
        v.visitPre(datum);
        if (left != null) left.traverse(v);
        v.visitIn(datum);
        if (right != null) right.traverse(v);
        v.visitPost(datum);
    }
}
    
```

No Silver Bullets

37

- These are all rules of thumb; but there is no panacea, and every rule has its exceptions
- You can only learn by doing – we can't do it for you
- Following software engineering rules only makes success more likely!