

Introduction to Software Testing (2nd edition) Chapter 2

Model-Driven Test Design

Paul Ammann & Jeff Offutt

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

Updated September 2016

Complexity of Testing Software

- No other engineering field builds products as **complicated** as software

صحت

- The term **correctness** has no meaning
 - Is a **building** correct?
 - Is a **car** correct?
 - Is a **subway system** correct?
- Like other engineers, we must use **abstraction** to manage **complexity**
 - This is the purpose of the **model-driven test design** process
 - The “model” is an abstract structure

نرم افزار پیچیده ترین محصولی است که مهندس های رشته های مختلف می تونن تولید بکنن
 نرم افزار از این نظر پیچیده است که اشکالاتش به سختی می تونه بروز پیدا بکنه
 ما توی تست نمی تونیم درست بودن سیستم رو نشون بدیم فقط می تونیم تلاش بکنیم که Failures ها
 رو پیدا بکنیم و شناسایی بکنیم و هدف این است که با کمترین میزان تلاش بیشترین Failures ها رو
 کشف بکنیم و رفعشون بکنیم

تعریف correctness هم خیلی تعریف واضحی نیست مثلا وقتی که می گیم یک ماشین درست کار
 می کنه شاید مبهم باشه --> توی فانکشنال ها وقتی که ما نرم افزار رو بررسی میکنیم خیلی راحت
 می تونیم بگیم درست رفتار کرد یا نه ولی توی نان فانکشنال ها تعریف correctness خیلی سخت
 و مبهم میشه

نان فانکشنال ریکوارمنت ها مبتنی بر how هستن و فانکشنال ریکوارمنت ها مبتنی بر what هستن

what ینی چه چیزی ما قراره توی نرم افزار مون انجام بدیم و چه کاری
 how ینی چگونه و تا چه حدی مثلا پر فرمنس درسته خب باشه الان تا چه حدی درسته؟ پس رفتار
 مورد انتظار مون اینجا بررسی میشه با رفتار نرم افزار

model-driven test design ینی یکسری مدل از توی سورس کدمون بکشیم بیرون و بعد تست
 رو روی اون مدل ها انجام بدیم چرا این کارو انجام میدیم؟ ینی سورس کد رو مدل بکنیم و بعد اون
 مدلو تست بکنیم چرا؟ چون راحت تره

هر چقدر سطح abstraction رو بالاتر ببریم احتمال اینکه تست ها overlap کمتری داشته باشند
 بیشتر است چون جزئیاتی که مد نظر مون نیست توی abstraction نادیده گرفته میشه
 abstraction ینی یک نگاه سطح بالا --> پس میشه نادیده گرفتن جزئیات
 پس هر چقدر سطح abstraction بره بالا ینی جزئیات بیشتری رو نادیده بگیریم
 منظور از مدل در ساده ترین حالت ینی گراف

Software Testing Foundations (2.1)

**Testing can only show the presence
of failures**

Not their absence

توی تست یکی از هدف هامون اینه که نشون بدیم که Failures ها وجود دارن

می تونیم نشون بدیم وجود دارن ولی نمی تونیم نشون بدیم که وجود ندارن Failures

در تست این مسئله undesirable است --> توی تست نشون دادن این که برنامه خالی از failure است یک مسئله undesirable است

ینی این که اصلا با استفاده از یک کامپیوتر نمی تونیم حلش بکنیم undesirable

Testing & Debugging

- Testing : Evaluating software by observing its execution
- Test Failure : Execution of a test that results in a software failure
- Debugging : The process of finding a fault given a failure

Not all inputs will “trigger” a fault
into causing a failure

تست کردن ینی این که ببایم ورودی بدیم به برنامه و رفتارشو توی اجرا ببینیم که چجوری هستش و مورد انتظارمون هستش یا نه

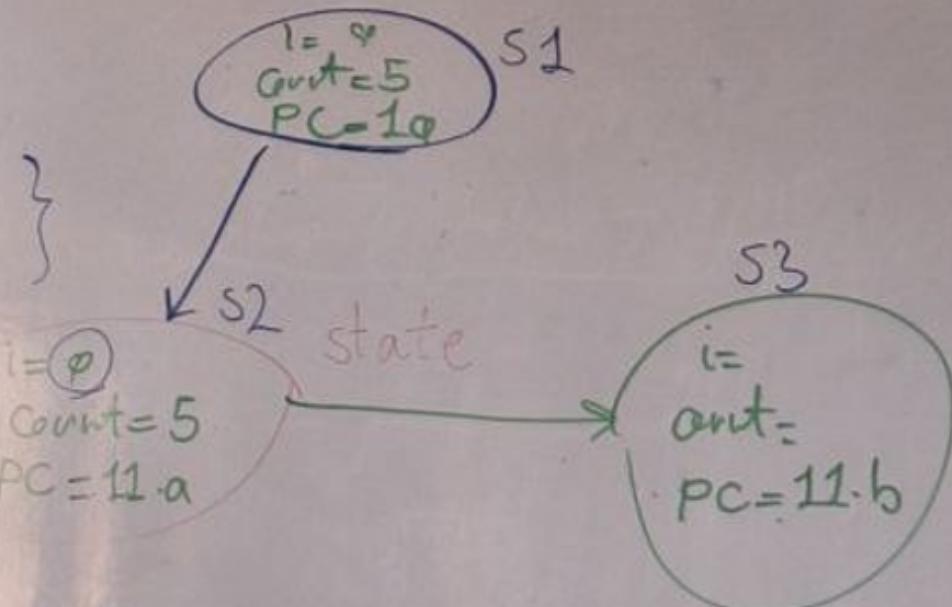
Test Failure: ینی رفتارش نتیجه اجرا متفاوت با اون نتیجه مورد نظر هستش

دیباگ کردن توی حوزه نرم افزار یک واژه مهم است توی فصل یک گفته شده --> دیباگ: فرایندی است که باعث میشه ما قدم به قدم جلو بریم که بررسیم به اون سر منشا **fault**

int i=0;

10. int count;

11. \rightarrow for (int i=1; i < count; i++) {



مثال:

فرض می کنیم count ما اینجا 5 است و مقدار اولیه آن هم دادیم 0
 $i=1$ اینجا میشه Fault

ایا هر وقت به fault رسیدیم در برنامه باعث ارور می شه یا نه؟
ارور ینی استیت داخلی سیستم بره توی یک استیتی که مورد انتظار برنامه نباشه --> شکلش کشیده شده

استیت برنامه در هر لحظه میشه:
مقدار آن
مقدار کانت

--> ینی چه خطی از برنامه یا چه قسمتی از برنامه داره اجرا میشه PC

اگر آن را اشتباها به جای صفر بذاریم یک این میشه Fault حالا این باعث ارور میشه یا نه؟
بله میشه چون انتظار داریم از S1 (وقتی که داریم خط 10 رو اجرا میکنیم ینی داخل استیت S1 هستیم) وارد S2 بشیم ینی استیت داخلی بعدی برنامه و انتظار داریم استیت S2 برنامه ای کماکان صفر باشه و PC=11.a باشه و count=5 ولی وقتی که این Fault رو مرتكب شدیم استیت مورد انتظارمون بعد از S1 میشه S2 ولی چون fault داریم بعد از S1 وارد S2 نمی شیم پس اینجا ارور داریم (چون انتظار داشتیم بعد از استیت S1 بره به S2 ولی اینطوری نیست و به یک استیت دیگه رفته : ارور)

ایا fault همیشه باعث ارور میشه یا نه؟ نه همیشه نمیشه مثلًا اگر آن کوچکتر از $(count \% 2)$ باشه بعضی وقتی باعث میشه استیت مورد انتظار استیت درست باشه و بعضی وقتی هم نه --> در هر حالتی بازم ارور دار میشه؟؟؟؟؟

Fault & Failure Model (**RIPR**)

Four conditions necessary for a failure to be observed

1. **Reachability** : The location or locations in the program that contain the fault must be reached
2. **Infection** : The state of the program must be incorrect
3. **Propagation** : The infected state must cause some output or final state of the program to be incorrect
4. **Reveal** : The tester must observe part of the incorrect portion of the program state

چی میشه که یک **fault** تبدیل به **Failures** میشه؟

Failures: ینی توی اون استیت نهایی کد خطا رخ بده و توی ظاهر نشون داده میشه --> ینی کاربر اون چیزی که قرار بود مشاهده بکنه از نرم افزار با اون چیزی که تولید شده متفاوت باشه برای اینکه یک **Fault** تبدیل به **Failures** بشه 4 تا شرط لازمه:

Reachability : اول از همه توی اون تستی که داریم انجام میدیم به اون نقطه بررسیم مثلاً اگر حلقه **for** توی یک تابعی است که این تابع در تست هیچ وقت فراخوانی نمیشه ینی تست رو جوری نوشتیم که این تابع هیچ وقت صدا زده نمیشه پس این **fault** هیچ وقت تبدیل به **Failures** نمیشه پس اولین شرط این هست که به اون مکان بررسیم

Infection : ینی الوده بشیم --> ینی تاثیرشو بذاره روی استیت داخلی ینی **fault** تاثیرشو روی استیت داخلی بذاره ینی مثل یک ویروسی که میاد داخل یک جامعه اول از همه باید اون فرد آلوده بشه تا بعد بتونه به همه منتقل بکنه پس به طور کلی ینی استیتمون خراب بشه

Propagation : ینی مثلاً اون ادمه که ویروس گرفته بیاد ویروسشو پخش بکنه بین ادما --> ینی این استیت S2 استیت مورد انتظار نبوده و این بیاد پخش بشه ینی همینطوری استیت بعدی هم باز به عنوان سومین استیتی که داریم مورد انتظار مون نباشه و استیت 4 هم نباشه و .. ینی هیچ وقت به اون مسیر درست برنگردد

Reveal : اشکار شدن --> ینی استیت نهایی خراب شده ولی کاربر نمی بینه ینی اون کسی که داره تست میکنه از دید تستر پنهان می مونه پس وقتی از دید تستر پنهان بمونه ینی **Reveal** نشده نکته: **missing code** ها هم **fault** حساب می شن ینی اگر یک تیکه از کد رو از دست بدیم یا.. هرچیزی که **miss** شده اینم می تونه **fault** در نظر گرفته بشه

مثال کلی:

یک سیستمی داریم که می خواهد لاگین بکنه و این سیستم مال بانکه و کاربر پسورد رو اشتباه می زنه بعد ما پسورد رو می گیریم ولی چک نمی کنیم و با نام کاربری سریع وارد اکانت می شیم و پول ها جابه جا میشه و **Faults** رخ میده و **Failures** اینه که پسورد رو چک نکردیم و اتفاقی که می افته اینه که استیت سیستم خراب میشه و تبدیل به ارور میشه و حتی ارور پخش هم میشه یعنی اتفاق هایی تنوی سیستم می افته و پول جابه جا میشه و تبدیل میشه به **Failures**

ash مثلا موجودی یک نفر در عرض یک ثانیه شده 10 میلیارد **reveal** اش یعنی تست کردیم و این حلقه چک کردن پسورد داخل یک فانکشن است و تستی رو که نوشتم هیچ وقت در هنگام اجرا کردن اون تست هیچ وقت این فانکشن صدای زده نمیشه که بخوایم به این نقطه بررسیم

RIPR Model

- Reachability
- Infection
- Propagation
- Reveability



شکلش:

تست باید اول از همه به اون نقطه `fault` اش بررسیم و `Reachability` باید باشه و بعد استیت داخلی سیستم آلوده بشه و بعد وقتی که استیت داخلی آلوده شد ینی ارور ظهر می‌پیدا کرد باید این ارور دائم باعث بشه که استیت های بعدی هم با استیت های مورد انتظار مون متفاوت باشند ینی برنگردیم به اون استیت اصلی و باعث بشه استیت نهایی سیستم اشتباه بشه و وقتی که استیت نهایی سیستم اشتباه شد اگر اون کسی که داره تست می‌کنه قسمتی که به اشتباه چک نکنه و نبینه این `reveal` نشه

Software Testing Activities (2.2)

■ Test Engineer : An IT professional who is in charge of one or more technical test activities

- Designing test inputs
- Producing test values
- Running test scripts
- Analyzing results
- Reporting results to developers and managers

■ Test Manager : In charge of one or more test engineers

- Sets test policies and processes
- Interacts with other managers on the project
- Otherwise supports the engineers

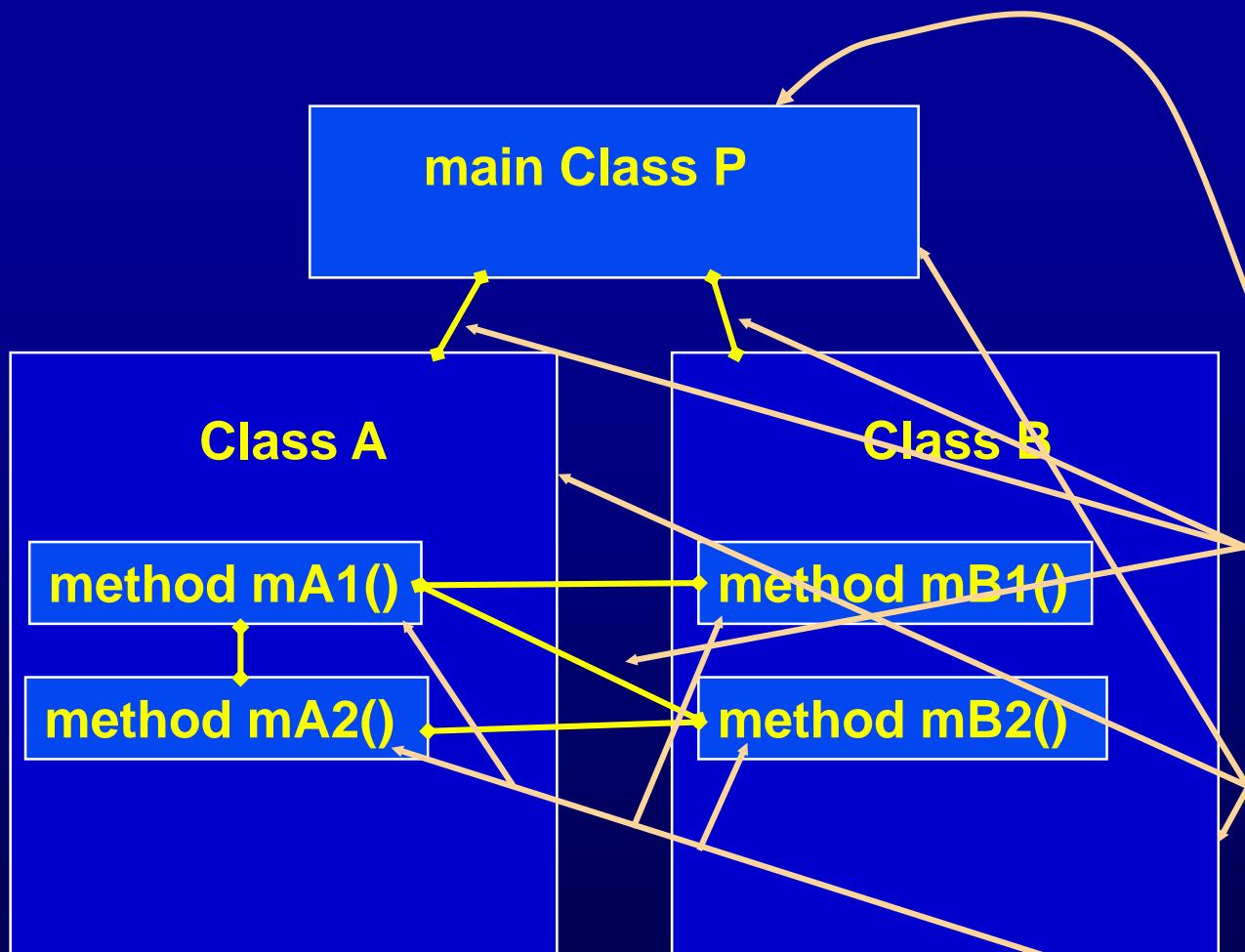
کارش دیزاین کردن تست ها است و چجوری دیزاین کنیم؟ براساس استراتژی Test Engineer و Test Manager ینی Test Manager خط و مشی رو مشخص میکنه و میگه براساس این استراتژی که انتخاب کردیم برای پروژه شما برو دیزاین بکن Test Engineer ها نتایج رو گزارش میدن به منیجرشون و دولوپرها اینکه Test Engineer تست رو دیزاین میکنه و اجرا میکنه ایا همشو یک نفر انجام میده؟ نه الزاماً و اینا هم سطح بندی داره

یکسری Test Engineer رو داره مدیریت می کنه و لولش بالاتر از Test Manager ها است و وقتی که مبتدی باشیم به عنوان Test Engineer استخدام می شیم و بعد که حرفه ای تر بشیم به عنوان Test Manager --> استراتژی رو مشخص میکنه و اینا رو می نویسن و مکتوبش می کنن و بعد تست منیجر مدیریت می کنه Test Engineer رو

--

تست پلن: چندتا صفحه داره و درشو که باز میکنیم یک داکیومنت است و تیم تست باید ایجادش بکنه و چیز های مختلفی تو ش مشخص میکنیم --> همون جوری که برای ریکوارمنت ها داکیومنت تهیه می کنیم برای تست هم باید داکیومنت داشته باشیم ینی از همون اول که پروژه استارت زده میشه باید تست پلن مشخص باشه و هرچقدر که می ریم جلو باید ورژن های قبلی رو دقیق تر بکنیم تست پلن مثلا اسکوپ رو مشخص میکنیم که اول از همه این پروژه توی چه حیطه ای می خواد کار بکنه - هدف از تست چیه - تست ریکوارمنت ها رو داخلش مشخص میکنیم - استراتژی ها رو مشخص میکنیم - چه منابعی برای تست نیاز داریم - چه سخت افزاری داریم - زمان بندی تست رو مشخص میکنیم

Traditional Testing Levels (2.3)



- **Acceptance testing :** Is the software acceptable to the user?
- **System testing :** Test the overall functionality of the system
- **Integration testing :** Test how modules interact with each other
- **Module testing (developer testing) :** Test each class, file, module, component
- **Unit testing (developer testing) :** Test each unit (method) individually

وقتی یک برنامه به زبان شی گرا نوشته شده باشد چگونه تستش می‌کنیم؟

هر فانکشنی از هر کلاس رو تست می‌کنیم اول از همه : **unit test** خوب پس برایم کوچکترین جز که فانکشن کلاس ها الان هست رو برایم جداگانه نگاه کنیم و تست کنیم یعنی برایم این 4 تا فانکشن رو ببینیم به تنها یکی درست کار می‌کنند یا نه

Module testing : حالا فرض می‌کنیم این فانکشن‌ها همچون درست هستند و بعد سه تا کلاسی که داریم اینجا رو بباییم تست بکنیم --> کل کلاس رو تست می‌کنیم یعنی ارتباط مژول هاش با هم، ارتباط فانکشن‌هاش با هم و ...

Integration testing : حالا این کلاس‌های مختلف چگونه با هم دارن تعامل پیدا می‌کنند --> ارتباط بین کلاس‌ها رو تست می‌کنیم

System testing : یعنی کل subsystem‌ها در کنار هم نشستن و قبل از اینکه ما برایم تحویل مشتری بدمون یکبار همه رو سر هم می‌کنیم و تست می‌کنیم

Acceptance testing : یعنی به کاربر نهایی نشون بدم و اونها تست بکنم

نکته: موقع طراحی نگاه از بالا به پایین است ولی موقع اجرا نگاه از پایین به بالا است

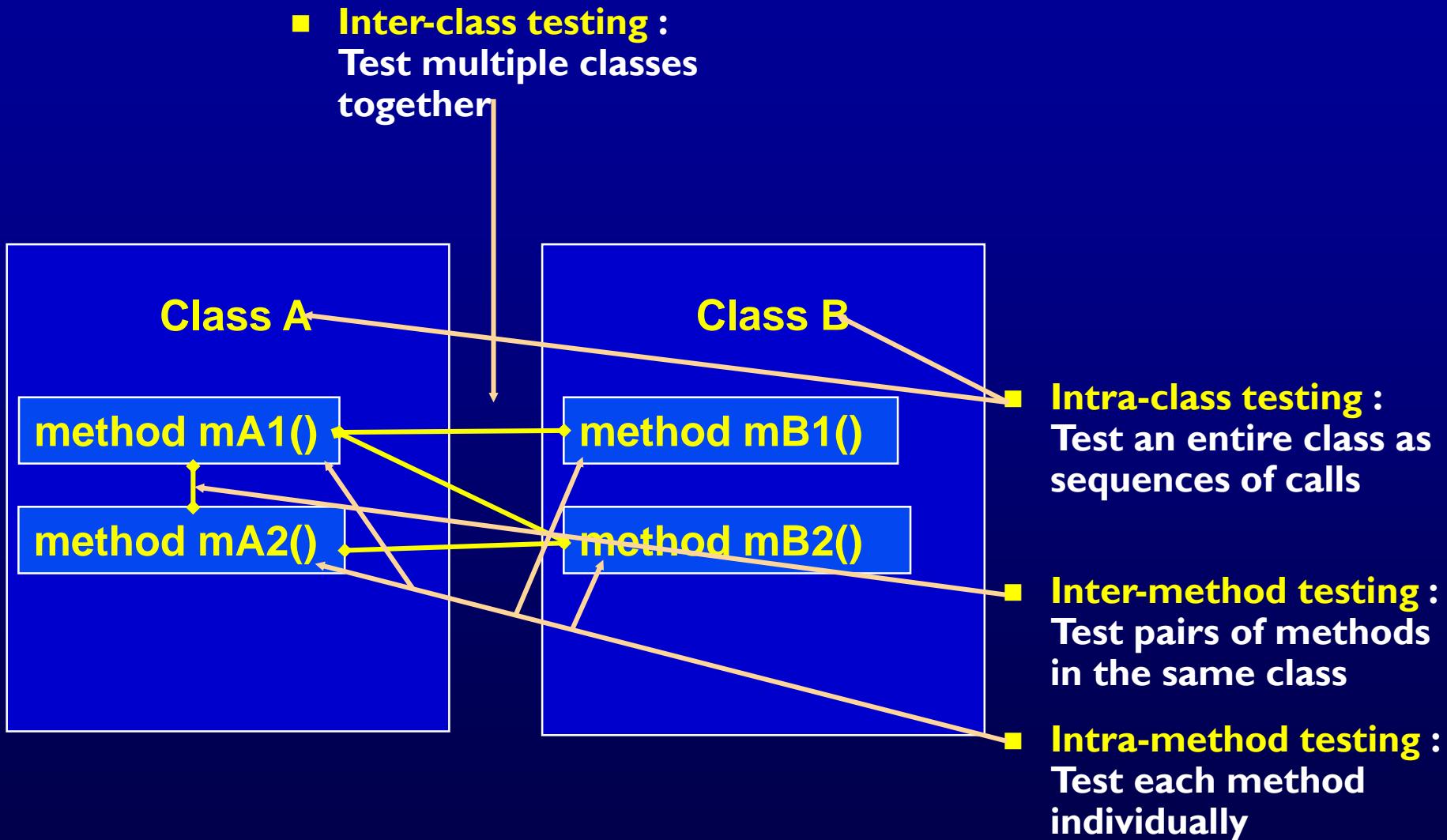
نکته: فرق System testing با validation

validation یعنی ما برایم نگاه بکنیم که ایا با اون چیزی که کاربر مورد انتظارش هست مطابقت

داره یا نه --> validation توی یک بخشیش میشه

ولی سیستم تست خودمون توی شرکت خودمون داریم یک سیستمی که سر هم شده رو تست می‌کنیم

Object-Oriented Testing Levels



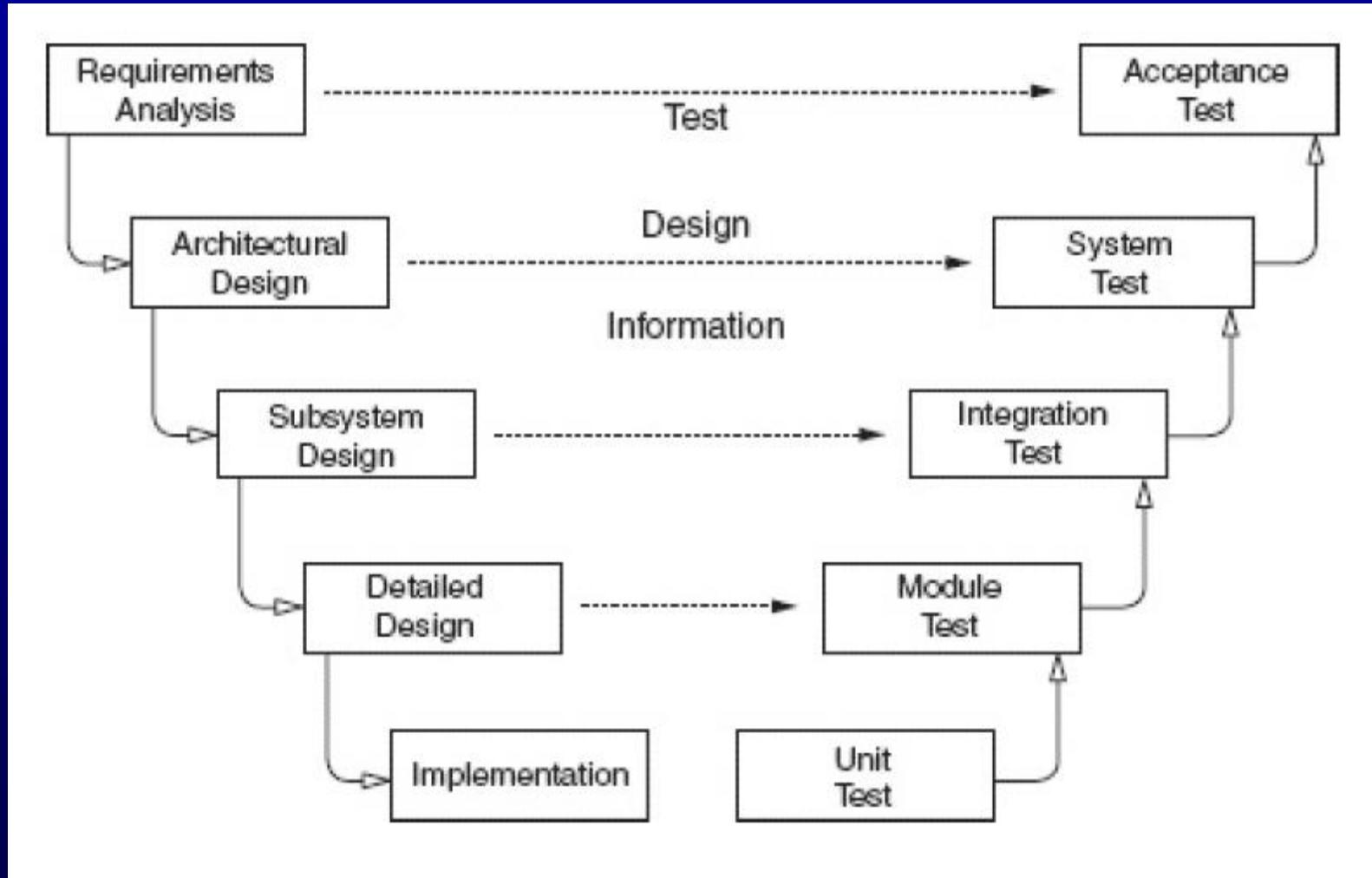
اگر نخوایم از ۷ مدل استفاده بکنیم --> یک رویکرد دیگری داریم به صورت زیر:
Intra-method testing : ینی داخل یک چیز رو قراره تست بکنیم --> داخل متند رو تست
می کنیم

Inter-method testing : ینی بین دو تا چیز رو قراره تست بکنیم --> روابط بین متدهای یک
کلاس رو تست و چک میکنیم

Intra-class testing : داخل یک کلاس رو چک میکنیم که ایا متدها و متغیرهایی که مال
کلاس هستند درست کار می کنن یا نه

Inter-class testing : تعاملات بین دو تا کلاس هستش

Software development activities and testing levels – the “V Model”



۷ مدل:

وقتی که متداول‌زیمون traditional باشه ینی این چندتا گام رو قراره طی بکنیم لازمه که از همون اول کار به فکر بشیم

گام اول توی متداول‌زی traditional ریکوارمنت در اوردن است از طریق پرسش نامه یا.. بهش دست پیدا میکنیم--> ریکوارمنت ها رو که در میاریم باید همون لحظه به فکر acceptance test باشیم ینی چی؟ تست پذیرفته شده که مشتری این رو می‌پذیره

acceptance test یکی از انواع تست است ینی ما نرم افزار که تولید کردیم و رفتیم به یوزر نهایی تحويل دادیم بگه همینه که می خواستم و این ینی نرم افزار acceptance test رو پاس کرده همون جوری که ریکوارمنت در میاریم برای نرم افزار باید تست ریکوارمنت هم دربیاریم --> تست ریکوارمنت در میاریم و اینارو مكتوب میکنیم و اخر کار که خواستیم تحويل یوزر بدیم همه اینا رو باید چک بکنیم

مرحله بعد دیز این سطح بالاست --> طراحی سطح بالا اولین گامش طراحی معماری است
معمار نرم افزار تصمیمات کلان و تاثیر گذار نرم افزار رو میگیره و اگر خراب کاری کرد کلی سرمایه از دست می‌ره

مرحله بعدی اینه که وقتی که معماری کلی مشخص شد --> اگر قراره subsystem های مختلفی داشته باشیم این subsystem ها هر کدامشون در درونشون قراره چه اتفاقی بیوفته subsystem ینی هر کدامشون به طور مجزا برای خودشون کاری رو انجام بدن ولی سطح بالاتر از اون مازول ها

توی مرحله معماری خود سیستم و این که چه نان فانکشنال ریکوارمنت هایی رو باید براورده بکنه مد نظر گرفته میشه و توی مرحله پایین تر توی این مرحله مشخص میشه هر subsystem به چه نحوی باید تعامل داشته باشه

detailed design : حالا وارد جزئیات میشیم ینی توی هر subsystem قراره چه کلاس هایی داشته باشیم و چجوری قراره با هم تعامل داشته باشند

هرچقدر که بیایم پایین تر بهش میگن طراحی سطح پایین ینی هیچ مرزبندی توی داکیومنت ها و توی ورژن های مختلف نیست بعدم تبدیل به کد میشن طرف مقابل:

برای requirements analysis فعالیت نظیرش acceptance test میشه این کاربرهای نهایی که ایا این نرم افزار اون چیزی هست که می خواستن یا نه

ایا همیشه یوزر نهایی تعیین کننده است؟ خیر - فقط کاربر نهایی مهم نیست صاحبان پروژه هم نظرشون توی تست نهایی محصول مهم است

نکته:

هزینه - زمان - کیفیت این سه تا فاکتور یک مثلث رو می سازن که هیچ وقت توی یک پروژه نرم افزاری هر سه تاشون با هم براورده نمیشن مثلای کاربری داریم که می گه سریع پروژه رو می خوایم و کیفیتش هم می خود خوب باشه و هزینه اش هم پایین باشه --> هیچ وقت به چنین مرحله ای نمی رسیم و یکی از این سه تا باید قربانی بشه

توی مرحله requirements analysis هنوز محصول ساخته نشده ولی همینجور که داریم نیازمندی هارو استخراج میکنیم باید acceptance test رو طراحی بکنیم - طراحی بکنیم ینی این که داکیومنت بکنیم و بگیم چه تست ریکوارمنت هایی احتیاج داریم

acceptance test رو دولوپرا انجام میدن یا نیروی جدآگانه براش می گیریم؟ نیروی جدآگانه چون این کاری نیست که دولوپر بخود انجام بده توی طراحی acceptance test و نیروی جدآگانه جدآگانه باید استخدام بشه و کسی باید باشه که به اون حوزه مسلط باشه

نکته: توی acceptance test ما باید مشخص بکنیم که چه بخش هایی از نرم افزار قراره که کاور بشه و اگر سپردم دست خود یوزر ممکنه یه سری چیزهای دیگه رو امتحان بکنه که مد نظر ما نیست --> پس باید مشخص بکنیم که دقیقا توی چه شرایطی باید acceptance test رو انجام بدیم

نکته: توی مرحله requirements analysis که هستیم وقتی که می خوایم ریکوارمنت ها رو استخراج کنیم همون موقع تست ریکوارمنت های مربوط به acceptance test رو استخراج کنیم ولی قرار نیست انجام بدیم چون هنوز محصولی نداریم و محصول نهایی وقتی است که از اونور بیایم پایین و بعد بریم بالا سیستم تست: ما سیستم رو سر هم کردیم و درستش کردیم و خودمون تستش میکنیم --> انجامش: وقتی که subsystem ها به هم وصل شدن و سیستم نهایی تولید شد قبل از اینکه تحويل کاربر بدیم انجامش بدیم کی طراحیش میکنیم؟ ایا وقتی که subsystem ها اماده شدن؟ نه --> به محض این که معماری رو استخراج کردیم و همین لحظه سیستم تست رو انجام میدیم

توی بعضی از کتاب ها integration test و مازول تست رو یکی در نظر می گیرن - integration test توی این مدل ۷ در نظر میگیره که هر subsystem الان اماده شده و خود subsystem رو می ریم چک میکنیم که فانکشنال و نان فانکشنالش درست است یا نه یعنی الان مجموعه ای از کلاس ها رو داریم و الان میریم این مجموعه ای از کلاس ها که یک subsystem رو تشکیل میدن چک میکنیم

مازول : مجموعه ای از unit ها است مثلا مازول رو در نظر گرفته که یک کلاس است یا یک فایل از کدها هستش یا یک پکیج است بسته به این که زبانمون شی گرا هست یا نه --> توی قسمت مازول تست بررسی میکنیم که ایا این فانکشن هایی که توی یک کلاس هستن که unit تست رو با موفقیت پاس کردن حالا در کنار هم در قالب یک کلاس واحد ایا درست دارن کار می کنن یا نه

اون اوایل که زبان های شی گرا نبودن unit به هر فانکشن گفته میشد اما بعدا که زبان های شی گرا اومدن unit به هر کلاس گفته شد توی این مدل فرض شده که فانکشن unit است --> کلا هر برنامه ای که داریم به کوچکترین واحدی که بتونیم به صورت مستقل تستش بکنیم میتونیم به عنوان unit در نظر بگیریم و کی انجام میشه؟ موقعی که کدمون رو پیاده سازی کردیم و توسط پروگرامها انجام میشه

نکته: موقع دیزاین از بالا طراحی میکنیم و میایم پایین ولی موقع انجام از پایین می ریم بالا

Coverage Criteria (2.4)

- Even small programs have too many inputs to fully test them all
 - **private static double computeAverage (int A, int B, int C)**
 - On a 32-bit machine, each variable has over 4 billion possible values
 - Over 80 octillion possible tests!!
 - Input space might as well be infinite
- Testers search a huge input space
 - Trying to find the fewest inputs that will find the most problems
- Coverage criteria give structured, practical ways to search the input space
 - Search the input space thoroughly
 - Not much overlap in the tests

Coverage criteria یا معیارهای پوشش:

چه معیارهایی می‌توانیم برای تست داشته باشیم؟

برای تست بتنونیم جوری تست رو انجام بدیم که با کمترین تعداد تست بیشترین کارایی رو داشته باشیم

ینی تا جایی که می‌توانیم باید تست ها overlapping کمی داشته باشند یعنی اون یکی تست نتونه

اون یکی تست رو پوشش بده و اگر نتونست پوشش بده دیگه اون تست رو نمی‌یاریم

-1 Coverage criteria که داشته باشیم می‌توانه به ما کمک بکنه که کی تست رو متوقف بکنیم

مثلماً توی طراحی تست جوری تست ها رو طراحی می‌کنیم که مطمئن باشیم خط به خط هر خط

توی سورس کد دقیقاً یکبار رو حداقل اجرا شده باشه

-2- یکی دیگش: ینی ما تست ها رو جوری طراحی بکنیم که مطمئن باشیم هر فانکشنال که توی

دادکیومنت ها هست دست کم یکبار مورد استفاده قرار گرفته پس تستش سبک‌تر می‌شه نسبت به

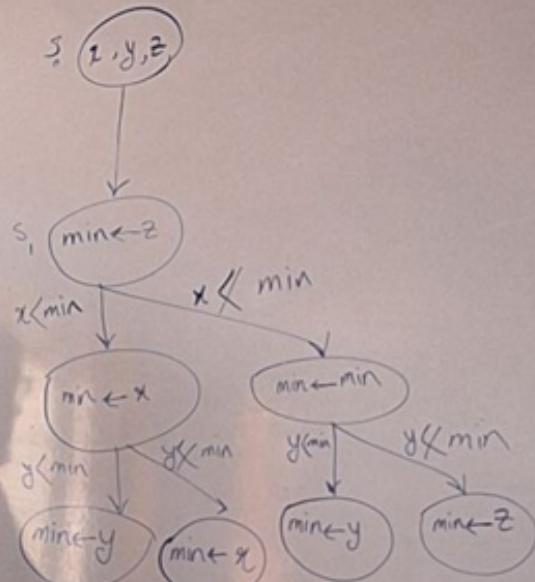
حالت قبلی که بالا گفتیم --> every state

-3 every function: هر فانکشن دست کم یکبار صدا زده شده باشه

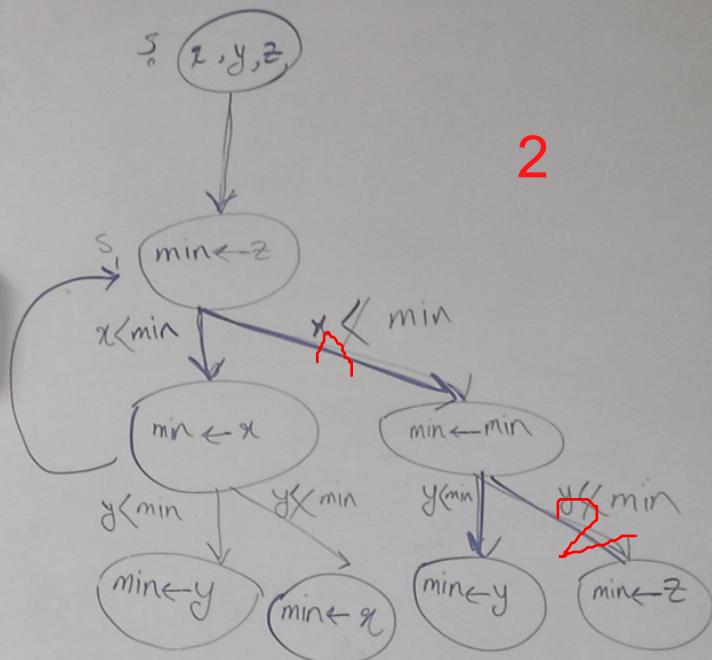
نکته: یکی از بخش های تست پلان Coverage criteria هست ینی موقع دیزاین کردن تست هامون

کی اکتفا می‌کنیم به این که دیزاینمون دیگه تموم شده

1



2



مثال: فرض میکنیم یک برنامه است که می خواهد بین 3 تا عدد مینیمم رو پیدا بکنه

Coverage criteria 1: که می تونیم تعریف بکنیم می تونیم بگیم ما تست ها را جوری طراحی میکنیم که این مجموعه تست ها مطمئن باشیم که Coverage every state رو میکنه پس به تعداد برگ تست نیاز داریم یعنی 4 نا تست

Coverage criteria 2: های دیگه ای که می تونیم تعریف کنیم به صورت زیر است:

اگر دور داشته باشیم میشه every edge یعنی دورها برآمدون مهم است زوج یال: مثل روی شکل که کشیدم یعنی اول 1 بعد 2

Advantages of Coverage Criteria

- Maximize the “bang for the buck”
- Provide traceability from software artifacts to tests
 - Source, requirements, design models, ...
- Make regression testing easier
- Gives testers a “stopping rule” ... when testing is finished
- Can be well supported with powerful tools

Coverage criteria ها معیارهای تست هستن مثلًا براساس پروژه میگیم که انتخاب میکنیم every state در سورس کد هستش ینی خط به خط سورس کد میخوایم حداقل یکبار در مجموعه تست ها اجرا بشه که این تست سنگینی هستش bang for the buck : مثلًا می تونیم اینطوری بگیم که سرمایمون زیاد بشه مثلًا یکی 1 دلارشو 5 تا کرد در حالت کلی پس: Coverage criteria باعث میشه که ما بتونیم سرمایه خروجی رو بتونیم حفظ بکنیم از این طریق که رسک تحويل نرم افزار به faulter ؟؟ کاهش پیدا بکنه software artifacts: تمام خروجی هایی که از ابتدای پروژه تا انتهای پروژه داریم به عنوان software artifacts تلقی میشن--> کدمون بارزترین software artifacts هستش حالا قبل از کد ما یک سند داریم که ریکوارمنت ها رو تو ش می نویسیم که این خودش یک نوع software artifacts هستش پس در حالت کلی تمام مراحلی که در طول ایجاد نرم افزار طی میشن و منجر به خروجی میشن این خروجی ها software artifacts هستن مثلًا هر نموداری که توی مرحله دیزاین میکشیم

coverage criteria می تونیم روی هر کدوم از این software artifacts ها اعمال بکنیم مثلًا می تونیم بگیم روی uml یا مثلًا سورس کد every state ایجاد بشه regression testing: ینی وقتی که نسخه اولیه نرم افزار رو تحويل دادیم بعد هر تغیری که از آنجا به بعد اعمال بکنیم نیاز داره به regression testing و regression testing چیه؟ مطمئن بشیم که تست های قبلی الان با این تغییر جدید پاس میشن مثلًا یک قابلیت جدید اضافه بکنیم و این باعث بشه که تمام قابلیت های قبلی از کار بیو قتن

Test Requirements and Criteria

- **Test Criterion** : A collection of rules and a process that define test requirements
 - Cover every statement
 - Cover every functional requirement
- **Test Requirements** : Specific things that must be satisfied or covered during testing
 - Each statement might be a test requirement
 - Each functional requirement might be a test requirement

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures ...

- | | |
|-------------------------------|------------------------------------------------|
| 1. Input domains
2. Graphs | 3. Logic expressions
4. Syntax descriptions |
|-------------------------------|------------------------------------------------|

-
Test Requirements : مثلاً توی سیستم بانک یک ریکوارمنت میشه لاگین کردن و Test Requirements ینی اگر یک ریکوارمنت داشته باشیم که کاربر بتونه وارد اکانت خودش بشه توی صفحه لاگین اون موقع Test Requirements میشه این که تست بکنیم که کاربر با موفقیت بتونه وارد اکانت خودش بشه از طریق صفحه لاگین پس ینی تست کردن هر کدام از ریکوارمنت ها

تست ریکوارمنت جملات مبهم و کلی هستن و ما نیاز داریم که دقیق تر بشیم مثلاً تست کنیم که کاربر به درستی توانسته انتقال پول رو انجام بده که این یک جمله کلی است چون توی این منتقل کردن پول ممکنه خیلی اتفاق بیوافته پس توی سناریوهای مختلف specification test رو باید برآش مطرح بکنیم مثلاً سناریوهایی که ممکنه پیش بیاد برآش به این صورت است که:

1- اگر همه چی اوکی بود بیاد انتقال انجام بشه

2- موجودی نداشته باشه

3- اگر شماره حساب مقصد رو اشتباه زد یه جوری متوجه این بشه

...

یکی از روش هایی که برای تست بررسی میکنیم اینه که 1- گراف سورس کد رو بکشیم 2- بیایم input domain رو بررسی بکنیم مثلاً بگیم این برنامه توی X می ره و X ما int است و به جای اینکه بیایم کل اینتیجرها رو چک بکنیم مثلاً بیایم فقط اینتیجرهای مثبت و منفی و صفر رو بررسی کنیم ینی بیایم پارتیشن بندی بکنیم 3- logic expression : ینی بیایم با استفاده از logic خواسته هامون رو مطرح بکنیم 4- syntax description مطمئن بشیم که سینتکس سورس کدمون درست است --> اینا روش های تست کردن است

Old View : Colored Boxes

- **Black-box testing** : Derive tests from external descriptions of the software, including specifications, requirements, and design
- **White-box testing** : Derive tests from the source code internals of the software, specifically including branches, individual conditions, and statements
- **Model-based testing** : Derive tests from a model of the software (such as a UML diagram)

MDTD makes these distinctions less important.

The more general question is:

from what abstraction level do we derive tests?

Black-box testing: مثل یک جعبه ای که سیاه است محتویات داخلش دیده نمیشه --> پس یک جعبه داریم که سورس کدmon رو گذاشتیم داخلش و فقط می تونیم ورودی و خروجی ها رو نگاه بکنیم و تست بکنیم پس اگر به این روش بخوایم سورس کد رو بررسی بکنیم ینی اینکه فقط ورودی بهش میدیم و خروجیشو نگاه میکنیم

White-box testing: مثل یک جعبه ای که شفاف است محتویات داخلش دیده میشه --> ینی سورس کد رو گذاشتیم داخل یک جعبه شفاف پس نه تنها ورودی و خروجی رو می تونیم ببینیم بلکه خط به خط کد رو می تونیم ببینیم مثلا فانکشن ها رو می تونیم ببینیم یا ...

نکته: **White-box testing** اینجوری نیست که فقط روی سورس کد اعمال بشه بلکه روی هر چیزی می تونه اعمال بشه مثلا روی اکتیویتی دیاگرام هم می تونه اعمال بشه **gray-box testing**: یه چیزی بینابین اون دوتا هست ینی اگر توی روش تست هم از **gray-box testing** و هم از **Black-box testing** استفاده کنیم میشه **Model-based testing**

Model-Driven Test Design (2.5)

- *Test Design* is the process of designing input values that will effectively test software
- Test design is one of **several activities** for testing software
 - Most mathematical
 - Most **technically challenging**

دیزاین کردن یک نرم افزار ینی این که بتوانیم **input value** ها را دیزاین بکنیم تا بتوانیم با اطمینان بالا بگیم که بسیاری کارهای کد رو کاور کردیم با استفاده از این **input value**ها

Types of Test Activities

- Testing can be broken up into four general types of activities
 - I. **Test Design** → I.a) Criteria-based
 - 2. **Test Automation** I.b) Human-based
 - 3. **Test Execution**
 - 4. **Test Evaluation**
- Each type of activity requires different skills, background knowledge, education and training
- No reasonable software development organization uses the same people for requirements, design, implementation, integration and configuration control

Why do test organizations still use the same people for all four test activities??

This clearly wastes resources

فرایند تست 4 مرحله داره:

سخت ترین قسمت تست مربوط به **Test Design** است و بعدش **Test Evaluation** نکته: همه این 4 تا وظیفه رو به یک نفر نباید بدم کارهای سخت رو باید بدم به افراد توأم‌مند **Test Design** دو روش داره:

every state Criteria-based: ینی این که یک معیار یا معیارها برای تست داشته باشیم مثلاً و اون متخصص‌ها بیان برای ما نرم افزار رو چک بکنن براساس این معیارها - نیاز به دانش کامپیوتری داره

Human-based: ینی اینکه رفتارهای عجیب و غریب رو بتونیم از سمت کاربر شناسایی بکنیم و سر اونا نرم افزار رو تست بکنیم پس این روش داره راجع به رفتارهای غیرمعمول کاربر داره تست رو انجام میده مثلاً توى انتقال پول کاربر حساب مبدا و مقصد رو یکی بزنه - این نوع تست نیاز به دانش کامپیوتری نداره ولی به دانش مسئله و علوم روانشناسی نیاز داره مثال توى علوم روانشناسی: مثلاً یک فروشگاه آنلاین داریم و میخوایم کاربر که سرچ میکنه محصولات مورد نظر رو بهش بده بعد هدف اینه که این نرم افزار در سطح نان فانکشنال ریکوارمنت‌ها در سطح بالایی باشه ینی کاربر به مایع ظرفشویی میگه ریکا و اینو سرچ میکنه و می بینه چیزی و اشن نمیاد و ناراضی میشه پس تستر میاد ریکا رو سرچ میکنه که ببینه تست درست جواب میده یا نه ینی براش با این کلمه بازم براش مایع ظرفشویی میاد یا نه

نکته: توى شرکت از جفتش باید استفاده بشه

1. Test Design—(a) Criteria-Based

Design test values to satisfy coverage criteria or other engineering goal

- This is the most technical job in software testing
- Requires knowledge of :
 - Discrete math
 - Programming
 - Testing
- Requires much of a traditional CS degree
- Test design is analogous to software architecture on the development side
- Using people who are not qualified to design tests is a sure way to get ineffective tests

1. Test Design—(b) Human-Based

Design test values based on domain knowledge of the program and human knowledge of testing

- This is much **harder** than it may seem to developers
- Criteria-based approaches can be blind to special situations
- Requires **knowledge** of :
 - Domain, testing, and user interfaces
- Requires almost **no traditional CS**
 - A background in the **domain** of the software is essential
 - An **empirical background** is very helpful (biology, psychology, ...)
 - A **logic background** is very helpful (law, philosophy, math, ...)

2. Test Automation

Embed test values into executable scripts

- This is slightly less technical
- Requires knowledge of programming
- Requires very little theory
- Often requires solutions to difficult problems related to observability and controllability
- Can be boring for test designers
- Who is responsible for determining and embedding the expected outputs ?
 - Test designers may not always know the expected outputs
 - Test evaluators need to get involved early to help with this

: Test Automation

اين ورودي هايى که ديزاينر طراحی کرد اينارو ميديمش به يك نيروى سطح پايين و بهش ميگيم
اينارو توی script جا بده و اتومات ران کن تا خروجي هاش اتومات بررسی بشن

3. Test Execution

Run tests on the software and record the results

- This is **easy** – and trivial if the tests are well automated
- Requires basic computer skills
 - Interns
 - Employees with no technical background
- Asking qualified test **designers** to execute tests is a sure way to convince them to look for a **development job**
- If, for example, GUI tests are not well automated, this requires a lot of **manual labor**
- Test executors have to be very **careful** and **meticulous** with bookkeeping

: Test Execution

یک بخش هایی از تست از اتومات بودن خارج میشے مثلا میخوایم یک زیر سیستمی رو تستش بکنیم و به یک سری ورودی نیاز داره که اینا هم تیم بهش میده ولی یک سنسوری هم داریم که داره سخت افزاری کار میکنه وسط کار --> اگر وسط کار تابع ما ببیاد یک مقدار سنسوری بخونی یا حواسش باشه که سنسور از یک مقداری بالاتر نره این قسمت های سخت افزاری رو باید با استفاده از محیط های شبیه سازی شده در اختیار کد قرار بدیم مثلا یک تیکه کد بنویسیم که داره شبیه سازی سنسور رو انجام میده یا اینکه بباییم واقعا یک سنسور بذاریم سخت افزاری و وصلش کنیم به سیستم و تستش بکنیم

Test execution ③ \rightarrow عصب حایی از test از اتومات بودن خارج است . و باید بصورت حد امانت \Rightarrow جوں من وان بعنوان input از آن ببرآمده دار . " \leftarrow مثلا لازم است بصورت شبیه سازی سده آن ورودی را به برنامه دار .

4. Test Evaluation

Evaluate results of testing, report to developers

- This is much **harder** than it may seem
- Requires **knowledge** of :
 - Domain
 - Testing
 - User interfaces and psychology
- Usually requires almost **no traditional CS**
 - A background in the **domain** of the software is essential
 - An **empirical background** is very helpful (biology, psychology, ...)
 - A **logic background** is very helpful (law, philosophy, math, ...)
- This is **intellectually** stimulating, rewarding, and challenging
 - But not to typical CS majors – they want to **solve problems** and **build things**

: Test Evaluation

نتایج تست رو ارزیابی میکنه

نایج را بررسی و جزئیات و فیدبک \rightarrow Feedback \rightarrow Test evaluation ④
با developer و manager



Other Activities

- **Test management** : Sets policy, organizes team, interfaces with development, chooses criteria, decides how much automation is needed, ...
- **Test maintenance** : Save tests for reuse as software evolves
 - Requires cooperation of test **designers** and automators
 - Deciding when to trim the test suite is partly policy and partly technical – and in general, **very hard** !
 - Tests should be put in **configuration control**
- **Test documentation** : All parties participate
 - Each test must document “**why**” – criterion and test requirement satisfied or a rationale for human-designed tests
 - Ensure **traceability** throughout the process
 - Keep **documentation** in the automated tests

Organizing the Team

- A mature test organization needs **only one test designer** to work with several test automators, executors and evaluators
- **Improved automation will reduce the number of test executors**
 - Theoretically to zero ... but not in practice
- Putting the **wrong people** on the **wrong tasks** leads to **inefficiency**, **low job satisfaction** and **low job performance**
 - A qualified test designer will be **bored** with other tasks and look for a job in development
 - A qualified test evaluator will **not understand** the benefits of test criteria
- Test evaluators have the **domain knowledge**, so they **must** be free to add tests that “blind” engineering processes will not think of
- The four test activities are **quite different**

**Many test teams use the same people for
ALL FOUR activities !**

توی یک تیم نقش های مختلفی وجود داره وقتی تست رو اتومات میکنیم ینی توی مرحله دیزاین تست یکسری ورودی مشخص کردیم و این ورودی ها توسط یک نفر که مسئولیت اتومات کردن رو بر عهده داره توی `script` های قابل اجرا شدن جاسازی میشه تا مطمئن بشیم که لازم نیست دستی تست رو انجام بدیم --> خوبیش: سرعتش می ره بالا - `regression test` رو راحت میکنه - خطای انسانی هم کاهش میده

Applying Test Activities

**To use our people effectively
and to test efficiently
we need a process that**

**lets test designers
raise their level of abstraction**

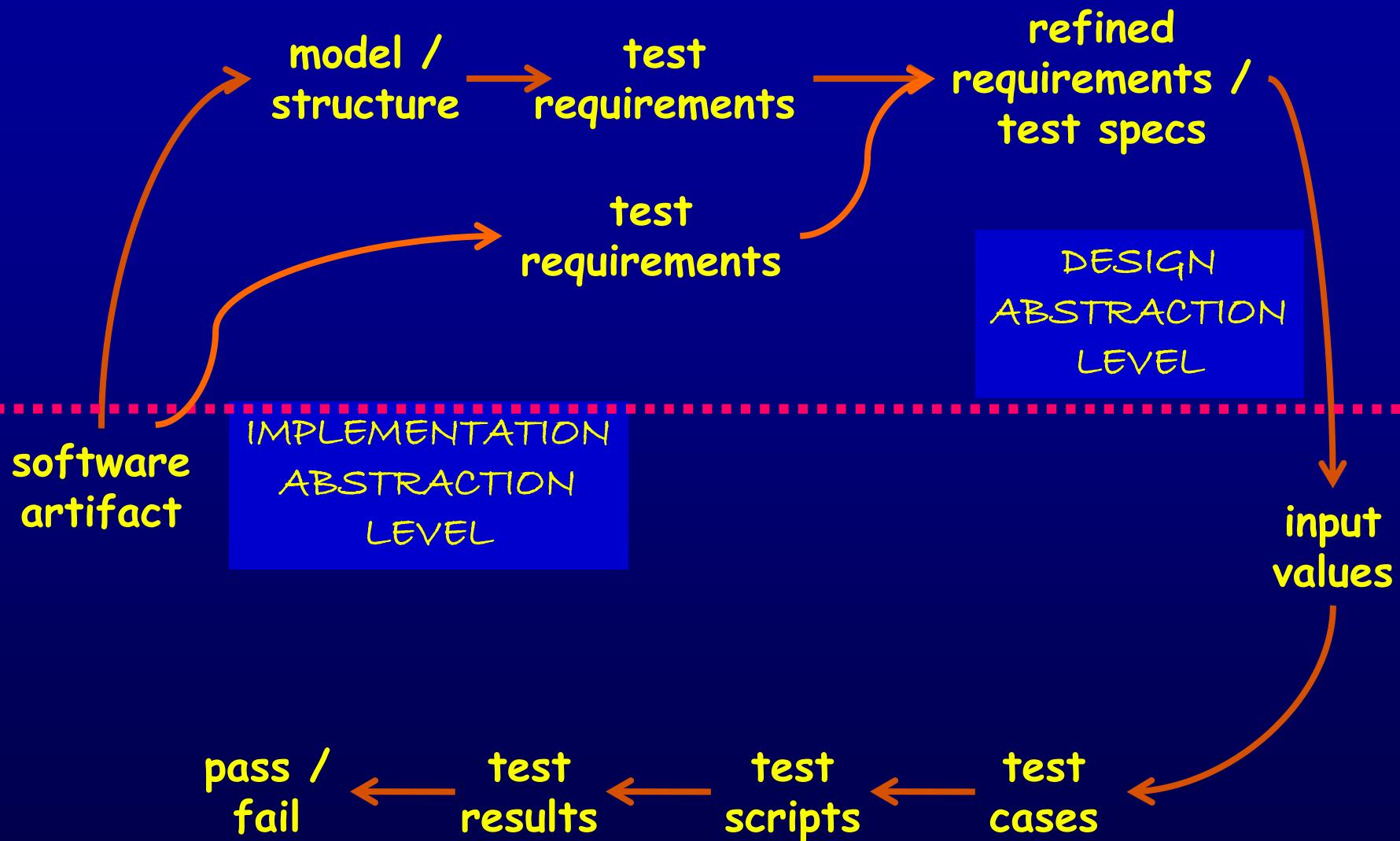
هر چقدر هم سطح abstraction بالا ببریم باعث میشه که نست بهتر و دیزاین بهتری داشته باشیم و توی مرحله بعدش تسلی پیدا میکنه

Using MDTD in Practice

- This approach lets one test designer do the math
- Then traditional **testers** and **programmers** can do their parts
 - Find values
 - Automate the tests
 - Run the tests
 - Evaluate the tests
- Just like in **traditional engineering** ... an engineer constructs models with calculus, then gives direction to carpenters, electricians, technicians, ...

Test designers become technical experts

Model-Driven Test Design



دیزاین کردن تست دو روش داره که قبل اینو گفته

روشی است که ما استفاده میکنیم برای تست کردن دو روش داره:
روشی است که Criteria-based تست رو داره هندل رو میکنه
روشی است که Human-based تست رو داره هندل میکنه
هر دوی اینا بالای خط هستن

وقتی که سورس کد رو داریم و وقتی که می خوایم Criteria-based رو انجام بدیم اول از همه لازمه که توی روش model driven یک مدلی استخراج بکنیم از خودمون مثلا یک گراف استخراج میکنیم و بعد بیایم اون گراف رو نگاه بکنیم و تست ریکوارمنت ها رو براساس داکیومنت های موجود و گراف استخراج بکنیم وقتی که به تست ریکوارمنت و تست specification روی یک گراف نگاه میکنیم --> تست ریکوارمنت ها توی معیارهای پوشش که every edge هست ینی هر یال رو بخوایم پوشش

بدیم تست ریکوارمنت نظیر هر یال هستش ولی تست specification نظیر هر مسیر هست توی گراف نکته: همه این مسیرها رو توی human-based testing به دست اورده میشه مثلا یکی از این مسیرها این است که کاربر برای انتقال مبدا و مقصد رو یکی بزنه پس این مسیر رو توی human-based testing به دست اورده میشه

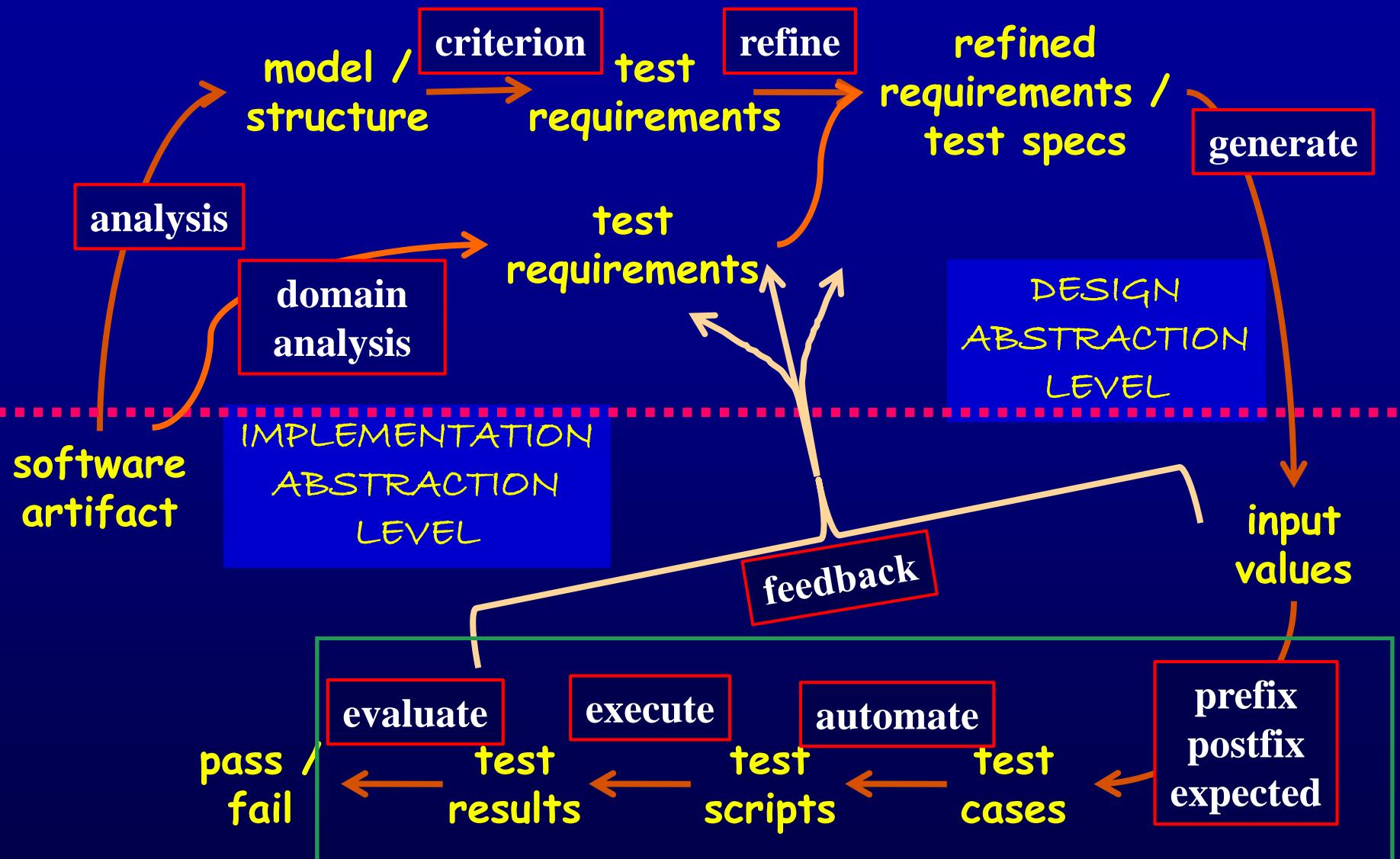
توی مسیر human-based testing دیگه هیچ مدلی استخراج نمیشه ینی اون کسی که داره براساس علوم روانشناسی و مسائل مربوط به پروژه تست رو انجام میده دیگه مدلی نداره و مستقیماً تست ریکوارمنت ها رو اضافه می کنیم توی داکیومنت و به دنبالش تست specification و اینجا فاز دیزاین طراحی میشه و خروجی فاز دیزاین input value ها میشه و از این به بعدش می ره توی فاز اتومیت کردن

test cases: یک زوج ورودی و خروجی داریم ینی یک تست است مثلا برنامه داره اعداد اینتیجر رو می گیره و اگر مثلا ورودی $x=2$ باشه به ازاش خروجی میشه 10 که این می شه یک test cases 10 پس این input value ها تبدیل به test cases میشن که البته تست کیس علاوه بر input value یک سری چیزای دیگه هم احتیاج داره (توی فصل بعد گفته میشه)

test scripts: که مربوط به اتومات سازی هست ینی بیایم این تست ها رو داخل تیکه کدهای اتومات جا بدیم و اجرашون بکنیم تا نتایج به دست بیاد ینی test results و بعد از این یا پاس میشه یا نمیشه و این فیدبک داده میشه به دولوپرها

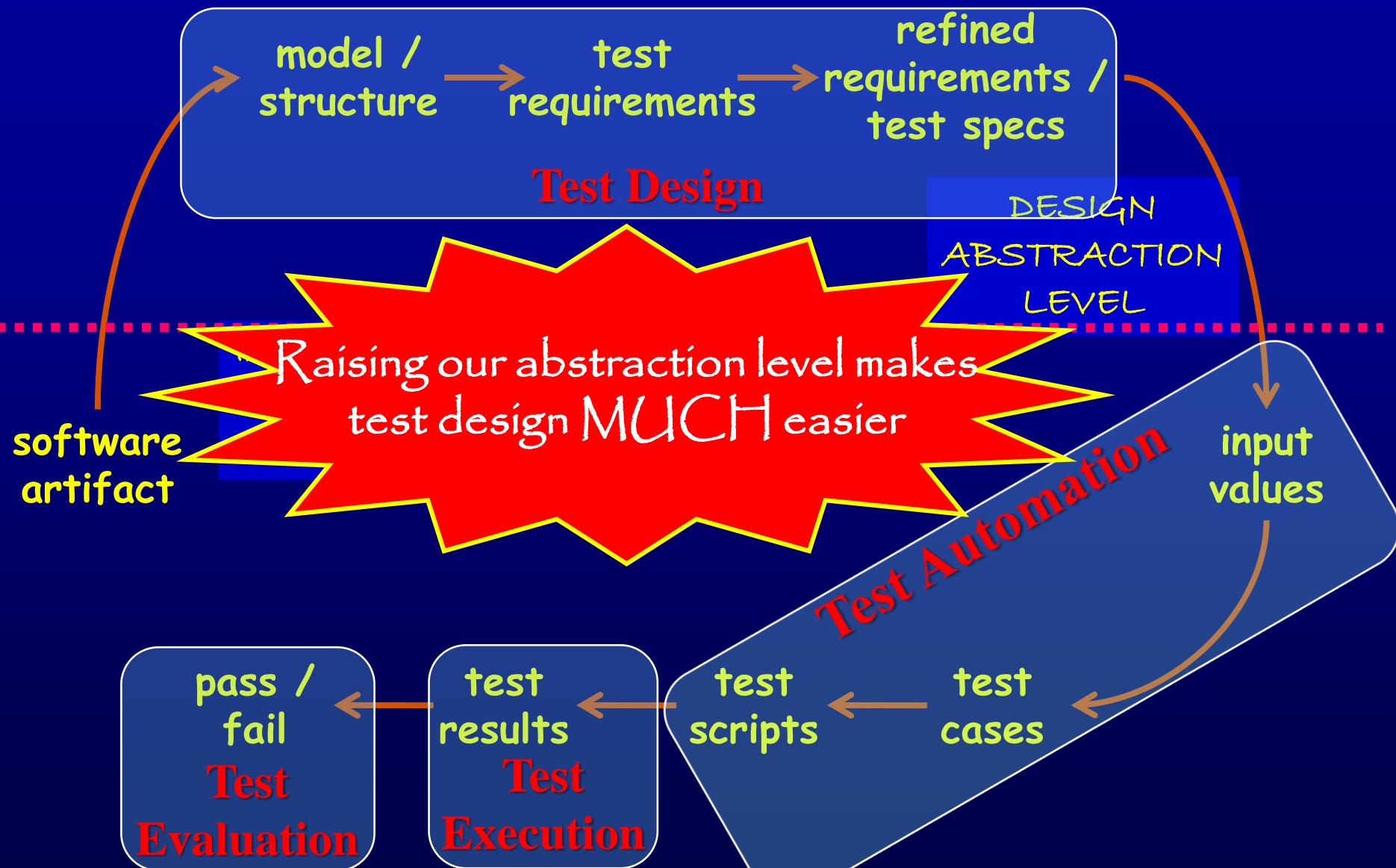
نکته: توی این روش لزوماً کد رو تست نمیکنیم کلا هر software artifacts رو می تونیم تست بکنیم

Model-Driven Test Design – Steps



رنگ سبز: توی این مراحل هم مثل prefix می تونه فیدبک داده بشه به کسایی که دارن تست ریکوارمنت استخراج می کنن

Model-Driven Test Design–Activities



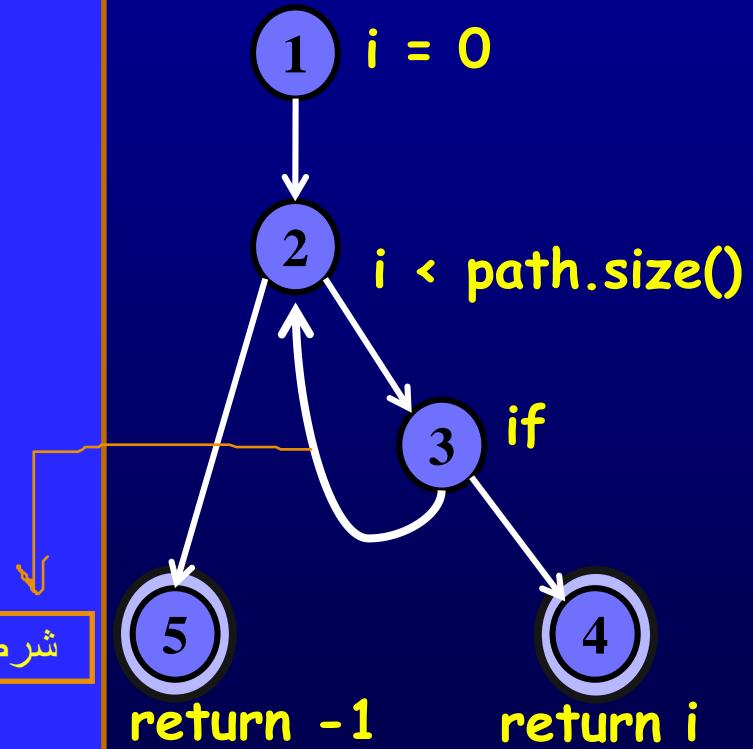
Small Illustrative Example

Software Artifact : Java Method

```
/**  
 * Return index of node n at the  
 * first position it appears,  
 * -1 if it is not present  
 */  
public int indexOf (Node n)  
{  
    for (int i=0; i < path.size(); i++)  
        if (path.get(i).equals(n))  
            return i;  
    return -1;  
}
```

شرط if برقرار نیست و اسه همین برگشتنه

Control Flow Graph



اين تيکه کد داره 1- رو مиде اگر او نود حضور نداشته باشه
به اين گراف Control Flow Graph می گيم يني بگيم از هر استتيت هاي
بعدی ميريم

اولين قدم توی کد اينه که $i=0$ در نظر گرفته بشه : استيت 1

استيت 2: شرط رو چك ميكنيم

نکته: توی Control Flow Graph استيت هاي نهايی رو با دونا خط باید نشون بدیم و استيت
اولیه رو با نقطه چین باید بکشیم که اینجا حالا نشون نداده ولی ما باید نقطه چین بذاریم

نکته: برای کال کردن یک فانکشن این که یک استيت جدأگانه در نظر بگیریم یا نگیریم بستگی داره
به طراحمون

برای `++i` اينو فرض کرده که توی استيت 2 یا استيت 3 داره انجام ميشه و برash مهم نبوده و اسه
همين برash یک استيت جدأگانه در نظر نگرفته ولی اگه برای ما مهم بود می توئیم اينو یک استيت
جدأگانه هم در نظر بگيريم

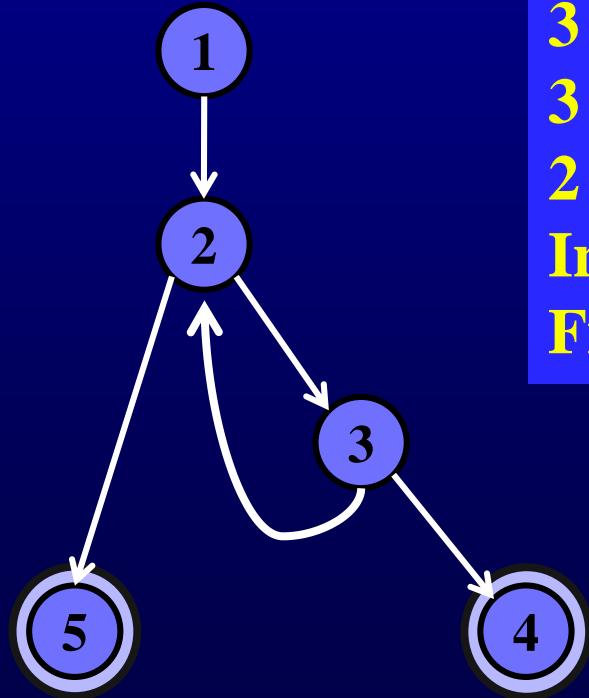
نکته: از استيت 1 به 2 `--> pc` ما تغيير مي肯ه

Example (2)

Support tool for graph coverage

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

Graph
Abstract version



Edges

1 2

2 3

3 2

3 4

2 5

Initial Node: 1

Final Nodes: 4, 5

6 requirements for
Edge-Pair Coverage

1. [1, 2, 3]

2. [1, 2, 5]

3. [2, 3, 4]

4. [2, 3, 2]

5. [3, 2, 3]

6. [3, 2, 5]

++

1
2

Test Paths

[1, 2, 5]

[1, 2, 3, 2, 5]

[1, 2, 3, 2, 3, 4]

Find values ...

معیارهای پوشش این گراف:

بنی هر یال حداقل یکبار دیده بشه: every edge

بنی اینجا برامون مهمه که سه تا استیت پشت سر هم حتما دیده بشن every edge-pair معادل همون تست specification میشه --> حداقل تعداد مسیرهای ممکن که باهاش می تونیم تمام این زوج یال ها رو پوشش بدیم 3 تا است (نکته: چون زوج یاله داخل اون مسیرها باید سه تا نود رو پشت سر هم ببینیم بنی سه تا سه در نظر بگیریم)

نکته ++: اگر معیار پوشش every edge باشه دومی می تونه اولی رو کاور بکنه و دیگه نیاز به نوشتن اولی نیست ولی اگر زوج یال باشه دومی دیگه نمیتونه اولی رو کارو کنه

نکته: معیار پوشش و کلا کارایی که توی تست میکنیم هم باید داکیومنت بکنیم کجا داکیومنت میکنیم؟
داخل تست پلن

Types of Activities in the Book

Most of this book is about test design

Other activities are well covered elsewhere

بیشتر این کتاب در مورد طراحی تست است سایر فعالیت ها در جاهای دیگر به خوبی پوشش داده شده است