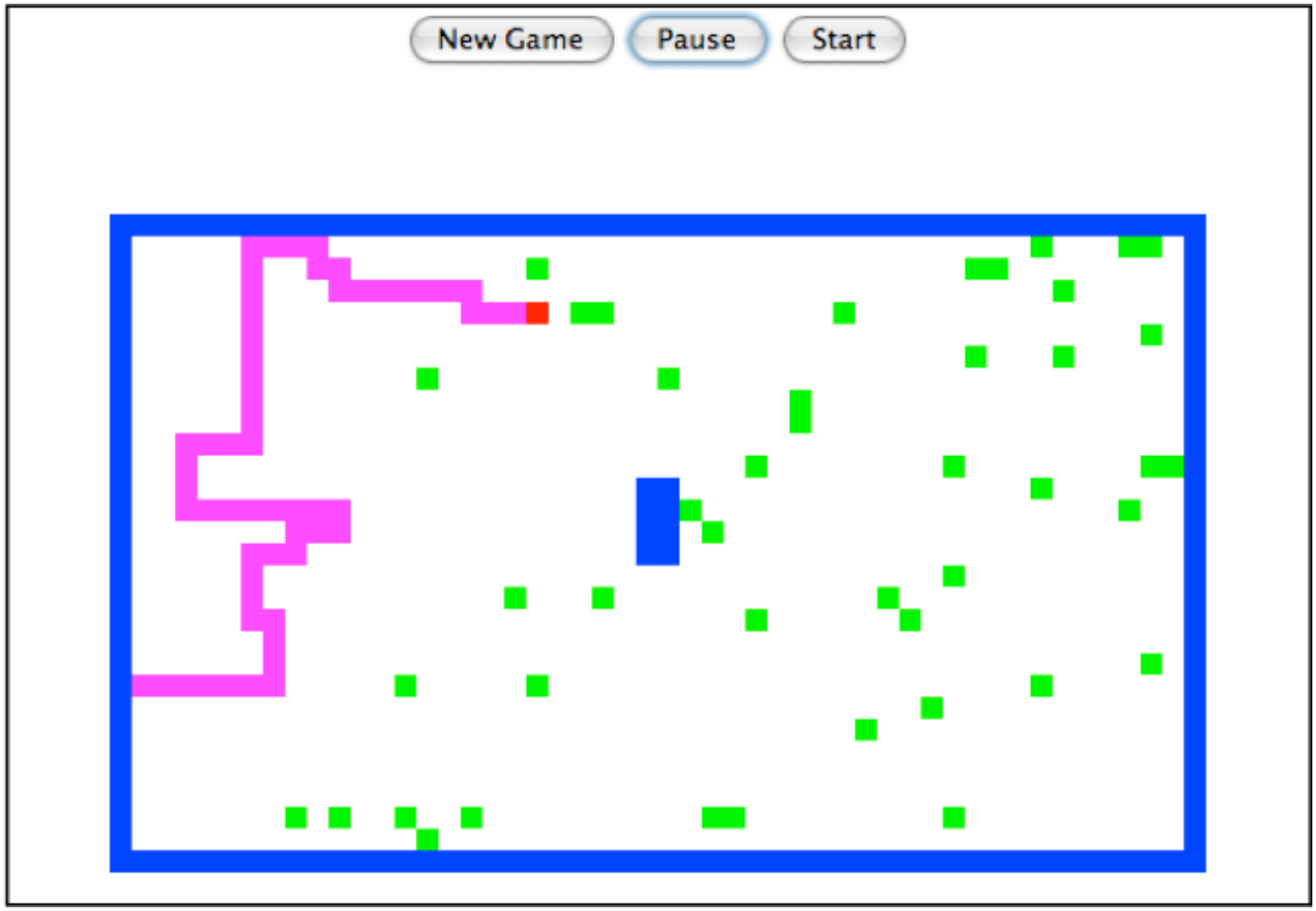
Object-Oriented Programming

February 27–28, 2012
CS 60: Principles of Computer Science

Assignment 5 due February 27: Spamseeker
Assignment 6 due March 5: Spampede!
Midterm out March 5–6
Midterm due back 5pm Friday, March 9

Spampede Demo

http://www.cs.hmc.edu/~dodds/Applets/SpampedeBasic/Spampede.html



Today's Outline

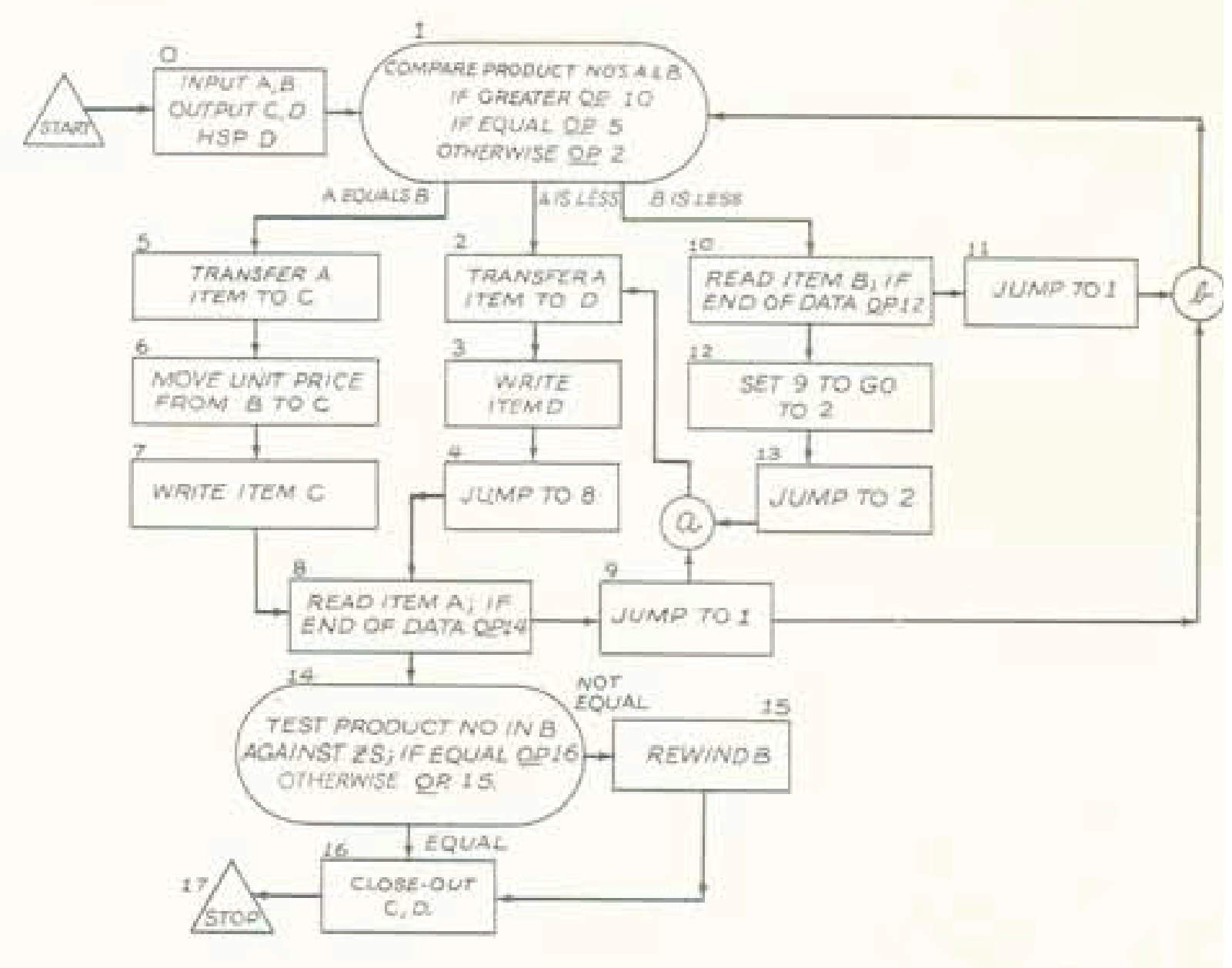
A bit of history

- ✓ Programming in the small
- ✓ Programming in the large

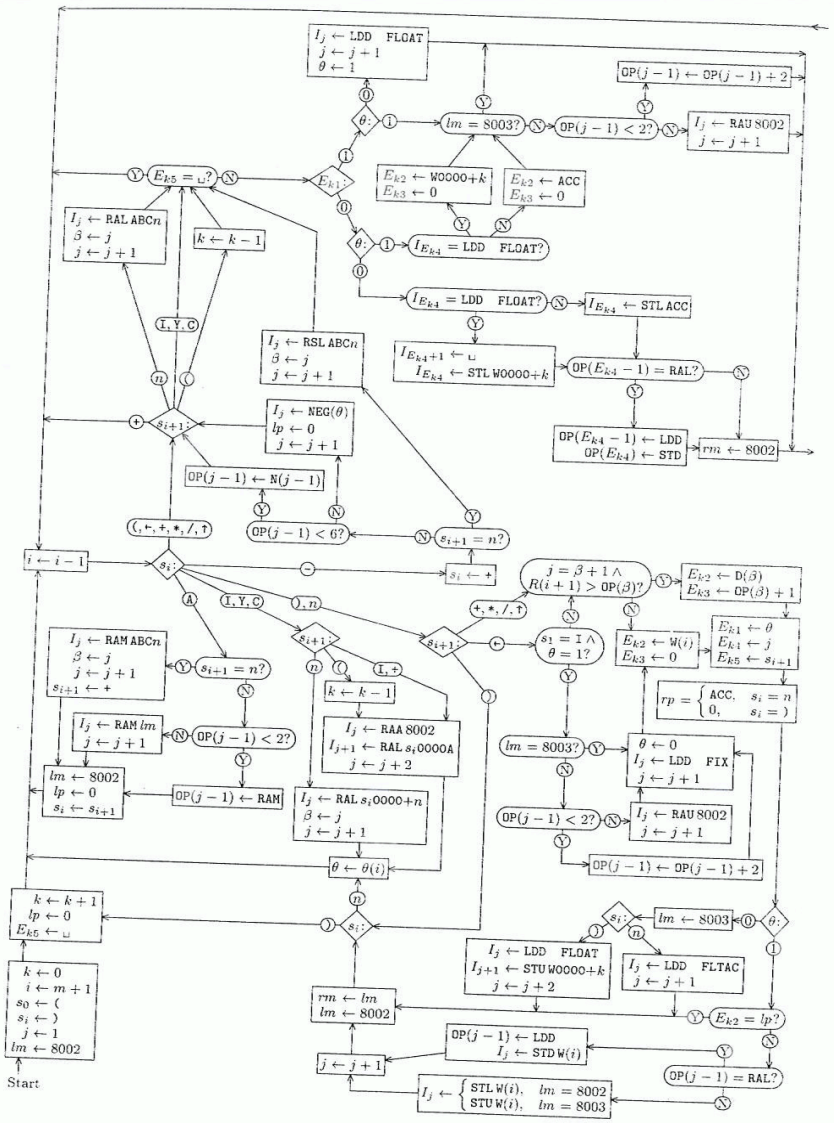
Inheritance in practice

Spampede, Parts 1 and 2 (of 3)

Flowcharts


START
0
INPUT A,B
OUTPUT C,D
HSP D
1
COMPARE PRODUCT NOS A & B
IF GREATER GP 10
IF EQUAL GP 5
OTHERWISE GP 2
A EQUALS B
A IS LESS
B IS LESS
5
TRANSFER A ITEM TO C
2
TRANSFER A ITEM TO D
10
READ ITEM B; IF END OF DATA GP 12
11
JUMP TO 1
6
MOVE UNIT PRICE FROM B TO C
3
WRITE ITEM D
12
SET 9 TO GO TO 2
7
WRITE ITEM C
4
JUMP TO 8
13
JUMP TO 2
8
READ ITEM A; IF END OF DATA GP 14
9
JUMP TO 1
14
TEST PRODUCT NO IN B AGAINST ZS; IF EQUAL GP 16 OTHERWISE GP 15
NOT EQUAL
15
REWIND B
EQUAL
16
CLOSE-OUT C, D
17
STOP

Flowcharts



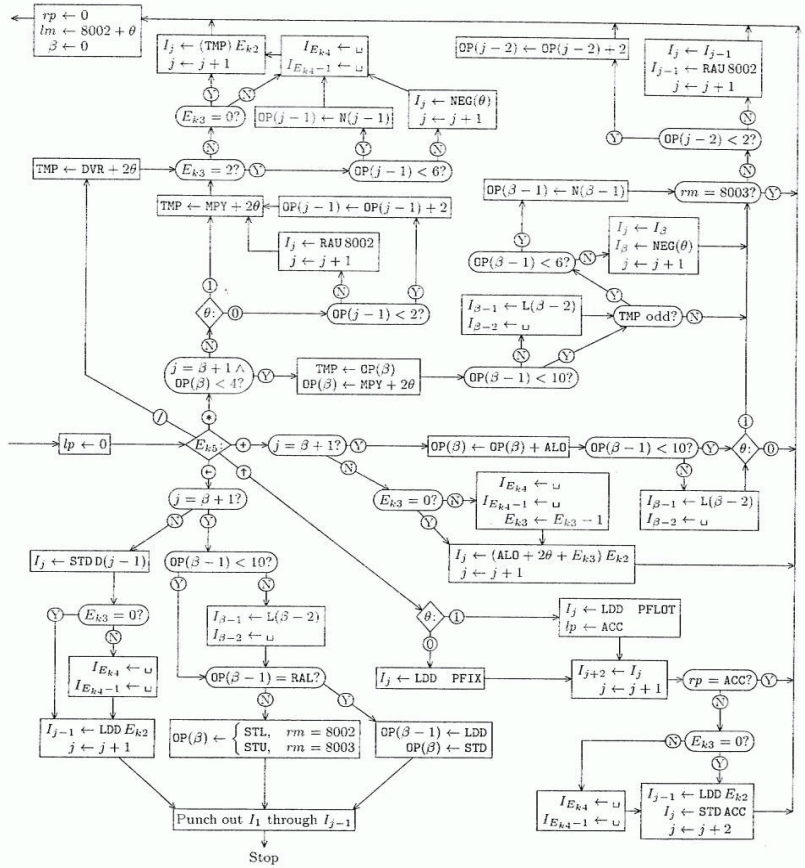


FIGURE 1. Compilation by RUNCIBLE. In this diagram, I_j refers to the jth instruction compiled; OP(j) means the operation code of instruction I_j, and D(j) stands for the DATA address. Several functions have been defined to simplify the notation:

$R(i) = (1, 1, 4, 4, 6)$, if $s_i = (, \uparrow, *, /, +)$, respectively;

$\mathrm{L}(j) = \mathtt{RAL}\ \mathrm{D}(j)$, if $\mathrm{OP}(j) = \mathtt{LDD}$ and $\mathrm{D}(j) \neq$ ' ', otherwise $\mathrm{L}(j) = I_j$;

$$\theta(i) = \begin{cases} 0, & \text{if } s_i = \mathtt{I} \text{ or fixed-point } n; \\ 1, & \text{if } s_i = \mathtt{Y}, \mathtt{C}, \text{ or floating-point } n; \end{cases} \qquad \mathrm{W}(i) = \begin{cases} \mathtt{WOOOO} + k, & \text{if } s_i =); \\ \mathtt{ACC}, & \text{if } s_i = n; \end{cases}$$

$$\mathrm{N}(j) = \begin{cases} \mathrm{OP}(j) + 1, & \text{if OP}(j) \text{ is even;} \\ \mathrm{OP}(j) - 1, & \text{if OP}(j) \text{ is odd;} \end{cases} \qquad \mathtt{NEG}(\theta) = \begin{cases} \mathtt{RSL\ 8002}, & \text{if } \theta = 0; \\ \mathtt{RSU\ 8003}, & \text{if } \theta = 1. \end{cases}$$

Plastic Templates for Programmers

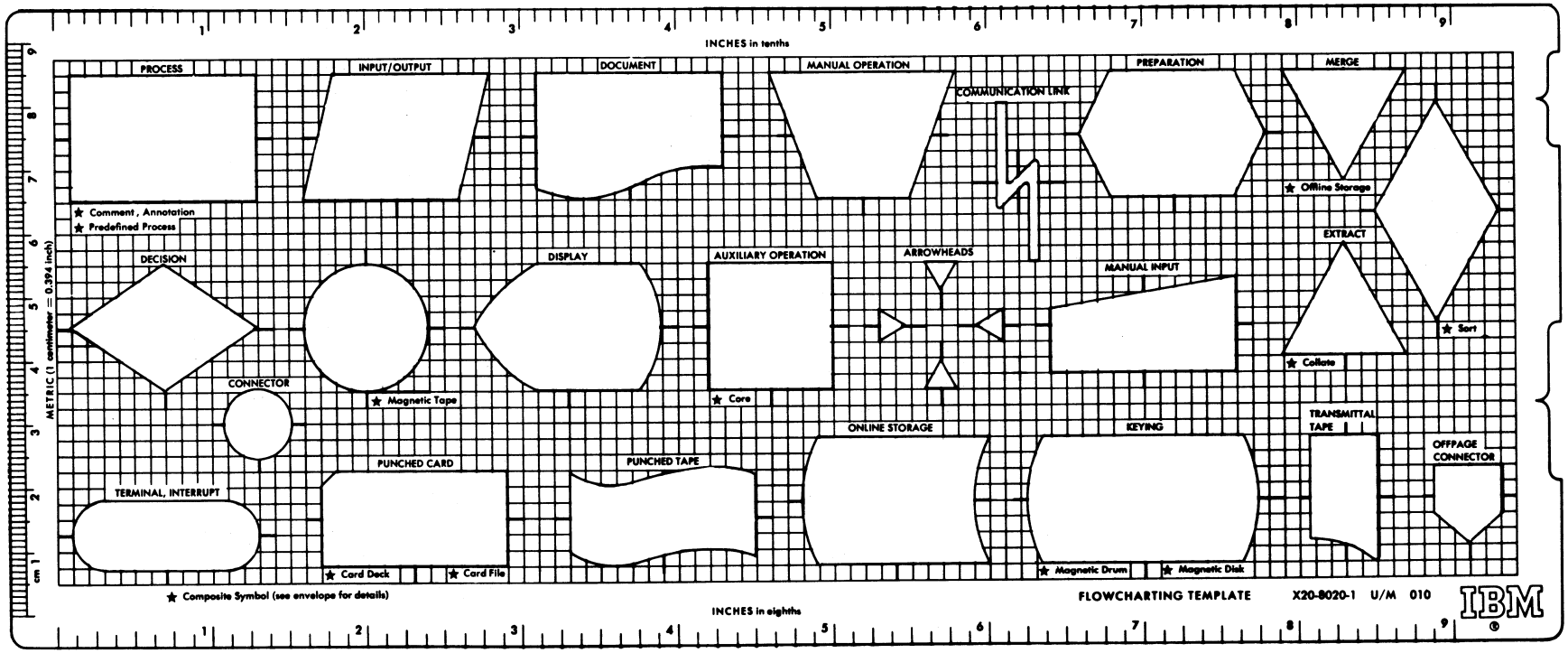
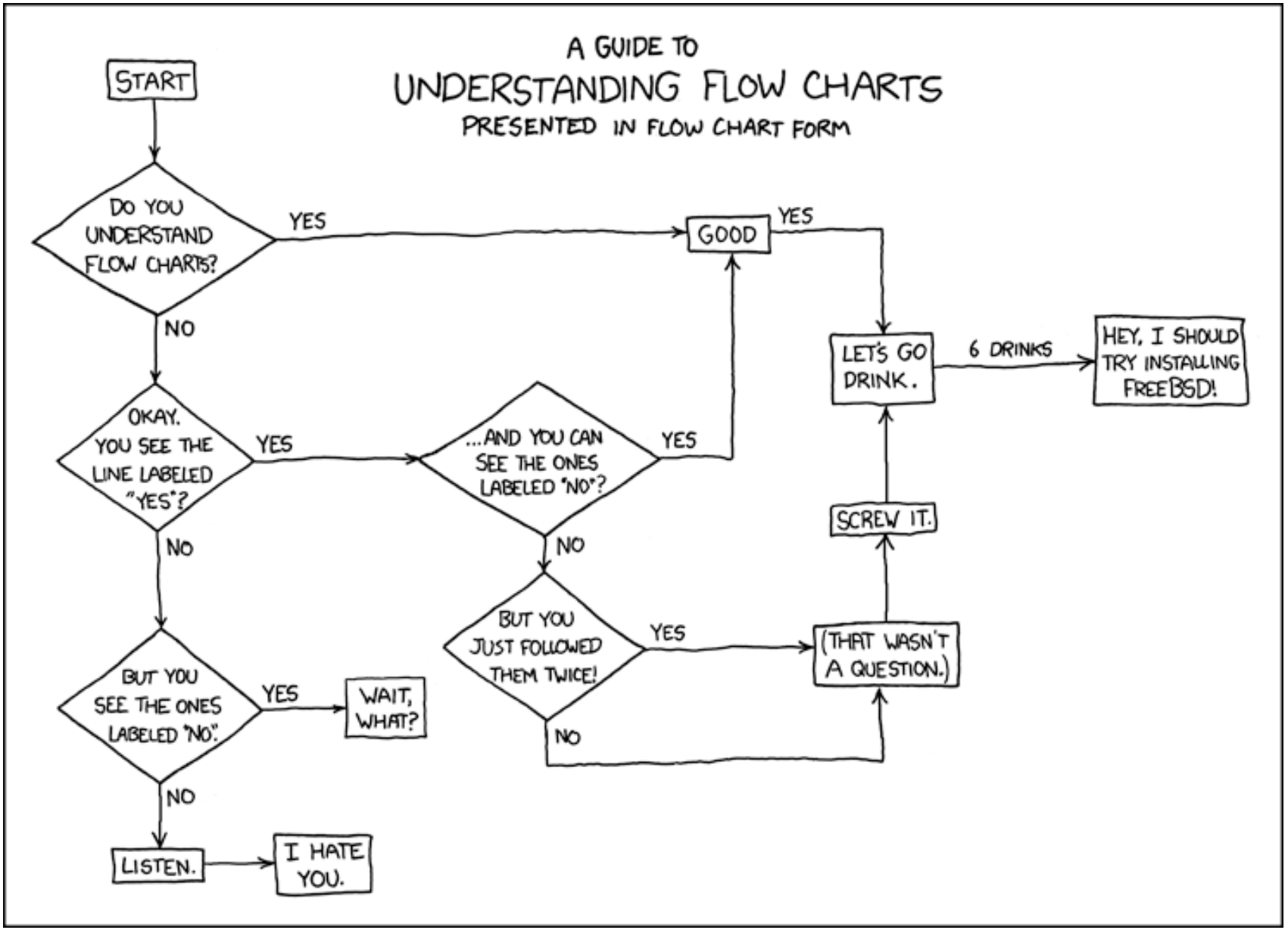


Figure 2. Flowcharting template (actual size)

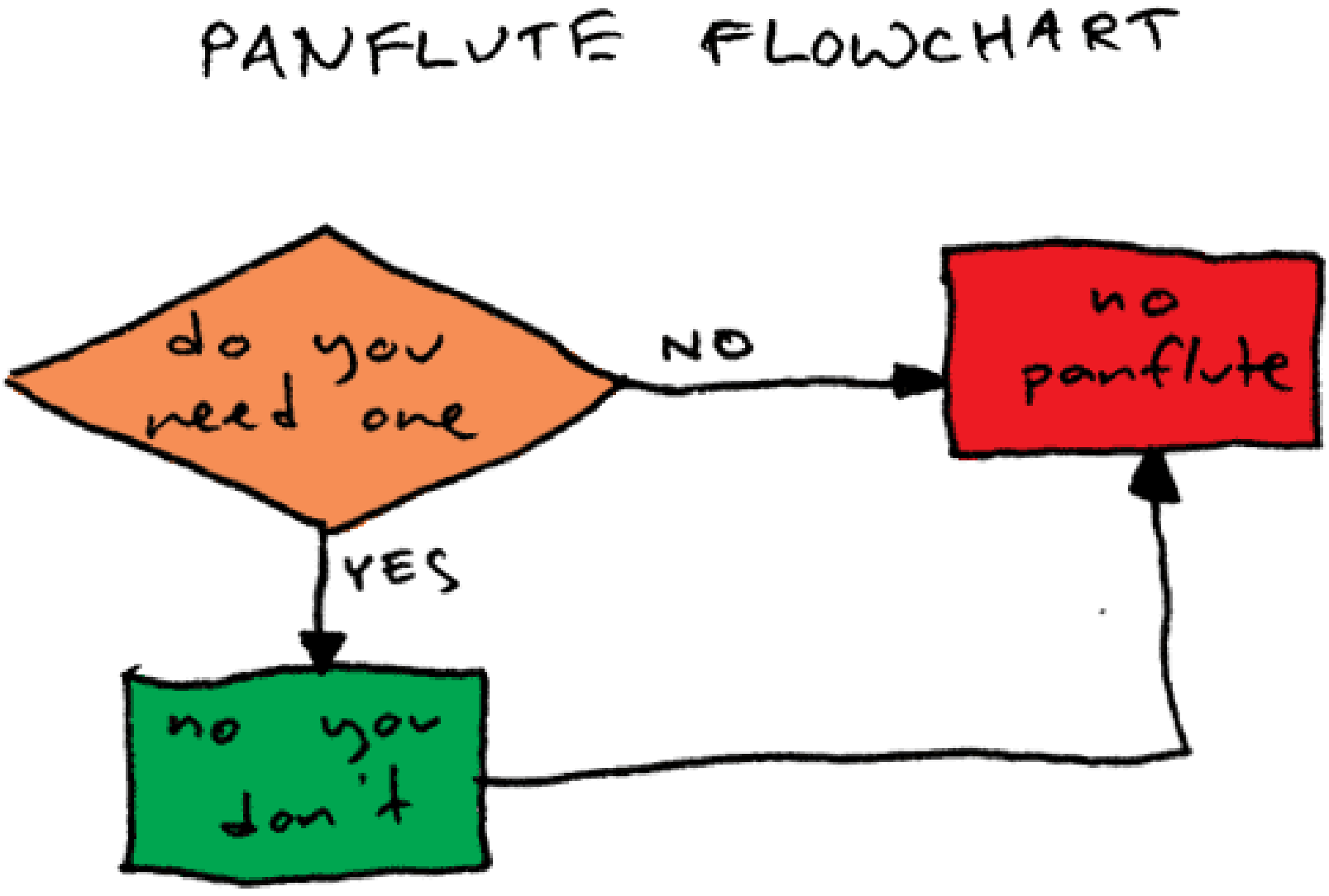
More Flowcharts

(http://xkcd.com/518)



More Flowcharts

(http://www.toothpastefordinner.com/index.php?date=020605)



From Spaghetti Code to Structured Programming

```
      SUBROUTINE OBACT(TODO)
      INTEGER ACT,LENGTH,NCHAR
      INTEGER TODO,DONE
   13 IF(TODO) 14,12,14
   14 ACT=MOD(TODO,10)
      TODO=TODO/10
      GOTO(9,42,43,9,99,45,9,9,9),ACT
      GOTO 13
   42 CALL COPY
      GOTO 127
   43 CALL MOVE
      GOTO 144
   99 NCHAR=-NCHAR
   44 CALL DELETE
      GOTO 127
   45 CALL PRINT
      GOTO 144
    9 CALL BADACT(ACT)
      GOTO 12
  127 L=L+N
  144 DONE=DONE+1
      CALL RESYNC
      GO TO 13
   12 RETURN
      END
```

```
      SUBROUTINE OBACT(TODO)
      INTEGER ACT, LENGTH, NCHAR
      INTEGER TODO, DONE
  100 IF ( TODO.NE.0 ) THEN
       ACT = MOD(TODO,10)
       TODO = TODO/10
       IF(ACT.EQ.1.OR.ACT.EQ.4.OR.
     &    ACT.EQ.7.OR.ACT.EQ.8.OR.
     &    ACT.EQ.9) THEN
         CALL BADACT(ACT)
         GOTO 200
       ELSEIF ( ACT.EQ.2 ) THEN
         CALL COPY
         LENGTH = LENGTH + NCHAR
       ELSEIF ( ACT.EQ.3 ) THEN
         CALL MOVE
       ELSEIF ( ACT.EQ.5 ) THEN
         NCHAR = -NCHAR
         CALL DELETE
         LENGTH = LENGTH + NCHAR
       ELSEIF ( ACT.EQ.6 ) THEN
         CALL PRINT
       ELSE
         GOTO 100
       ENDIF
       DONE = DONE + 1
       CALL RESYNC
       GOTO 100
      ENDIF
  200 RETURN
```

```
SUBROUTINE OBACT(TODO)
INTEGER ACT, LENGTH, NCHAR
INTEGER TODO, DONE
DO WHILE ( TODO.NE.0 )
  ACT = MOD(TODO,10)
  TODO = TODO/10
  SELECT CASE (ACT)
    CASE (1,4,7,8,9)
      CALL BADACT(ACT)
      EXIT
    CASE (2)
      CALL COPY
      LENGTH = LENGTH + NCHAR
    CASE (3)
      CALL MOVE
    CASE (5)
      NCHAR = -NCHAR
      CALL DELETE
      LENGTH = LENGTH + NCHAR
    CASE (6)
      CALL PRINT
    CASE DEFAULT
      CYCLE
  END SELECT
  DONE = DONE + 1
  CALL RESYNC
ENDDO
RETURN
END
```

Abstract Data Types

```
def makeStack(): ...

def push(stack, n): ...

def pop(stack): ...

def isEmpty(stack): ...
```

We don't need to know or care how these stacks are actually implemented.

```
st = makeStack()
push(st, 3)
push(st, 4)
pop(st)
```

Abstract Data Types

```
def makeStackLL(): ...
def pushLL(stack, n): ...
def popLL(stack): ...
def isEmptyLL(stack): ...
```

```
def makeStackArr(): ...
def pushArr(stack, n): ...
def popArr(stack): ...
def isEmptyArr(stack): ...
```

We do have to be sure not to mix our types.

```
st1 = makeStackLL()
st2 = makeStackArr()
pushLL(st1, 3)
pushArr(st2, 4)
popLL(st2)          # whoops!
```

Data + Code = Objects

Objects: ADTs that "come with" the right operations.

```
class StackLL:
  def push(self, n): ...

  def pop(self): ...

  def isEmpty(self): ...
```

```
class StackArr:
  def push(self, n): ...

  def pop(self): ...

  def isEmpty(self): ...
```

```
st1 = StackLL()
st2 = StackArr()
st1.push(3)
st2.push(4)
st2.pop()
```

Classes

It is possible to have objects without classes! (E.g., JavaScript)

But, classes are popular because they provide a lot of additional functionality:

- ✓ Classes let the user create objects
- ✓ Classes let the user define new types
- ✓ Classes have an inside and outside (`public`/`private`)
- ✓ Last (and least?): classes support inheritance

Inheritance: The Plan

We have a class `C` that *almost* does what we want.

We know it's a bad idea to duplicate `C`'s source code and start making changes.

Goal: Say only what changes we want, starting with `C`

- ✓ What new fields and methods we want (**`extension`**)
- ✓ What methods we want to replace (`override`)

Note: need to be careful about accessibility. Who has access?

- ✓ `public`: everyone
- ✓ `private`: the current class
- ✓ `protected`: the current class and inheriting classes can access
- ✓ `package`: any class in the current package (group of classes)

Example Superclass: Person

```
class Person extends Object
{
    public Person( String name )  { this.name = name; }

    protected String name;                  // !

    public boolean isAsleep( int hr )  { return hr > 22 || hr < 7; }

    public String toString()      { return name; }

    public void status( int hr ) {
       if ( this.isAsleep( hr ) )
          System.out.println( "Now asleep: " + this );
       else
          System.out.println( "Now awake: " + this );  }
}
```

```
Person p = new Person("Pat");
p.status(23);
```

Example Subclass: Student

```
class Student extends Person                    // !
{
    // Constructor (constructors are never inherited)
    public Student( String name, int units ) {
       super(name);                             // !
       this.units = units; }

    // Extension (adding new fields and/or methods)
    protected int units;
    void study() { System.out.println( "Done studying." ); }

    // Override (replacing inherited methods)
    public boolean isAsleep( int hr ) { return 2 < hr && hr < 8; }
    public String toString() {
       String result = super.toString();        // !
       return result + " units: " + units;  }
}
```

```
Student s = new Student("Pat", 19);
s.status(23);
```

Inheritance in Java

When `Student` inherits from (`extends`) Person, then:

1. The *code* for `Person` is *reused* in `Student`
2. Java will let you treat a `Student` as a `Person`!
   (After all, it has all the fields and methods a `Student` does.)

```
Person p = new Student("Pat", 19);
p.status(23);
```

Another Inheritance Example

```
class Person extends Object
{
  public String getName() {...}
  // no getMajor method defined here
}

class Mudder extends Person
{
  // no getName defined here
  public String getMajor() {...}
}

class General extends Person
{
  // no getName defined here
  public String getMajor() {...}
}
```

```
Person P  = new Person();
Person Q  = new Person();
Mudder M  = new Mudder();
General G = new General();
Object Ob = new General();

// which lines are not ok?


System.out.println( M.getName() );
P = M;
System.out.println( P.getMajor() );
G = Q;
G = Ob;
System.out.println( G.equals(Ob) );
System.out.println(
          G.equals(new General()) );

// Where did .equals come from?
```

Important Programming Tip

Inherit only when classes have an "is-a" relationship.

- ✓ a `Kangaroo` is an `Animal`
- ✓ a `Circle` is a `Shape`
- ✓ a `SpamMaze` is a `Maze`

If you're still not sure, think about what Java will let you do.

- ✓ If I'm expecting an `Animal`, would I be ok getting a `Kangaroo`?
- ✓ If I'm expecting a `Shape`, would I be ok getting a `Circle`?
- ✓ If I'm expecting a `Circle`, would I be ok getting an `Point`?
- ✓ If I'm expecting a `Point`, would I be ok getting an `Circle`?

Alternative: Containment

Much more common is the "has-a" relationship.

- ✓ A `Circle` has a central `Point`
- ✓ A `Maze` has a two-dimensional array of `MazeCells`

All we need is a field of the appropriate type.

No inheritance necessary!

Dangers of Inheritance (1)

```
class InstrumentedHashSet extends HashSet {
  private int addCount = 0;  // count of elements added
  ...
  public boolean add(Object o) {
     addcount++;
     return super.add(o);
  }

  public boolean addAll(Collection c) {
     addCount += c.size();
     return super.addAll(c);
  }

  public int getAddCount() { return addCount; }
}
```

```
InstrumentedHashSet ihs = new InstrumentedHashSet();
ihs.addAll(Arrays.asList(new String[]{"A","B","C"}));
// ihs.getAddCount() == ???
```

From Joshua Bloch's *Effective Java*

Dangers of Inheritance(2)

Object-Oriented Systems Engineering
Object reuse

Careless Code Reuse Causes Killer Kangaroos

Mutant Marsupials Take Up Arms Against Australian Air Force

The reuse of some object-oriented code has caused tactical headaches for Australia's armed forces. As virtual reality simulators assume larger roles in helicopter combat training, programmers have gone to great lengths to increase the realism of their scenarios, including detailed landscapes and - in the case of the Northern Territory's Operation Phoenix- herds of kangaroos (since disturbed animals might well give away a helicopter's position). The head of the Defense Science & Technology Organization's Land Operations/Simulation division reportedly instructed developers to model the local marsupials' movements and reactions to helicopters. Being efficient programmers, they just re-appropriated some code originally used to model infantry detachment reactions under the same stimuli, changed the mapped icon from a soldier to a kangaroo, and increased the figures' speed of movement. Eager to demonstrate their flying skills for some visiting American pilots, the hotshot Aussies "buzzed" the virtual kangaroos in low flight during a simulation. The kangaroos scattered, as predicted, and the visiting Americans nodded appreciatively... then did a double-take as the kangaroos reappeared from behind a hill and launched a barrage of Stinger missiles at the hapless helicopter. Apparently the programmers had forgotten to remove that part of the infantry coding. The lesson? Objects are defined with certain attributes, and any new object defined in terms of an old one inherits all the attributes. The embarrassed programmers had learned to be careful when reusing object-oriented code, and the Yanks left with a newfound respect for Australian wildlife. Simulator supervisors report that pilots from that point onward have strictly avoided kangaroos, just as they were meant to.

From June 15, 1999, Defense Science and Technology Organization Lecture Series, Melbourne, Australia, and staff reports. Item taken from Software Testing and Quality Engineering magazine, Volume 1, Issue 6 (November/December 1999).

The Morals of These Stories

Inheritance is not the solution for all problems.
There are many other ways to get code reuse, including

- ✓ Interfaces
- ✓ Generics
- ✓ Functions (*e.g.*, `map` in Racket)

Inheritance works best when the superclass is *designed for inheritance*

- ✓ I.e., when the programmer thought about what subclasses might want to know or access or change.