SE350: PRINCIPLES OF OBJECT ORIENTED DESIGN

Jane Cleland-Huang
Office: CDM 836
Hours: Tuesday/Thursday 3.30pm-4.15pm
Phone: 312-362-8863
Email: jhuang@cs.depaul.edu

Questions

- How do we solve programming problems when we need to apply different algorithms or different solutions according to the current context and/or state of the system?
- How can we effectively test our programs?

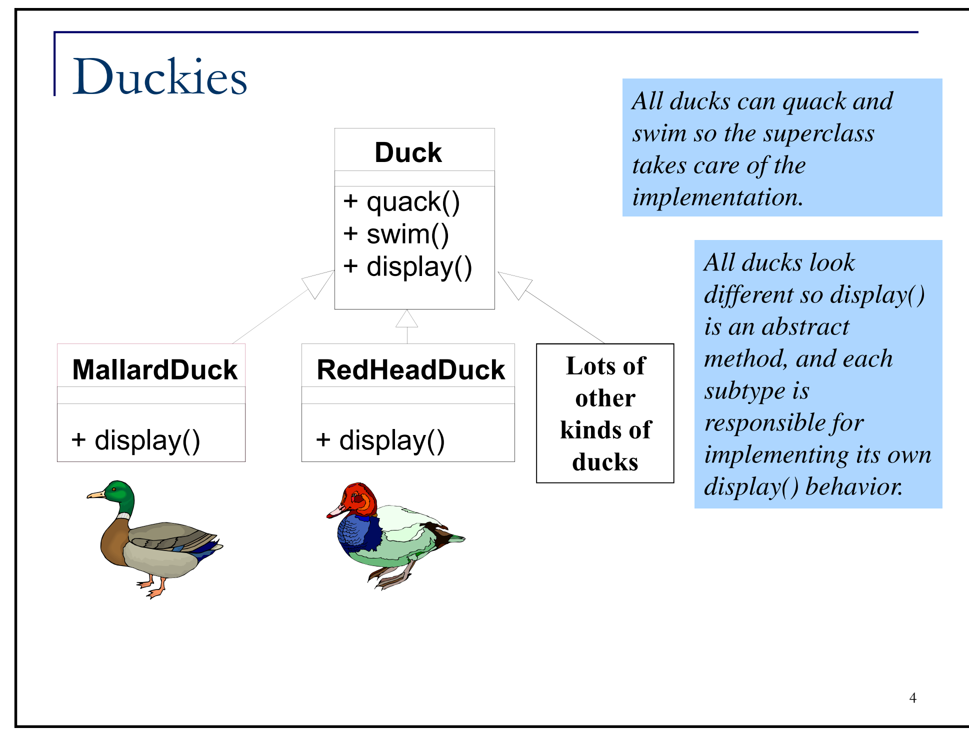
2

A Duck Story

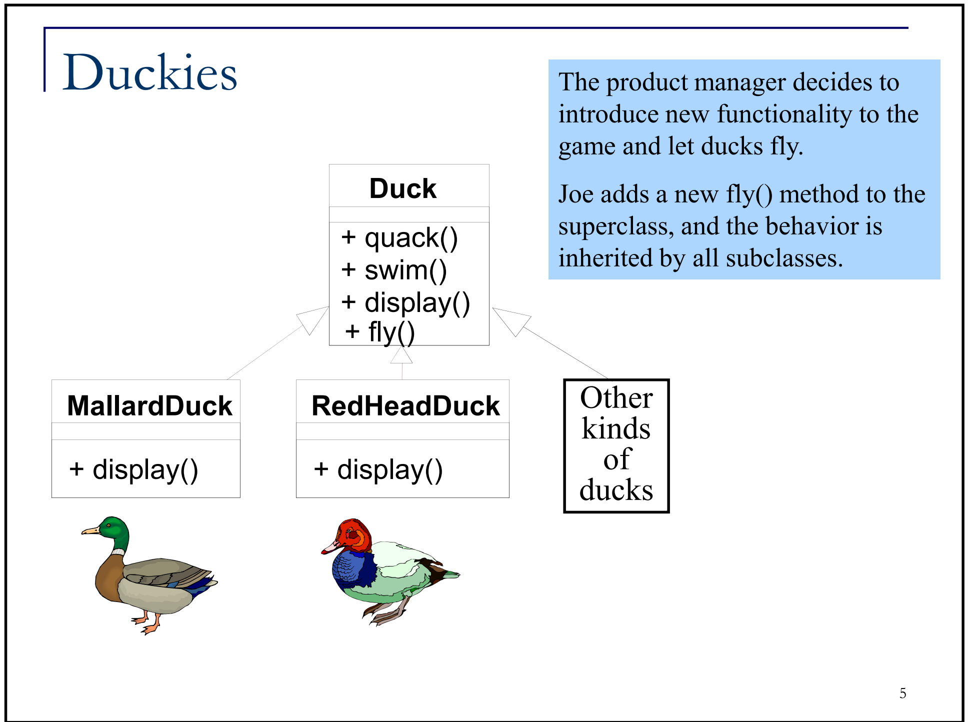
- Joe works for a company creating a simulation game called Quackers. He is an OO programmer and his job is to create the necessary functionality for the game.
- The game must support the following features:
  - It should include a variety of ducks
  - All ducks can swim
  - All ducks can quack

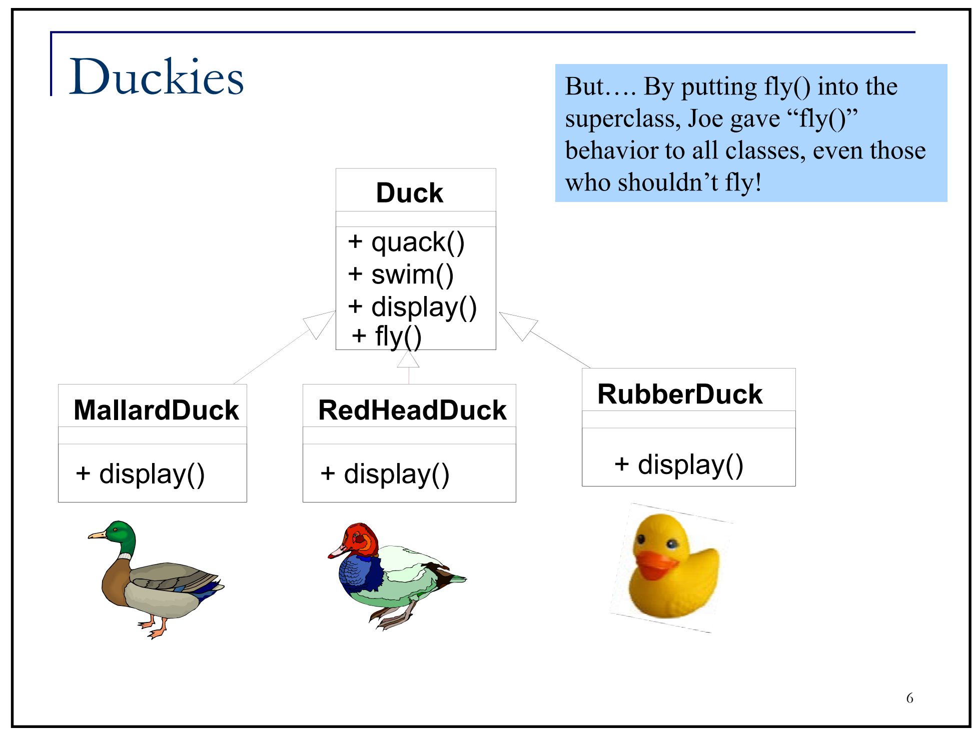
3

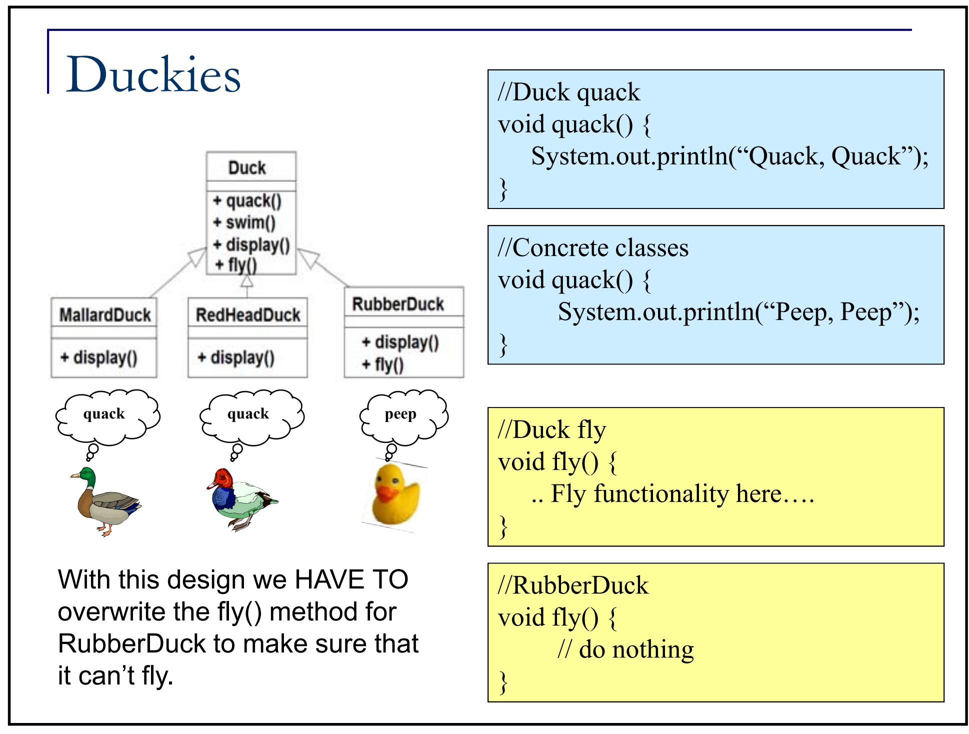
Duckies

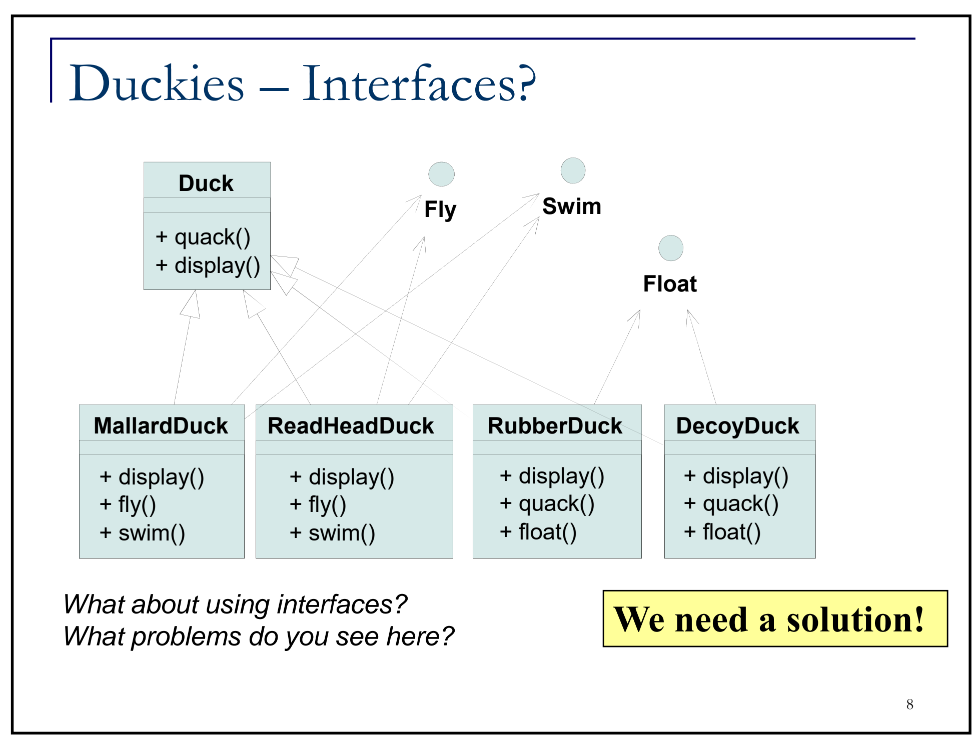


4


Duckies
The product manager decides to introduce new functionality to the game and let ducks fly.
Joe adds a new fly() method to the superclass, and the behavior is inherited by all subclasses.
Duck
+ quack()
+ swim()
+ display()
+ fly()
MallardDuck
+ display()
RedHeadDuck
+ display()
Other kinds of ducks
5


Duckies
But…. By putting fly() into the superclass, Joe gave "fly()" behavior to all classes, even those who shouldn't fly!
Duck
+ quack()
+ swim()
+ display()
+ fly()
MallardDuck
+ display()
RedHeadDuck
+ display()
RubberDuck
+ display()
6


Duckies
Duck
+ quack()
+ swim()
+ display()
+ fly()
MallardDuck
+ display()
RedHeadDuck
+ display()
RubberDuck
+ display()
+ fly()
quack
quack
peep
//Duck quack
void quack() {
System.out.println("Quack, Quack");
}
//Concrete classes
void quack() {
System.out.println("Peep, Peep");
}
//Duck fly
void fly() {
.. Fly functionality here….
}
//RubberDuck
void fly() {
// do nothing
}
With this design we HAVE TO overwrite the fly() method for RubberDuck to make sure that it can't fly.


Duckies – Interfaces?
Duck
+ quack()
+ display()
Fly
Swim
Float
MallardDuck
+ display()
+ fly()
+ swim()
ReadHeadDuck
+ display()
+ fly()
+ swim()
RubberDuck
+ display()
+ quack()
+ float()
DecoyDuck
+ display()
+ quack()
+ float()
What about using interfaces?
What problems do you see here?
We need a solution!
8

Principles of OO Design

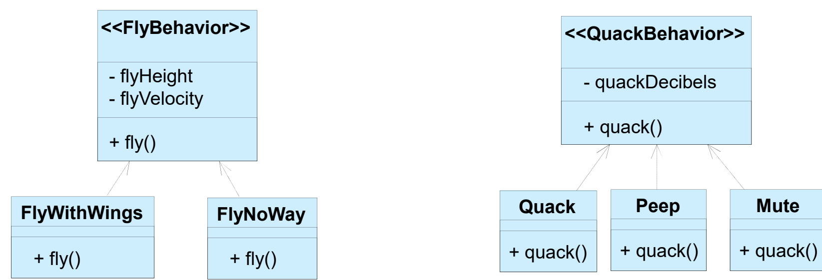
Throughout this course we will be learning about several different principles of OO design. Today we will look at three fundamental ones..

1. Separate and encapsulate the part of your design that will vary.

Think about what varies in the Quackers game.

9

Separate out the parts that will change



- Fly() and Quack() behaviors will vary.
- Create separate hierarchies of classes to represent each of these behaviors.
- Each duck will reference the appropriate fly and quack behaviors.

10

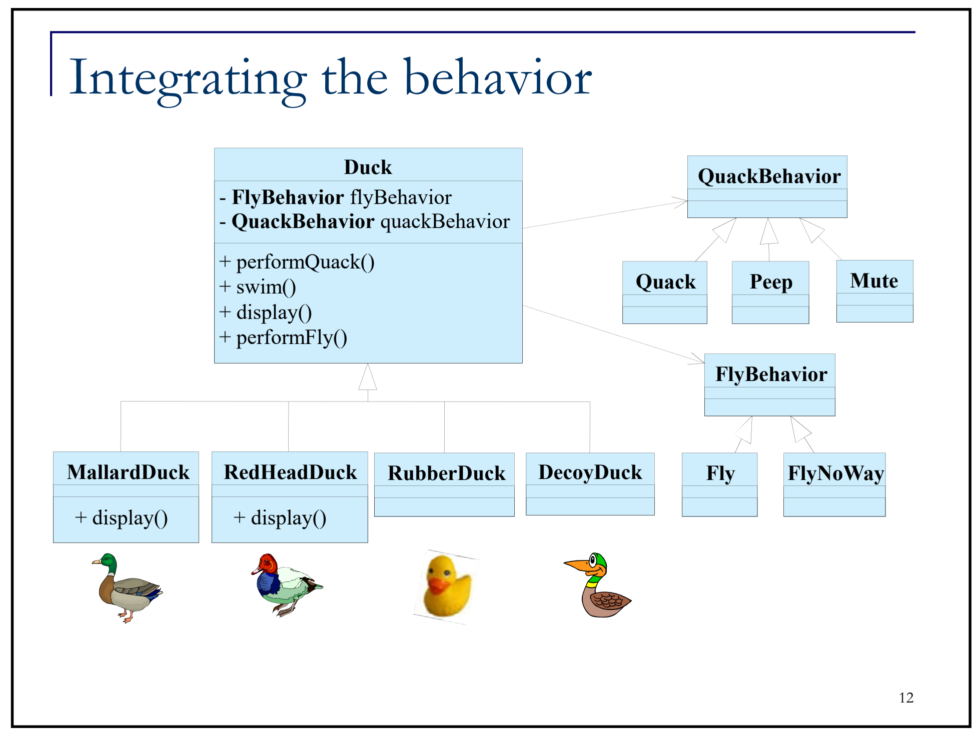
Principles of OO Design

Throughout this course we will be learning about several different principles of OO design. Today we will look at three fundamental ones..

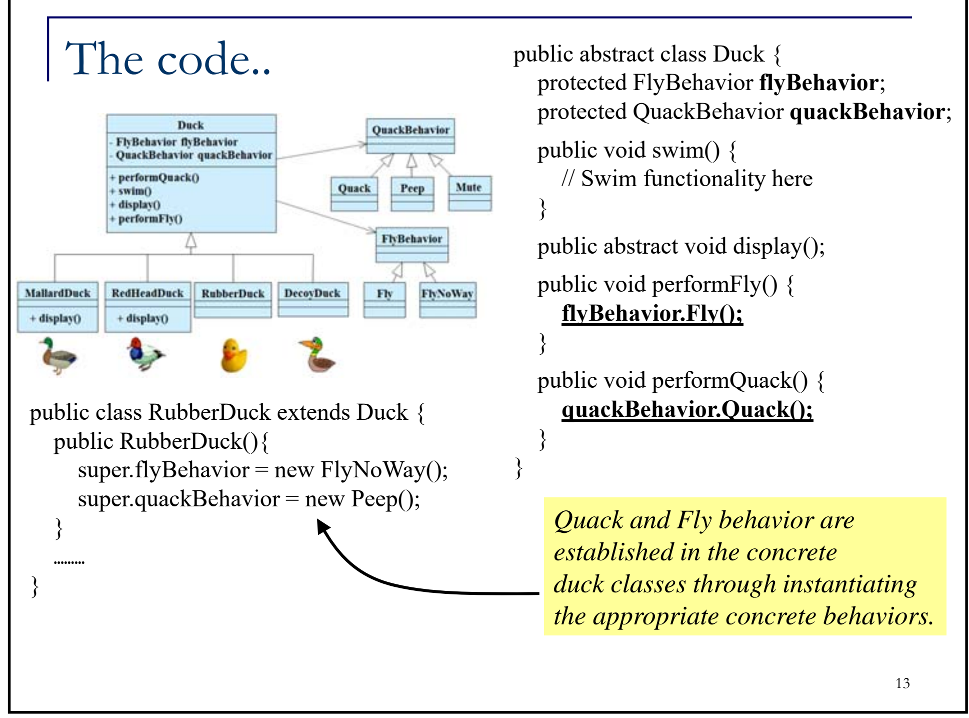
1. Separate and encapsulate the part of your design that will vary.
2. Program to an interface and not an implementation.

11

Integrating the behavior



12



Principles of OO Design

Throughout this course we will be learning about several different principles of OO design. Today we will look at three fundamental ones..

1. Separate and encapsulate the part of your design that will vary.
2. Program to an interface and not an implementation.
3. **Favor object composition over inheritance.**

14

Duckies

- Joe is given a new challenge by his boss. The competition is getting ahead.
- His challenge is to create a duck shooting game and to create different types of ducks dynamically during the game.
- Ducks needs to change their behavior too if different things happen to them.

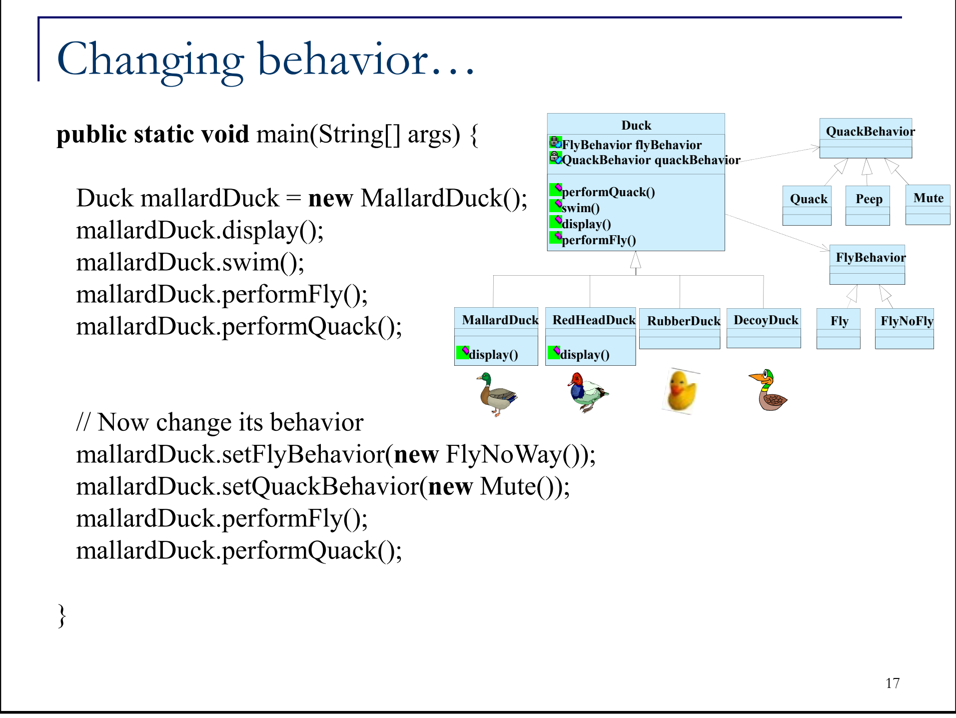
15

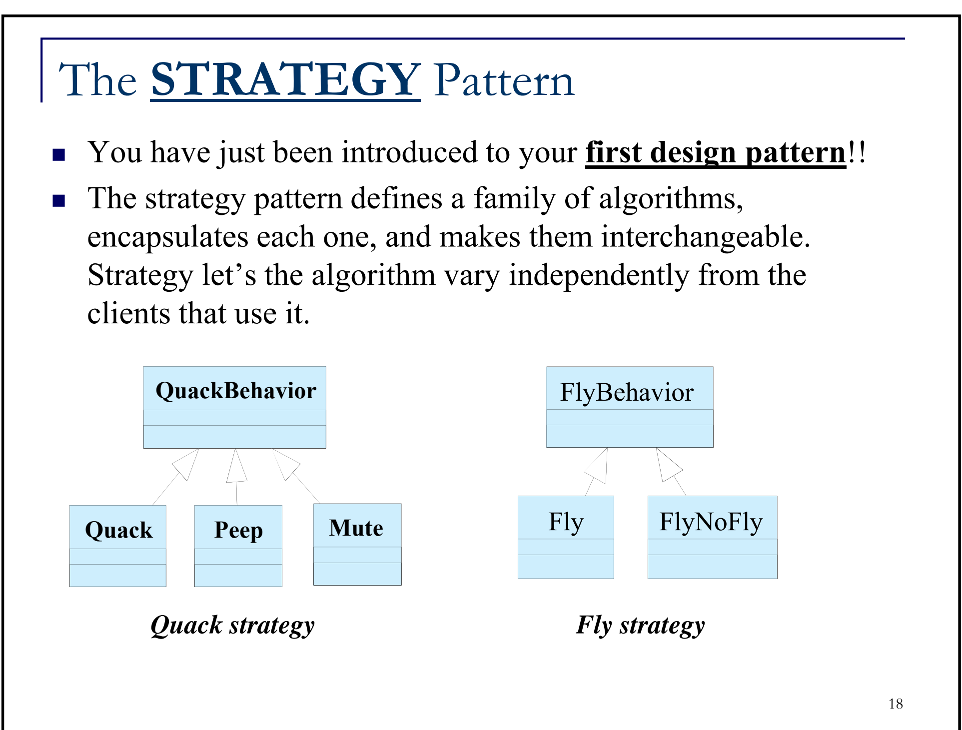
Duckies

Duck
FlyBehavior flyBehavior QuackBehavior quackBehavior
performQuack() swim() display() performFly() setFlyBehavior(FlyBehavior) setQuackBehavior(QuackBehavior)

```
public void setQuackBehavior(QuackBehavior qb){
    quackBehavior = qb;
}
public void setFlyBehavior (FlyBehavior fb){
    flyBehavior = fb;
}
```

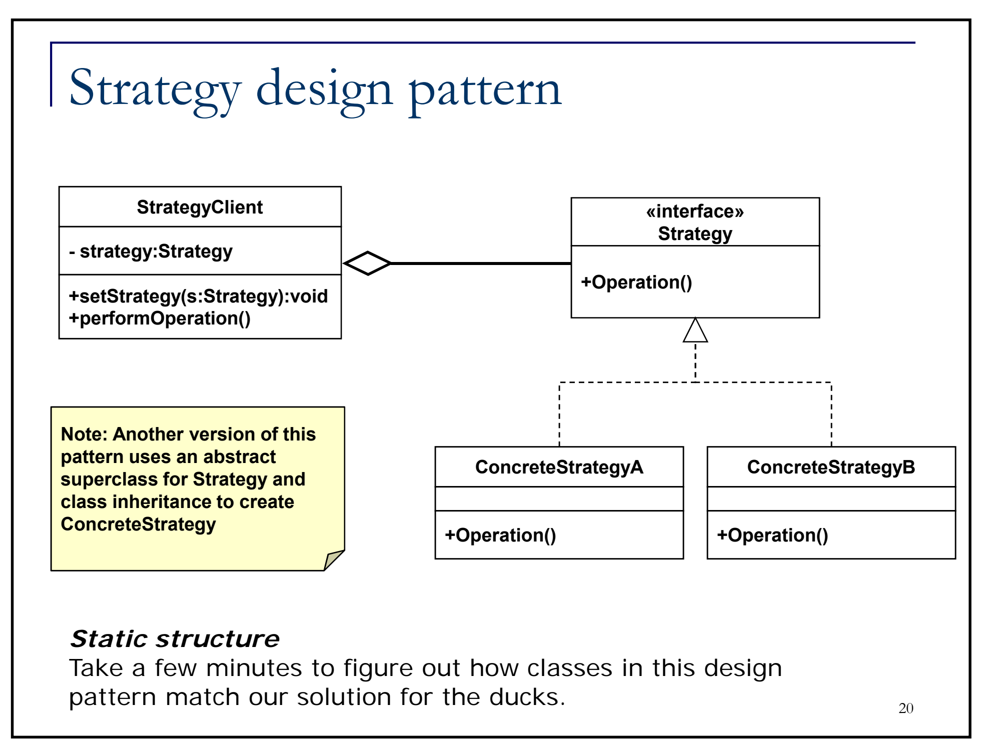
16


Changing behavior…
public static void main(String[] args) {
Duck mallardDuck = new MallardDuck();
mallardDuck.display();
mallardDuck.swim();
mallardDuck.performFly();
mallardDuck.performQuack();
// Now change its behavior
mallardDuck.setFlyBehavior(new FlyNoWay());
mallardDuck.setQuackBehavior(new Mute());
mallardDuck.performFly();
mallardDuck.performQuack();
}
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
QuackBehavior
Quack
Peep
Mute
FlyBehavior
MallardDuck
display()
RedHeadDuck
display()
RubberDuck
DecoyDuck
Fly
FlyNoFly
17


The STRATEGY Pattern
You have just been introduced to your first design pattern!!
The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy let's the algorithm vary independently from the clients that use it.
QuackBehavior
Quack
Peep
Mute
FlyBehavior
Fly
FlyNoFly
Quack strategy
Fly strategy
18

Strategy Design Pattern

19



Strategy design pattern (aka Policy)

Use the Strategy pattern when:

- You have a variety of ways to perform an action
- You might not know which approach to use until runtime
- You want to easily add new ways to perform an action
- You want to keep code maintainable as you add behaviors

Description

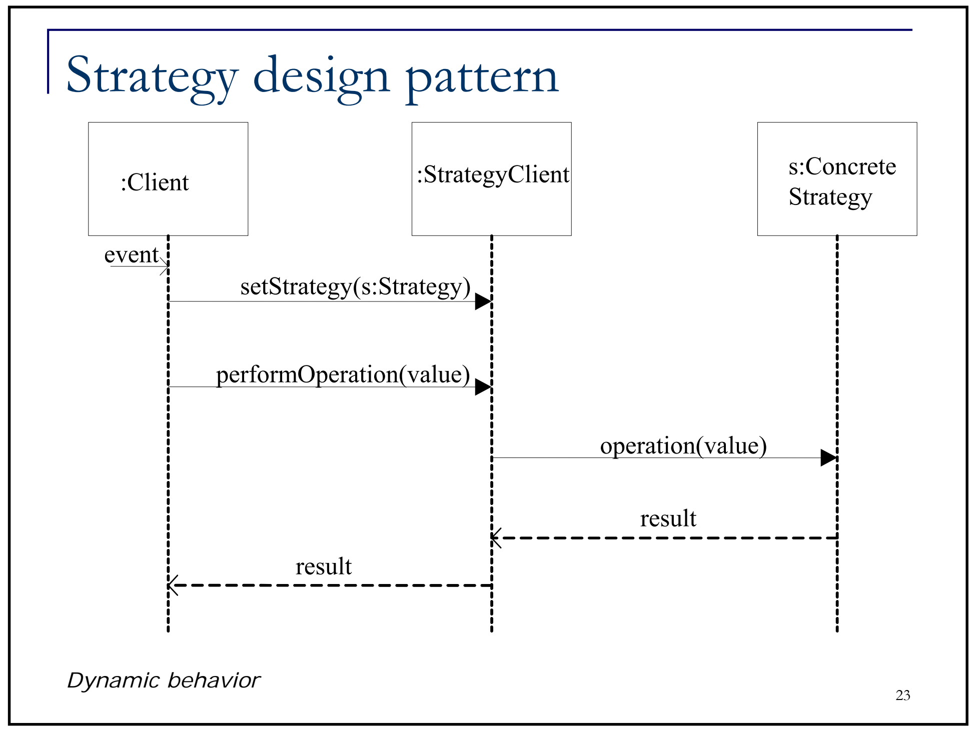
The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy let's the algorithm vary independently from the clients that use it.

21

Strategy: Implementation

- *Strategy*. Interface that defines the common operation or operations that each variant behavior must provide
- *ConcreteStrategy*. Implements the methods of the Strategy interface to provide a specific variant behavior
- *StrategyClient*. Provides operations for selecting the appropriate Strategy and performing the operations provided by Strategy

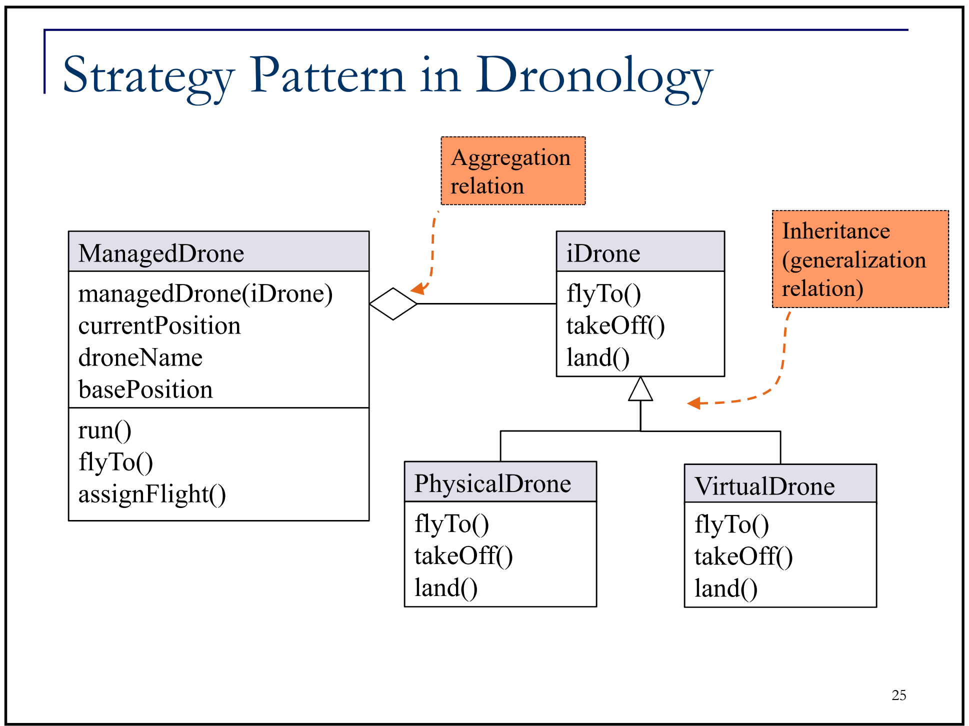
22

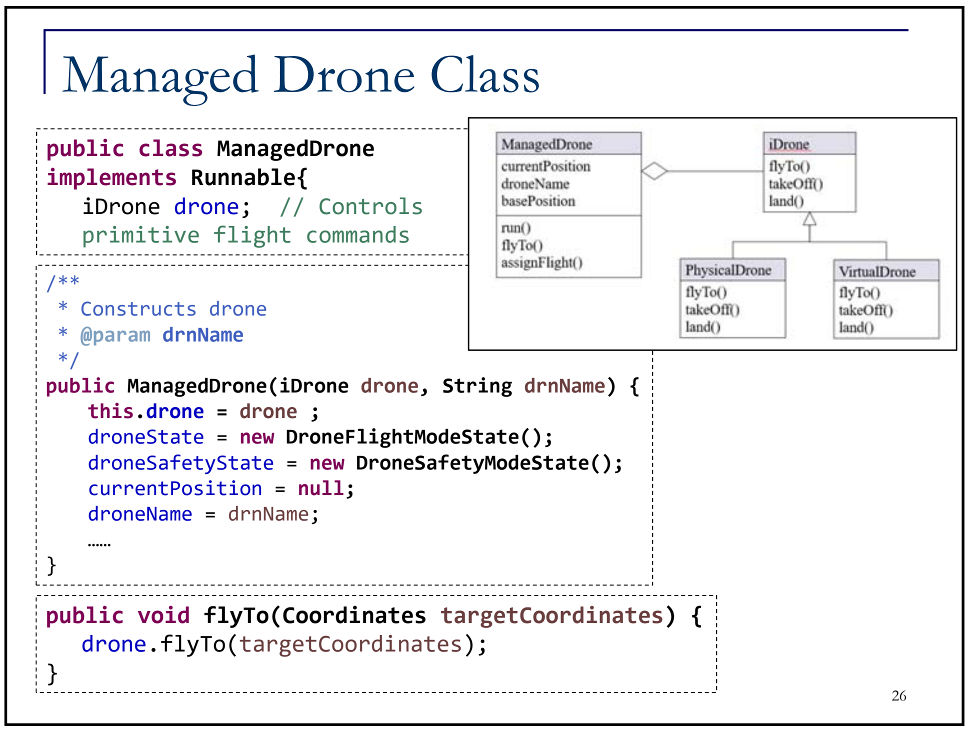


Dronology Requirements

A framework for interactively coordinating the safe flight of drone formations.

R2	The system will either be in physical mode or virtual mode but never simultaneously in both.
R3	When the system is in physical mode only physical drones will be used.
R4	When the system is in virtual mode only virtual drones will be used.
R5	The system will not switch modes at runtime.


Strategy Pattern in Dronology
Aggregation relation
Inheritance (generalization relation)
ManagedDrone
managedDrone(iDrone)
currentPosition
droneName
basePosition
run()
flyTo()
assignFlight()
iDrone
flyTo()
takeOff()
land()
PhysicalDrone
flyTo()
takeOff()
land()
VirtualDrone
flyTo()
takeOff()
land()
25


Managed Drone Class
public class ManagedDrone
implements Runnable{
iDrone drone; // Controls
primitive flight commands
/**
* Constructs drone
* @param drnName
*/
public ManagedDrone(iDrone drone, String drnName) {
this.drone = drone ;
droneState = new DroneFlightModeState();
droneSafetyState = new DroneSafetyModeState();
currentPosition = null;
droneName = drnName;
……
}
public void flyTo(Coordinates targetCoordinates) {
drone.flyTo(targetCoordinates);
}
ManagedDrone
currentPosition
droneName
basePosition
run()
flyTo()
assignFlight()
iDrone
flyTo()
takeOff()
land()
PhysicalDrone
flyTo()
takeOff()
land()
VirtualDrone
flyTo()
takeOff()
land()
26

Strategy: Who chooses it?

- In the basic form of the pattern, an external Client sets the strategy for the StrategyClient and invokes performOperation(…) on it
- *Implementation variant:* StrategyClient invokes setStrategy(…) on itself to select appropriate Strategy, based on value or characteristics of parameter passed into performOperation(…)

  This variant can make use of the **reflection mechanisms** in Java to determine the type of object passed to performOperation(…) and choose the strategy accordingly

27

Strategy: Benefits & Drawbacks

+ Easier maintenance since each behavior is defined in its own class
+ Easier to extend the behavior of an object: just create a new class
± You must identify a generic common interface for all Strategies that must also be specific enough for various concrete Strategies

28

Strategy Pattern

- In this example we want to be easily able to interchange the style for printing invoices.
- For example:
  - Plain invoice
  - Fancy invoice
  - html invoice
- Before we get started let's draw a UML class diagram for our solution.

29

InvoiceFormatter interface

```
package Invoices;
public interface InvoiceFormatter {

    // Format header
    String formatHeader();

    // Format line item
    String formatLineItem(LineItem item);

    // Format footer
    String formatFooter(double invoiceTotal);
}
```

Methods that concrete classes MUST override

** Comments etc have been removed for display purposes.*

30

Concrete FancyFormatter

```
package Invoices;
public class FancyFormatter implements InvoiceFormatter {
   public String formatHeader() {
      return "\n~~~~~~\nI N V O I C E\n~~~~~~\n\n\n";
   }
   public String formatLineItem(LineItem item)  {
      return (String.format("%s: $%.2f\n", item.toString(),
         item.getPrice()));
   }
   public String formatFooter(double total) {
      return (String.format("\nPLEASE REMIT THE
         FOLLOWING AMOUNT: $%.2f\n", total));
   }
}
```

31

Line Item (a supporting class)

```
package Invoices;
public class LineItem {
   private String description;
   private double price;
   public LineItem (String description, double price){
      this.description = description;
      this.price = price;
   }
   public double getPrice() { return price;}
   public String toString() { return description;}
}
```

32

Invoice

```
public class Invoice {
   private List<LineItem> items = new ArrayList<LineItem>();
   public void addItem(LineItem item) {
      items.add(item);
   }

   public String format(InvoiceFormatter formatter) {
      StringBuilder sb = new StringBuilder();
      sb.append(formatter.formatHeader());
      double invoiceTotal = 0.0;
      for (Iterator it = items.iterator (); it.hasNext (); ) {
         LineItem i = (LineItem)it.next ();
         sb.append(formatter.formatLineItem(i) + "\n“);
         invoiceTotal += i.getPrice();
      }
      sb.append(formatter.formatFooter(invoiceTotal));
      return sb.toString();
   }
}
```

A formatter strategy is received as an argument and then used to format the header, line items, and footer.

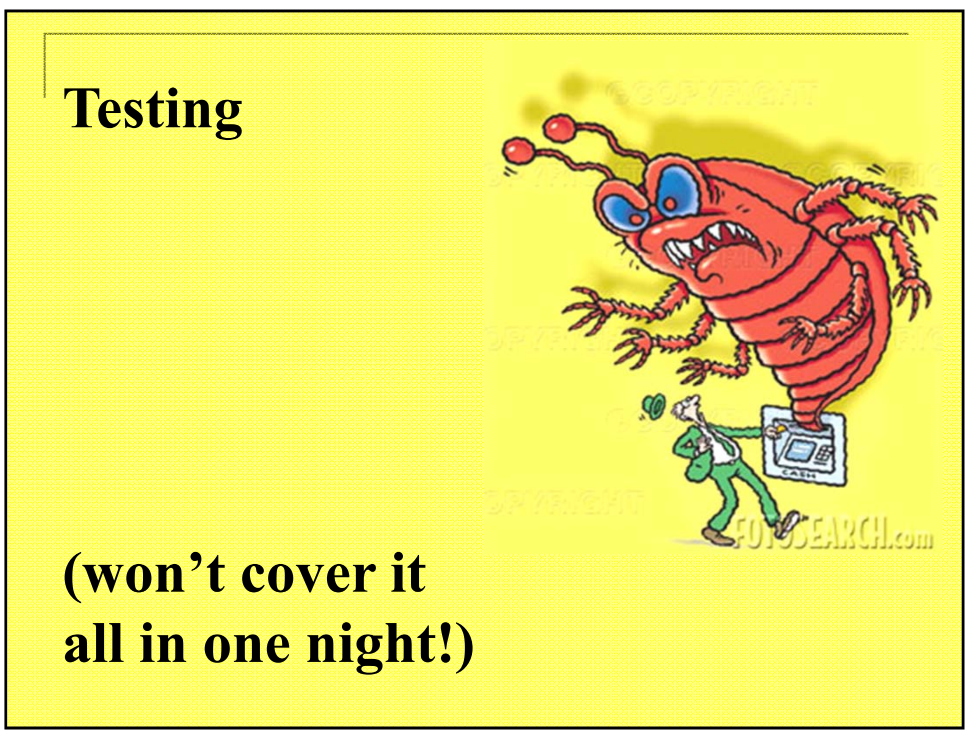
InvoiceTester

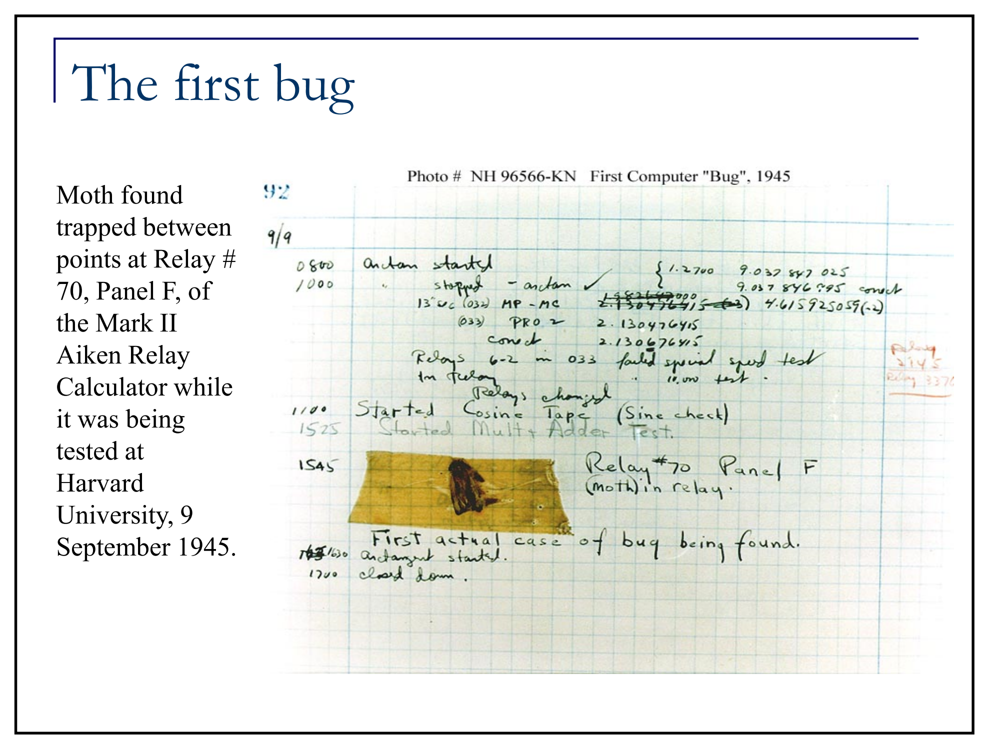
Create two types of formatter.

```
package Invoices;
public class InvoiceTester {

   public static void main(String[] args) {
      final Invoice invoice = new Invoice();
      final InvoiceFormatter formatter = new SimpleFormatter();
      final InvoiceFormatter formatter2 = new FancyFormatter();

      // Add line items  to the invoice.
      invoice.addItem(new LineItem("Hammer", 19.95));
      invoice.addItem(new LineItem("Assorted nails", 9.75));
      ……

      String displayInvoice = invoice.format(formatter);
      System.out.println(displayInvoice);
   }
}
```

Pass the desired formatter to invoice.format(..)

34


Testing
(won't cover it
all in one night!)
FOTOSEARCH.com


The first bug
Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945.
Photo # NH 96566-KN First Computer "Bug", 1945
92
9/9
0800 Arctan started
1000 stopped - arctan ✓
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

Why we TEST

- We test software because we cannot guarantee its correctness – under normal development practices.
- Testing is the art of devising and executing test cases that have a high likelihood of finding errors.
- A small subset of faults accounts for most failures during operation.
  We need to 'test smart' in order to find these faults.
- A high-quality product will experience few failures.
  Remember the five 9s of reliability!

Verification and Validation

- Testing is just part of a broader topic referred to as Verification and Validation (V&V)
- Pressman:
  - Verification: Are we building the product right?
  - Validation: Are we building the right product?
- IEEE standard 1012-1998:
  - Requirements validation is the process of evaluating an implemented system to determine whether it conforms to the specified requirements.
- SWEBOK:
  - Validation is the process of ensuring that the engineer has understood the requirements correctly, in other words "Have we got the right requirements?"

Some words of wisdom

To what extent do you agree with these quotes?

- *"Testing only to end user requirements is like inspecting a building based on the work done by the interior designer at the expense of the foundations, girders, and plumbing"*
  Boris Beizer.
- *"Optimism is the occupational hazard of programming; testing is the treatment."*
  Kent Beck
- *"The first mistake that people make is thinking that the testing team is responsible for assuring quality."*
  Brian Marick.

Who Tests the Software?

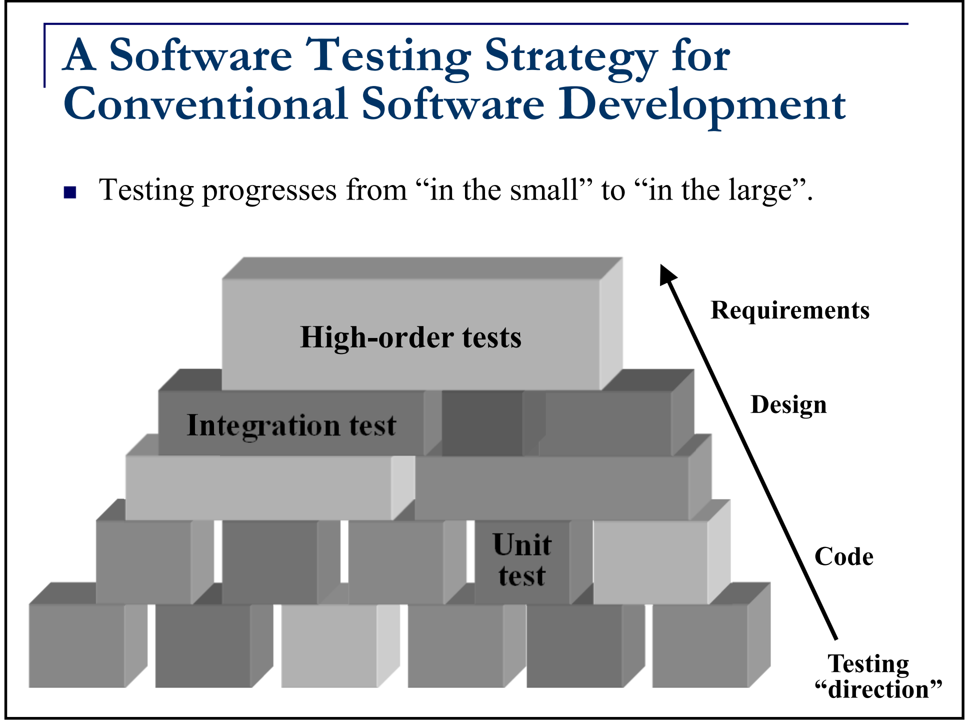
developer

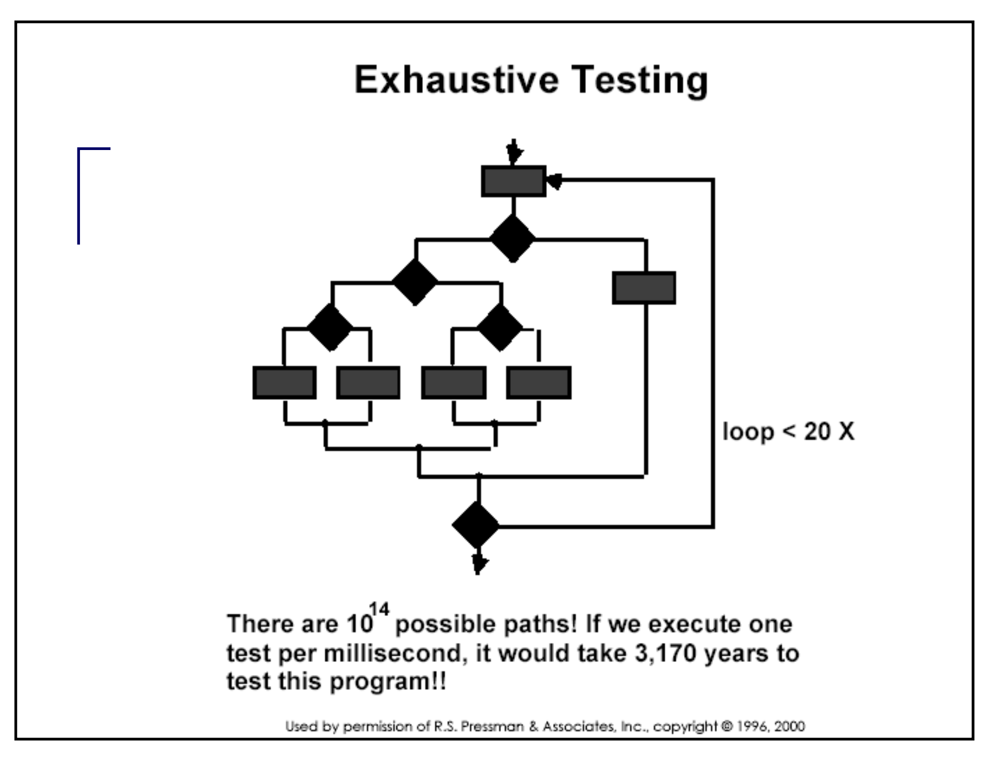
Understands the system
but, will test "gently"
and, is driven by "delivery"

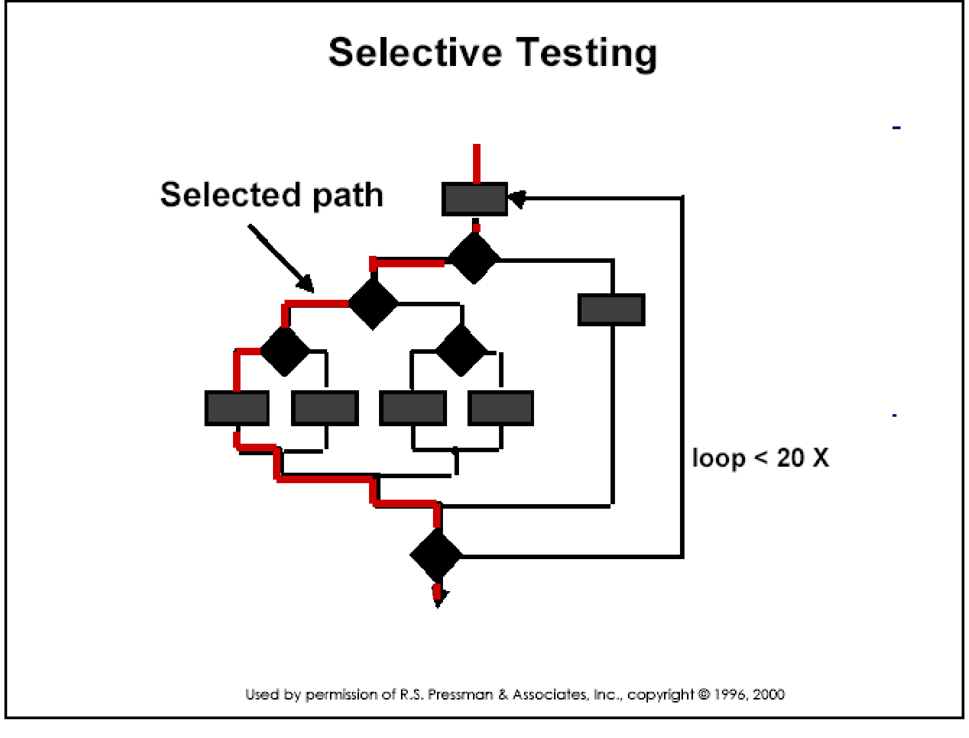
independent tester

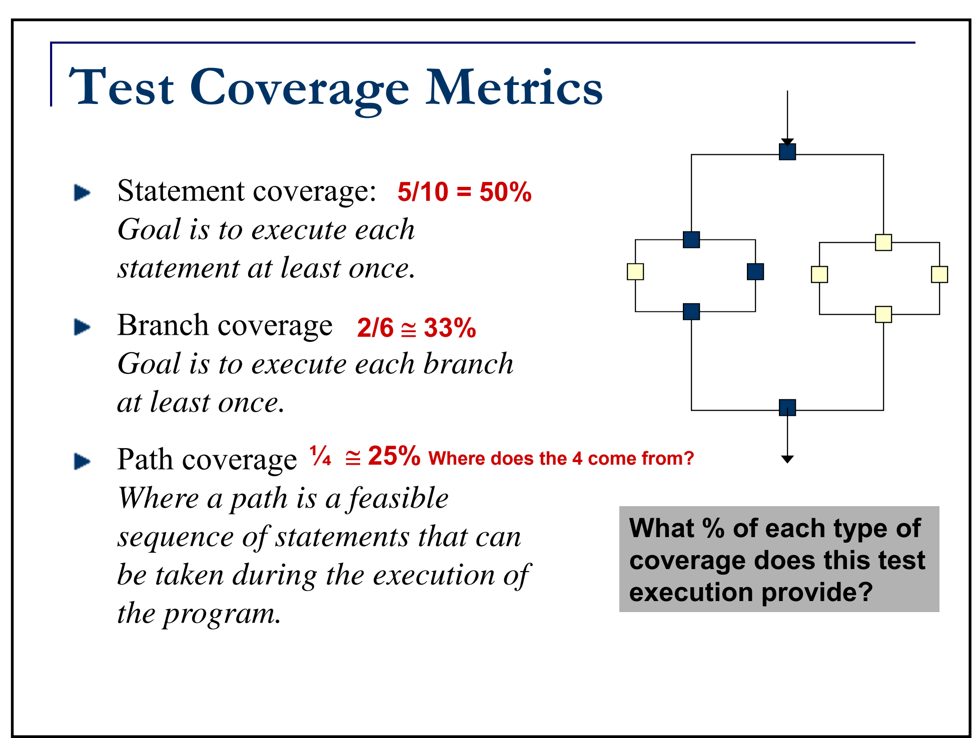
Must learn about the system,
but, will attempt to break it
and, is driven by quality

Used by permission of R.S. Pressman & Associates, Inc., copyright © 1996, 2000


A Software Testing Strategy for Conventional Software Development
Testing progresses from "in the small" to "in the large".
High-order tests
Integration test
Unit test
Requirements
Design
Code
Testing "direction"


Exhaustive Testing
loop < 20 X
There are 10^14 possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!
Used by permission of R.S. Pressman & Associates, Inc., copyright © 1996, 2000


Selective Testing
Selected path
loop < 20 X
Used by permission of R.S. Pressman & Associates, Inc., copyright © 1996, 2000


Test Coverage Metrics
Statement coverage: 5/10 = 50%
Goal is to execute each statement at least once.
Branch coverage 2/6 ≅ 33%
Goal is to execute each branch at least once.
Path coverage ¼ ≅ 25% Where does the 4 come from?
Where a path is a feasible sequence of statements that can be taken during the execution of the program.
What % of each type of coverage does this test execution provide?

A Strategic Approach to Testing

- Specify product requirements in a quantifiable manner long before testing commences.
  - Quantify all non-functional requirements.
  - Remove ambiguities (etc, etc)
- State testing objectives explicitly. For example:
  - Test effectiveness
  - Test coverage
  - Mean time to failure
  - The cost to find and fix defects
  - Remaining defect density
  - Test work-hours per regression test.

A Strategic Approach to Testing

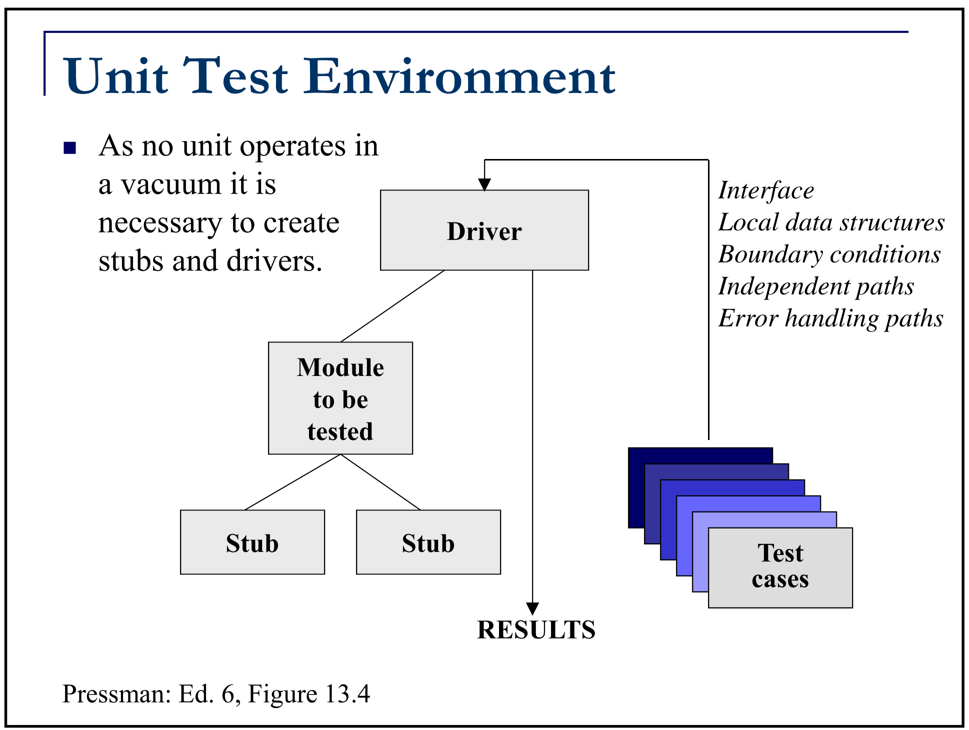
- Understand the users of the software and develop a profile for each user category.
  - Focus testing on actual use of the product.
- Develop a testing plan that emphasizes "rapid cycle testing."
  - Test field "trialable" increments of functionality
  - Feedback generated from rapid cycle tests can be used to control quality levels and subsequent test strategies.
- Build "robust" software that is designed to test itself.
  - Software should be designed with inbuilt antibugging techniques.
  - It should be capable of diagnosing certain classes of errors.
  - Design should accommodate automated testing and regression testing.

A Strategic Approach to Testing

- Use effective formal technical reviews as a filter prior to testing.
  - FTRs have been shown to be as effective as testing in uncovering errors.
- Conduct FTRs to assess the test strategy and test cases themselves.
  - Uncover inconsistencies, omissions, and outright errors in the testing approach.
- Develop a continuous improvement approach for the testing process.
  - Test strategy should be measured.
  - Metrics collected should be used as part of a statistical process control approach for software testing.

Unit Testing

- Focuses on a single software component or module.
- Design description guides test generation to
  - Ensure coverage of important control paths
  - Test the boundaries of the module.
- Focuses on internal processing logic and data structures.
- Specific tests
  - Does information flow correctly into and out of the unit?
  - Does data stored in local data structures maintain its integrity during ALL steps in the algorithm's execution.
- Common errors
  - Incorrect arithmetic precedence
  - Incorrect initializations
  - Precision inaccuracy


Unit Test Environment
As no unit operates in a vacuum it is necessary to create stubs and drivers.
Driver
Module to be tested
Stub
Stub
RESULTS
Interface
Local data structures
Boundary conditions
Independent paths
Error handling paths
Test cases
Pressman: Ed. 6, Figure 13.4