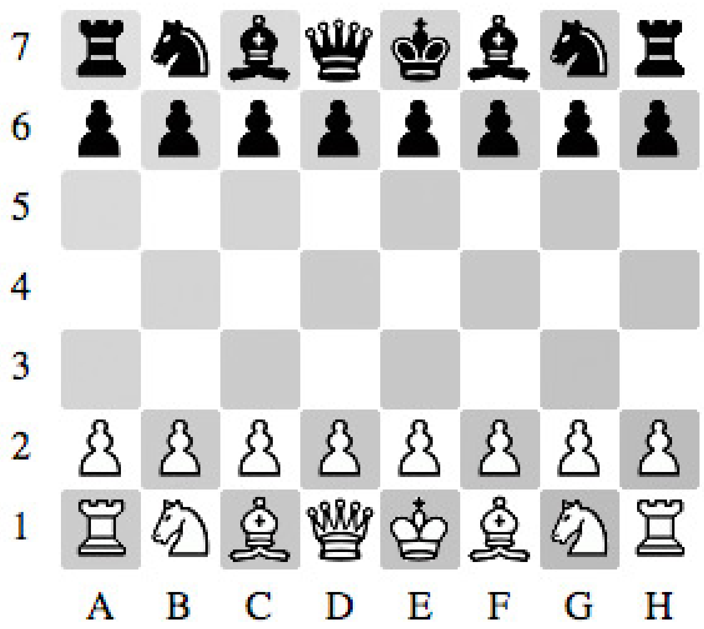
Victoria University
f Wellington, New Zealan
Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa

SWEN221: Software Development

11: Testing II

David J. Pearce & Nicholas Cameron & James Noble
Engineering and Computer Science, Victoria University

Simplified Chess




```
7|r|n|b|q|k|b|n|r|
6|p|p|p|p|p|p|p|p|
5|_|_|_|_|_|_|_|_|
4|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
2|P|P|P|P|P|P|P|P|
1|R|N|B|Q|K|B|N|R|
  a b c d e f g h
```

- Board dimensions: **7 rows** by **8 columns**
- No **check** or **check mate**. Win by **taking King**.
- No **draws**, but lose if cannot move any piece or can only move King.
- Pawns always move one space (no **double move**; no **en-passant**)
- **No castling**. Pawns can only be **promoted** to captured piece, and cannot move onto last row if no captured piece.
- **See:** https://www.chessvariants.com/rules/simplified-chess

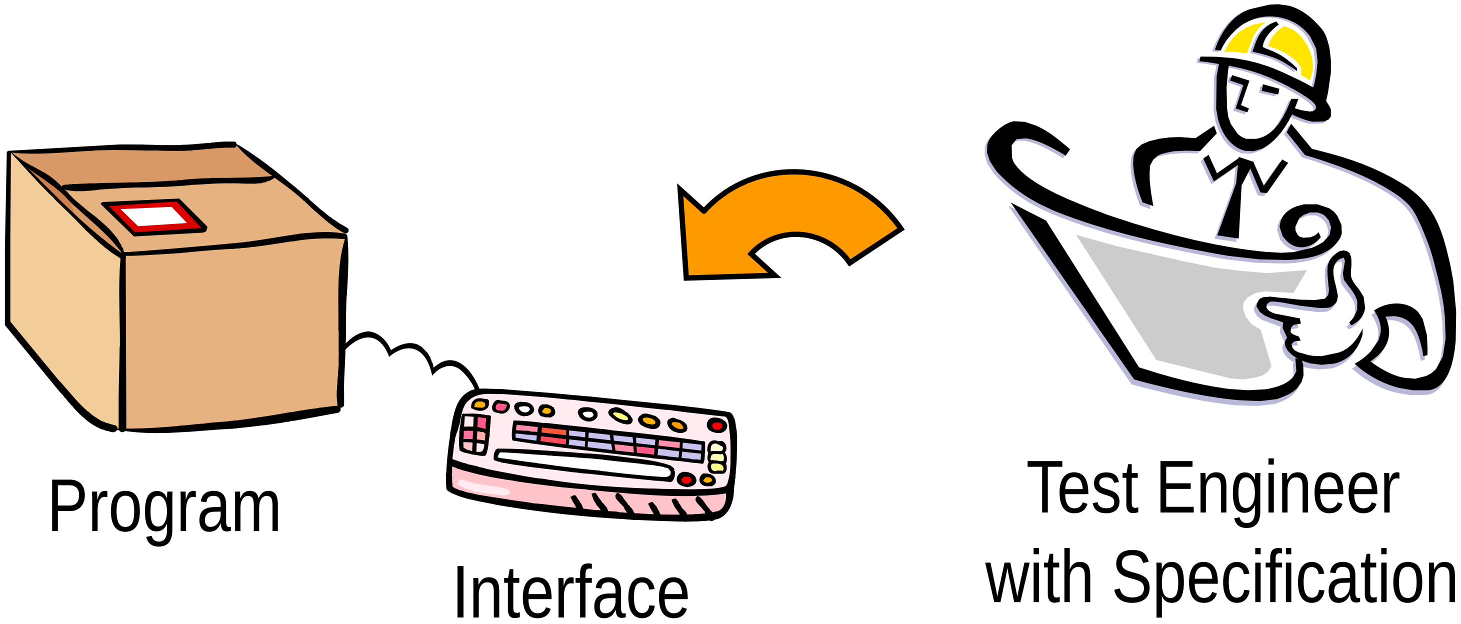
Long Algebraic Notation

```
String input = "e2-e3 e6-e5; Bf1-d3";
// Define the expected output
String expected =
             "7|r|n|b|q|k|b|n|r|\n"+
             "6|p|p|p|p|_|p|p|p|\n"+
             "5|_|_|_|_|p|_|_|_|\n"+
             "4|_|_|_|_|_|_|_|_|\n"+
             "3|_|_|_|B|P|_|_|_|\n"+
             "2|P|P|P|P|_|P|P|P|\n"+
             "1|R|N|B|Q|K|_|N|R|\n"+
             "  a b c d e f g h";
// Execute the move sequence
Game game = execute(input);
// Check got expected output
assertEquals(expected,game.toString());
```

"In **long algebraic notation**, moves specify both the starting and ending squares separated by a hyphen, for example: e2-e4 or Nb1-c3. Captures are still indicated using "x": Rd3xd7. "

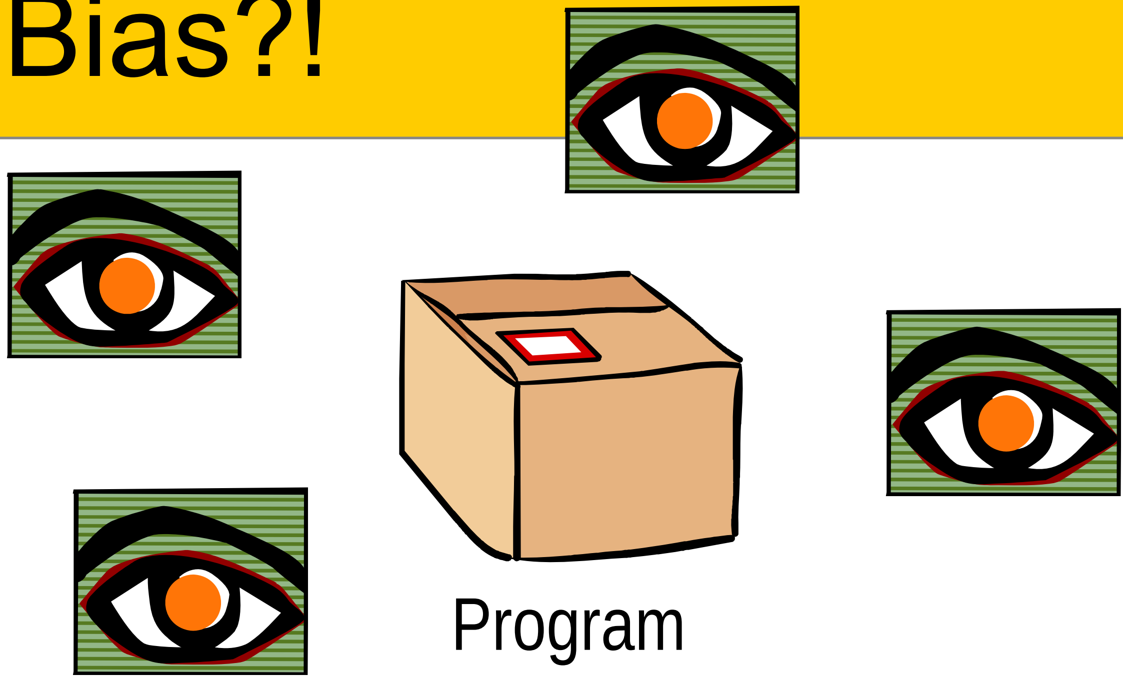
https://en.wikipedia.org/wiki/Algebraic_notation_(chess)

Black-box Testing



- Testing *without knowledge of implementation*
  - Test cases generated directly from specification
  - Gives unbiased approach
  - Robust to implementation changes

What Bias?!



- Biases introduced by programmer:
  - Programmer may misinterpret specification
    - This misinterpretation may be repeated in his test
  - Programmer may "believe" a particular part is well-coded
    - He/she might omit tests because of this to save time
  - Programmer unlikely to represent target audience
    - What he/she finds acceptable others may not
  - Bottom-line: more eyeballs = greater chance of finding problems

Black-Box Testing (cont'd)

- Test **typical inputs**
  - Values that your program is likely to encounter
  - E.g. single pawn move for ChessView
- Test **boundary conditions**
  - Values at edges of valid input domain
  - E.g. off-by-one error:

```
int nextDay(int day) {
 // 1 <= day <= 7
 if(day > 7) { return 1; }
 else { return day + 1; }
}
```

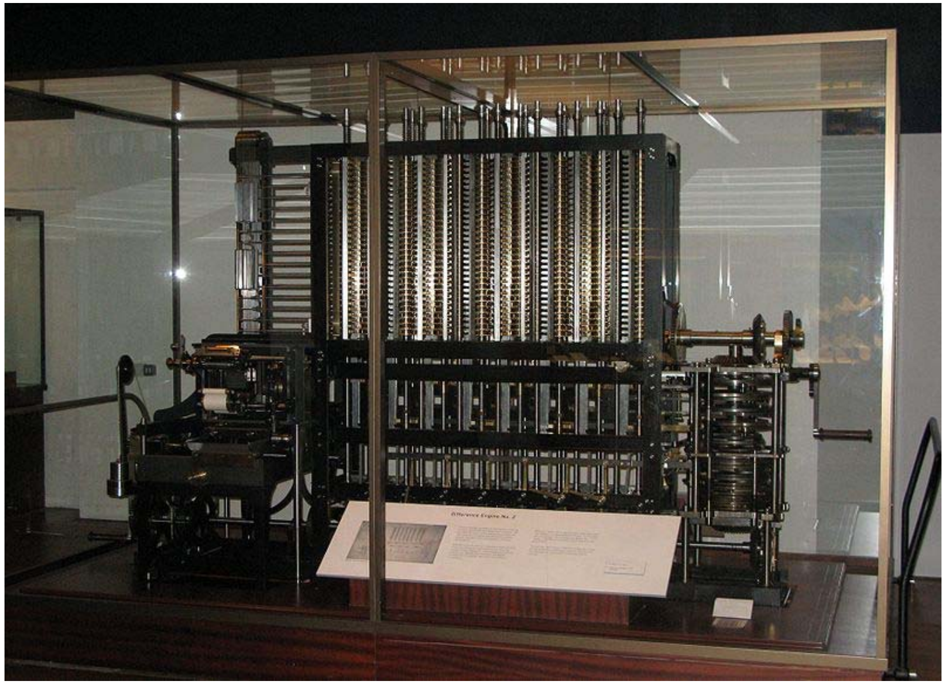
Quiz: Find Good Boundary Tests

```
class TableRow<T> {
 private List<T> rows;

 public TableRow() { this.rows = new ArrayList<T>(); }

 public TableRow(List<T> rows) { this.rows = rows; }

 public T get(int index) { return rows.get(index); }

/**
  * Copy elements from this TableRow into parameter to
  */
 void copy(List<T> to) {
  for(int i=0;i!=rows.size();++i) {
   to.add(rows.get(i));
}}}
```

Quiz: Space for answers!

```
@Test void testAdd1() {



}
@Test void testAdd2() {



}
@Test void testAdd3() {



}
@Test void testAdd4() {



}
```

White-Box Testing (A.K.A. Glass-Box)



- Testing *with complete knowledge of implementation*
  - Test cases generated by looking at program code
  - Aim to reach high-degree of *code covereage*
  - Gives potentially biased approach
  - Not robust to implementation changes

White-Box testing

```
int sumSmallest(List<Integer> v1, List<Integer> v2) {
 // sum smallest list
 int r = 0;
 if(v1.size() < v2.size()) {
  for(int i=0;i != v1.size();++i) { r += v1.get(i); }
 } else {
  for(int i=0;i != v2.size();++i) { r += v2.get(i); }
 }
 return r;
}
```

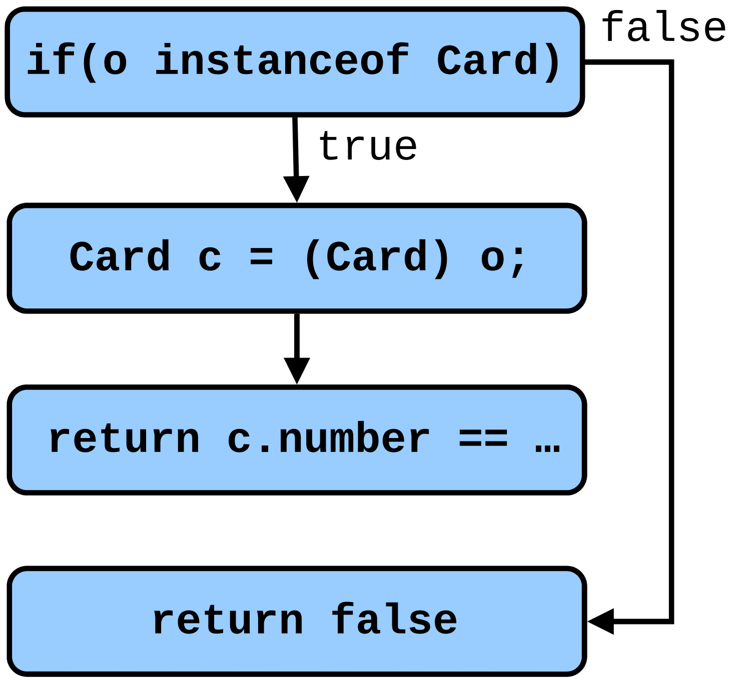
- What's wrong with these test cases?
  - (v1=[1,5,4,3], v2=[4,2,3])
  - (v1=[4], v2=[5])
  - (v1=[5], v2=[])

Code-Coverage

- Want test cases to cover X% of code
  - E.g. > 85% of code covered by tests
  - But, how to measure code coverage?
- Example *Coverage Criteria*:
  - **Function Coverage:** number of methods invoked / # methods
  - **Statement Coverage**: number of statements executed / # statements
  - **Branch Coverage**: number of branches where both true and false side tested / # branches
- Calculating Code Coverage
  1. Select Criteria
  2. Construct Control-Flow Representation (next slide)
  3. Mark nodes Executed Based on Tests
  4. Compute Coverage

Control-Flow Graph

```
public boolean equals(Object o) {
 if(o instanceof Card) {
   Card c = (Card) o;
   return c.number == number
          && c.suit == suit;
  }
  return false;
}
```



- Computing Coverage
  - Must be clear what counts and what doesn't
  - Requires precise representation of program
  - Control-Flow Graph (CFG) useful here
  - Nodes represent statements
  - Edges represent branching

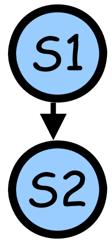
Control-Flow Graph (cont'd)

- Unit Statements
  - No branching (i.e. only one way through)
  - One node in CFG for each of these
  - E.g. assignment, method call, return, etc

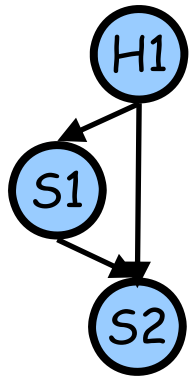
- Branching Statements
  - Cause branches
  - One node in CFG for "header"
  - E.g. ifs, for/while loops, etc

CFG Construction Examples

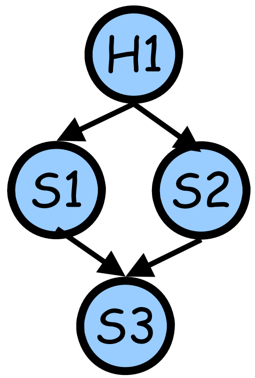
```
S1 ; S2
```



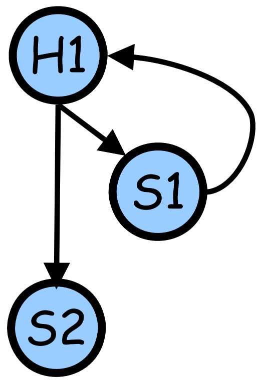

```
if(…) { S1 }
S2
```




```
if(…) { S1 } else { S2 }
S3
```

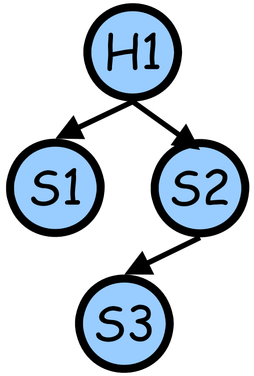



```
while(…) { S1 }
S2
```

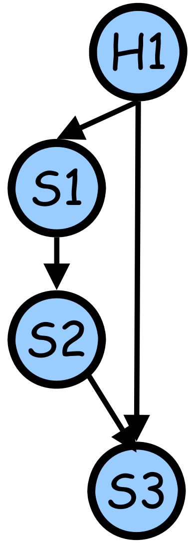


More Construction Examples

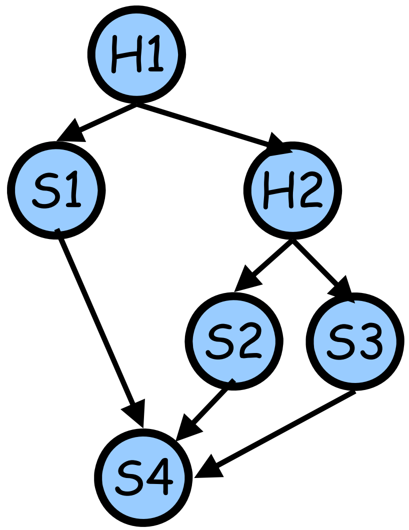
```
if(…) { return; }
else { S2 }
S3
```




```
if(…) { S1 ; S2 }
S3
```




```
if(…) { S1 }
else if(…) { S2 }
else { S3 }
S4
```



Example

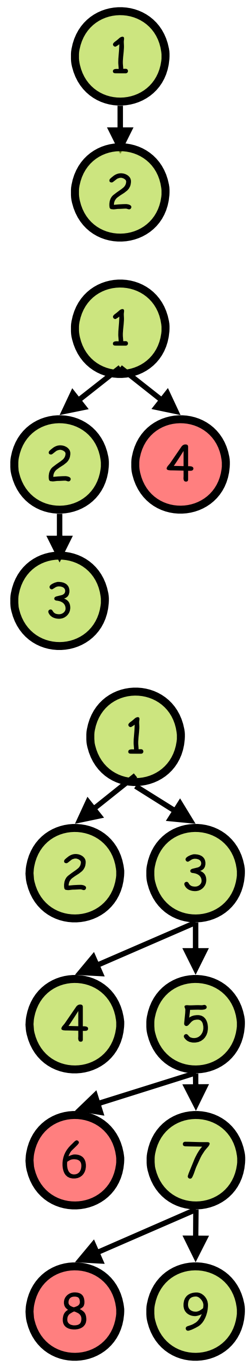
```
class Card {
 private int number, suit;

 public Card(int n, int s) { number = n; suit = s; }

 public boolean equals(Object o) {
  if(o instanceof Card) {
   Card c = (Card) o;
   return c.number == number && c.suit == suit;
  }
  return false;
 }

 public int compareTo(Card c) {
  if(suit > c.suit) { return -1; }
  else if(suit < c.suit) { return 1; }
  else if(number < c.number) { return -1; }
  else if(number > c.number) { return 1; }
  else { return 0; }
}}
```

Example (continued)

```
@Test void testEquals() {
 assertTrue(new Card(1,2).equals(new Card(1,2)));
}
@Test void testCompareEquals() {
 assertTrue(new Card(1,2).compareTo(new Card(1,2)) == 0);
}
@Test void testCompareLess() {
 assertTrue(new Card(2,3).compareTo(new Card(2,1)) < 0);
}

@Test void testCompareGreater() {
 assertTrue(new Card(2,1).compareTo(new Card(2,3)) > 0);
}
```

- Based on these, Calculate (as %):
  - Method Coverage
  - Statement Coverage
  - Branch Coverage

```
class Card {
 private int number, suit;

 public Card(int n, int s) { number = n; suit = s; }

 public boolean equals(Object o) {
  if(o instanceof Card) {
   Card c = (Card) o;
   return c.number == number && c.suit == suit;
  }
  return false;
 }

 public int compareTo(Card c) {
  if(suit > c.suit) { return -1; }
  else if(suit < c.suit) { return 1; }
  else if(number < c.number) { return -1; }
  else if(number > c.number) { return 1; }
  else { return 0; }
}}
```



Method Coverage = 3 / 3 = **100%**
Statement Coverage = 12 / 15 = **80%**
Branch Coverage = 2 / 5 = **40%**