CS1007 lecture #23 notes

thu 25 apr 2002

- news
- software engineering
- inheritance
- reading: ch 3.9 and ch 10; ch 7.1-7.6

news.

- homework #6 due Thu May 2
- final exam:
  - Tue May 14 9am-12noon (AM class, section 002)
  - Thu May 16 1pm-4pm (PM class, section 001)

software engineering.

- reading: ch 3.9, ch 10
- the *software life cycle*:

$$\begin{tikzcd}
\boxed{\text{development}} \arrow[r] & \textit{release} \arrow[r] & \boxed{\text{use}} \arrow[r] & \boxed{\text{maintenance}} \arrow[r] & \boxed{\text{retirement}}
\end{tikzcd}$$

- there are several *process models*:
  - *build-and-fix model*
  - *waterfall model*
  - *iterative model*
  - *evolutionary model*

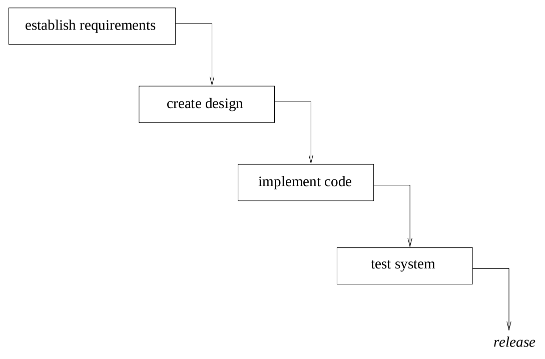
build-and-fix model.

- the oldest model
- probably what you've been doing when you write your homework...

$$\begin{tikzcd}
\boxed{\text{write program}} \arrow[r] & \boxed{\text{modify program}} \arrow[r, leftrightarrow] & \textit{release}
\end{tikzcd}$$

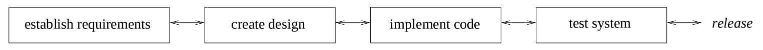
waterfall model.

- developed as software evolved into large projects, involving many lines of code, many files and many programmers working together on the same large project...



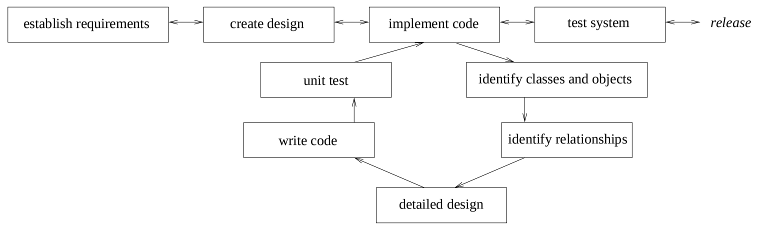
iterative model.

- developed after it was recognized that the waterfall model was unrealistic
- each step can be (and usually is) revisited
- especially common in large companies, where multiple people are working on the same project and the people who, for example, “establish requirements” are not the same people who “create design” or “implement code” or “test system”



evolutionary model.

- evolved, again from companies where large software projects are developed and maintained, particularly after the introduction of the “object-oriented” way of thinking
- emphasizes *modularity* and allows for software re-use as well as testing of individual modules to make sure that each piece is robust and correct before it is added to the whole

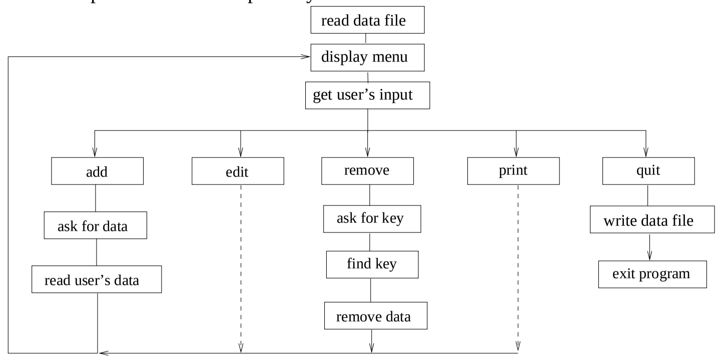


command-line interface and menu processing.

- last class, we developed a flow chart to handle typical database processing operations
- these are:
  - add data
  - edit data
  - remove data
  - print data
- typically, a program begins by reading the data from a data file – for your homework, this will be a text data file – into variables inside the program (e.g., Vectors or arrays)
- then the program will manipulate the data in the Vector(s)/array(s), until the user is ready to quit
- finally, the program will write the manipulated data back to the data file

menu processing flow chart.

- we developed this flow chart partially in class...



inheritance.

- reading: ch 7.1-7.6
- *inheritance* is the means by which classes are created out of other classes
- a cornerstone of object-oriented programming
- idea is to create classes that can be re-used from one application to another
- classes contain *data objects* and *methods*
- you want to be able to change the *data type* of the data objects and still be able to use the same methods
- you also want to be able to change the flavor of what the methods do

inheritance tree (1).

- think of the most primitive Java class, `Object` as being at the root of the inheritance tree
- all other classes are "children" or *subclasses* of that class
- here is an example of the inheritance tree for Applet:

```
Object
   |
Component
   |
Container
   |
Panel
   |
Applet
```

inheritance tree (2).

- as you move DOWN the inheritance tree from the root to the leaf, you are *extending* subclasses from parent classes
  - parent classes are also called *superclasses*
  - or *base classes*
  - children classes are *derived* from their parents
- as you move UP the inheritance tree from the leaf to the root, you can say that each subclass is a *more specific* version of its parent
- this is known as the *is-a* relationship between a subclass and the parent class that the child extends

overriding methods.

- for homework #5, you had to *extend* the `Applet` class and *override* the `paint()` method
- if you traverse the inheritance tree for `Applet`, you will find that `paint()` is defined in the `Component` class
- in your homework, you wrote a new version of the `paint()` method to draw something cool — this was called *overriding* the method
- when your homework is executed, your `paint()` method is the one that is invoked, instead of the one in a superclass
- the rule is: *the version of any method that is invoked is the definition closest to the leaf of the tree*
- if you want to refer to the version of the method in a class's superclass, you use the `super` reference
- so in order to invoke the version of `paint()` that is defined in the `Component` class, you call: `super.paint()`
- (but this will do nothing since the default version of paint is empty)

overloading methods.

- in addition to changing precisely what a method does, you can also change the arguments to that method.
- this is very useful if you are changing the data type of data objects defined in the class.
- you can create a new version of a method which has different arguments from the version of the method defined in the class's superclass.
- this is what happens when we use different versions of the `StringTokenizer` constructor last class:

```
StringTokenizer tokenizer = new StringTokenizer( line );
```

versus

```
StringTokenizer tokenizer = new StringTokenizer( line,"|" );
```

other terminology...

- *polymorphism*
  - "having many forms"
  - lets us use different implementations of a single class
  - we talked about this in relation to interfaces
  - a polymorphic reference can refer to different types of objects at different times
- *abstract* class
  - represents a generic concept in a class hierarchy
  - cannot be instantiated — can only be extended