L2 Test Design

With FitNesse

Slide deck on Canvas > Modules > Assignments-Labs-Activities

TriType Example in FitNesse (Demo)

NextDate Example in FitNesse: Lab

NextDate

A Date object represents a date using three integers: day, month, and year. The nextDate() methods returns a new Date object that gives the date of the next day.

The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise.

The year must be between 1900 and 2200 (1899 and 2201 are not valid years.)

A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

A Date object cannot be formed with invalid values!

Why is this little function important?

- Utility function
- Deals with data critical to lots of applications
- If it is wrong, it can wreak havoc!

Y2K 

We'll be as thorough as practicably possible!

Spec - explicit info

A Date object represents a date using three integers: day, month, and year. The nextDate() methods returns a new Date object that gives the date of the next day. The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise. The year must be between 1900 and 2200 (1899 and 2201 are not valid years). A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

What are the important bits of info here?

Spec – important bits

A Date object represents a date using three integers: day, month, and year. The nextDate() methods returns a new Date object that gives the date of the next day. The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise. The year must be between 1900 and 2200 (1899 and 2201 are not valid years). A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

Attributes

Inputs or initial states that can affect the behavior

Things I need to determine or set up before calling the tested behavior

Attributes

~~Inputs or~~ initial states that can affect the behavior

- Day
- Month
- Year

Other domain info we know

(not in the spec, but we must know them to function in a civilized society)

Month:

- 1, 3, 5, 7, 8, 10, 12 have 31 days
- 4, 6, 9, 11 have 30 days
- 2 (Feb) has 28 days in non-leap year or 29 days in leap year
- < 1 and > 12 are always invalid

Day:

- < 1 and > 31 are always invalid

Characteristics – let’s start with the obvious

Characteristics – let's start with the obvious

- Day
- Month
- Year

We can make these "smart" later when we partition them, or introduce new characteristics to partition them properly

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise. The year must be between 1900 and 2200 (1899 and 2201 are not valid years). A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

What can we consider next?

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise. The year must be between 1900 and 2200 (1899 and 2201 are not valid years). A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

Characteristics – let's be smart

- Day
- Month
- Year *stress boundary values*
- "Year in valid range?"

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise. The ~~year must be between 1900 and 2200 (1899 and 2201 are not valid years)~~. A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

What can we consider next?

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method isLeapYear() returns true if the year of the Date object is a leap year; false otherwise. The ~~year must be between 1900 and 2200 (1899 and 2201 are not valid years)~~. A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

Characteristics – let's be smart

- Day
- Month
- Year
- "Year in valid range?"
- "Leap Year?"

Characteristics – let's be smart

- Day
- Month
- Year
- "Year in valid range?"
- "Leap Year"
  - *How many ways a year can be a leap year?*

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method ~~isLeapYear()~~ method returns true if the year of the Date object is a leap year; false otherwise. The ~~year must be between 1900 and 2200 (1899 and 2201 are not valid years)~~. A leap year is a year that is divisible by 4; but if it's a century year, it should also be divisible by 400.

Characteristics – let's be smart

- Day
- Month
- Year
- "Year in valid range?"
- "Leap Year?"
- "Century Year?"

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method ~~isLeapYear()~~ returns true if the year of the Date object is a leap year; false otherwise. The ~~year must be between 1900 and 2200 (1899 and 2201 are not valid years)~~. A ~~leap year~~ is a year ~~that is divisible by 4~~; but if it's a ~~century year~~, it should also be ~~divisible by 400~~.

What else?

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method ~~isLeapYear()~~ returns true if the year of the Date object is a leap year; false otherwise. The ~~year must be between 1900 and 2200 (1899 and 2201 are not valid years)~~. A ~~leap year~~ is a year ~~that is divisible by 4~~; but if it's a ~~century year~~, it should also be ~~divisible by 400~~.

Characteristics – let's be smart

Let's go back to the spec…

A Date object represents a date using three integers: ~~day~~, ~~month~~, and ~~year~~. The nextDate() methods returns a new Date object that gives the date of the next day. The method ~~isLeapYear()~~ returns true if the year of the Date object is a leap year; false otherwise. The ~~year must be between 1900 and 2200 (1899 and 2201 are not valid years)~~. A ~~leap year~~ is a year ~~that is divisible by 4~~; but if it's a ~~century year~~, it should also be ~~divisible by 400~~.

What is the importance of nextDate?

Consider boundary values

- Day
- Month:
- Year

upper boundary values are more important

- “Year in valid range?”
- “Leap Year?”
- “Century Year?”

Other domain info we know

Month:

- 1, 3, 5, 7, 8, 10, 12 have 31 days
- 4, 6, 9, 11 have 30 days
- 2 (Feb) has 28 days in non-leap year or 29 days in leap year
- < 1 and > 12 are always invalid

Day:

- < 1 and > 31 are always invalid

Add additional characteristics

- Day
- Month:

stress boundary values (go outside valid ranges)

- Year
- “Year in valid range?”
- “Leap Year?”
- “Century Year?”
- “Day in valid range?”
- “Month in valid range?”

Other domain info we know

Month:

- 1, 3, 5, 7, 8, 10, 12 have 31 days
- 4, 6, 9, 11 have 30 days
- 2 (Feb) has 28 days in non-leap year or 29 days in leap year
- ~~< 1 and > 12 are always invalid~~

Day:

- ~~< 1 and > 31 are always invalid~~

Otherwise Date itself is invalid!

Add additional characteristics

- Day
- Month
- Year
- “Year in valid range?”
- “Leap Year?”
- “Century Year?”
- “Day in valid range?”
- “Month in valid range?”
- “Type of month” {30D, 31D, Feb}
- “Date valid?”

Additional boundary value considerations

Month:

- 1, 3, 5, 7, 8, 10, 12 have 31 days ✓ upper
- 4, 6, 9, 11 have 30 days ✓ upper

Many upper boundary values for day due to different # of days in different months

- 2 (Feb) has 28 days ✓ upper in non-leap year or 29 days ✓ upper in leap year
- < 1 and > 12 are always invalid

Day:

- < 1 and > 31 are always invalid

Set of characteristics so far: anything else?

- Day *many upper boundary values, they change depending on Month*
- Month
- Year
- “Year in valid range?”
- “Leap Year?”
- “Century Year?”
- “Day in valid range?”
- “Month in valid range?”
- “Type of month” {30D, 31D, Feb}
- “Date valid?”

Other domain info we know

Month:

- 1, 3, 5, 7, 8, 10, 12 have 31 days
- 4, 6, 9, 11 have 30 days
- 2 (Feb) has 28 days in non-leap year or 29 days in leap year
- < 1 and > 12 are always invalid

Day:

- < 1 and > 31 are always invalid

No additional characteristics

- Day
- Month
- Year
- “Year in valid range?”
- “Leap Year?” *already considered!*
- “Century Year?”
- “Day in valid range?”
- “Month in valid range?”
- “Type of month”

Full set of characteristics

- Day
- Month
- Year
  - *upper boundary values are more important*
  - *many upper boundary values*
  - *stress boundary values (go outside valid ranges)*
- “Year in valid range?”
- “Leap Year?”
- “Century Year?”
- “Day in valid range?”
- “Month in valid range?”
- “Type of month”
- “Date valid?”

A big input space model!

- Day
- Month
- Year
- ValidY
- LeapY
- CentY
- ValidD
- ValidM
- MonthT
- Valid

Two options:

- One big design: add combinatorial constraints
- Several input space models: few constraints

Too Big! Let's divide and conquer

- Day
- Month
- Year
- ValidY
- LeapY
- CentY
- ValidD
- ValidM
- MonthT
- Valid

Start easy: handle pure syntactic validity first!

Input Space Model 1: Syntactic Validity

- Day
- Month
- Year
- ValidD
- ValidM
- ValidY
- Valid

- *all boundary values,*
- *upper boundary values more important,*
- *stress boundary values*

Input Space Model 1 – Syntactic Validity - Let's partition the characteristics

- Day = { <1, 1, 2, 3..29, 30, 31, >31 }
- Month = { <1, 1, 2, 3..10, 11, 12, >12 }
- Year = { <1900, 1900, 1901, 1902..2198, 2199, 2200, >2200 }
- ValidD = { Y, N }
- ValidM = { Y, N }
- ValidY = { Y, N }
- Valid = { Y, N }

Sanity check for each partition

- Disjoint ✓
- Complete ✓
- Uniform ✓

Input Space Model 1 – Syntactic Validity - Let's combine blocks

- Day = { <1, 1, 2, 3..29, 30, 31, >31 }
- Month = { <1, 1, 2, 3..10, 11, 12, >12 }
- Year = { <1900, 1900, 1901, 1902..2198, 2199, 2200, >2200 }

Apply BC to these

- ValidD = { Y, N }
- ValidM = { Y, N }
- ValidY = { Y, N }
- Valid = { Y, N }

If ValidD, ValidM, ValidY = Y, then make sure Valid = Y so that we don't pick semantically invalid dates (this is a constraint)

- AC: Would generate too many unnecessary test cases!
- Don't need to combine non-ordinary values since once one of Day, Month, Year is invalid, the other two don't matter: we just need to isolate one characteristic at a time, no need to check interactions
- Strategy: Base Choice The base case is (D: 3...29, M: 3...10, Y:1902..2198)
- ValidD, ValidM, ValidY determined by Day, Month, Year block: we can ignore them when forming combinations
- Try to keep Valid = Y when possible not to mix syntactic validity with semantic validity

Input Space Model 1 – Syntactic validity : BC Combinations

Case	Day	Month	Year	ValidD	ValidM	ValidY	Valid
1.1 BC	3..29	3..10	1902..2198	Y	Y	Y	Y
1.2 Vary Year	3..29	3..10	<1900	Y	Y	N	N
1.3 Vary Year	3..29	3..10	1900	Y	Y	Y	Y
1.4 Vary Year	3..29	3..10	1901	Y	Y	Y	Y
1.5 Vary Year	3..29	3..10	2199	Y	Y	Y	Y
1.6 Vary Year	3..29	3..10	2200	Y	Y	Y	Y
1.7 Vary Year	3..29	3..10	>2200	Y	Y	N	N
1.8 Vary Month	3..29	<1	1902..2198	Y	N	Y	N
1.9 Vary Month	3..29	1	1902..2198	Y	Y	Y	Y
1.10 Vary Month	3..29	2	1902..2198	Y	Y	Y	Y
…							
1.14 Vary Day	<1	3..10	1902..2198				
…							

Some of cases will overlap with other cases in other models: remove them later when that happens

1 + 3 x 6 = 19 cases

Input Space Model 2: 30D Month

- Day
- Month
- Year
- MonthT
- Valid

- *upper boundary values are important*
- *many upper boundary values*
- *stress boundary for day*

Input Space Model 2 – 30D: Let's partition the characteristics

- Day = { 1..28, 29, 30, 31 } 4 blocks
- Month = { 4..9 {4, 6, 9}, 11 } 2 blocks
- Year = { 1900..2200 } 1 block, no year roll-over
- MonthT = { 30D } 1 block
- Valid = { Y, N } *determined by others*

Partition sanity check
- Disjoint ✓
- Complete ✓
- Uniform ✓

AC: 4 x 2 = 8 cases

some cases may overlap with ISM1, e.g.:

	Case	Day	Month	Year	MonthT			Valid
ISM2	2.1	1..28	4..9	1900..2200	30D			Y

	Case	Day	Month	Year	ValidD	ValidM	ValidY	Valid
ISM1	1.1 BC	3..29	3..10	1902..2198	Y	Y	Y	Y

In actual test cases: choose a representative that covers both and remove one test case

	A	B	C	D	E	F	G	H
1	Case	Day	Month	Year	MonthT			Valid
2		3...28	4...9	1902...2198	30			Y
3		29	4...9	1900...2200	30			Y
4		30	4...9	1900...2200	30			Y
5		31	4...9	1900...2200	30			N
6		3...28	11	1900...2200	30			Y
7		29	11	1900...2200	30			Y
8		30	11	1900...2200	30			Y
9		31	11	1900...2200	30			N
10								

Input Space Model 3: 31D Month

- Day
- Month
- Year
- MonthT
- Valid

- *upper boundary values are important*
- *many upper boundary values*
- *stress boundary for day*

Input Space Model 3 – 31D: Let's partition the characteristics

- Day = { 1..29, 30, 31, 32} 4 blocks
- Month = { 1, 3..10, 12 } 2 blocks
- Year = { 1900..2199, 2200 } 2 blocks with upper boundary value, because year roll-over due to Dec being in Month
- MonthT = { 31D } 1 block
- Valid = { Y, N } *determined by others*

2200.12.31 is invalid since there is not nextdate()

Sanity Check
- Disjoint ✓
- Complete ✓
- Uniform ✓

AC: 4 x 3 x 2 = 24 cases

some cases may overlap with Model 1, e.g.,

	Case	Day	Month	Year	MonhtT			Valid
ISM3	3.1	1..29	1	1900..2199	31D			Y

	Case	Day	Month	Year	ValidD	ValidM	ValidY	Valid
ISM1	1.9	3..29	1	1902..2198	Y	Y	Y	Y

In actual test cases: choose a representative that covers both and remove one test case

Input Space Model 4: Feb

- Day
- Month
- Year
- MonthT
- LeapY
- CentY
- Valid

- *upper boundary values are important*
- *many upper boundary values*
- *stress boundary values*

Input Space Model 4 - Feb: Let's partition the characteristics

- Day = { 1..26, 27, 28, 29, 30} 5 blocks
- Month = { 2 } 1 block
- Year = { 1900..2200 } 1 block
- MonthT = { Feb } 1 block
- LeapY = { Y, N } 2 blocks
- CentY = { Y, N } 2 blocks
- Valid = { Y, N }

Sanity Check
- Disjoint ✓
- Complete ✓
- Uniform ✓

AC: 5 x 2 x 2 cases = 20 cases

some cases may overlap with Model 1, e.g.,

	Case	Day	Month	Year	MonthT	LeapY	CenttY	Valid
ISM4	4.1	1..26	2	1900..2200	Feb	Y	Y	Y

	Case	Day	Month	Year	ValidD	ValidM	ValidY	Valid
ISM1	1.10	3..29	2	1902..2198	Y	Y	Y	Y

In actual test cases: choose a representative that covers both and remove one test case

Anything important missing?

- Perhaps, perhaps not!
- I'll let you decide!

Move this design to FitNesse

- Identify redundant test cases
  - Test cases in ISM1 and other ISMs will overlap (perhaps others too)
- Choose representative inputs for each test case
  - Take care not to include redundant test cases when they overlap
  - For redundant test cases, identify the overlapping blocks, remove one test case
  - For each pair of overlapping blocks, choose a representative from their intersection
- Determine oracles for each test case
- Complete the lab