

Introduction to Software Testing

Chapter 6

Input Space Partition Testing

Paul Ammann & Jeff Offutt

<https://www.cs.gmu.edu/~offutt/softwaretest/>

Ch. 6 : Input Space Coverage

Four Structures for Modeling Software

Input Space

Graphs

Logic

Syntax

Applied
to

Applied to

Applied
to

Source

FSMs

Specs

DNF

Source

Specs

Design

Use cases

Source

Models

Integ

Input

چوری براساس ورودی تست رو انجام بدیم : **input space**

نه تنها متغیرهای محلی بلکه هر متغیری که در برنامه حضور داره مثل متغیرهای که در زمان اجرا دارن توسط کاربر مقدار می‌گیرن

Benefits of ISP

- Equally **applicable** at several levels of testing
 - Unit
 - Integration
 - System
- Easy to apply with **no automation**
- No **implementation knowledge** is needed
 - Just the input space

Input Domains

- Input domain: all possible inputs to a program
 - Most input domains are so large that they are effectively infinite
- *Input parameters* define the scope of the input domain
 - Parameter values to a method
 - Data from a file
 - Global variables
 - User inputs
- We partition input domains into regions (called *blocks*)
- Choose at least one value from each block

Input domain: Alphabetic letters

Partitioning characteristic: Case of letter

- Block 1: upper case
- Block 2: lower case

مجموعه تمام مقدارهایی که ممکنه به برنامه داده بشه: **input space** یا **Input domain** افراز: ینی مجموعه رو به زیر مجموعه هایی بشکنیم که دو به دو اشتراکشون تھی باشه و اجتماع کلشون بشه مجموعه اولیه : **input space** توى

اولین کاری افراز مجموعه ورودی بعد افراز مناسب پیدا میکنیم
این دوتا شرط خیلی مهمه: وقتی می خوایم **input space** رو انجام بدیم یک کارش اینه که رویکرد:

: **interface based** ینی فقط مرکز هستیم روی خود ورودی ها نه رفتار اون **unit** که داریم بررسی میکنیم --> مجموعه ورودی برنامه رو نگاه بکنیم و اونا رو پارتبیشن بندی بکنیم --> فضای ورودی رو پارتبیشن بندی بکنیم و برامون مهم نیست چه کاری قراره روش انجام بشه **functional based**: توی این رفتار هم در نظر می گیریم --> نه تنها ورودی رو در نظر میگیریم بلکه رفتار اون **unit** که داریم تست میکنیم فانکشنالیتیش هم در نظر میگیریم و بعد پارتبیشن ندی رو لحاظ میکنیم --> این توی سیستم های پیچیده خیلی سخت میشه پس در کل وقتی مناسب است که سیستم بزرگ و پیچیده نباشه

یک روش افراز بندی فایلها: مثال روی **interface** ... هست الان این:
وضعیت مرتب بندی فایل پارتبیشن بندی کنیم --> مثلا ورودی ها 3 مدل فایل می تونن باشن : فایل های که صعودی مرتب شدن یا اونایی که نزولی مرتب شدن یا اونایی که قاطی هستن --> ایا این معیار افراز درستیه؟

توضیح بهتر در مورد مثالش: یک برنامه رو میخوایم تست بکنیم و به ما میگن برنامه رو به روش **input space** مدلینگ طراحی بکن و میگیم به روش **interface** ... اینا؟ و میگن بله حالا ورودی یک فایل از استرینگ هاست و ما میایم افراز میکنیم مجموعه تمام فایل های ممکن از استرینگ ها این میشه **input space** و این رو افراز می کنیم واز هر زیرمجموعه یک نماینده تولید میکنیم

مثال: یک تابع داریم:

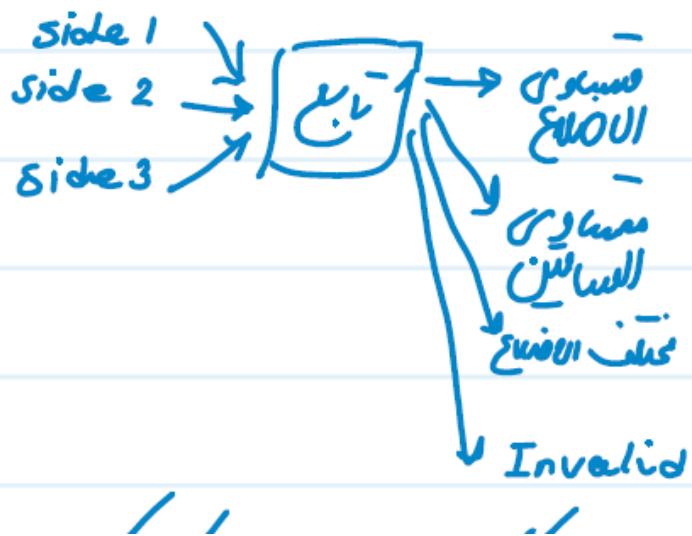
تابع اندازه سه تا ضلع مثلث رو می‌گیره و توی خروجی میگه چه مثلثی است
فرض میکنیم ضلع 1 و ضلع 2 و ضلع 3 ورودی ها ما هستن و هر سه تای اینا مقدارهای اینتیجر
ما هستن و مابنارو میدونیم به عنوان دیزاینر و می خوایم با روش input space mo ... تست رو
طراحی بکنیم

خروجی ها یا مثلث متساوی الاضلاع است یا ...

چه روش interface .. باشیم چه functional

قدم اول: باید به فکر پارسیشن بندی input space باشیم

توی این دوتا روش اینه که چه معیاری برای پارسیشن بندی انتخاب بکنیم متفاوت است توی روش
interface فقط به ورودی نگاه میکنیم و اونارو صرف نظر اینه که این تابع داره چه کاری انجام
میده پارسیشن بندی می کنیم --> توی شکل: وضعیت هر سه ضلع نسبت به صفر چجوریه اینو
بررسی میکنیم (اینجا نگاه نکردیم تابع داره چی کار میکنه و چون میدونیم سه تا اینت می‌گیره
اینجوری او مدیم افزار بندی کردیم) --> این دوتا ویژگی افزار رو داره پس اوکیه افزار است



Interface based

لـ $\{ \text{shape}, \text{square}, \text{triangle} \}$

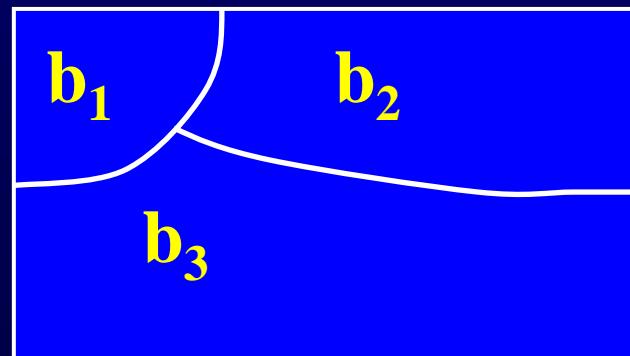
$$\begin{aligned}
 S_1 &= 0 & 3 \rightarrow 5 = 103 \\
 S_1 &\geq 0 & 3 = 27 \\
 S_1 &\leq 0
 \end{aligned}$$

✓

Partitioning Domains

- Domain D
- Partition scheme q of D
- The partition q defines a set of blocks, $B_q = b_1, b_2, \dots, b_Q$
- The partition must satisfy two properties :
 - I. Blocks must be *pairwise disjoint* (no overlap)
$$b_i \cap b_j = \emptyset, \forall i \neq j, b_i, b_j \in B_q$$
 2. Together the blocks *cover* the domain D (complete)

$$\bigcup_{b \in B_q} b = D$$



مثال ص4:

فرض میکنیم `input dom` .. حروف الفبای انگلیسی چجوری پارتبیشن انجام میدیم --> مجموعه ورودی ها حروف الفبای انگلیسی است ینی هر بار قراره یک حرف از حروف الفبای انگلیسی به برنامه داده بشه --> معیار واسه پارتبیشن بندی: دو دسته بکنیم --> حروف الفبای بزرگ و کوچیک --> معیار درست است؟ بله چون اشتراکی با هم ندارند و اجتماععش میشه کل ورودی های ممکن

معیار پارتبیشن بندی خوب:
اشتراکشون تهی باشه
اجتماعشون بشه کلشون

What is a characteristic?

“A feature or quality belonging typically to a person, place, or thing and serving to identify it.”

Input: people

Characteristics: hair color, major

Concrete
level

Blocks:

A=(red, black, brown, blonde, other)

B=(cs, swe, ce, math, ist, other)

abstract
level

Abstraction:

A = [a1, a2, a3, a4, a5]

B = [b1, b2, b3, b4, b5, b6]

برای پارتيشن بندی باید یک معیار داشته باشیم
ادما رو می تونیم براساس معیار های مختلفی پارتيشن بندی کنیم --> برای افزایش: رنگ چشم ها -
جنسیت

Examples

- Example characteristics
 - Whether X is null
 - Order of the list F (sorted, inverse sorted, arbitrary, ...)
 - Min separation of two aircraft
 - Input device (DVD, CD, VCR, computer, ...)
 - Hair color, height, major, age
- Partition characteristic into blocks
 - Blocks may be single-value or a set of values
 - Each value in a block should be **equally useful** for testing
- Each abstract test has one block from each characteristic

Choosing Partitions

- Defining partitions is not hard, but is easy to get wrong
- Consider the “*order of elements in list F*”

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

but ... something's fishy ...

Length 1 : [14]

The list will be in all three blocks ...

That is, disjointness is not satisfied

Solution:

Two characteristics that address just one property

C1: List F sorted ascending

- c1.b1 = true
- c1.b2 = false

C2: List F sorted descending

- c2.b1 = true
- c2.b2 = false

مثال:

فرض میکنیم یک فایلی از استرینگ ها داریم به اسم F که ورودی برنامه است معیار پارتیشن بندی این باشه: اگر صعودی باشه پارتیشن b1 و اگر نزولی باشه پارتیشن b2 و اگر ترتیب دلخواه پارتیشن b3 حالا این پارتیشن بندی مناسبه؟ نیست چون می تونیم یک مثالی بزنیم که به بیش از یکی از این پارتیشن ها تعلق داشته باشه --> توی فایلی به طول 1 فقط توش رشته 14 باشه --> اگر ما یک فایل داشته باشیم که فقط داخلش عدد 14 ریخته باشه الان توی کدوم یکی از این سه تا پارتیشن قرار میگیره؟ نمی تونیم یه جورایی بگیم پس این معیار مناسبی نیست پس می تونیم بگیم:

اونایی که صعودی هستن و طولشون بیش از یک است و اونایی که نزولی هستن و بیش از یک و اونایی که طولشون یک است و اونایی که ترتیب دلخواه دارن

نکته: وقتی که میخوایم معیار پوشش روی این پارتیشن ها تعریف بکنیم هرچقدر بلاک های حاصل از یک افزار زیاد باشن ما تستر به دردرس می افتمیم پس به دنبال این هستیم که پارتیشن بندی رو جوری انتخاب بکنیم که تعداد افزارها خیلی کم باشه --> مثلا اینجا بگیم دوبار پارتیشن بندی میکنیم: یک افزار یک: اونایی که صعودی هستن b1 و اونایی که صعودی نیستن b2

یک افزار بندی دیگه براساس نزولی بودن --> اونایی که نزولی هستن و اونایی که نزولی نیست حالا یه بار با اون پارتیشن بندی تست رو تولید بکنیم و یه بار هم با اون پارتیشن بندی تست رو تولید بکنیم و این خیلی بهتر از اینه که به 4 تا برسیم و اینجا الان شده دوتا --> برای فایل تک کلمه اگر معیار صعودی باشه میشه صعودی است و برای وقتی که معیار نزولی باشه فایل تک کلمه میشه نزولی

نکته: چندتا characteristic داشته باشیم با تعداد بلاک های کم خیلی بهتر از این است که یک داشته باشیم با تعداد بلاک های زیاد --> اولویت تعداد بلاک رو کم بکنیم است

Modeling the input domain

- Step 1 : Identify testable functions
 - Step 2 : Find all inputs, parameters, & characteristics
 - Step 3 : Model the input domain
 - Step 4 : Apply a test criterion to choose combinations of values (6.2)
 - Step 5 : Refine combinations of blocks into test inputs
-
- The diagram illustrates the five steps of modeling the input domain. Steps 1 and 2 are grouped under a bracket labeled 'Concrete level'. Step 3 is grouped under a bracket labeled 'Move from imp level to design abstraction level'. Step 4 is entirely within a bracket labeled 'Entirely at the design abstraction level'. Step 5 is grouped under a bracket labeled 'Back to the implementation abstraction level'.

وقتی که میخوایم space mo .. داشته باشیم:

توی مرحله اول باید function ها رو اگر توی سطح unit test هستیم شناسایی بکنیم
توی مرحله دوم: تمام ورودی های این تابع رو شناسایی بکنیم

اگر پارامتر هایی داریم که روش تاثیر داره مثل ورودی های کاربر او نا رو در نظر بگیریم
و بگیم این مجموعه ورودی رو براساس چه ویژگی پارسیشن بندی بکنیم

مرحله سوم: ...input dom

مرحله چهارم: بلاک ها رو چجوری با هم ترکیب بکنیم
5: معیارشون رو

نکته: characteristic ینی ویژگی که میتوانیم براساسش افزایش روی مجموعه ورودی ها رو انجام بدیم می تونیم بلاک بندی و پارسیشن بندی بکنیم مثل صعودی بودن یا نبودن فایل یک ویژگی هستش
مشخص بکنیم که توی کدام یکی از این زیرمجموعه ها قرار میگیره
به هر زیرمجموعه حاصل از افزایش یک بلاک میگن

نکته: هر چقدر تعداد characteristic ها رو کم بکنیم و تعداد بلاک ها زیاد باشه تست هایی که باید در بیاریم تعدادشون می ره بالا

مثال مثلث: یک تابع داریم سه تا ضلع رو میگیره و در جواب میگه نوع مثلثی که با این سه ضلع ساخته میشه چیه

حالا میخوایم تست طراحی بکنیم و ورودیها سه تا ضلع است و میخوایم براساس input space برایم ینی براساس ورودی ها تست رو طراحی بکنیم
میخوایم تابع رو تست بکنیم :

مجموعه input space میشه یک مجموعه ای که مقادیر این سه تا ضلع مثلث رو توش ریختیم
حالا میخوایم تست رو انجام بدیم --> مثلا اگر از این مثلث متساوی الاضلاع یکی رو انتخاب بکنیم و بگیم به متدهای کافی است--> اگر ما یک پارسیشن بندی خوبیب داشته باشیم از یک بلاک می تونیم یک سمپل برداریم ینی یکیشو که بر میداریم برآمون کفایت میکنه ینی عناصری که توی یک بلاک قرار میگیرن باید از دید تست ارزش یکسانی داشته باشند
پارسیشن بندی ینی مجموعه ورودی رو تقسیم بکنیم

معیار یا characteristic برای این این مجموعه ورودی رو تقسیم بندی بکنیم نوع مثلث بود ینی ما رفتیم توی دل تابع و نگاه کردیم رفتار تابع چیه و با این سه تا ورودی میخواهد چه کاری انجام بده و فهمیدم که میخواهد نوع مثلث رو بگه --> الان مثلا معیار C1 تایپ مثلث است الان این مثلث با معیار نوع میندازیم توی مثلث متساوی الاضلاع --> این غلطه چون متساوی الاضلاع ها رو توی یک بلاک گذاشتیم و مختلف الاضلاع رو توی یک بلاک دیگه --> چرا غلطه؟ چون اشتراک دارن چون مثلث متساوی الاضلاع می تونه متساوی الساقین باشه --> پس شرط می ذاریم حالا میگیم:

متساوی الاضلاع

متساوی الساقین که متساوی الاضلاع نیست

نامعتبر

مختلف الاضلاع

هر characteristic افزایهای خاص خودش رو داره

ینی رفتیم توی دل تابع و فهمیدیم با این سه تا مقدار می خواهد بگه نوع مثلث چیه پس این میشه functionality و معیار رو میگیریم نوع مثلث

Steps 1 & 2

Identify testable functions

Find inputs, parameters, characteristics

Example IDM (syntax)

- Method *triang()* from class *TriangleType* on the book website :
 - <https://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java>
 - <https://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java>

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }
public static Triangle triang (int Side1, int Side2, int Side3)
// Side1, Side2, and Side3 represent the lengths of the sides of a triangle
// Returns the appropriate enum value
```

IDM for each parameter is identical

Characteristic : *Relation of side with zero*

Blocks: negative; positive; zero

اینجا او مده characteristic رو رابطه هر کدوم از سه تا ضلع با صفر در نظر میگیریم ینی
براساس علامت میایم ورودی رو تولید میکنیم:

یک بلاک می ذاریم اونایی که هر سه تاشون مثبت هستن

یک بلاک دیگه: هر سه تاشون منفی هستن

یک بلاک دیگه: اونی که فقط اولیه منفی است

....

تعداد بلاک ها اینجا زیاد میشه

Example IDM (behavior)

- Method *triang()* again :

The three parameters represent a *triangle*

The IDM can combine all parameters

Characteristic : *Type of triangle*

Blocks: Scalene; Isosceles; Equilateral; Invalid

Steps 1 & 2—IDM

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//           else return true if element is in the list, false otherwise
```

Parameters and Characteristics

Two parameters : list, element

Characteristics based on syntax :

list is null (block1 = true, block2 = false)

list is empty (block1 = true, block2 = false)

Characteristics based on behavior :

number of occurrences of element in list

(0, 1, >1)

element occurs first in list

(true, false)

element occurs last in list

(true, false)

مثال: یک تابع داریم که یک لیست رو میگیره و یک عنصر اگر اون لیست یا اون عنصر نال باشه نال پوینتر اکسپشن باید تولید باشه در غیر اینصورت اگر عنصر توی لیست باشه `true` در این صورت تابع می خواهد سرچ کنه یعنی لیست میگیریم با یک عنصر و این عنصر رو توی لیست سرچ میکنیم

از روش `..interface` :

افراز 1: لیست نال باشه یا نباشه

افراز 2: عنصر نال باشه یا نباشه

... --> شکل پایین

الان یک ویژگی معرفی کردیم براساس `interface` براساس شکل بالا

نکته: توی `interface` ها چه روش `functional` و چه روش `input sapce` فقط اکسپشن کجا میزنیم برآمون مهم است و `hapy path` ها و کاری نداریم به این که بدنه تابع ساده است یا پیچیده

list

null

null

notnull

not null

element

null

not null

null

not null

Step 3

Model input domain

Partition characteristics into blocks

Choose values for blocks

حالا میخوایم ویژگی رو براساس functionality بذاریم: ینی براساس این میخوایم ببینیم تعداد رخداد عنصر در لیست چقدر است و افزایش رو براساس این انجام بدیم --> تعداد رخداد 0 - تعداد رخداد یه بار - بزرگتر از یه بار --> این میشه functionality

ورودی رو براساس این حالت پارتیشن بندی کنیم --> برای اکسپشن : نال بودن هپی پس: یک لیست با یک عنصر --> عنصر توی لیست نیست یا بار هست یا بیشتر از یه بار هست

معیار c1 : بودن یا نبودن

معیار c2: توی عکس نوشته شده

نکته: هرچی تعداد بلاک ها بره بالا باعث میشه هزینه بیاد بالا و تست های بیشتری هم طراحی میشه

← تعداد رخداد ران characteristic value در list : (۱ < ۵ , ۱)

① exception handling → بودن null

② Happy path → ① → بودن یا نبودن b1

b1 = true → بودن ، نبودن ، بیمار بودن؟

b2 = false → بسته باز نباشد

b3 = true

b4 = true

آنچه کرام C را آنها بزم نسم ، سلسلی یه حسابت test دارد. از حسابت زیاری ندارد، نزدیک نزدیک است های زیاره) برای آن تولید سود که تولید آن هزینه رارد. وسایه لازم به این مفرغه نباشد، در این حالت ۲) مناسب نیست.

triang(): Relation of side with zero

- 3 inputs, each has the same partitioning

Characteristic	b_1	b_2	b_3
q_1 = “Relation of Side 1 to 0”	positive	equal to 0	negative
q_2 = “Relation of Side 2 to 0”	positive	equal to 0	negative
q_3 = “Relation of Side 3 to 0”	positive	equal to 0	negative

- Maximum of $3*3*3 = 27$ tests
- Some triangles are **valid**, some are **invalid**
- Refining the characterization can lead to more tests ...

اینجا به جای اینکه بیاد سه تاشو باهم در نظر بگیره ۳ تا characteristic معرفی کرده اولی: مثلث هایی میشون که ضلع اولشون مثبته یا صفره یا منفی است دومی: برای ضلع دوم سوم: برای ضلع سوم اینجا تعداد characteristic ها بالا رفته و این بهتره

اینکه چجوری افزار بندی داشته باشیم بر میگردد به خودمون و بودجه که تا چقدری ریز بشیم

Refining triang()'s IDM

Second Characterization of triang()'s inputs

Characteristic	b_1	b_2	b_3	b_4
$q_1 = \text{"Refinement of } q_1\text{"}$	greater than 1	equal to 1	equal to 0	negative
$q_2 = \text{"Refinement of } q_2\text{"}$	greater than 1	equal to 1	equal to 0	negative
$q_3 = \text{"Refinement of } q_3\text{"}$	greater than 1	equal to 1	equal to 0	negative

- Maximum of $4*4*4 = 64$ tests
- Complete only because the inputs are integers ($0 \dots 1$)

Values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side 1	2	1	0	-1

Test boundary conditions

triang() : Type of triangle

Geometric Characterization of *triang()*'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles	equilateral	invalid

What's wrong with this partitioning?

- Equilateral is also isosceles !
- We need to **refine** the example to make characteristics valid

Correct Geometric Characterization of *triang()*'s Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = “Geometric Classification”	scalene	isosceles, not equilateral	equilateral	invalid

فضای حالت براساس نوع مثلث:
مختلف الاضلاع
متساوی الساقین
متساوی الاضلاع
نامعتبر

مشکلشو بالا گفتم که بعضی هاشون با هم اشتراک دارن یعنی مثلث متساوی الاضلاع می تونه متساوی الساقین هم باشه یعنی اگه هر سه ضلعش یک باشه توی هردوتای اینا میشینه و غلط است این

برای بهتر شدن --> بالا گفتم --> با این روش بلاک ها اشتراک ندارن

ولی روش بهتر --> ص 19

Values for triang()

Possible values for geometric partition q_1

Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Yet another `triang()` IDM

- A different approach would be to break the geometric characterization into four separate characteristics

Four Characteristics for `triang()`

Characteristic	b_1	b_2
$q_1 = \text{"Scalene"}$	True	False
$q_2 = \text{"Isosceles"}$	True	False
$q_3 = \text{"Equilateral"}$	True	False
$q_4 = \text{"Valid"}$	True	False

- Use constraints to ensure that
 - `Equilateral = True` implies `Isosceles = True`
 - `Valid = False` implies `Scalene = Isosceles = Equilateral = False`

روش بهتر:

مثلث رو اینطوری نگاه بکنیم به جای اینکه بیایم نوعش رو با یک characteristic بسنجدیم با 4 تا characteristic در نظر بگیریم و هر کدام از بلاک ها بشه true or false ینی یه بار بگیم مثلث مختلف الاضلاع است اگر بود توی بلاک b1 و اگر نبود توی بلاک b2

مثلا اینجا اگر مثلثی هر سه تا ضلعش یک باشه مشکلی نداره --> چون مسئله اینجا اینکه بلاک های یک characteristic با هم اشتراک نداشته باشند و کاری به characteristic نداریم مثلث توی characteristic اول میره توی بلاک b1 و توی characteristic دوم هم می ره توی بلاک b1 و این مشکلی نداره از characteristic های مختلف قطعا با هم دیگه اشتراک دارن : نکته

بروری

Test Case 1 : $\left\{ \begin{array}{l} (3, 4, 5) \\ (1, 1, 1) \end{array} \right\}$

$(3, 4, 5) \rightarrow$ Expected Result = Scalene

$(1, 1, 1) \rightarrow$ Expected Result = Equilateral

مثال: این کل مثلث ها است:

فضای ورودی رو (کل مثلث هایی که میتوانیم تولید کنیم بدیم به برنامه و برنامه بگه بهمن نواعش رو)

8 تا مثلث داریم اینجا --> سوال : ما میخوایم از توی این فضای حالت یه جوری چندتاشو بکشیم
بیرون و نمیخوایم همشو تست بکنیم و فقط چندتاشو بدیم به برنامه و سوال اینکه کدومو بدیم؟
ما میایم اینجا افزار بندی میکنیم اینارو چجوری؟ یه جوری که تیکه هایی که داریم ک اگر از هر
تیکه یک سمپل کشیدیم بیرون کافی باشه

مثلا یه نفر براساس نوع مثلث افزار بندی میکنه: نامعتبرها توی یک افزار - مختلف .. توی یک
افزار - متساوی الاضلاع توی یک افزار - متساوی الساقین --> حالا ویژگی که داریم اینجا چیه
میشه نوع مثلث

حالا مثلا یکی دیگه میگه ما میدونیم تعداد characteristic رو ببریم بالا --> اونایی ک متساوی
الاضلاع هستن رو می ذاریم توی یک بلاک و اونایی که نیستن رو میداریم توی یک بلاک دیگه -->
اینجا دوتا بلاک داریم --> تعداد بلاک های برای متساوی الاضلاع خوبه ولی اونایی که نیستن خوب
نیست برآشون

کل مثلث ها: رودی	
(1,1,1)	(1,1,2)
(2,2,2)	(2,2,3)
1,2	
1,2	
(3,4,5)	(-1,0,2)
(2,3,4)	(0,0,0)

یک characteristic دیگه رو یه جور دیگه بلاک بندی میکنیم:

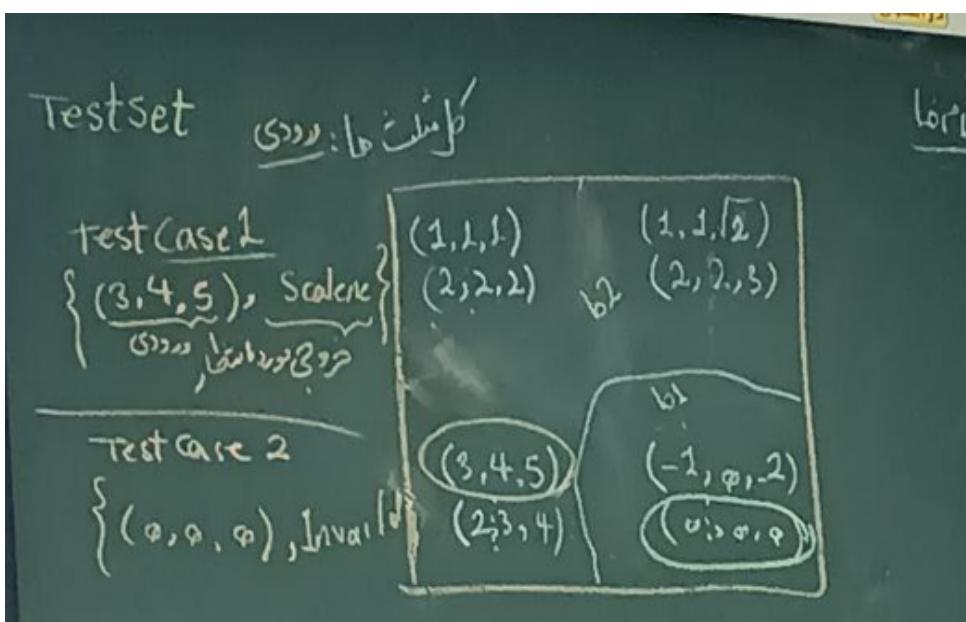
نامعتبرها می‌ذاریم توی بلاک b1 و معتبرها توی بلاک b2 بینی الان معیار پارتیشن بندی رو معتبر بودن یا نبودن می‌گیریم

حالا که بلاک بندی کردیم در در قدم بعدی باید بیایم از هر بلاک یک سپل بکشیم بیرون و همونو تست بکنیم --> اونایی که دورش خط کشیده اورده بیرون و اینا میشه ورودی هایی که برای تولید تست کیس نیاز داریم

تست کیس دو بخش: یکیش خروجی مورد انتظاره مثلا اگر ورودی ما 3 و 4 و 5 بود خروجی مورد انتظار باید مختلف الاصلاع باشه --> حالا دوتا تست کیس داریم اینجا براساس همون افزایی که میشه و می‌کشیم بیرون

نکته: خروجی مورد انتظار اون چیزی که داکیومنت ها داره بهمون میگه ما فقط اینجا پارتیشن بندی میکنیم که از بین این همه ورودی داریم بفهمیم کدوم یکی رو به عنوان نماینده بدیم به تست

نکته: سپل هایی که توی یک بلاک می‌افتن ارزششون باید یکسان باشه



IDM hints

- More characteristics → more tests
- More blocks → more tests
- Do not use program source
- Design more characteristics with fewer blocks
 - Fewer mistakes
 - Fewer tests
- Choose values strategically
 - Valid, invalid, special values
 - Explore boundaries
 - Balance the number of blocks in the characteristics

Modeling the input domain

- Step 1 : Identify testable functions
 - Step 2 : Find all inputs, parameters, & characteristics
 - Step 3 : Model the input domain
 - Step 4 : Apply a test criterion to choose combinations of values (6.2)
 - Step 5 : Refine combinations of blocks into test inputs
-
- The diagram illustrates the progression of steps in modeling the input domain. It starts with Step 1 at the implementation level, moves up to Step 2 and Step 3 at the design abstraction level, then down to Step 4 and Step 5 at the implementation level. A large curly brace on the right side groups Steps 1 through 3, labeled 'Move from imp level to design abstraction level'. Another large curly brace on the right side groups Steps 4 and 5, labeled 'Entirely at the design abstraction level'. A third curly brace on the right side groups Step 5, labeled 'Back to the implementation abstraction level'.

Step 4 – Choosing combinations of values (6.2)

- After partitioning characteristics into blocks, testers design tests by combining blocks from different characteristics
 - 3 Characteristics (abstract): A, B, C
 - Abstract blocks: A = [a1, a2, a3,a4]; B = [b1, b2]; C = [c1, c2,c3]
- A test starts by combining one block from each characteristic
 - Then values are chosen to satisfy the combinations
- We use **criteria** to choose **effective** combinations

:combination برای

یک مسئله داریم که اومدیم فضای حالتش رو یکبار براساس characteristic a پارتیشن بندی کردیم و 4 تا بلاک بهمون داده و یه بار هم براساس characteristic b پارتیشن بندی کردیم و داده b1,b2

یکی از معیارهای پوشش این است که all combination انتخاب بکنیم یعنی این بلاک ها رو همه رو با هم قاطع میکنیم --> مثلا یه دونه تست کیس شماره یک طراحی می کنیم که بخارط ویژگی a توی بلاک a1 و بخارط ویژگی b توی بلاک b1 --> یعنی یک بلاک از یک characteristic باید با تمام بلاک ها از characteristic دیگر ترکیب بشه --> در این حالت تعداد ضرب بلاک های موجود تست کیس نیاز داریم و بعضی هاشون هم infeasible میشه

$$A = [a_1, a_2, a_3, a_4]$$

$$B = [b_1, b_2]$$

Testcase All Combinations

- | | | |
|-------------|---|-------------|
| 1: (a1, b1) | { | 1: (a1, b1) |
| 2: (a1, b2) | | 2: (a2, b2) |
| 3: (a2, b1) | | 3: (a3, b2) |
| 4: (a2, b2) | | 4: (a4, b2) |

All combinations criterion (ACoC)

The most obvious criterion is to choose all combinations

All Combinations (ACoC) : Test with all combinations of blocks from all characteristics.

a1 b1 c1	a2 b1 c1	a3 b1 c1	a4 b1 c1
a1 b1 c2	a2 b1 c2	a3 b1 c2	a4 b1 c2
a1 b1 c3	a2 b1 c3	a3 b1 c3	a4 b1 c3
a1 b2 c1	a2 b2 c1	a3 b2 c1	a4 b2 c1
a1 b2 c2	a2 b2 c2	a3 b2 c2	a4 b2 c2
a1 b2 c3	a2 b2 c3	a3 b2 c3	a4 b2 c3

مثال: دو تا characteristic داریم :

دومی: مثلث مختلف الاصلع باشد یا نباشد

اولی: مثلث متساوی الساقین باشد یا نباشد

حالا که میخوایم توانی این روش که میخوایم برآشون تست کیس تولید بکنیم --> میایم سمبول از دل ورودی ها می کشیم

مثال یک سمبولی باید تولید بکنیم که مختلف ال.. باشد و متساوی الساقین باشد --> است میره کنار

تست کیس دو: مختلف ال.. باشد و متساوی الساقین نباشد --> این اوکیه

سوم: مختلف ال.. نیست ولی متساوی الساقین است

4 : دو تاش نیست --> اوکی است --> نامعتبر است

یکسری تست ها توانی ترکیب کردن ها infeasible در میار د پس وظیفه تستر هست که قیود رو اضافه کنه

قید 1: بلاک b1 از characteristic اول با بلاک b1 از characteristic دوم همزمان امکان پذیر نیستن

$$\text{base Test} \leftarrow 1 + \sum_{i=1}^{\infty} (\beta_i - 1)$$

$$A = [a_1, a_2, a_3, a_4]$$

$$B = [b_1, b_2]$$

Test case:

base Test:

Test Case All Combination	base Test:
1: (a1, b1)	1: (a1, b1) { (a3, b1)
2: (a1, b2)	2: (a2, b2) } Non-base Test:
3: (a2, b1)	3: (a3, b2) { (a1, b2)
4: (a2, b2)	4: (a4, b2) } (a2, b1) { (a4, b1) { (a3, b2)

All combinations criterion (ACoC)

- Number of tests is the product of the number of blocks in each characteristic :
$$\prod_{i=1}^Q (B_i)$$
- The syntax characterization of triang()
 - Each side: >I, I, 0, <I
 - Results in $4*4*4 = 64$ tests
- Most form invalid triangles

How can we get fewer tests ?

ایا این روش all combination روشن خوبی است؟

نه --> الان برای یک مثال ساده 2 به توان 4 بلاک تولید شد اگر مسئله پیچیده بشه تعداد تست کیس هایی که نیاز داریم خیلی زیاده میشه پس باید یه کاری کنیم که تعداد تست کیس ها کم بشن

Example

Input: students

Characteristics: Level, Mode, Major, Classification

Blocks:

Level: (grad, undergrad)

Mode: (full-time, part-time)

Major: (cs, swe, other)

Classification: (in-state, out-of-state)

Abstract IDM:

A = [a1, a2] C = [c1, c2, c3]

B = [b1, b2] D = [d1, d2]

ورودی هاش : دانش اموزان مختلف است
4 تا ویژگی معرفی میکنیم -->
1: سطح تحصیلات
mode :2
3: گرایش
4: دسته بندی (بومی - غیربومی)

الان اینجا تست کیس ها میشه : $2^2 \cdot 3^2 \cdot 2^2$ میشه 24 ---> شاید بعضی هاش infeasible بشن ولی
در کل 24 تا تست کیس داریم

In-class exercise

All combinations criterion (ACoC)

Consider this abstract IDM

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];

C = [c1, c2, c3]; D = [d1, d2]

How many tests are needed to satisfy ACoC?

In-class exercise (*answer*)

All combinations criterion (ACoC)

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];
C = [c1, c2, c3]; D = [d1, d2]

Number of tests: $2*2*3*2 = 24$

a1 b1 c1 d1	a1 b2 c1 d1	a2 b1 c1 d1	a2 b2 c1 d1
a1 b1 c1 d2	a1 b2 c1 d2	a2 b1 c1 d2	a2 b2 c1 d2
a1 b1 c2 d1	a1 b2 c2 d1	a2 b1 c2 d1	a2 b2 c2 d1
a1 b1 c2 d2	a1 b2 c2 d2	a2 b1 c2 d2	a2 b2 c2 d2
a1 b1 c3 d1	a1 b2 c3 d1	a2 b1 c3 d1	a2 b2 c3 d1
a1 b1 c3 d2	a1 b2 c3 d2	a2 b1 c3 d2	a2 b2 c3 d2

ISP criteria – each choice

- We should try at least one value from each block

Each Choice Coverage (ECC) : Use at least one value from each block for each characteristic in at least one test case.

- Number of tests is the number of blocks in the largest characteristic : $\text{Max}_{i=1}^Q (B_i)$

یکی از معیارهای پوشش دیگه --> خیلی ضعیف است

از هر بلاکی که داریم بیا حداقل یه بار استفاده بکن

یک تست کیسی انتخاب بکن که a_1 میداریم توش و با b_1 ترکیب کردیم و برای تست کیس دوم دیگه سراغ a_1 و b_1 نمی‌ریم چون اینارو یه بار استفاده کردیم فرقی نداره a_3 با b_1 بگیریم یا با b_2 چون هدف این بوده که هر بلاک یه بار ظاهر بشه

تعداد تست کیس‌ها: نگاه میکنیم کدام characteristic تعداد بلاک هاش بیشتر است به همون تعداد تست کیس داریم اینجا

$$A = [a_1, a_2, a_3, a_4]$$

$$B = [b_1, b_2]$$

Testcase All combination

- | | |
|-----------------|---|
| 1: (a_1, b_1) | $\left\{ \begin{array}{l} 1: (a_1, b_1) \\ 2: (a_2, b_2) \\ 3: (a_3, b_2) \\ 4: (a_4, b_2) \end{array} \right.$ |
| 2. (a_1, b_2) | |
| 3: (a_2, b_1) | |
| 4: (a_2, b_2) | |
| : | |

In-class exercise

Each choice criterion (ECC)

Apply ECC to our previous example

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];
C = [c1, c2, c3]; D = [d1, d2]

1. How many tests are needed for ECC?
2. Design the (abstract) tests

In-class exercise (*answer*)

Each choice criterion (ECC)

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];
C = [c1, c2, c3]; D = [d1, d2]

Number of tests: $\max(2,2,3,2) = 3$

a1	b1	c1	d1
a2	b2	c2	d2
a1	b1	c3	d1

ISP criteria – base choice (BCC)

- ECC is simple, but very few tests
- The base choice criterion recognizes :
 - Some blocks are more important than others
 - Using diverse combinations can strengthen testing
- Lets testers bring in domain knowledge of the program

Base Choice Coverage (BCC) : Choose a base choice block for each characteristic. Form a base test by using the base choice for each characteristic. Choose subsequent tests by holding all but one base choice constant and using each non-base choice in each other characteristic.

- Number of tests is one base test + one test for each other block $1 + \sum_{i=1}^Q (B_i - 1)$

معیار :base choice

برای هر ویژگی بیا یک بلاک رو مشخص بکن و بگو کدام بلاک از همه مهمتر است میشه بلاک base

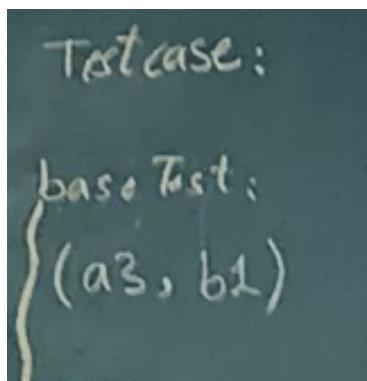
مثلا میدونیم اکثر مثلث هایی که کاربر میده اکثرشون مختلف الاضلاع است پس بلاک base رو می داریم مثلث مختلف الا ..

اگر به ورودی ها براساس ویژگی a نگاه بکنیم مهمترین ورودی میشه اونایی که توی بلاک a3 قرار دارن این میشه بلاک base ما چون میخوایم کاری بکنیم روی همین بلاک که حساسیتش بالا است مانور بدیم

و مثلا b1 فرکانس رخدادنش خیلی زیاده و کاربر اینو میده بیشتر بهمون و روی این مرکز بشیم پس بلاک بیس میشه b1

حالا اگر به این نتیجه برسیم که که هیچ کدام از اینا اهمیتی بر دیگری نداره --> کدام انتخاب میشه؟ رندوم انتخاب میکنیم --> محدودیتی نداریم دیگه تست کیس هایی توی این حالت:

تست کیس دو دسته میشه --> base test , non base test
توی base تست کیس هایی که هستن --> بیا تستی طراحی بکن که تمام بلاک های base نشسته باشن توی جایگاهشون --> از هر ویژگی بیس بلاک رو انتخاب میکنیم و تست رو می سازیم باهاش



non base test : میایم از روی بیس تست ها رو اینارو می سازیم --> تمام بیس ها رو ثابت نگه داریم به جز یکیشون و جای اون یکی بلاک های غیر بیس رو می ذاریم --> به ازای هر بیس تست اینو انجام میدیم --> از بین این a_1 و a_3 که داشتیم و به جای بیسی که ثابت نگه نداشتیم بلاک های غیر بیس رو می ذاریم --> برای ثابت نگه داشتن a_1 اینطوری رفتیم : شکل پایین بعدش a_3 رو ثابت میگیریم

نکته: نمی تونیم تستی رو پیدا بکنیم که بلاک بیس تو ش نباشه

Non-base Test:	
	(a_1, b_2)
	(a_2, b_1)
	(a_4, b_1)
	(a_3, b_2)

Base choice notes

- The base test must be **feasible**
 - That is, all base choices must be **compatible**
- **Base choices** can be
 - Most likely from an **end-use** point of view
 - Simplest
 - Smallest
 - First in some ordering
- **Happy path** tests often make good base choices
- The base choice is a **crucial design decision**
 - Test designers should **document** why the choices were made

In-class exercise

Base choice criterion (BCC)

Apply BCC to our previous example

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];
C = [c1, c2, c3]; D = [d1, d2]

1. How many tests are needed for BCC?
2. Pick base values and write one base test
3. Design the remaining (abstract) tests

In-class exercise (*answer*)

Base choice criterion (BCC)

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];
C = [c1, c2, c3]; D = [d1, d2]

Number of tests: $1(\text{base})+1+1+2+1 = 6$

Base	a1 b1 c1 d1
A	a2 b1 c1 d1
B	a1 b2 c1 d1
C	a1 b1 c2 d1
C	a1 b1 c3 d1
D	a1 b1 c1 d2

ISP criteria – multiple base choice

- We sometimes have more than one logical base choice

Multiple Base Choice Coverage (MBCC) : Choose at least one, and possibly more, base choice blocks for each characteristic. Form base tests by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

- If M base tests and m_i base choices for each characteristic:

$$\text{تعداد بیس تست ها میشه} : M + \sum_{i=1}^Q (M * (B_i - m_i))$$

For our example: Two base tests: a1, b1, c1, d1 a2, b2, c2, d2

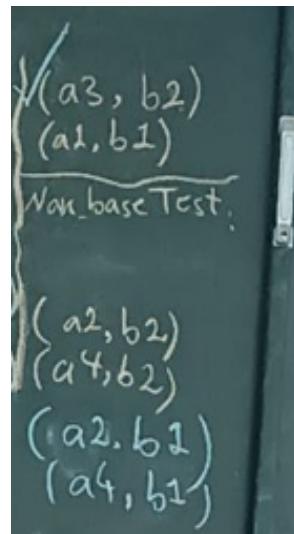
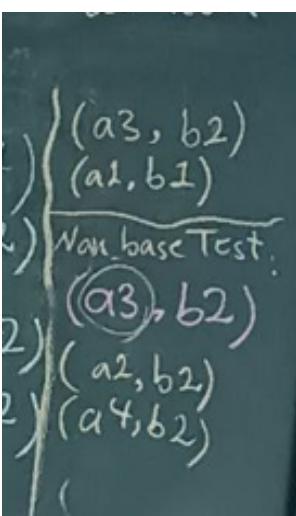
Tests from a1, b1, c1, d1: a1, b1, c3, d1

Tests from a2, b2, c2, d2: a2, b2, c3, d2

:multiple base معیار

برای هر characteristic به جای اینکه یک بلاک رو به عنوان بیس انتخاب بکنیم چندتا رو بیس انتخاب بکنیم مثلا از A هم a_1 بیس باشه و هم a_3 و از B مثلا فقط b_1 بیس باشه --> الزاما چندتا نمی خوایم ولی حداقل یکی میخوایم
???

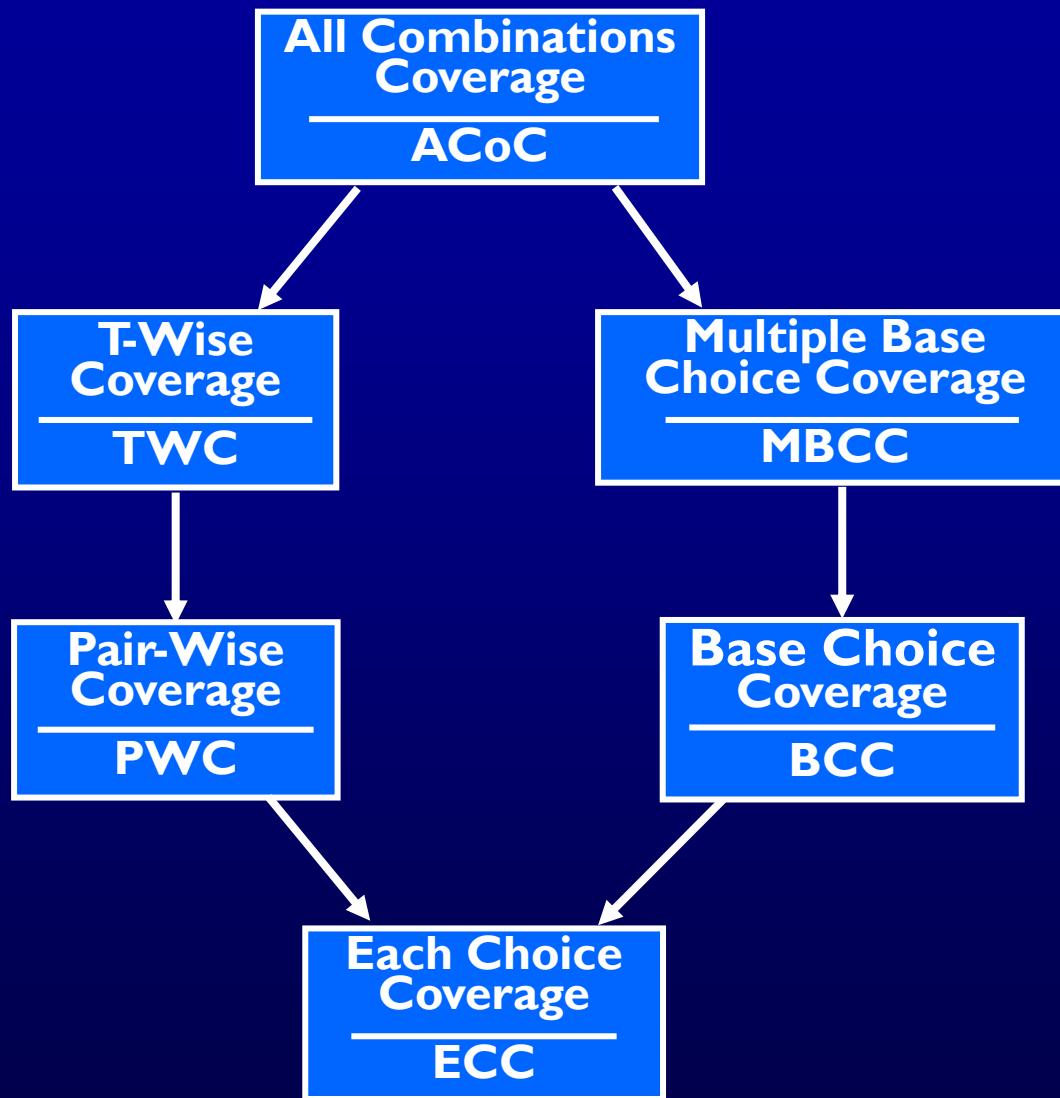
هر بیس تست برای هر characteristic حداقل یکبار ظاهر بشه برای بیس تست ها --> الان که a_1 اضافه شد ما این حالت رو داریم
حالا اگر b_2 هم جز بیس های characteristic B باشه --> دیگه نمیاریمش چون یه بار ظاهر شده توی شکل پایین --> فقط کافیه هر بلاک بیس حداقل یه بار ظاهر بشه
حالا برای non base test ها : از روی این بیس تست ها اینارو می سازیم --> یکی از بیس تست ها رو میگیریم و بعد می ریم سراغ بعدی --> به ازای هر کدام از بلاک های بیس که انتخاب میشه بقیه باید ثابت باشن این میشه نان بیس ها
برای a_3 نه b_1 میشه و نه b_2 چون از B هر دو تاشون بیس هستن و باید نان بیس بذاریم کنار a_3
پس کلا نداریم چیزی برآش



$$A = [\begin{matrix} a_1 & a_2 & \overset{a_3}{\cancel{a_3}} & a_4 \\ b_1 & b_2 \end{matrix}]$$

T_b

ISP Coverage Criteria Subsumption



نکته کلی:

بعضی از ترکیب ها infeasible میشن و اونارو نادیده می گیریم

اما اگر تعداد infeasible ها خیلی رفت بالا توی روشهی که بیس داریم چه یه دونه بیس و چه multiple base تعداد تست کیس های infeasible اگر خیلی رفت بالا ما می فهمیم توی بلاک بندی ها خطای داشتیم و خوب انتخاب نکردیم و باید ویژگی و بیس اینارو عوض بکنیم و همینطور قیدها هم بنویسیم یعنی اونایی که باعث infeasible میشن

Constraints Among Characteristics

(6.3)

- Some combinations of blocks are **infeasible**
 - “less than zero” and “scalene” ... not possible at the same time
- These are represented as **constraints among blocks**
- Two general types of constraints
 - A block from one characteristic **cannot be** combined with a specific block from another
 - A block from one characteristic can **ONLY BE** combined with a specific block from another characteristic
- Handling constraints depends on the criterion used
 - **ACC, PWC, TWC** : Drop the infeasible pairs
 - **BCC, MBCC** : Change a value to another non-base choice to find a feasible combination

Example Handling Constraints

```
public boolean findElement (List list, Object element)
```

// Effects: if list or element is null throw NullPointerException

// else return true if element is in the list, false otherwise

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted	More than one, all identical
B : match	element not found	element found once	element found more than once	
Invalid combinations : (A1, B3), (A4, B2)				

element cannot be in a one-element list more than once

If the list only has one element, but it appears multiple times, we cannot find it just once

Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

**Simple, straightforward, effective,
and widely used**
