Lecture 17

ECE 1145: Software Construction and Evolution

Systematic Testing
(CH 34)

Announcements

- Iteration 6: Compositional Design due Nov. 7
  - Bonus: EtaCiv due Dec. 12
- Relevant Exercises: 34.3 (Equivalence Classes on Breakthrough game described in exercise 5.4)
- Midterm grades posted (see Gradescope)

Questions for Today

How do we write tests efficiently?

Review: TDD

Tests can only demonstrate the presence of defects, **not the absence of defects.**

Review: TDD

Tests can only demonstrate the presence of defects, **not the absence of defects.**

→ How can we increase the chance of finding defects?

Systematic Testing

Systematic testing is a planned and systematic process with the goal of improving the chance of finding defects while limiting the number of test cases

Systematic Testing

Systematic testing is a planned and systematic process with the goal of improving the chance of finding defects while limiting the number of test cases

It is a **complement** to TDD, not an alternative

Testing Approaches

Testing approach depends on the **complexity** of the unit under test:

Testing Approaches

Testing approach depends on the **complexity** of the unit under test:

- **No testing** : small methods (e.g., get/set methods)

Testing Approaches

Testing approach depends on the **complexity** of the unit under test:

- **No testing** : small methods (e.g., get/set methods)
- **Explorative testing** : made based on experience, not following any rigid method; low cost, best for medium-complexity methods

Testing Approaches

Testing approach depends on the **complexity** of the unit under test:

- **No testing** : small methods (e.g., get/set methods)
- **Explorative testing** : made based on experience, not following any rigid method; low cost, best for medium-complexity methods
- **Systematic testing** : follow a rigid method for generating test cases in order to increase the probability of finding defects
  - Costly, best for highly complex methods or systems where reliability is very important (e.g., for safety)

Testing Approaches

Testing approach depends on the **complexity** of the unit under test:

- **No testing** : small methods (e.g., get/set methods)
- **Explorative testing** : made based on experience, not following any rigid method; low cost, best for medium-complexity methods
- **Systematic testing** : follow a rigid method for generating test cases in order to increase the probability of finding defects
  - Costly, best for highly complex methods or systems where reliability is very important (e.g., for safety)

→Balance the **effort** of testing with expected increase in **reliability** (e.g., focus on testing default configurations)

Systematic Testing

Black-box testing: The UUT is treated as an opaque “black box”

White-box testing: The full implementation of the UUT is known (transparent or “white box”)

Systematic Testing

Black-box testing: The UUT is treated as an opaque “black box”
→ Use the **specification** of the UUT and a **general knowledge** of programming techniques, constructs, and common programming mistakes to guide testing

White-box testing: The full implementation of the UUT is known (transparent or “white box”)
→ Actual code can be inspected to generate test cases

Systematic Testing

Black-box testing: The UUT is treated as an opaque "black box"
→ Use the **specification** of the UUT and a **general knowledge** of programming techniques, constructs, and common programming mistakes to guide testing

White-box testing: The full implementation of the UUT is known (transparent or "white box")
→ Actual code can be inspected to generate test cases

We will focus on black-box testing → Implementation-independent

- Equivalence Class Partitioning
- Boundary Value Analysis

Equivalence Class Partitioning

Example: **Math.abs(int x)** in the Java system libraries

- Calculates the absolute value of an integer
- If the argument is not negative, return the argument
- If the argument is negative, return the negation of the argument

Equivalence Class Partitioning

Example: **Math.abs(int x)** in the Java system libraries

- Calculates the absolute value of an integer
- If the argument is not negative, return the argument
- If the argument is negative, return the negation of the argument

Test case table:

Unit under test: Math.abs	
Input	Expected output
x = 37	37
x = 38	38
x = 39	39
x = 40	40
x = 41	41

Equivalence Class Partitioning

Example: **Math.abs(int x)** in the Java system libraries

- Calculates the absolute value of an integer
- If the argument is not negative, return the argument
- If the argument is negative, return the negation of the argument

Test case table:

Unit under test: Math.abs	
Input	Expected output
x = 37	37
x = 38	38
x = 39	39
x = 40	40
x = 41	41

Do these tests ensure that the function is reliably implemented?

Are these test the best test cases to check?

Equivalence Class Partitioning

Example: Math.abs(int x) in the Java system libraries

- Calculates the absolute value of an integer
- If the argument is not negative, return the argument
- If the argument is negative, return the negation of the argument

Test case table:

Unit under test: Math.abs	
Input	Expected output
x = 37	37
x = 38	38
x = 39	39
x = 40	40
x = 41	41

Nope!

Do these tests ensure that the function is reliably implemented?

Are these test the best test cases to check?

Equivalence Class Partitioning

An **Equivalence Class** (EC) is a subset of all possible inputs to the UUT, where if one element in the subset reveals a defect during testing, we assume that **all other elements in the subset will reveal the same defect**.

Equivalence Class Partitioning

An **Equivalence Class** (EC) is a subset of all possible inputs to the UUT, where if one element in the subset reveals a defect during testing, we assume that **all other elements in the subset will reveal the same defect**.

→ We only need to choose one representative element from each EC

Equivalence Class Partitioning

Partition the input space into **equivalence classes,** choose a **representative** input value from each EC

We can have **valid** or **invalid** ECs

- Example: Math.sqrt(double x), positive vs. negative values of x
- Invalid ECs will typically define special processing, e.g., throwing an exception

Equivalence Class Partitioning

When partitioning, aim for **coverage** and **representation**

Coverage: Every possible input element belongs to at least one of the equivalence classes

Representation: If a defect is demonstrated by one member of an equivalence class, the same defect is assumed to be demonstrated by any other member of the class

Equivalence Class Partitioning

When partitioning, aim for **coverage** and **representation**

Coverage: Every possible input element belongs to at least one of the equivalence classes

Representation: If a defect is demonstrated by one member of an equivalence class, the same defect is assumed to be demonstrated by any other member of the class

Equivalence class table

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

Equivalence Class Partitioning

Often it is not easy to find equivalence classes!

- Apply heuristics, iterate, refine the ECs and test cases as you gain insight into the problem
- Take small steps

Equivalence Class Partitioning

Often it is not easy to find equivalence classes!

- Apply heuristics, iterate, refine the ECs and test cases as you gain insight into the problem
- Take small steps

Look for conditions in the **specifications** of the UUT indicating expected input and output values

- **Set** : define an EC for each value in the set and one EC containing all elements outside the set
- **Boolean** : define one EC for the true condition and one for the false condition
- **Range** : select one valid EC that covers the allowed range and two invalid ECs, one above and one below

Equivalence Class Partitioning

Example: Pay station must accept 5, 10, and 25 cent coins

Equivalence Class Partitioning

Example: Pay station must accept 5, 10, and 25 cent coins

Set? Boolean? Range?

Equivalence Class Partitioning

Example: Pay station must accept 5, 10, and 25 cent coins

Set : define an EC for each value in the set and one EC containing all elements outside the set

Equivalence Class Partitioning

Example: Pay station must accept 5, 10, and 25 cent coins

Set : define an EC for each value in the set and one EC containing all elements outside the set

Condition	Invalid ECs	Valid ECs
Allowed coins	$\notin \{5, 10, 25\}[1]$	$\{5\}[2]; \{10\}[3]; \{25\}[4]$

Equivalence Class Partitioning

Example: A method to recognize a properly formatted programming language identifier that is required to start with a letter

Set? Boolean? Range?

Equivalence Class Partitioning

Example: A method to recognize a properly formatted programming language identifier that is required to start with a letter

Boolean : define one EC for the true condition and one for the false condition

Equivalence Class Partitioning

Example: A method to recognize a properly formatted programming language identifier that is required to start with a letter

Boolean : define one EC for the true condition and one for the false condition

Condition	Invalid ECs	Valid ECs
Initial character of identifier	non-letter [1]	letter [2]

Equivalence Class Partitioning

Example: A method to test if a position on a chess board is valid (columns a-h, rows 1-8)

Set? Boolean? Range?

Equivalence Class Partitioning

Example: A method to test if a position on a chess board is valid (columns a-h, rows 1-8)

Range : select one valid EC that covers the allowed range and two invalid ECs, one above and one below

Equivalence Class Partitioning

Example: A method to test if a position on a chess board is valid (columns a-h, rows 1-8)

Range : select one valid EC that covers the allowed range and two invalid ECs, one above and one below

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

Equivalence Class Partitioning

Heuristics are based on how the value conditions would typically be handled (because this is what they are designed to test)

Ranges:

```
if ( row < 1 || row > 8 ) { ... }
```

Sets:

```
if ( coin == 5 || coin == 10 || coin == 25 ) { ... }
// alternative
switch ( coin ) { case 5: case 10: case 25: { ... }}
```

Equivalence Class Partitioning

Heuristics are based on how the value conditions would typically be handled (because this is what they are designed to test)

Arithmetic computations:

Addition/subtraction: select one valid EC for 0, one valid EC for all other elements

Multiplication/division: select one valid EC for 1, and one valid EC for all other elements

Equivalence Class Partitioning

Example: HotCiv attacks

- Implementation 1: Value = attackStrength;
- Implementation 2: Value = attackStrength * dieValue;

What happens if you choose 1 as die value in all your test cases?

Equivalence Class Partitioning

Example: HotCiv attacks

- Implementation 1: Value = attackStrength;
- Implementation 2: Value = attackStrength * dieValue;

What happens if you choose 1 as die value in all your test cases?

Using the heuristics:

Condition:	Valid EC:
Die value	{1} [A], {2,3,4,5,6} [B]

Equivalence Class Partitioning

Once ECs are established, generate test cases by **picking elements from each EC**

Equivalence Class Partitioning

Once ECs are established, generate test cases by **picking elements from each EC**

If ECs don't overlap:

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

Equivalence Class Partitioning

Once ECs are established, generate test cases by **picking elements from each EC**

If ECs don't overlap:

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

ECs covered	Test case	Expected output
[1]	$x = -37$	$+37$
[2]	$x = 42$	$+42$

Equivalence Class Partitioning

Once ECs are established, generate test cases by **picking elements from each EC**

If ECs don't overlap:

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

ECs covered	Test case	Expected output
[1]	$x = -37$	$+37$
[2]	$x = 42$	$+42$

If ECs overlap:

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

(Must provide both row and column)

Equivalence Class Partitioning

Once ECs are established, generate test cases by **picking elements from each EC**

If ECs don't overlap:

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

ECs covered	Test case	Expected output
[1]	$x = -37$	$+37$
[2]	$x = 42$	$+42$

If ECs overlap:

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

(Must provide both row and column)

ECs covered	Test case	Expected output
[1], [4]	(' ',0)	illegal
[2], [4]	('i',-2)	illegal
[3], [4]	('e',0)	illegal
[1], [5]	(' ',9)	illegal
[2], [5]	('j',9)	illegal
[3], [5]	('f',12)	illegal
[1], [6]	(' ',4)	illegal
[2], [6]	('i',5)	illegal
[3], [6]	('b',6)	legal

?

Equivalence Class Partitioning

With many ECs for many independent conditions, avoid combinatorial explosion:

1. Until all **valid** ECs have been covered, define a test case that covers **as many uncovered valid ECs** as possible
2. Until all **invalid** ECs have been covered, define a test case that only involves **one invalid EC**

Equivalence Class Partitioning

With many ECs for many independent conditions, avoid combinatorial explosion:

1. Until all **valid** ECs have been covered, define a test case that covers **as many uncovered valid ECs** as possible
2. Until all **invalid** ECs have been covered, define a test case that only involves **one invalid EC**

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

Equivalence Class Partitioning

With many ECs for many independent conditions, avoid combinatorial explosion:

1. Until all **valid** ECs have been covered, define a test case that covers **as many uncovered valid ECs** as possible
2. Until all **invalid** ECs have been covered, define a test case that only involves **one invalid EC**

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

ECs covered	Test case	Expected output
[1], [6]	(' ',5)	illegal
[2], [6]	('j',3)	illegal
[3], [4]	('b',0)	illegal
[3], [5]	('c',9)	illegal
[3], [6]	('b',6)	legal

- One valid test case
- Only invalid row or column (not both)
- Test lower and higher than allowed range

Equivalence Class Partitioning

Allowing only one condition to be invalid at a time avoids **masking**

Equivalence Class Partitioning

Allowing only one condition to be invalid at a time avoids **masking**

Example:

columns a-h
rows 1-8

```
/** Demonstration of masking of defects.
*/
public class ChessBoard {
  public boolean valid(char column, int row) {
    if ( column < 'a' ) { return false; }
    if ( row < 0 ) { return false; }
    return true;
  }
}
```

ECs covered	Test case	Expected output
[1], [4]	(' ',0)	illegal

Equivalence Class Partitioning

Allowing only one condition to be invalid at a time avoids **masking**

Example:

columns a-h
rows 1-8

Not covered!

```
/** Demonstration of masking of defects.
*/
public class ChessBoard {
  public boolean valid(char column, int row) {
    if ( column < 'a' ) { return false; }
    if ( row < 0 ) { return false; }
    return true;
  }
}
```

ECs covered	Test case	Expected output
[1], [4]	(' ',0)	illegal

But this test case passes!

Correct column checking **masks** the defect in row checking

Equivalence Class Partitioning

Allowing only one condition to be invalid at a time avoids **masking**

Example:

columns a-h
rows 1-8

```
/** Demonstration of masking of defects.
*/
public class ChessBoard {
  public boolean valid(char column, int row) {
    if ( column < 'a' ) { return false; }
    if ( row < 0 ) { return false; }
    return true;
  }
}
```

ECs covered	Test case	Expected output
[1], [4]	(' ',0)	illegal

ECs covered	Test case	Expected output
[1], [6]	(' ',5)	illegal
[2], [6]	('j',3)	illegal
[3], [4]	('b',0)	illegal
[3], [5]	('c',9)	illegal
[3], [6]	('b',6)	legal

Equivalence Class Partitioning

1. Review the requirements for the UUT, identify **conditions,** and use heuristics to find ECs for each condition
   - Create an **equivalence class table**
2. Review the ECs and consider the **representation** of elements in each EC. Repartition the EC if elements are not representative.
3. Review to verify **coverage**
4. Generate test cases from the ECs. Apply heuristics to generate a minimal set of test cases.
   - Create a **test case table**
5. Review test cases to find what is missing, **iterate**!

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

1. Identify conditions

Equivalence Class Partitioning

Example:

valid years 1900 – 3000

valid months 1 – 12

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

1. Identify conditions

Equivalence Class Partitioning

Example:

valid years 1900 – 3000

valid months 1 – 12

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

1. Identify conditions: Apply range heuristics

Equivalence Class Partitioning

Example:

valid years 1900 – 3000

valid months 1 – 12

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

1. Identify conditions: Apply range heuristics

Condition	Invalid ECs	Valid ECs
year	< 1900 [1]; > 3000 [2]	$1900 - 3000$ [3]
month	< 1 [4]; > 12 [5]	$1 - 12$ [6]

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

2. Evaluate representativeness

Condition	Invalid ECs	Valid ECs
year	< 1900 [1]; > 3000 [2]	$1900 - 3000$ [3]
month	< 1 [4]; > 12 [5]	$1 - 12$ [6]

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

2. Evaluate representativeness : What about leap years?

→ Repartition

Condition	Invalid ECs	Valid ECs
year	< 1900 [1]; > 3000 [2]	$1900 - 3000$ [3]
month	< 1 [4]; > 12 [5]	$1 - 12$ [6]

Equivalence Class Partitioning

Example:

a leap year is divisible by 4, and a year that is evenly divisible by 100 is a leap year only if it is also evenly divisible by 400

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

2. Evaluate representativeness : What about leap years?

→ Repartition

Condition	Invalid ECs	Valid ECs
year	< 1900 [1]; > 3000 [2]	$1900 - 3000$ [3]
month	< 1 [4]; > 12 [5]	$1 - 12$ [6]

Condition	Invalid ECs	Valid ECs
year (y)		$\{y \| y \in [1900; 3000] \wedge y\%400 = 0\}$ [3a] $\{y \| y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [3a]\}$ [3b] $\{y \| y \in [1900; 3000] \wedge y\%4 = 0 \wedge \notin [3a] \cup [3b]\}$ [3c] $\{y \| y \in [1900; 3000] \wedge y\%4 \neq 0\}$ [3d]

Condition	Invalid ECs	Valid ECs
month		$1 - 2$ [6a]; $3 - 12$ [6b]

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

3. Verify coverage : all possible values of (year, month) belong to one or another EC

Condition	Invalid ECs
year	< 1900 [1]; > 3000 [2]
month	< 1 [4]; > 12 [5]

Condition	Invalid ECs	Valid ECs
year (y)		$\{y \mid y \in [1900; 3000] \wedge y\%400 = 0\}$ [3a] $\{y \mid y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [3a]\}$ [3b] $\{y \mid y \in [1900; 3000] \wedge y\%4 = 0 \wedge \notin [3a] \cup [3b]\}$ [3c] $\{y \mid y \in [1900; 3000] \wedge y\%4 \neq 0\}$ [3d]

Condition	Invalid ECs	Valid ECs
month		$1 - 2$ [6a]; $3 - 12$ [6b]

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

4. Generate test cases : apply heuristics

Condition	Invalid ECs
year	< 1900 [1]; > 3000 [2]
month	< 1 [4]; > 12 [5]

Condition	Invalid ECs	Valid ECs
year (y)		$\{y \mid y \in [1900; 3000] \wedge y\%400 = 0\}$ [3a]
		$\{y \mid y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [3a]\}$ [3b]
		$\{y \mid y \in [1900; 3000] \wedge y\%4 = 0 \wedge \notin [3a] \cup [3b]\}$ [3c]
		$\{y \mid y \in [1900; 3000] \wedge y\%4 \neq 0\}$ [3d]

Condition	Invalid ECs	Valid ECs
month		$1-2$ [6a]; $3-12$ [6b]

ECs covered	Test case	Expected output
$[3a], [6a]$	$y = 2000; m = 2$	-
$[3b], [6b]$	$y = 1900; m = 5$	-
$[3c], [6b]$	$y = 2004; m = 10$	5
$[3d], [6a]$	$y = 1985; m = 1$	-
[1]	$y = 1844; m = 4$	[exception]
[2]	$y = 4231; m = 8$	[exception]
[4]	$y = 2004; m = 0$	[exception]
[5]	$y = 2004; m = 13$	[exception]

Equivalence Class Partitioning

Example:

```
public interface weekday {
  /** calculate the weekday of the 1st day of the given month.
      @param year the year as integer. 2000 means year 2000 etc. Only
      years in the range 1900-3000 are valid. The output is undefined
      for years outside this range.
      @param month the month as integer. 1 means January, 12 means
      December. Values outside the range 1-12 are illegal.
      @return the weekday of the 1st day of the month. 0 means Sunday
      1 means Monday etc. up til 6 meaning Saturday.
  */
  public int weekday(int year, int month)
     throws IllegalArgumentException;
}
```

4. Generate test cases : apply heuristics

Only one invalid parameter at a time

Condition	Invalid ECs
year month	< 1900 [1]; > 3000 [2] < 1 [4]; > 12 [5]

Condition	Invalid ECs	Valid ECs
year (y)		$\{y \mid y \in [1900; 3000] \wedge y\%400 = 0\}$ [3a] $\{y \mid y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [3a]\}$ [3b] $\{y \mid y \in [1900; 3000] \wedge y\%4 = 0 \wedge \notin [3a] \cup [3b]\}$ [3c] $\{y \mid y \in [1900; 3000] \wedge y\%4 \neq 0\}$ [3d]

Condition	Invalid ECs	Valid ECs
month		$1-2$ [6a]; $3-12$ [6b]

ECs covered	Test case	Expected output
$[3a]$, $[6a]$	$y = 2000; m = 2$	-
$[3b]$, $[6b]$	$y = 1900; m = 5$	-
$[3c]$, $[6b]$	$y = 2004; m = 10$	5
$[3d]$, $[6a]$	$y = 1985; m = 1$	-
[1]	$y = 1844; m = 4$	[exception]
[2]	$y = 4231; m = 8$	[exception]
[4]	$y = 2004; m = 0$	[exception]
[5]	$y = 2004; m = 13$	[exception]

Equivalence Class Partitioning

Example:

```
/** return true iff x+y < T */
public boolean isMoreThanSumOf(int x, int y)
```

Condition	Invalid ECs	Valid ECs
x		?
y		?

Equivalence Class Partitioning

Example:

```
/** return true iff x+y < T */
public boolean isMoreThanSumOf(int x, int y)
```

Condition	Invalid ECs	Valid ECs
x		?
y		?

Condition	Invalid ECs	Valid ECs
x+y	-	$< T$ [1]; $\geq T$ [2]

Can’t just look at input parameters – need the condition on x + y

Equivalence Class Partitioning

Example:

```
/** return true iff x+y < T */
public boolean isMoreThanSumOf(int x, int y)
```

Condition	Invalid ECs	Valid ECs
x		?
y		?

Condition	Invalid ECs	Valid ECs
x+y	-	$< T$ [1]; $\geq T$ [2]

ECs covered	Test case	Expected output
[1]	$x = 12; y = 23; T = 100$	true
[2]	$x = -23; y = 15; T = -10$	false

Can't just look at input parameters – need the condition on x + y

Equivalence Class Partitioning

Example:

```
/** format a string representing of a double. The string is always
    6 characters wide and in the form ###.##, that is the double is
    rounded to 2 digit precision. Numbers smaller than 100 have '0'
    prefix. Example: 123 -> '123.00'; 2,3476 -> '002.35' etc. If the
    number is larger or equal to 999.995 then '***.**' is output to
    signal overflow. All negative values are signaled with '---.--'
*/
public String format(double x);
```

Equivalence Class Partitioning

Example:

```
/** format a string representing of a double. The string is always
    6 characters wide and in the form ###.##, that is the double is
    rounded to 2 digit precision. Numbers smaller than 100 have '0'
    prefix. Example: 123 -> '123.00'; 2,3476 -> '002.35' etc. If the
    number is larger or equal to 999.995 then '***.**' is output to
    signal overflow. All negative values are signaled with '---.--'
*/
public String format(double x);
```

Condition	Invalid ECs	Valid ECs
overflow / underflow	$\geq$ 1000.0 [1]; $<$ 0.0 [2]	
2 digit rounding		(,00x round up) [3];
		(,00x round down) [4]
prefix		no '0' prefix [5]
		exact '0' prefix [6]
		exact '00' prefix [7]
		exact '000' prefix [8]
output suffix		'.yx' suffix ($x \neq 0$)[9]
		'.x0' suffix ($x \neq 0$)[10]
		exact '.00' suffix [11]

Equivalence Class Partitioning

Example:

```
/** format a string representing of a double. The string is always
    6 characters wide and in the form ###.##, that is the double is
    rounded to 2 digit precision. Numbers smaller than 100 have '0'
    prefix. Example: 123 -> '123.00'; 2,3476 -> '002.35' etc. If the
    number is larger or equal to 999.995 then '***.**' is output to
    signal overflow. All negative values are signaled with '---.--'
*/
public String format(double x);
```

Condition	Invalid ECs	Valid ECs
overflow / underflow	≥ 1000.0 [1]; < 0.0 [2]	
2 digit rounding		(,00x round up) [3]; (,00x round down) [4]
prefix		no '0' prefix [5] exact '0' prefix [6] exact '00' prefix [7] exact '000' prefix [8]
output suffix		'.yx' suffix ($x \neq 0$)[9] '.x0' suffix ($x \neq 0$)[10] exact '.00' suffix [11]

ECs covered	Test case	Expected output
[1]	1234.456	'***.**'
[2]	-0.1	'—.–'
[3][5][9]	212.738	'212.74'
[4][6][10]	32.503	'032.50'
[3][7][11]	7.995	'008.00'
[4][8][9]	0.933	'000.93'

Boundary Value Analysis

Boundary value: an element that lies right on or next to the edge of an equivalence class

→Not always applicable, depends on the requirements and implementation

Boundary Value Analysis

Boundary value: an element that lies right on or next to the edge of an equivalence class

→Not always applicable, depends on the requirements and implementation

Example: Check for a valid chess board position

```
if ( row <= 1 ) return false; // should have been row < 1
```

Boundary Value Analysis

Boundary value: an element that lies right on or next to the edge of an equivalence class

→Not always applicable, depends on the requirements and implementation

Example: Check for a valid chess board position

```
if ( row <= 1 ) return false; // should have been row < 1
```

Set and Boolean conditions generate ECs where boundaries are **part of the ECs themselves**

Systematic Testing: Summary

Key point: Observe unit **preconditions** – don't generate ECs and test cases for conditions that a unit cannot/should not handle

Systematic Testing: Summary

Key point: Observe unit **preconditions** – don't generate ECs and test cases for conditions that a unit cannot/should not handle

Example: For a game with a user interface, it is unnecessary to test presence of a game piece at a "from" location if the user must click on a piece to move

→This is the case for HotCiv!

→Check/define method preconditions

Systematic Testing: Summary

Key point: Systematic testing assumes **competent programming**

Systematic Testing: Summary

Key point: Systematic testing assumes **competent programming**

Example:

```
public int abs(int x) {
  switch(x) {
   case 1: return 1;
   case 2: return 2;
   case 3: return 3;
   case 7123: return 8222;
   case -1: return -1;
   ...
}
```

Tests can't necessarily detect bad code!

Systematic Testing: Summary

Key point: Heuristics do not guarantee sufficient test cases – ensure that important test cases are not omitted

Systematic Testing: Summary

Key point: Heuristics do not guarantee sufficient test cases – ensure that important test cases are not omitted

→ Consider use cases

Next time: Code Coverage