



Software Testing

Course Overview

Dr. Elham Mahmoudzadeh
Isfahan University of Technology
Mahmoudzadeh@iut.ac.ir

2023

*As the software industry moves into the second decade of the 21st century, **software quality** is increasingly becoming essential to all businesses and knowledge of **software testing** is becoming necessary for all software engineers.*

با ورود صنعت نرم افزار به دهه دوم قرن بیست و یکم، کیفیت نرم افزار به طور فزاینده ای برای همه مشاغل ضروری می شود و دانش تست نرم افزار برای همه مهندسان نرم افزار ضروری می شود.

Course Overview



Grading policy

- ❖ 25% on individual homework (individually solve homework assignments)
- ❖ 25% on midterm exam.
- ❖ 50% on Final exam.
- ❖ +15% on project(?, in groups two or three students)
- Late policy: no credit for late work.

Course Communication

- Email: Mahmoudzadeh@iut.ac.ir
- Skype: elham.Mahmoudzadeh
- Yekta.iut.ac.ir/Software Testing/Messages

References

- 1- P. Ammann, J. Offutt, “**Introduction to Software Testing**”, Cambridge University Press, 2nd Edition, 2017.
 - Available at: <https://cs.gmu.edu/~offutt/softwaretest/>
- 2- Lecture notes of Dr. P. Muller, ETH Zurich: “Software Architecture and Engineering Testing”, 2018.
- 3- Lecture notes of Dr. Darko Marinov, University of Illinois at Urbana-Champaign: “Software Testing”, 2016.
- 4-Lecture notes of Dr. Debra Richardson, UC Irvine: “Analysis and Testing are Creative”, 2002.

Course goals

- Analysis of **structure** and **behavior** of the software.
- How to form an **abstract model** of the software artifact.
- Teach software engineers **how to test**.
- Design **suitable test cases**.
- Become a **good tester**.

Advice

- ❖ Don't get behind: first week especially is very fast!
- ❖ Attend lectures: material is not all in textbook.
- ❖ Do the readings on time.

Software is a Skin that Surrounds Our Civilization



Quote due to Dr. Mark Harman

Testing in the 21st Century

- Software defines behavior
 - network routers, finance, switching networks, other infrastructure
- Today's software market :
 - is much bigger
 - is more competitive
 - has more users
- Embedded Control Applications
 - airplanes, air traffic control
 - spaceships
 - watches
 - ovens
 - remote controllers
 - PDAs
 - memory seats
 - DVD players
 - garage door openers
 - cell phones
- Agile processes put increased pressure on testers
 - Programmers must unit test – with no training or education!
 - Tests are key to functional requirements – but who builds those tests ?

Industry is going through a revolution in what testing means to the success of software products

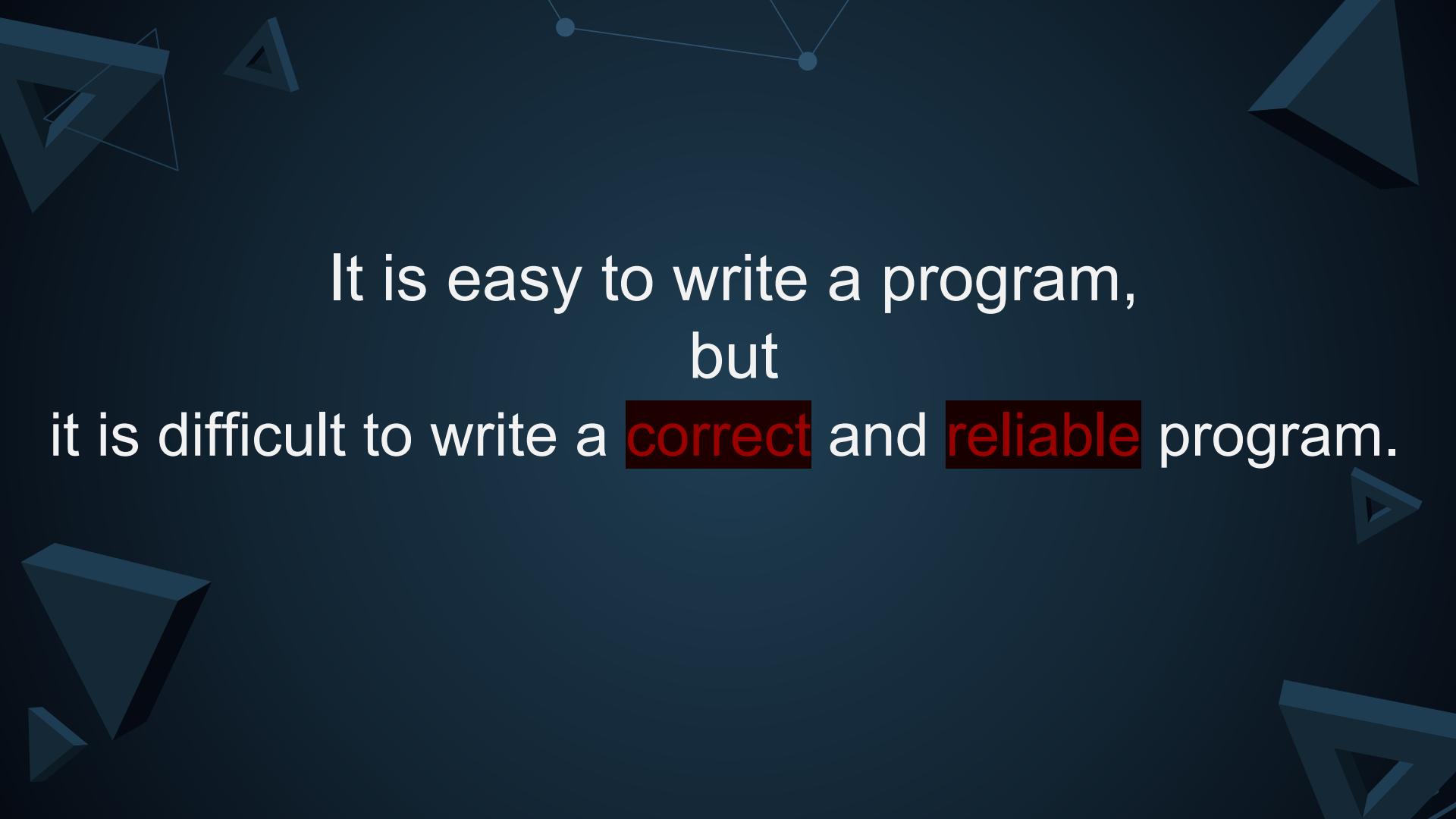
Testing in the 21st Century

- More **safety** critical, **real-time** software
- Embedded software is ubiquitous ... check your pockets
- Enterprise applications means bigger programs, more users
- Paradoxically, free software increases our expectations !
- Security is now all about **software faults**
 - Secure software is **reliable** software
- The **web** offers a new deployment platform
 - Very **competitive** and very **available** to more users
 - Web apps are **distributed**
 - Web apps must be **highly reliable**

بیشتر نرم افزار تعییه شده در همه جا وجود دارد ... جیب های خود را بررسی کنید. برنامه های کاربردی سازمانی به معنای برنامه های بزرگتر، کاربران بیشتر است به طرز متناقضی، نرم افزار ایگان انتظارات ما را افزایش می دهد!

وب یک پلت فرم استقرار جدید ارائه می دهد
بسیار رقابتی و بسیار در دسترس برای کاربران بیشتر
- برنامه های وب توزیع می شوند
- برنامه های وب باید بسیار قابل اعتماد باشند

Industry desperately needs our inventions !



It is easy to write a program,
but

it is difficult to write a **correct** and **reliable** program.

Some Costly Failures



NASA Mars space missions

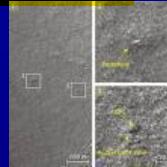
- Priority inversion (2004)
- Different metric systems (1999)

BMW airbag problems (1999)

- Recall of 15,000+ cars
- Ariane 5 crash (1996)
 - Uncaught exception of numerical overflow

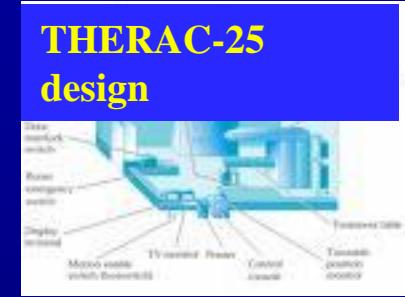
Spectacular Software Failures

- NASA's Mars lander: September 1999, crashed due to a units integration fault



Mars Polar
Lander
crash
site?

- THERAC-25 radiation machine : Poor testing of safety-critical software can cost *lives* : 3 patients were killed
- Ariane 5 explosion : Millions of \$\$



Intel's Pentium FDIV fault(an early alarm of the need for better testing) : Public relations nightmare



We need our software to be
dependable

Testing is one way to assess dependability

ما باید نرم افزارمان قابل اعتماد باشد تست یکی از راه های ارزیابی
قابلیت اطمینان است

Ariane 5:
exception-handling bug : forced self destruct on maiden flight (64-bit to 16-bit conversion: about 370 million \$ lost)

Northeast Blackout of 2003

508 generating units and 256 power plants shut down

Affected 10 million people in Ontario, Canada

Affected 40 million people in 8 US states

Financial losses of \$6 Billion USD

The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system



Costly Software Failures

- NIST report, “The Economic Impacts of Inadequate Infrastructure for Software Testing” (2002)
 - Inadequate software testing costs the US alone between \$22 and \$59 billion annually
 - Better approaches could cut this amount in half
- Huge losses due to web application failures
 - Financial services : \$6.5 million per hour (just in USA!)
 - Credit card sales applications : \$2.4 million per hour (in USA)
- In Dec 2006, amazon.com’s BOGO offer turned into a double discount
- 2007 : Symantec says that most security vulnerabilities are due to faulty software

اثرات اقتصادی
زیرساخت ناکافی
برای تست نرم افزار

زیان پولی در سراسر جهان به دلیل نرم افزار ضعیف خیره کننده است

World-wide monetary loss due to poor software is staggering

*“Even when you think you’ve tested **everything**
that you can **possibly** imagine, **you’re wrong**”*

Glenn E. Reeves
(Pathfinder’s Software Team Leader)

Goal of Testing(I)

- Find faults (“Debug” Testing): a test is successful if the program fails
- Provide confidence (Acceptance Testing)
 - of reliability
 - of (probable) correctness
 - of detection of particular faults

هدف از تست

عیوب را پیدا کنید (تست «اشکالزدایی»): در صورت شکست برنامه، آزمایش موفقیتآمیز است. ارائه اطمینان (تست پذیرش) از قابلیت اطمینان از صحبت (احتمالی) تشخیص عیوب خاص

خطا عبارت است از انحراف رفتار مشاهده شده از رفتار مورد نیاز (مطلوب).

Goal of Testing(II)

- An **error** is a **deviation** of the observed behavior from the **required (desired) behavior**
 - Functional requirements (e.g., user- acceptance testing)
 - Nonfunctional requirements (e.g., performance testing)

تست فرآیند اجرای یک برنامه با هدف یافتن خطای است.

- Testing is a process of **executing a program** with the **intent** of **finding an error**

الزمات عملکردی (به عنوان مثال، آزمایش پذیرش کاربر)

- الزمات غیر کاربردی (به عنوان مثال، تست عملکرد)

- A **successful test** is a test that **finds errors**

Why does Software **contains** Bugs?

- Our ability to predict the behavior of our implementations is limited
 - Software is extremely complex
 - No developer can understand the whole system
- We make mistakes
 - Unclear requirements, miscommunication
 - Wrong assumptions (e.g., behavior of operating system)
 - Design errors (e.g., capacity of data structure too small)
 - Coding errors (e.g., wrong loop condition)

چرا نرم افزار حاوی اشکال است؟
 • توانایی ما برای پیش بینی رفتار پیاده سازی هایمان محدود است
 - نرم افزار بسیار پیچیده است
 - هیچ توسعه دهنده ای نمی تواند کل سیستم را درک کند
 • ما اشتباه می کنیم
 - الزامات نامشخص، عدم ارتباط

- مفروضات اشتباه (به عنوان مثال، رفتار سیستم عامل)
 - خطاها طراحی (به عنوان مثال، ظرفیت ساختار داده بسیار کم)
 - خطاها کدنویسی (به عنوان مثال، وضعیت حلقه اشتباه)

محدودیت های تست

"تست برنامه را می توان برای نشان دادن وجود اشکالات استفاده کرد، اما هرگز برای نشان دادن عدم وجود آنها نمیتوان استفاده کرد."

Limitations of Testing

"Program testing
can be used to **show the presence of bugs**,
but never to **show their absence!**"

[E. W. Dijkstra]

- It is **impossible** to **completely** test any nontrivial module or any system
 - **Theoretical** limitations: **termination**
 - **Practical** limitations: prohibitive in **time and cost**

ازمایش کامل هر مازول غیر ضروری یا هر سیستمی غیرممکن است

- محدودیت های نظری: خانمه

- محدودیت های عملی: از نظر زمان و هزینه بازدارنده است

Increasing Software Reliability

- Fault Avoidance

- Detect faults statically without executing the program
- Includes development methodologies, reviews, and program verification

- Fault Detection

- Detect faults by executing the program
- Includes testing

- Fault Tolerance

- Recover from faults at runtime (e.g., transactions)
- Includes adding redundancy (e.g., n-version programming)

جتناب از خطای

- تشخیص خطاها به صورت ایستا بدون اجرای برنامه
- شامل روش‌های توسعه، بررسیها و تأیید برنامه است

• تشخیص خطای

- تشخیص عیوب با اجرای برنامه
- شامل تست است

• تحمل خطای

- بازیابی از خطاها در زمان اجرا (به عنوان مثال، تراکنش‌ها)
- شامل اضافه کردن افزونگی (به عنوان مثال، برنامه نویسی نسخه n)

What we will talk about next...

- ❖ Why Software Testing?

Introduction to Software Testing *(2nd edition)* **Chapter 1**

Why Do We Test Software?

Paul Ammann & Jeff Offutt

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

*Updated September 2015
First version, 28 August 2011*

Introduction

- To teach software engineers how to test.

- Is useful for
 - a programmer who needs to unit test her own software,
 - a full-time tester who works mostly from requirements at the user level,
 - a manager in charge of testing or development,
 - any position in between.

Software Faults, Errors & Failures

- Software Fault : A static defect in the software
- Software Error : An incorrect internal state that is the manifestation of some fault
- Software Failure : External, incorrect behavior with respect to the requirements or other description of the expected behavior

خطای نرم افزار: یک نقص استاتیک در نرم افزار
خطای نرم افزار: یک حالت داخلی نادرست که مظہر نقص است
خرابی نرم افزار: رفتار خارجی و نادرست با توجه به الزامات یا سایر توضیحات رفتار مورد انتظار

Faults in software are equivalent to design mistakes in hardware.

Software does not degrade.

نقص در نرم افزار معادل اشتباهات طراحی در سخت افزار است.
نرم افزار تخریب نمی شود.

Fault and Failure Example

- A patient gives a doctor a list of symptoms

- Failures

یک بیمار لیستی از علائم را به پزشک می دهد

ناراحتی ، درد

- The doctor tries to diagnose the root cause, the ailment

- Fault

پزشک سعی می کند علت اصلی، بیماری را تشخیص دهد

- The doctor may look for anomalous internal conditions (high blood pressure, irregular heartbeat, bacteria in the blood stream)

- Errors

پزشک ممکن است به دنبال شرایط داخلی غیر عادی باشد

Most medical problems result from external attacks (bacteria, viruses) or physical degradation as we age.

Software faults were there at the beginning and do not “appear” when a part wears out.

ایرادات نرم افزاری در ابتدا وجود داشت و زمانی که یک قطعه فرسوده می شود، ظاهر نمی شود

A Concrete Example

Fault: Should start searching at 0, not 1

```
public static int numZero (int [ ] arr)
{ // Effects: If arr is null throw NullPointerException
// else return the number of occurrences of 0 in arr
int count = 0;
for (int i = 1; i < arr.length; i++)
{
    if (arr [ i ] == 0)
    {
        count++;
    }
}
return count;
```

Error: i is 1, not 0, on the first iteration
Failure: none

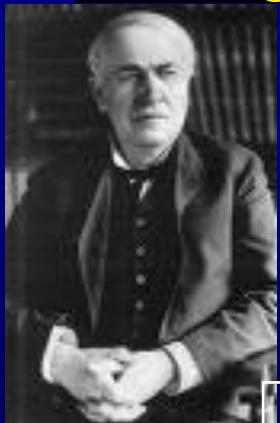
Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

The Term Bug

- *Bug* is used informally
- Sometimes **speakers mean fault, sometimes error, sometimes failure**
... often the speaker doesn't know what it means !
- This class will try to use words that have **precise, defined, and unambiguous meanings**



“It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and [it is] then that 'Bugs'—as such **little faults and difficulties** are called—show themselves and months of intense watching, study and labor are requisite. . .” – Thomas Edison

“an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of **error**. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.” – Ada, Countess Lovelace
(notes on Babbage's Analytical Engine)

What Does This Mean?

**Software testing is getting more
important**

**What are we trying to do when we test ?
What are our goals ?**

Validation & Verification (IEEE)

حوزه‌ی کیفی

- **Validation** : The process of **evaluating software** at the end of software development to **ensure compliance** with intended usage

فرآیند ارزیابی نرم افزار با درنظر گرفتن اینکه اون نرم افزار داره در حوزه کاربردی خودش درست کار میکنه یعنی اینجا دانش دامنه‌ی نرم افزار را میخاهیم.
چقدر منطبق است با نیازمندی هایی که اندیوژرها کفتند

- Depends on **domain knowledge**; that is, knowledge of the application for which the software is written.

سناریوهای طبق خاسته
ی مشتری باشه

اعتبار سنجی: فرآیند ارزیابی نرم افزار در پایان توسعه نرم افزار برای اطمینان از انطباق با استفاده مورد نظر بستگی به دانش حوزه دارد. یعنی دانش برنامه‌ای که نرم افزار برای آن نوشته شده است.

- **Verification** : The process of **determining** whether the products of a given phase of the software development process fulfill the requirements established during the previous phase.

داکیومنت‌ها باید یک سیستم را تعریف کنند.
بیشتر یک رویکرد فنی و تکنیکال است

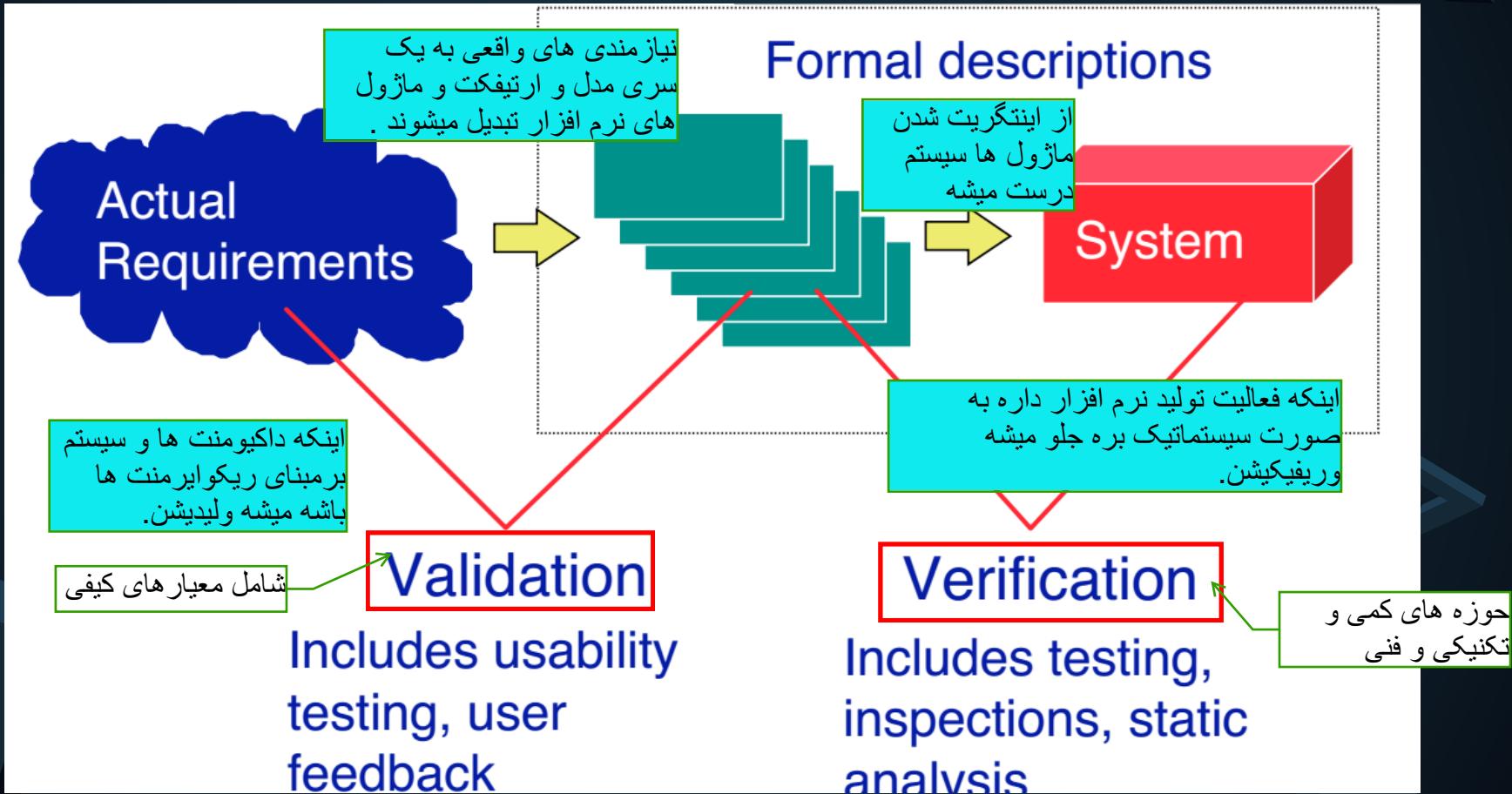
آیا نسبت به فرآیند های قبلی نرم افزار داره خوب پیش میره؟ مثلاً اینکه میگفتیم دیاگرام هامون باید باهم consistent باشند و نباید کانفلیکتی بینشون باشند.

- Is a more **technical activity** that uses **knowledge about the individual software artifacts, requirements, and specifications**.

افراد باید دانش نرم
افزاری داشته باشند.

تایید: فرآیند تعیین اینکه آیا محصولات یک مرحله معین از فرآیند توسعه نرم افزار الزامات تعیین شده در مرحله قبل را برآورده می‌کنند یا خیر.
یک فعالیت فنی تر است که از دانش در مورد مصنوعات، الزامات و مشخصات نرم افزار فردی استفاده می‌کند.

Validation and Verification



Testing Goals Based on Test Process Maturity

سطح مختلفی از تست.
تست در سطح های مختلف
میتواند باشد.

- Level 0 : There's no difference between testing and debugging
- Level 1 : The purpose of testing is to show correctness
- Level 2 : The purpose of testing is to show that the software doesn't work
- Level 3 : The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4 : Testing is a mental discipline that helps all IT professionals develop higher quality software

نمایش میزان صحت
و درستی

نمایش فیلورها و جاهایی که درست کار
نمیکنند

کامل ترین سطح تست
که به کیفیت نرم افزار
از ابعاد مختلف نگاه
نمیکنند

اهداف آزمون بر اساس بلوغ فرآیند آزمون

سطح 0: هیچ تفاوتی بین تست و اشکال زدایی وجود ندارد

سطح 1: هدف از تست نشان دادن درستی است

سطح 2: هدف از آزمایش نشان دادن این است که نرم افزار کار نمی کند

سطح 3: هدف از آزمایش اثبات چیز خاصی نیست، بلکه کاهش خطر استفاده از نرم افزار است.

سطح 4: تست یک رشته ذهنی است که به همه متخصصان فناوری اطلاعات کمک می کند نرم افزار با کیفیت بالاتر توسعه دهد.

Level 0 Thinking

- Testing is the same as debugging

تست همان دیباگ است یعنی خیلی فرقی نداره که رفتار برنامه درست است یا نه بلکه میخاهد برنامه دیباگ شه تا خطاهاش مشخص شه.
این روش هیچ کمکی نمیکنه که بفهمیم نرم افزار Reliable هست یا نه؟

- Does not distinguish between incorrect behavior and mistakes in the program

- Does not help develop software that is reliable or safe

سطح 0 تفکر

تست همان اشکال زدایی است

بین رفتار نادرست و اشتباه در برنامه تمایز قائل نمی شود

به توسعه نرم افزار قابل اعتماد یا ایمن کمک نمی کند

این چیزی است که ما در مقطع کارشناسی رشته های CS آموزش می دهیم

This is what we teach undergraduate CS majors

Level 1 Thinking

- Purpose is to **show correctness**
- Correctness is **impossible to achieve**
- What do we know if **no failures?**
 - Good software or bad tests?
- Test engineers have **no:**
 - Strict goal
 - Real stopping rule
 - Formal test technique
 - Test managers are **powerless**

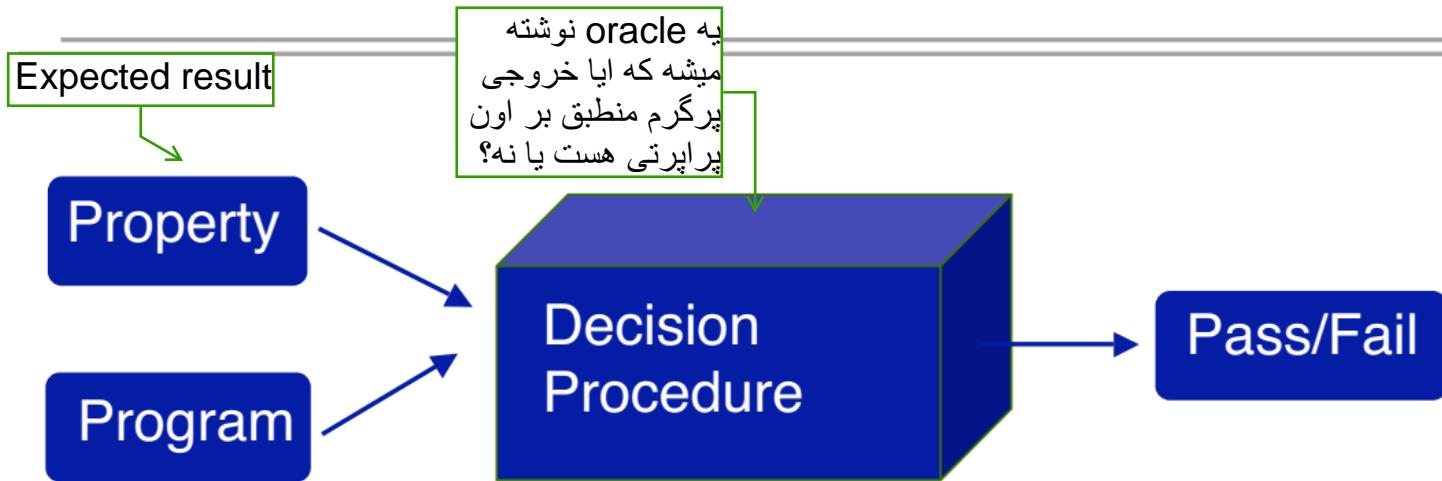
This is what hardware engineers often expect

مشخص کردن صحت و درستی برنامه.
نشون بدیم برنامه داره درست کار میکنه.
اینکه بگیم برنامه ما ۱۰۰ درصد داره
صحیح کار میکنه غیر ممکن است.
جون فضای حالت مختلف برای تست انقدر
بزرگ میشه که غیرممکن است به ازای
همه ی ترکیب ها تست کنیم نرم افزار را.
همه ی جایگشت ها و ترکیب های
متغیرهای ورودی.

سطح 1 تفکر
هدف نشان دادن درستی است
رسیدن به صحت غیرممکن است
اگر شکستی نداشته باشیم چه می دانیم?
– نرم افزار خوب یا تست بد؟
مهندسين آزمون هیچ:
– هدف دقیق
– قانون توقف واقعی
– تکنیک تست رسمی
– مدیران آزمون ناتوان هستند
این همان چیزی است که مهندسان سخت افزار اغلب انتظار دارند

شما هرگز نمیتوانید به انچه که میخواهید دست پیدا کنید (درستی کامل) چون **undecidable** است.

You can't ~~ever~~ always get what you want



Correctness properties are **undecidable**
the **halting problem** can be embedded in almost
every property of interest

Level 2 Thinking

- Purpose is to **show failures**

نمایش فیلورها.
در حدی که میتوانیم تست کیس هارا طوری طراحی کنیم که فیلورها قابل مشاهده باشند. این تا حدی قابل انجام است ولی یک فعالیت منفی است چون داریم ایرادات دولوپرها را پیدا میکنیم.

- Looking for **failures** is a **negative activity**

- Puts **testers and developers** into an **adversarial relationship**

- What if there are **no failures?**

هدف نشان دادن شکست هاست
جستجوی شکست یک فعالیت منفی است
تسترها و توسعه دهنگان را در یک رابطه خصم‌مانه قرار می‌دهد
اگر فیلوری وجود نداشته باشد چه؟

شاید ما درست تست نکردیم شاید همه مواردی که ممکن بوده اتفاق بیفته را در نظر نگرفتیم.
شاید هم نرم افزار واقعاً درست بوده.



این نوع از تست را
برخی از شرکت ها
انجام میدن.

Level 3 Thinking

ریسک کارنگردن به سری از ویژگی ها
خیلی زیاده میتواند اعتباری باشه یا
ریسک سرویسی داشته باشه یا میتواند
ریسک غیرقابل جبران داشته باشه.

- Testing can only show the presence of failures
- Whenever we use software, we incur some risk
- Risk may be small and consequences unimportant
- Risk may be great and consequences catastrophic
- Testers and developers cooperate to reduce risk

اهمیت ثبت نام اصلی نسبت به ثبت نام مقدماتی خیلی بیشتر است و اگه توی اصلی خطای رخ داد اثر و عواقب بدتری داره. یا مثلا خطأ در بحث لاجین کردن در سیستم خیلی مهم تره تا خطأ توی تماس با ما. ما وقتی داریم نرم افزار را تحلیل میکنیم بعضی از یوزکیس ها و مسیرها حتما باید درست کار کنند.

تست به منظور کاهش ریسک انجام میشه یعنی ریسک ها را در نرم افزار شناسایی میکنیم مسیرهای حیاتی و کریتیکال را درمیاریم و میگیم عواقب درست کار نکردن این یوزکیس ها چقدر است؟ چه میزان اهمیت دارن؟ به همان میزان فعالیت تست را انجام بدهیم تا ریسک کار با نرم افزار کم بشه.

سطح 3 تفکر آزمایش فقط می تواند وجود نقص را نشان دهد هر زمان که از نرم افزار استفاده می کنیم، ریسک هایی را متحمل می شویم خطر ممکن است کوچک و عواقب آن بی اهمیت باشد خطر ممکن است بزرگ و عواقب فاجعه بار باشد از مایش کنندگان و توسعه دهندهای برای کاهش ریسک همکاری می کنند این چند شرکت نرم افزاری "روشنفکر" را توصیف می کند

This describes a few “enlightened” software companies

له فیلور ممکن است کوچک باشد ولی اثار بزرگی داشته باشد و ممکن است بزرگ باشد و اثار کوچکی داشته باشد. یا موارد و حالات دیگه.

کامل ترین مرحله‌ی ازمنون که مستقیماً به افزایش کیفیت نرم افزار کمک می‌کننده که مهندسین تست میشن لیدرهای پروژه. که هم ترین مسئولیت را دارن که کیفیت نرم افزار را ارزیابی کنند و هم ارتقا بدند.

Level 4 Thinking

A mental discipline that increases quality

- Testing is only one way to increase quality
- Test engineers can become technical leaders of the project
- Primary responsibility to measure and improve software quality
- Their expertise should help the developers

تنها راه افزایش کیفیت نرم افزار استفاده از تست است.

This is the way “traditional” engineering works

مسئولیت اصلی اندازه‌گیری و بهبود کیفیت نرم افزار تخصص آنها باید به توسعه دهندگان کمک کنداش روشی است که مهندسی "سترنی" کار می‌کند

سطح 4 تفکر

یک نظم ذهنی که کیفیت را افزایش می‌دهد
تست تنها یک راه برای افزایش کیفیت است
مهندسان آزمون می‌توانند رهبران فنی پروژه شوند

Software Qualities

چقدر نرم افزار داره درست کار میکنه؟
چقدر به جواب هاش میتوانیم اطمینان
داشته باشیم؟
نرم افزار قابلیت هندل
کردن عملکردهای
مختلف را داشته باشه.
در حوزه ها و دامین
های مختلف کار کنه

External properties
(that can be verified)
timeliness
interoperability

Dependability
properties

correctness robustness
safety reliability

Process oriented
(internal) properties
maintainability reusability
modularity

External properties
(that can be validated)

user-friendliness
usability

چقدر کاربرها راحت
هستند در کار با
سیستم؟ چقدر راحت
پیدا میکنند سرویس
ها را؟ گم نشن توی
مسیرهای مختلف

ادم ها و یوز ها چقدر
راحت میتوانند با نرم
افزار کار کنند؟

در تست بیشتر به
خصوصیات
Dependability
میپردازیم.

چقدر میتوانه خودش را از
خطا ها ریکاور کنه؟

Where Are You?

دیباگ

یافتن درستی

یافتن خطاهای

Are you at level 0, 1, or 2 ?

Is your organization at work at level
0, 1, or 2 ?
Or 3?

ما امیدواریم که به شما آموزش دهیم که در محل کار خود "عامل تغییر" شوید ... حامیان تفکر سطح 4

کاهش ریسک ناشی
از خطاهای

We hope to teach you to become
“change agents” in your workplace ...

Advocates for level 4 thinking

افزایش کیفیت نرم
افزار و تضمین
کیفیت نرم افزار

Tactical Goals : Why Each Test ?

If you don't know why you're conducting each test, it won't be very helpful

تست اگر مستند نشه پیچیده میشه و سردرگم میشیم. باید بدانیم چرا یه تستی داریم انجام میدیم باید بدانیم هر تست را داریم با چه نیازمندی ای هست.

- Written test objectives and requirements must be documented
- What are your planned coverage levels?
- How much testing is enough?
- Common objective – spend the budget ... test until the ship-date ...
 - Sometimes called the “date criterion”

اهداف و الزامات آزمون کتبی باید مستند باشد سطوح پوشش برنامه ریزی شده شما چیست؟

اهداف تاکتیکی: چرا هر آزمون؟ اگر نمی دانید چرا هر آزمون را انجام می دهید، خیلی مفید نخواهد بود

تا یک زمان مشخصی
باید یه حد قابل قبولی
از تست انجام شده
باشد.

stopping rule
هیچ ای وجود نداره.

چقدر آزمایش کافی است؟
هدف مشترک - صرف بودجه ... آزمایش تا تاریخ ارسال ...
- گاهی اوقات "معیار تاریخ" نامیده می شود

Why Each Test ?

If you don't start planning for each test when the functional requirements are formed, you'll never know why you're conducting the test

نرم افزار باید قابلیت
نگهداری داشته باشه
مثلاً مازولار باشه

■ 1980: “The software shall be easily **maintainable**”

باید کاملاً ریکوایرمنت ها مستند شده باشند و یه پلنی داشته باشیم
حتی نیازمندی ها هم باید اولویت داشته باشند و دسته بندی بشوند
و برای تست یه پلن و برنامه بریزیم.

1980: ”نرم افزار باید به راحتی قابل نگهداری باشد
الزمات قابلیت اطمینان آستانه؟
هر آزمون سعی می کند چه واقعیتی را تأیید کند؟“

■ **Threshold reliability** requirements?

مثلاً اگه saftey نرم افزار به لول
۷ درصد بر سه ایا اوکیه؟ یا باید مثلاً
۹۹ درصد باشه؟

تیم های تعریف نیازمندی ها به آزمایش کننده نیاز دارند! اگر زمانی که
الزمات عملکردی شکل گرفت، برای هر آزمون برنامه ریزی نکنند،
هرگز نمی دانند چرا آزمون را انجام می دهید.

■ What fact does each test try to verify?

تسترها باید هم گام باشند با افرادی که دارند ویژگی های نرم افزار را مشخص میکنند.
تا تستر وقتی داره ارزیابی نرم افزار را انجام میده منطبق باشه ارزیابیش با نیازمندی
های مشتری و کاربر.

طبق چه معیاری میایم Verification انجام
ییدیم؟ اینها از تیم ریکوایرمنت ها میدارند یعنی اون
یعنی که دارن نیازمندی ها را مینویسند باید حتماً
یه تستر هم بینشون باشه.

■ Requirements definition teams need testers!

Cost of Not Testing

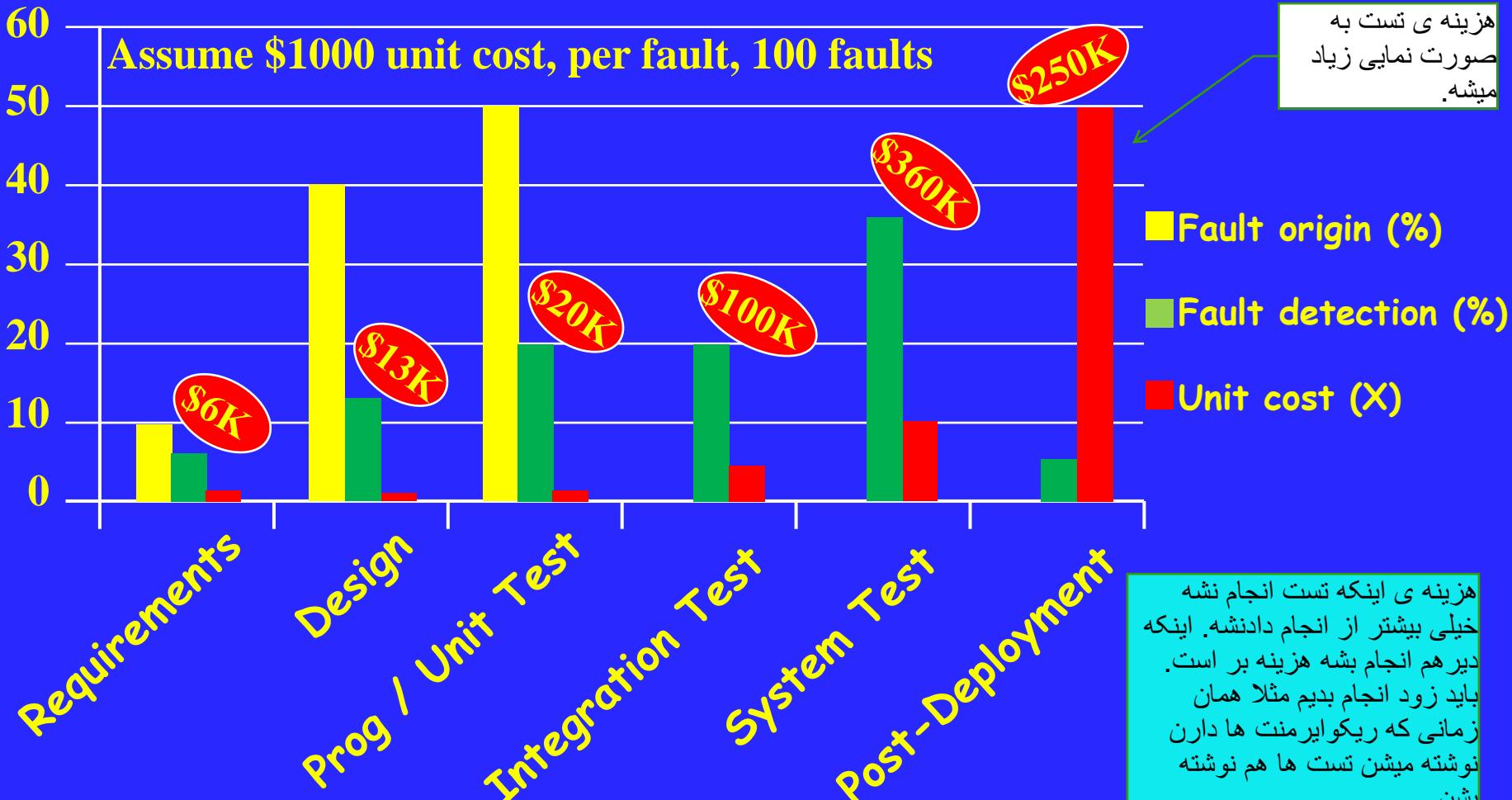
Poor Program Managers might say:
"Testing is too expensive."

- Testing is the **most time consuming** and **expensive part** of software development
- Not testing is even **more expensive**
- If we have **too little testing effort early**, the **cost** of testing **increases**
- Planning for testing **after development** is **prohibitively expensive**

هزینه عدم آزمایش
تست زمان برترین و پر هزینه ترین بخش توسعه نرم افزار است
تست نکردن حتی گرانتر است

اگر تلاش آزمایشی خیلی کمی داشته باشیم، هزینه آزمایش افزایش می یابد
برنامه ریزی برای آزمایش پس از توسعه بسیار گران است
مدیران برنامه ضعیف ممکن است بگویند: "تست بسیار گران است".

Cost of Late Testing



So...

Avoiding the work early
(requirements analysis and unit testing)
saves money in the short term.

But it leaves faults in software that are like little bombs, ticking away, and the longer they tick, the bigger the explosion when they finally go off.

بنابراین...

اجتناب از کار زود هنگام (تجزیه و تحلیل نیازمندی ها و آزمایش واحد) در کوتاه مدت باعث صرفه جویی در هزینه می شود.

ما در نرمافزارها که مانند بمبهای کوچک هستند، ایرادهایی به جا میگذارد و هر چه بیشتر تیکک میزنند، وقتی در نهایت خاموش میشوند، انفجار بزرگتر میشود.

Impact of Software on Testing

The type of software and its characteristics impact in different ways the testing and analysis activities.

- different emphasis may be given to the same properties
- different (new) properties may be required
- different (new) testing and analysis techniques may be needed

تأثیر نرم افزار بر تست

- نوع نرم افزار و ویژگی های آن به طرق مختلف بر فعالیت های تست و تحلیل تأثیر می گذارد.
- ممکن است بر ویژگی های یکسان تأکید متفاوتی داده شود
 - ممکن است ویژگی های متفاوت (جديد) مورد نیاز باشد
 - ممکن است به تکنیک های مختلف (جديد) آزمایش و تجزیه و تحلیل نیاز باشد

Different emphasis on different properties

سطح Dependability بر اساس نوع نرم افزار متفاوت است
مثلاً اگه سیستم ما Safety-critical است یه سطحی میخاده و
اگه محصول یه مارکت بزرگ است یه سطح دیگه.

Dependability requirements

- They differ radically between
 - Safety-critical applications

مثلاً availability خیلی مهم تر از Dependability است. اینکه سیستم مواره کار کنه خیلی مهمه برآمون

- Flight control systems have strict safety requirements
- Telecommunication systems have strict robustness requirements

Mass-market products

- Dependability is less important than time to market
- Vary within the same class of products
- Reliability and robustness are key issues for multi-user operating systems and less important for single users operating systems.

حتی در یک محصول و پروداکت هم ورزن های مختلفش میتوانه dependability ش متفاوت باشه.

- الزامات قابلیت اطمینان
- آنها به شدت با هم تفاوت دارند
- برنامه های کاربردی حیاتی ایمنی
- سیستم های کنترل پرواز الزامات ایمنی سختگیرانه ای دارند
- سیستم های مخابراتی الزامات استحکام سختی دارند
- محصولات بازار انبوه قابلیت اطمینان کمتر از زمان عرضه به بازار اهمیت دارد
- در یک کلاس از محصولات متفاوت است
- قابلیت اطمینان و استحکام مسائل کلیدی برای سیستم عامل های چند کاربره است و برای سیستم عامل های تک کاربر اهمیت کمتری دارد.

توانایی برگشت از خطأ و
ریکاور شدن زمانی که
خطایی رخ میده.

اون Threshold safety
که برای میگذاریم باید بالای
میگذاریم باید بالای
۹۹ درصد باشه.

Different type of software may require different properties

چه پر اپراتی هایی برای چه نوع نرم افزار هایی مهم هستند در نتست اهمیت دارند.

- **Timing properties**
 - deadline satisfaction is a key issue for real time systems,
 - performance is important for many applications
- **Synchronization properties**
 - absence of deadlock is important for concurrent or distributed systems,
- **External properties**
 - user friendliness is an issue for GUI, irrelevant for embedded controllers.

- خواص زمان بندی
- رضایت از ضرب الجل یک مسئله کلیدی برای سیستم های زمان واقعی است،
- عملکرد برای بسیاری از برنامه ها مهم است
- ویژگی های همگام سازی
- عدم وجود بن بست برای سیستم های همزمان یا توزیع شده مهم است،
- خواص خارجی
- کاربر پسند بودن یک مشکل برای رابط کاربری گرافیکی است که برای کنترلرهای تعییه شده بی ربط است.

Different Testing for checking the **same** properties for different software

- Test selection criteria based on structural coverage are different for
 - procedural software (statement, branch, path,...)
 - object oriented software (coverage of combination of polymorphic calls and dynamic bindings,...)
 - concurrent software (coverage of concurrent execution sequences,...)
- Absence of deadlock can be statically checked on some systems, require the construction of the reachability space for other systems.

• معیارهای انتخاب آزمون بر اساس پوشش سازه متفاوت است

• نرم افزار رویه ای (گزاره، شاخه، مسیر,...)

• نرم افزار شی گرا (پوشش ترکیبی از فراخوانی های چند شکلی و پیوندهای پویا,...)

• نرم افزار همزمان (پوشش اجرای همزمان

• دنباله ها,...)

• عدم وجود بن بست را می توان به صورت ایستاد در برخی سیستم ها بررسی کرد، نیاز به ساخت

فضای دسترسی برای سایر سیستم ها دارد.

Principles of effective software testing techniques

اصول تکنیک های تست نرم افزار موثر

• حساسیت: بهتر است هر بار شکست بخورید تا گاهی اوقات

- **Sensitivity:** better to fail every time than sometimes
 - Run time deadlock analysis works better if it is **machine independent.** تحلیل بن بست زمان اجرا اگر مستقل از ماشین باشد بهتر عمل می کند.
- **Redundancy:** making **intentions explicit**
 - Increase the capabilities of catching **specific faults early or more efficiently** افزونگی: واضح ساختن مقاصد قابلیت تشخیص زودهنگام یا کارآمدتر عیب های خاص را افزایش دهید
- **Partitioning:** **divide and conquer**
 - Can be handled by suitably partitioning the **input space**
- **Restriction:** making the **problem easier**
 - Can **reduce hard** (unsolvable) problems to **simpler** (solvable) problems
- **Feedback:** **tuning** the development process
 - Learning from experience

• تقسیم بندی: تفرقه بینداز و غلبه کن

• می توان با پارتیشن بندی مناسب فضای ورودی کار کرد

• محدودیت: آسان کردن مشکل

• می تواند مسائل سخت (غیر قابل حل) را به مسائل ساده (قابل حل) کاهش دهد

• بازخورد: تنظیم فرآیند توسعه

• یادگیری از تجربه

Fundamental Principles of software testing(I)

اصول اساسی تست نرم افزار

پیک کاربران در یک زمانی چندبرابر بیشتر از سطح انتظار ما میشه که ما قبل انتظارشو داشتیم.

- Testing show the presence of defects.
 - Regardless of how thoroughly a product or application is tested, no one can guarantee that it is defect-free.
- Exhaustive testing is impossible.
 - It is impossible to test all possible combinations of data, inputs, and test scenarios.
- Early testing
 - Should be as soon as possible
- Defect clustering
 - Most defects are found in few modules.

به دلیل محدودیت زمان و هزینه زیاد به دلیل بی انتها بودن فضای ورودی نمیتوانیم همه چیز را تست کنیم.

روی برخی از مازول های خطا را بیشتر وقت و تمرکز بگذاریم.

خوشه بندی میتواند از یید معماری نرم افزار آشنه یا از دید ریسک ها باشه یا از دید ریکوایرمنت ها باشه یا از دید خطا خیز بودن مازول ها باشه.

- آزمایش وجود نقص را نشان می دهد.
- صرف نظر از اینکه یک محصول یا برنامه به طور کامل آزمایش شده است، هیچ کس نمی تواند تضمین کند که بدون نقص است.
- آزمایش جامع غیرممکن است.
- آزمایش تمام ترکیبات ممکن از داده ها، ورودی ها و سناریوهای آزمایش غیرممکن است.
- تست اولیه
- باید در اسرع وقت
- خوشه بندی نقص
- بیشتر عیوب در چند مازول یافت می شود.

Fundamental Principles of software testing(II)

در فاز integration میاد.
به صورت لوکالی ممکن است همه
چیز درست باشه ولی اجزا که کنار
هم قرار بگیرند ممکن است خطأ
ایجاد شه.

Pesticide Paradox

- Frequently review and update the test cases in order to cover various sections of the projects.
- Testing is dependent on the problem.
 - Same techniques cannot be used for multiple projects.
- The lack of the Fallacy of Errors
 - The software product should be tested not only on its technical aspects, but also on the expectations and needs of users.

چون سطح انتظار و
خصوصیت های
پروژه های مختلف
باهم فرق داره مثل
در یک پروژه زمان
پاسخ مهم تر از
قابلیت اطمینان است.

نه تنها باید Verification را
نجام بدم بلکه validation هم
باید انجام بدم یعنی انتظارات
کاربران را ببینیم براورده شده
یانه؟ باید این دو ترا باهم چک
کنیم.

ممکن است از نظر فنی همه چیز خوب باشه ولی نیاز کاربر اون چیزی
نیست که سیستم داره ارائه میده و بر عکس تمرکز من روی خاسته های
کاربر است ولی اهمیتی به جنبه های تکنیکال ندادیم اگه اینکارو کنیم دیگه
نرم افزار مون reusable و maintainable نیست یعنی کیفیت سیستم
پایین میاد.

- پارادوکس آفت کش ها
- موارد آزمایشی را به طور مکرر بررسی و به روز کنید تا بخش های مختلف پروژه ها را پوشش دهید.
- آزمایش به مشکل بستگی دارد.
- تکنیک های مشابه را نمی توان برای چندین پروژه استفاده کرد.
- فقدان مغالطه خطاهای
- محصول نرم افزاری باید نه تنها از نظر جنبه های فنی، بلکه بر اساس انتظارات و نیاز های کاربران نیز مورد آزمایش قرار گیرد.

Summary: Why Do We Test Software ?

هدف از تست: فالت ها را بیشتر تشخیص بدیم و زودتر تشخیص بدیم در حدی که میتوانیم باید فالت ها را تشخیص بدیم.

A tester's goal is to **eliminate faults**
as early as possible

We can never be perfect, but every time we eliminate a fault during unit testing (or sooner!), we save money.

هدف تستر این است که عیوب را در اسرع وقت از بین ببرد
ما هرگز نمیتوانیم کامل باشیم، اما هر بار که در طول آزمایش واحد (یا
زودتر!) یک عیوب را برطرف میکنیم، در هزینه صرفهجویی میکنیم.

- **Improve quality**
- **Reduce cost**
- **Preserve customer satisfaction**

- بهبود کیفیت
- کاهش هزینه
- حفظ رضایت مشتری

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

در عوض، سعی کنید «رفتار» نرم افزار را ارزیابی کنید تا تصمیم بگیرید که آیا رفتار با توجه به تعداد زیادی از عوامل قابل قبول است یا خیر.
بدیهی است که این پیچیده‌تر است.

یک راه حل این است که سطح انتزاع خود را بالا ببریم“

- The term **correctness** has no meaning

- Is a **building** correct?
- Is a **car** correct?
- Is a **subway system** correct?

هیچ رشته مهندسی دیگری به اندازه نرم افزار محصولی پیچیده نمی سازد
اصطلاح صحت معنا ندارد.
آیا ساختمان درست است؟
– ماشین درسته؟
آیا سیستم مترو درست است؟

- Instead, try to **evaluate software's “behavior”** to decide if the **behavior** is **acceptable** within consideration of a large number of factors.

- Obviously, this is **more complex**.

اـ حل برای ارزیابی نرم افزار: سطح انتظار را بالا ببریم. هنر کاهش جزئیات. حذف کردن شاخ و برگ ها

- One solution is to “raise our level of abstraction.”

به جای اینکه بگیم یه نرم افزار درسته، درستی را براساس یه سری فاکتور بررسی میکنیم مثل **reliability** پس میایم رفتار نرم افزار را بررسی میکنیم

چقدر نرم افزار **safe** است؟ چقدر در برابر حمله ها مقاوم است؟ جقدر **robust** است؟ چقدر در برابر حمله ها مقاوم است؟

برای غلبه بر پیچیدگی تست نرم افزار از مدل های
ابستركت استفاده میکنیم.

اگر طراحان بتوانند سطح انتزاع خود را بالا ببرند، کارآمدتر و مؤثرتر
هستند.

- مانند سایر مهندسان، ما باید از انتزاع برای مدیریت پیچیدگی استفاده کنیم
- این هدف از فرآیند طراحی تست مدل محور است
 - مدل «یک ساختار انتزاعی است

*Designers are more efficient and effective
if they can
raise their level of abstraction.*

- 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

مدل همه ی واقعیت
را بیان نمیکنه چون
جزئیاتش حذف شده

Model-Driven Test Design (MDTD) process(1)

- Breaks testing into a series of small tasks that simplify test generation.
- Then test designers isolate their task, and work at a higher level of abstraction by using mathematical engineering structures to design test values independently of the details of software or design artifacts, test automation, and test execution.

تست را به یک سری کارهای کوچک تقسیم می کند که تولید تست را ساده می کند.

سپس طراحان آزمون وظیفه خود را جدا میکنند و با استفاده از ساختارهای مهندسی ریاضی برای طراحی نقادیر آزمون مستقل از جزئیات نرم افزار یا مصنوعات طراحی، اتماسیون تست و اجرای آزمون، در سطح بالاتری از انتزاع کار میکنند.

ارویکردهای ریاضی مدلمن را میسازیم مثلا برنامه های را به یک گراف تبدیل میکنیم. گراف میشه یه نمایش انتزاعی از کدمن که گراف میشه مبنای تولید ولیو هامون برای تست. و مستقل از نرم افزار کار میکنه

The Model-Driven Test Design (MDTD) process(1)

طراحی تست کیس

- A key intellectual step in MDTD is **test case design**.
- Test case design can be the primary determining factor in whether tests successfully find failures in software.

یک مرحله فکری کلیدی در MDTD طراحی مورد آزمایشی است. طراحی کیس آزمایشی میتواند عامل اصلی تعیینکننده در یافتن موفقیت آمیز اینکه آیا تستها به خوبی فیلیورها را کشف کردند یا نه در نرمافزار باشد.
- Tests can be designed with a “**human-based**” approach.

آزمون ها را می توان با رویکرد «متنی بر انسان» طراحی کرد.

 - A test engineer uses domain knowledge of the software’s purpose and his or her experience to design tests that will be effective at finding faults.

تست ها را طوری دربیاریم که یک سری معیارهایی را کاور کنند. این رویکرد دوم است بر اساس معیارهای پوشش، تست کیس ها را تولید میکنیم.
- Alternatively, tests can be designed to satisfy well-defined engineering goals such as **coverage criteria**.

یک مهندس تست از دانش دامنه در مورد هدف نرم افزار و تجربه خود برای طراحی آزمایش هایی استفاده می کند که در یافتن عیوب موثر باشد.

از طرف دیگر، تست ها را میتوان برای برآورده کردن اهداف مهندسی به خوبی تعریف شده مانند معیارهای پوشش طراحی کرد.

Software Testing Foundations (1)

مثلاً یکی از coverage criteria یا پوشش عبارتی است یعنی کل تست کیس هامون در برنامه باید. به نحوی باشه که هر استیتمنتی توى برنامه حداقل یک بار اجرا شه . معیارهای متفاوتی از معیارهای پوشش است.

**Testing can only show the presence of failures,
Not their absence**

تست ها فقط می توانند وجود فیلورها را نشان دهد، نه عدم وجود آنها را

Software Testing Foundations (2)

- the problem of finding all failures in a program is **undecidable**.
- Testers often call a test **successful** (or **effective**) if it finds an error.

مساله‌ی یافتن تمام فیلیور‌ها در یک برنامه غیرقابل تصمیم‌گیری است.
تسترها معمولاً در صورت یافتن خطاء، آزمایشی را موفق (یا مؤثر) مینامند.

در صورتی موفق هستیم که خطای پیدا کنیم نمی‌گیم که همه خطاهای را چون پیدا کردن همه‌ی خطاهای یه مساله‌ی **undecidable** است.

Software Faults, Errors & Failures

- Software Fault : A static defect in the software
- Software Error : An incorrect internal state that is the manifestation of some fault
- Software Failure : External, incorrect behavior with respect to the requirements or other description of the expected behavior

فالت نرم افزار: یک نقص استاتیک در نرم افزار
ارور نرم افزار: یک حالت داخلی نادرست که مظهر ونشانه‌ی فالت است
فیلیویر نرم افزار: رفتار خارجی و نادرست با توجه به الزامات یا سایر
توضیحات رفتار مورد انتظار

Testing & Debugging

- **Testing** : Evaluating software by observing its execution

تست: ارزیابی نرم افزار با مشاهده نحوه اجرای آن

- **Test Failure** : Execution of a test that results in a software failure

Test Failure : اجرای آزمایشی که منجر به خرابی نرم افزار می شود

- **Debugging** : The process of finding a fault given a failure

همه ای انپوت ها منجر به مشاهده فیلور نمیشن.

اشکال زدایی: فرآیند یافتن عیب در صورت شکست

for a given fault, not all inputs will “trigger” the fault into creating incorrect output (a failure)

برای یک خطای معین، همه ورودیها باعث ایجاد خطأ در ایجاد خروجی نادرست (شکست) نمیشوند.

A Concrete Example

Fault: Should start searching at 0, not 1

```
public static int numZero (int [ ] arr)
{ // Effects: If arr is null throw NullPointerException
// else return the number of occurrences of 0 in arr
int count = 0;
for (int i = 1; i < arr.length; i++)
{
    if (arr [ i ] == 0)
    {
        count++;
    }
}
return count;
```

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Error: i is 1, not 0, on
the first iteration
Failure: none

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

So,

- it is often **very difficult** to relate a failure to the associated **fault**.
- Analyzing these ideas leads to the **fault/failure model**, which states that **four conditions** are needed for a failure to be observed.

اغلب بسیار دشوار است که یک شکست را به خطای مرتبط مرتبط کنیم.
تجزیه و تحلیل این ایده‌ها منجر به مدل خطا/شکست می‌شود که بیان می‌کند چهار شرط
برای مشاهده شکست لازم است.

دل می‌گه اگه ۴تا شرط برقرار باشه میشه تضمین کرد که یک فیلور قابل مشاهده است. که به این مدل
میگیم RIPR

Fault & Failure Model (RIPR)

Four conditions necessary for a failure to be observed

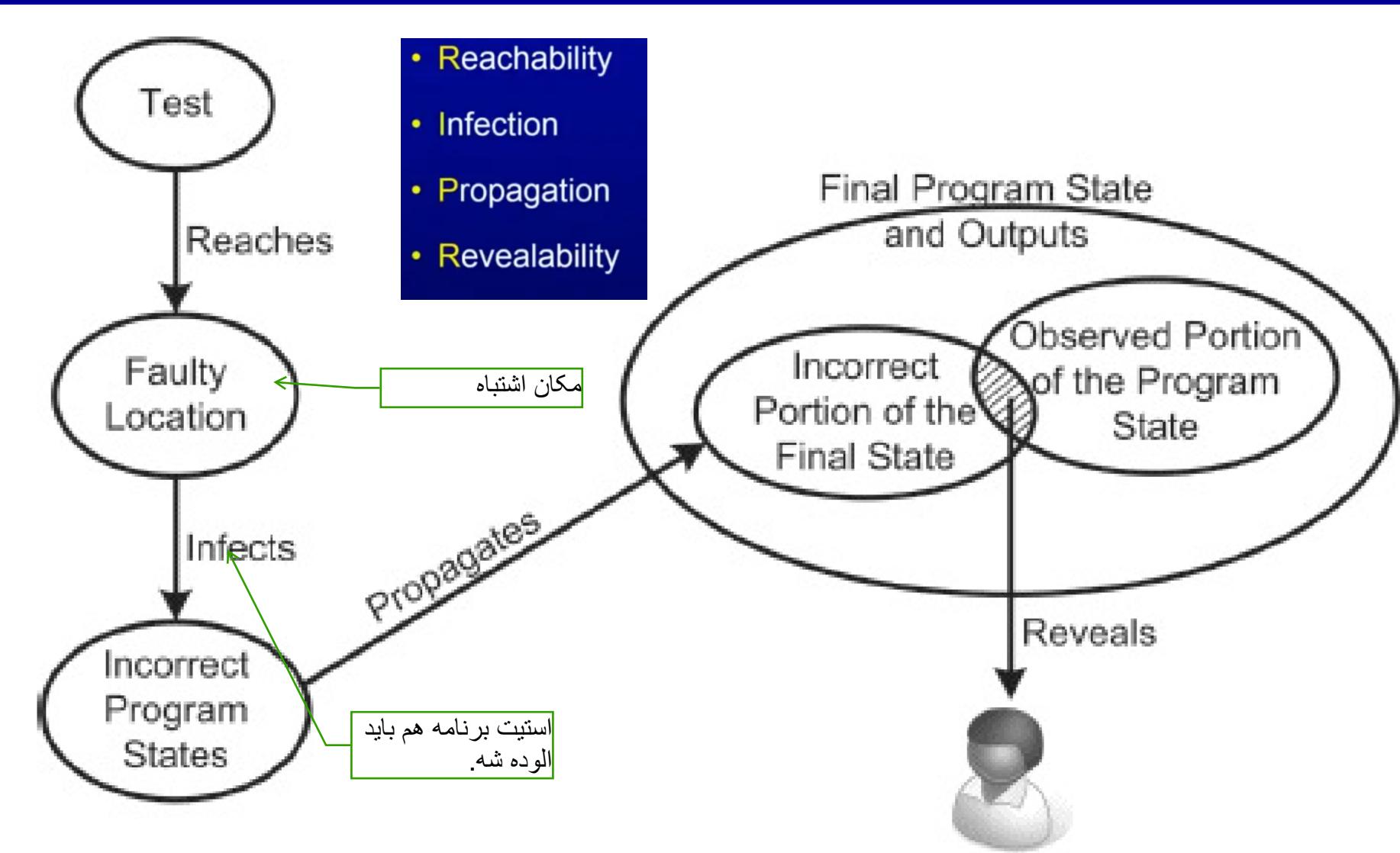
وقتی که برنامه داره اجرا میشه باید به مکانی که اشکال و فالت داره باید برسیم باید اون خطی که اشکال داره اجرا بشه.

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
استیت برنامه هم باید از اون فالته تاثیر بگیره و اشتباه بشه.
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
استیت ای که خطا داره باید توسط یوزر مشاهده شه.

1. قابلیت دسترسی: باید به مکان یا مکان هایی در برنامه که حاوی خطا هستند دسترسی داشت
2. عفونت: وضعیت برنامه باید نادرست باشد
3. انتشار: حالت آلوده باید باعث شود برخی خروجی ها یا وضعیت نهایی برنامه نادرست باشد.
4. آشکار: آزمایشگر باید بخشی از قسمت نادرست وضعیت برنامه را مشاهده کند

RIPR Model

با برقراری این ۴ تا
شرط گارانتی میشے
که خطای قابل مشاهده
است.



Is RIPR model can detect missing code?

- The RIPR model applies even when the **fault is missing code** (so-called **faults of omission**).
- when **execution** passes through the **location** where the **missing code** should be, the **program counter**, which is part of the **program state**, necessarily has the **wrong value**.

آیا مدل RIPR می تواند کم شده را تشخیص دهد؟
مدل RIPR حتی زمانی که خطا فاقد کد باشد (به اصطلاح خطاهای حذف) اعمال می شود.
هنگامی که اجرا از محلی می گذرد که کم شده باید باشد، شمارنده برنامه، که بخشی از وضعیت برنامه است، لزوماً دارای اشتباه است.

آیا مدل RIPR برای کدهای کم شده و کدهایی که ننوشتم هم قابل اعمال است؟ میتواند تشخیص بده که یک قسمتی از کد را ننوشتم؟ بله چون وقتی یه قسمتی از کد را ننویسیم یعنی PROGRAM counter داره اشتباه اپدیت میشه پس استینت برنامه اشتباه میشه که باعث میشه رفتار نرم افزار الوده شه و اگه کاربر ببینه میفهمه یک قسمت کد نیست.

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
- every engineer involved in software development can wear the hat of a test engineer. Because the person best positioned to define these test cases is often the designer of the artifact.

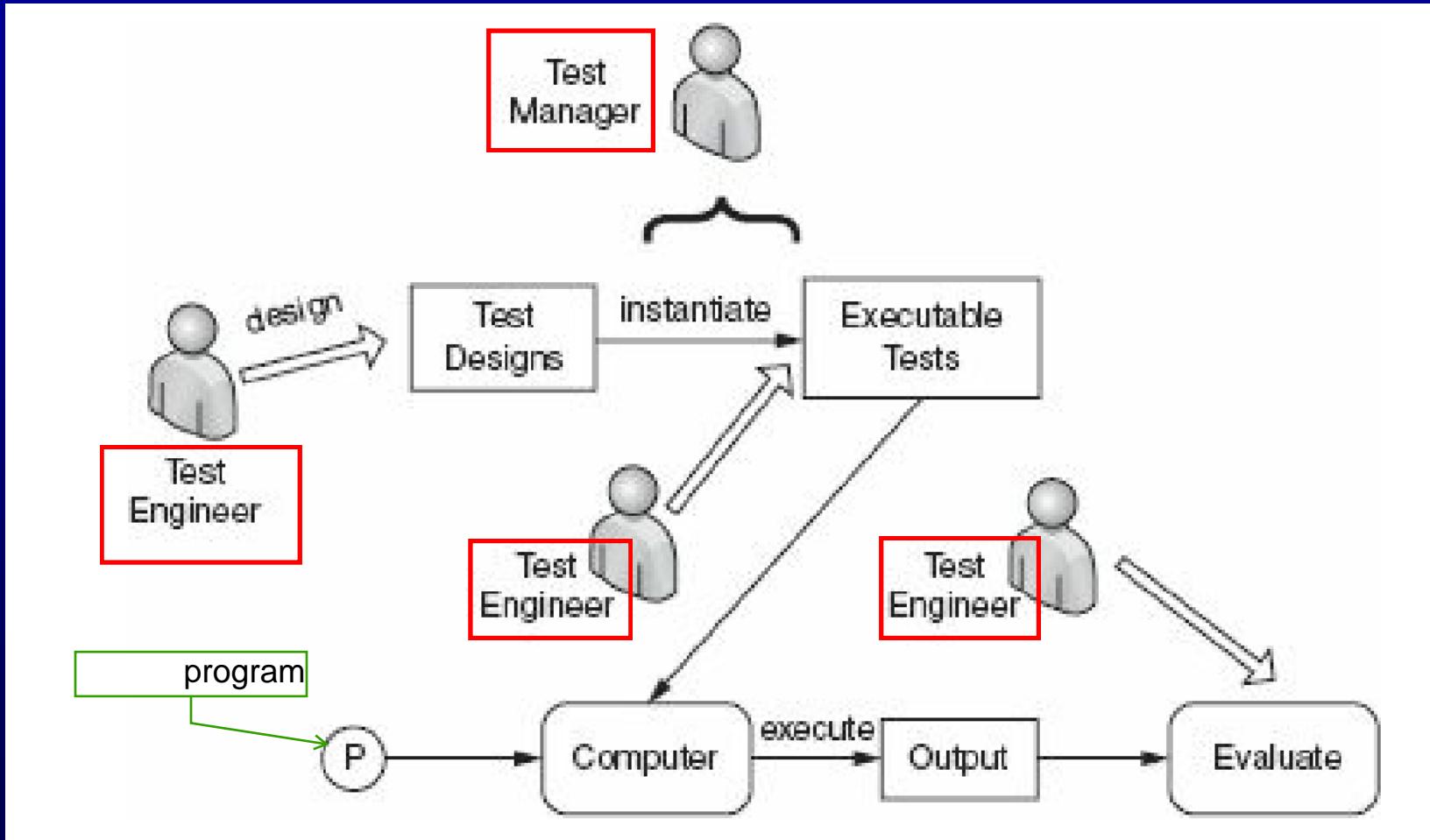
مهندس تست: یک متخصص فناوری اطلاعات که مسئول یک یا چند فعالیت تست فنی است

- طراحی ورودی های تست
- تولید مقادیر تست
- اجرای اسکریپت های تست
- تجزیه و تحلیل نتایج
- گزارش نتایج به توسعه دهندگان و مدیران

نتایج باید لوکالایز
بشن برای اون فالت
ما یعنی ترک بشن که
چه فالت هایی باعث
اون فیلورها شده؟

هر مهندس درگیر در توسعه نرم افزار می تواند کلاه یک مهندس آزمایشی را بر سر بگذارد.
زیرا شخصی که بهترین موقعیت را برای تعریف این موارد تست دارد، اغلب طراح مصنوع است.

Activities os Test Engineers



Important points

هدف اصلی -

- طراحی تست هایی که به طور سیستماتیک طبقات مختلف خطاها را با حداقل زمان و تلاش آشکار می کند
- یک تست خوب احتمال خطایابی بالایی دارد
- یک آزمایش موفق یک خطا را آشکار می کند

Main objective

- Design tests that systematically uncover different classes of errors with a minimum amount of time and effort
- A good test has a high probability of finding an error
- A successful test uncovers an error

ولی برای طراحی و لیوهای تست ادم هایی که درگیر کار هستند میتوانند ایده های خوبی بدن و تست کیس های خوبی طرح کنند.

Secondary benefits

- Demonstrate that software appears to be working according to specification (functional and non-functional)
- Data collected during testing provides indication of software reliability and software quality
- Good testers clarify the specification (creative work)

تستر های خوب میتوانند خود specification ها را دقیق تر و شفاف تر نمایند یعنی میتوانند در بیان بهتر نیازمندی ها

Testing is done best by independent testers.

- آزمایش کننده های خوب مشخصات را روشن می کنند (کار خلافانه) تست به بهترین وجه توسط آزمایش کنندگان مستقل انجام می شود.

مزایای ثانویه - نشان میدهد که به نظر می رسد نرم افزار مطابق با مشخصات کار می کند (عملکردی و غیر کاربردی)
- داده های جمع آوری شده در طول آزمایش، نشاندهنده قابلیت اطمینان و کیفیت نرم افزار است.

Software Testing Activities (2.2)

- 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**

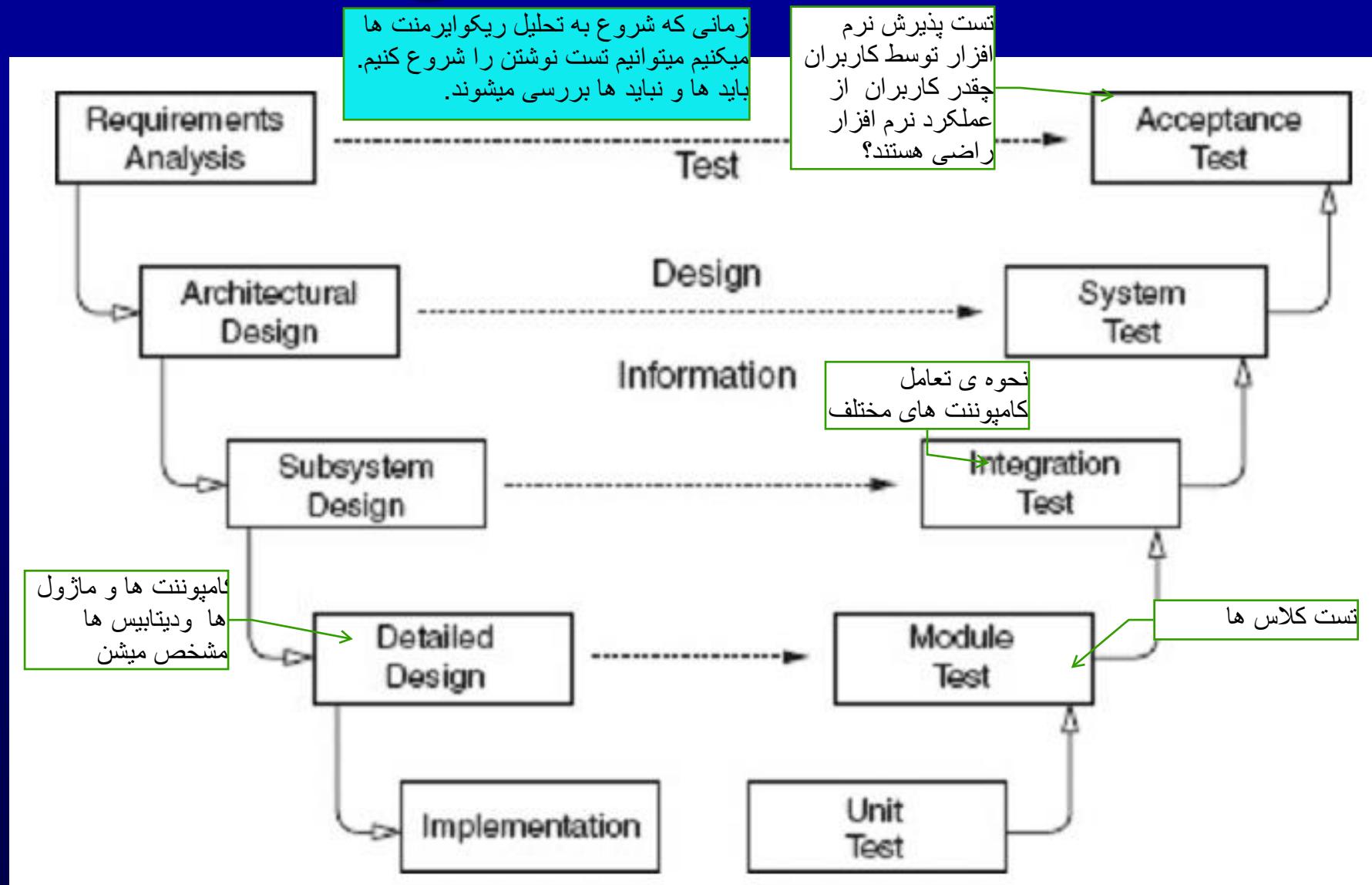
مدیر آزمون: مسئول یک یا چند مهندس آزمون

- سیاست ها و فرآیندهای تست را تنظیم می کند

- با سایر مدیران پروژه تعامل دارد

- در غیر این صورت از مهندسین پشتیبانی می کند

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



Traditional Testing Levels (2.3)

متدهایی که منسجم باشند و فقط یه کار را بکنند تست کردن شون
هم راحت تره single responsibility

هر چه سطح تست بالاتر بیاد، تست
کردن سخت تر میشه چون
کامپوننتهای بیشتری درگیر میشن

main Class P

با درنظر گرفتن ارتباطاتی که
ماژول ها باهم دارن تست کنیم
نحوه تعامل کلاس ها با هم
بررسی میشه مثلما میدنیم
کلاس A و کلاس B به تنها ی
درست کار میکنن حالا میخایم
بیننم در تعامل باهم دیگه هم ایا
هنووز درست کار میکنن؟

Class A

method mA1()

method mA2()

Class B

method mB1()

method mB2()

آزمون پذیرش: آیا نرم افزار مورد قبول کاربر است؟

تست یکپارچه سازی: نحوه تعامل ماژول ها با یکدیگر را تست کنید

تست سیستم: عملکرد کلی سیستم را تست کنید

تست ماژول (تست توسعه دهنده): هر کلاس، فایل، ماژول،
کامپوننت را تست کنید

تست واحد (تست توسعه دهنده): هر واحد (متد) را جداگانه تست کنید

کلاس A را مستقل
تست کنیم.

بالارفتن سطح
تست

تست خود متد مستقل
از بقیه متدها به
صورت لوکالی

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

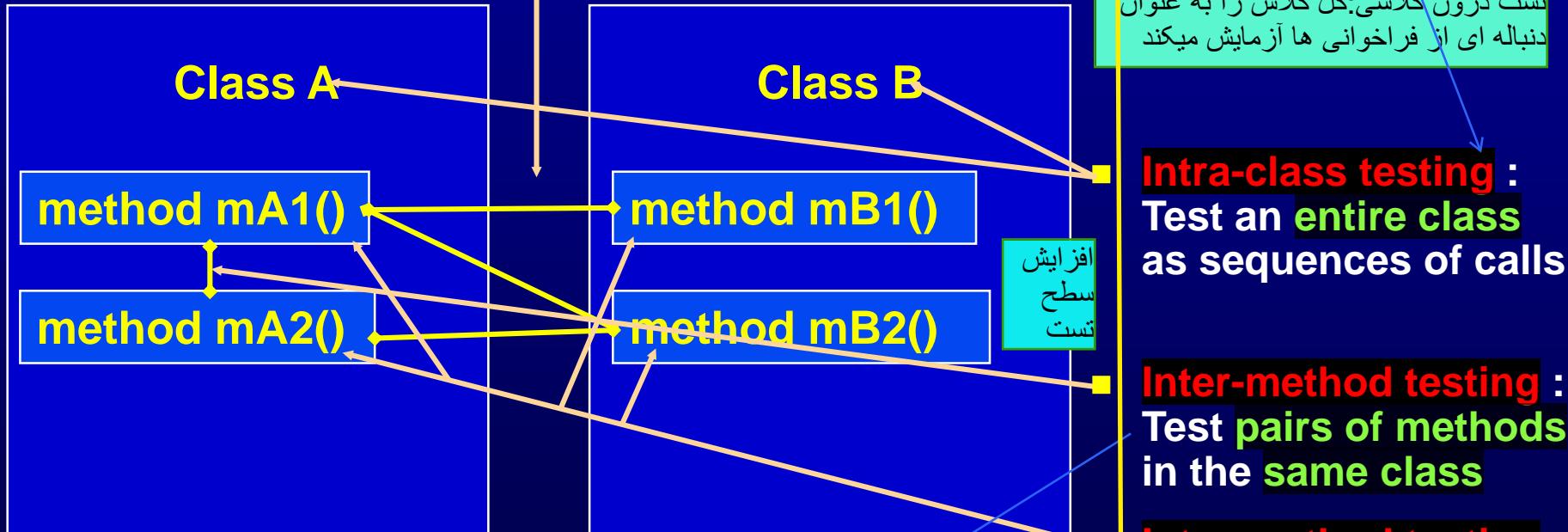
Object-Oriented Testing Levels

تست بین کلاسی: چندین کلاس را با هم آزمایش کنید

Inter-class testing :
Test multiple classes together

نگاه شی گرایانه به لول های مختلف تست

عادل مازول تستینگ
ست فقط مازول میشه
کلاس یعنی کلاس را
با درنظر گرفتن همه
متدهاش تست کنیم.



تست بین متدها: جفت متدها را در یک کلاس آزمایش کنید تست کردن ارتباطات بین متدهای یک کلاس معادل مازول تستینگ است

تست درون متدها: هر متده را جداگانه تست کنید

Unit Testing

- Testing **individual subsystems (collection of classes)**
- Goal: Confirm that **subsystem** is **correctly coded** and carries out the intended **functionality**.
- To achieve a reasonable **test coverage**, one has to **test each method** with **several inputs**
 - To cover **valid** and **invalid inputs**
 - To cover **different paths** through the **method**

- آزمایش زیرسیستم های فردی و منحصر به فرد (مجموعه ای از کلاس ها)
- هدف: تأیید کنید که زیرسیستم به درستی کدگذاری شده است و عملکرد مورد نظر را انجام می دهد.
- برای دستیابی به پوشش تست معقول، باید هر روش را با چندین ورودی آزمایش کرد
 - برای پوشش ورودی های معترض و نامعتبر
 - برای پوشش مسیر های مختلف از طریق متند

Module Testing

- Testing **elements of each modules**(classes, component, ...)
- Goal: Confirm that module is **correctly coded** and carries out the intended **functionality**.

• تست عناصر هر مژول (کلاس ها، مؤلفه ها، ...)

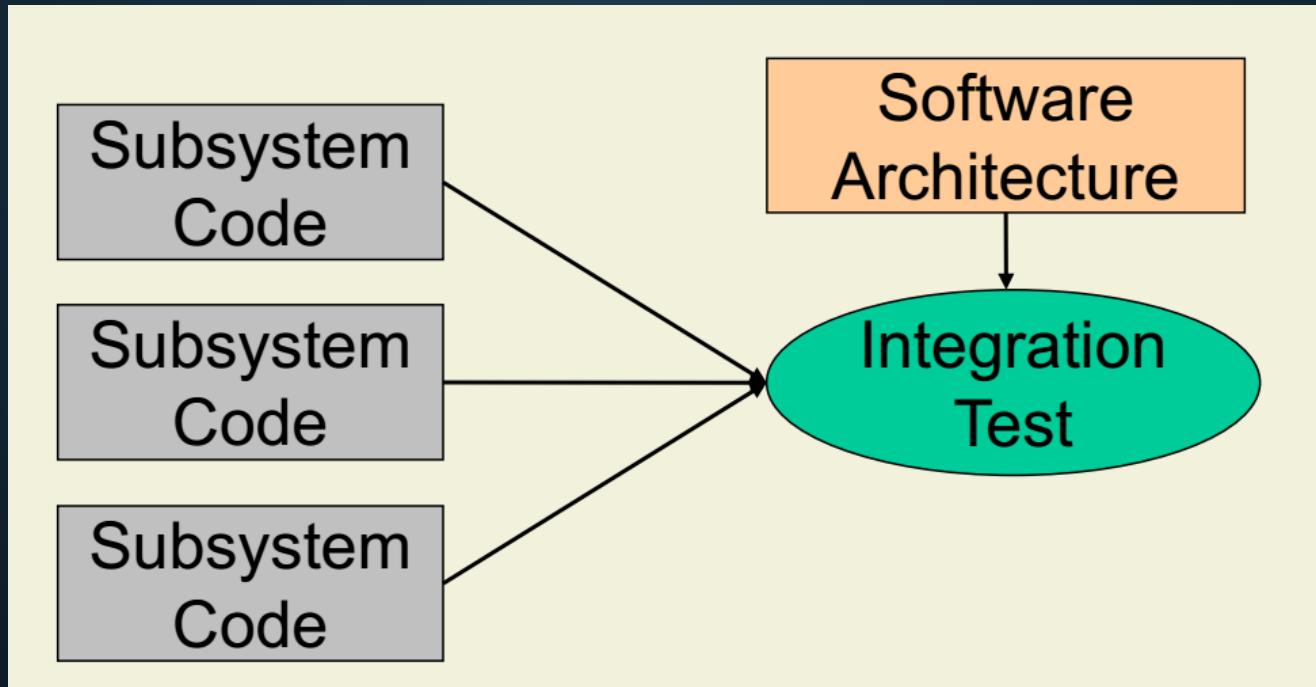
• هدف: تأیید کنید که مژول به درستی کدنویسی شده است و عملکرد مورد نظر را انجام می دهد.

Integration Testing

تست کردن اینترفیس های بین ساب سیستم ها

- Testing **groups of subsystems** and eventually the **entire system**
- Goal: Test **interfaces between subsystems**

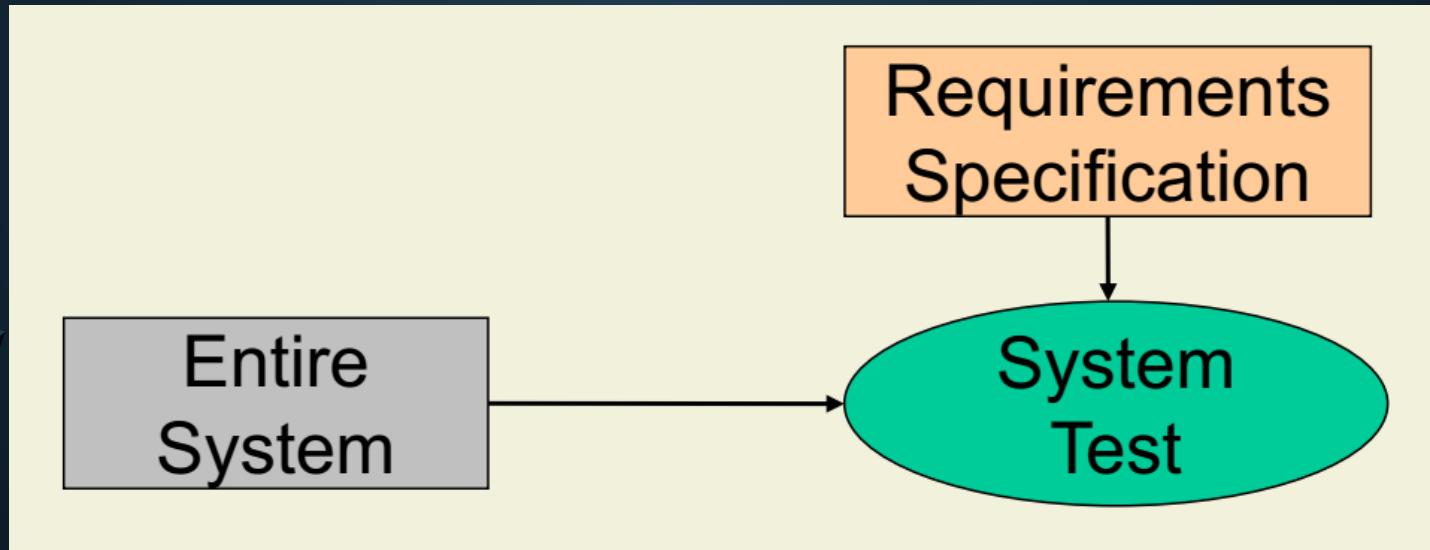
آزمایش گروه های زیرسیستم ها و در نهایت کل سیستم
هدف: تست روابط بین زیرسیستم ها



System Testing

- تست کل سیستم
- هدف: تعیین اینکه آیا سیستم الزامات (عملکردی و غیر عملکردی) را برآورده می کند یا خیر.

- Testing the entire system
- Goal: Determine if the system meets the requirements (functional and non-functional)



Acceptance Testing

- هدف: نشان دادن اینکه سیستم با نیازهای مشتری مطابقت دارد و آمده استفاده است
- توسط مشتری انجام می شود، نه توسط توسعه دهنده

- **Goal:** Demonstrate that the system meets **customer requirements** and is **ready to use**
- Performed by the **client**, not by the developer

تست آلفا

- مشتری از نرم افزار در سایت توسعه دهنده استفاده می کند
- نرم افزار مورد استفاده در یک تنظیمات کنترل شده، با توسعه دهنده آمده برای رفع اشکالات

- **Alpha test**

- Client uses the software at the developer's site
- Software used in a controlled setting, with the developer ready to fix bugs

- **Beta test**

- Conducted at client's site (developer is not present)
- Software gets a realistic workout in target environment

سیستم ها دارن داده های واقعی تولید میکنند و در حالت پروداکشن هستند.

تست بتا

- در سایت مشتری انجام شد (توسعه دهنده حضور ندارد)
- نرم افزار تمرینی واقع بینانه در محیط هدف می گیرد

Regression testing

- a **standard** part of the **maintenance phase** of software development.
- is done **after changes** are made to the software, to help **ensure** that the **updated software** still possesses the functionality it had **before the updates**.
- **Testing** that everything that used to work **still works** after changes are applied to the system

تست رگرسیون

بخشی استاندارد از مرحله تعمیر و نگهداری توسعه نرم افزار.

پس از ایجاد تغییرات در نرم افزار انجام می شود تا اطمینان حاصل شود که نرم افزار به روز شده همچنان عملکردی را که قبل از به روز رسانی داشته است دارد.

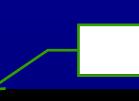
آزمایش اینکه همه چیزهایی که قبلاً کار میکرند، پس از اعمال تغییرات در سیستم همچنان کار میکنند

انتظار داریم مازول هایی که قبل از تغییر کار میکردن همچنان کار کنند. پس میایم تست میکنیم که همه چه مثل قبل درست کار میکنه بعد از اعمال تغییر.

مطمین شیم تغییرها خراب نکردن چیزی را. برای نگهداری نرم افزار استفاده میشوند.

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**

10²⁷
- Over **80 octillion possible tests!!**
- **Input space might as well be infinite**

در یک ماشین 32 بیتی، هر متغیر بیش از 4 میلیارد مقدار ممکن دارد.

- بیش از 80 اکتیلیون تست ممکن!!
- فضای ورودی نیز ممکن است بی نهایت باشد
آزمایشکنندگان فضای ورودی بزرگی را جستجو میکنند.
تلاش برای یافتن کمترین ورودی که بیشترین مشکل را پیدا کند

■ Testers search a huge input space

- Trying to find the **fewest inputs** that will find the **most problems**

ما توی این فضای بی نهایت سرج میکنیم و کمترین تعداد ورودی را انتخاب میکنیم که بیشترین خطاهارا اشکار کنند. پس نمیتوانیم همه ی ورودی ها را تست کنیم.

■ Challenges

- **How do we search?**
- **When do we stop?**

چالش ها
- چگونه جستجو کنیم؟ چطوری توی این فضای بی نهایت جستجو کنیم؟
چه زمانی متوقف می شویم؟ ۱۰ تا تست انتخاب کنیم یا ۲۰ تا؟ چقدر انتخاب کنیم تا مطمین شیم بیشترین میزان خطا اشکار شده؟ ضمن اینکه اون تست هایی که در اوردیم باهم اول لپ نداشته باشند

Coverage Criteria (2.4)

- Coverage criteria give **structured**, **practical** ways to search the input space, **how** to search and **when** to stop.

- Search the **input space** thoroughly, especially in the **corners**
- Not much **overlap** in the tests

یعنی اگه مقادیر A,B,C یه فالتی را برامون دراورد دیگه نیاز به یک تست دیگه نداریم که همان فالت را دربیاره برامون!

جاهايي که به احتمال زیاد دولوپر نادیده گرفته و خطأ توش هست را دیتکت کنیم.
پتانسیل خطأ زیاده

معیارهای پوشش روش های ساختار یافته و عملی برای جستجوی فضای ورودی، نحوه جستجو و زمان توقف را ارائه می دهد

- فضای ورودی را به طور کامل جستجو کنید، به خصوص در گوشه ها
- همپوشانی زیادی در تست ها وجود ندارد

Advantages of Coverage Criteria

■ Maximize the “bang for the buck”

بهره وری بیشتر میشه.

مزایای معیارهای پوشش

به حداقل رساندن "Bang for Buck"

■ Provide traceability from software artifacts to tests

- Source, requirements, design models, ...

ارائه قابلیت ردیابی از مصنوعات نرم افزاری تا آزمایشات
– منبع، نیازمندی ها، مدل های طراحی، ...

تست رگرسیون را آسان تر میکند

پس از پایان آزمایش به آزمایشکنندگان یک «قانون توقف»
میدهد

می توان به خوبی با ابزارهای قدرتمند پشتیبانی کرد

■ Make regression testing easier

■ Gives testers a “stopping rule” ... when testing is finished

قانون توقف میده

■ Can be well supported with powerful tools

Test Requirements and Criteria

- **Coverage Criterion** : A collection of rules and a process that define test requirements

- Cover **every statement**
- Cover **every functional requirement**

تست باید یه سری
قابلیت داشته باشه

عيار پوشش: مجموعه ای از قوانین و فرآيندي
كه الزامات آزمون را تعریف می کند.

- هر عبارت را میپوشاند

- هر نیاز کاربردی را پوشش میدهد

- لزامات آزمون: موارد خاصی که باید در طول آزمایش رعایت شوند یا پوشش داده شوند

- **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

رویکردهای مشخصی که در تست حتما باید برقرار باشه

- هر عبارت ممکن است یک نیاز
آزمایشی باشد

هر نیاز کاربردی ممکن است یک نیاز آزمایشی باشد

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

محققان آزمایش ده ها معیار را تعریف کرده اند، اما همه آنها در واقع تنها چند معیار در چهار نوع ساختار هستند ...

1. Input domains
2. Graphs

درنظر گرفتن
ووردي تابع

3. Logic expressions
4. Syntax descriptions

1. دامنه های ورودی
2. نمودارها گراف ها
3. عبارات منطقی
4. توضیحات نحوی و سینتکسی

Introduction to Software Testing (2nd edition) **Chapter 2**

Model-Driven Test Design

Paul Ammann & Jeff Offutt

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

پس راه حل این شد که برای غلبه بر پیچیدگی سطح انتزاع را بالا ببریم و دوم برای پیچیدگی فضای تست و فضای ورودی، از یک سری قوانین یا معیارهای پوشش استفاده میکنیم که این معیارها روی یک سری از ساختارهای خاصی تعریف میشوند. که ۴تا ساختار بررسی میشوند.

Updated September 2016

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

تست جعبه سیاه: آزمایش ها را از توضیحات خارجی نرم افزار، از جمله مشخصات، الزامات، و طراحی استخراج کنید

تست جعبه سفید: آزمایشها را از کد منبع داخلی نرم افزار استخراج کنید، به ویژه از جمله شاخه ها، شرایط فردی و عبارات

MDTD makes these distinctions less important.

The more general question is:

from what abstraction level do we derive tests?

MDTD این تمایزات را کمتر اهمیت می دهد.
سوال کلی تر این است:

از چه سطح انتزاعی تست ها را استخراج می کنیم؟

MBT and MDTD

- Model-based testing : Derive tests from a model of the software (such as a UML diagram)
 - Typically assumes that the model has been built to specify the behavior of the software and was created during a design stage of development.
- Our way is much overlap with MDTD and most of the concepts in this book can be directly used as part of MBT.

آزمایش مبتنی بر مدل: آزمایشها را از یک مدل نرم افزار استخراج میکند (مانند نمودار UML) - معمولاً فرض میکند که مدل برای مشخص کردن رفتار نرم افزار ساخته شده است و در مرحله طراحی توسعه ایجاد شده است.
روش ما بسیار با MDTD همپوشانی دارد و بیشتر مفاهیم این کتاب را می توان مستقیماً به عنوان بخشی از MBT استفاده کرد.

Difference between MDTD and MBT

- We derive our tests from **abstract structures** that are very **similar to models**

ما تست های خود را از ساختارهای انتزاعی که بسیار شبیه به مدل ها هستند استخراج می کنیم

- Differences**

- These **structures** can be created **after the software is implemented.**
- Create **idealized structures** that are **more abstract** than most modeling languages.
- MBT explicitly **does not use the source code implementation** to design tests. In this book, **abstract structures** can be created from the **implementation.**

- این ساختارها را می توان پس از پیاده سازی نرم افزار ایجاد کرد.
- ساختارهای ایده آلی ایجاد میکند که انتزاعی تر از بسیاری از زبان های مدل سازی هستند.
- MBT به صراحت از پیاده سازی کد منبع برای طراحی تست ها استفاده نمی کند.
در این کتاب می توان ساختارهای انتزاعی از پیاده سازی ایجاد کرد.

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

طراحی تست فرآیند طراحی مقادیر ورودی است که به طور موثر نرم افزار را آزمایش می کند
طراحی تست یکی از چندین فعالیت برای تست نرم افزار است.
- ریاضی ترین
- از نظر فنی چالش برانگیزترین

Types of Test Activities

- Testing can be broken up into four general types of activities

1. Test Design
2. Test Automation
3. Test Execution
4. Test Evaluation

بر اساس معیارها
مبتنی بر انسان

- I.a) Criteria-based
- I.b) Human-based

1. طراحی تست
2. تست اتوماسیون
3. اجرای تست
4. ارزیابی تست

هر نوع فعالیت به مهارت ها، دانش پیش زمینه، آموزش متفاوتی نیاز دارد
هیچ سازمان توسعه نرم افزار معقولی از افراد مشابه برای نیازمندی ها، طراحی، پیاده
سازی، یکپارچه سازی و کنترل پیکربندی استفاده نمی کند

- 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

برای سازمان های آزمون هنوز از افراد
یکسان برای هر چهار فعالیت آزمون
استفاده می کنند؟
این به وضوح منابع را هدر می دهد

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
- This is intellectually stimulating, rewarding, and challenging
- 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

این فنی ترین کار در تست نرم افزار است
نیاز به دانش:- ریاضیات گسسته- برنامه نویسی- آزمایش کردن
به یک مدرک CS سنتی نیاز دارد
این از نظر فکری محرك، پاداش و چالش برانگیز است
طراحی تست مشابه معماری نرم افزار در سمت توسعه است
استفاده از افرادی که صلاحیت طراحی تست ها را ندارند یک راه مطمئن برای
گرفتن تست های بی اثر است

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
- 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, ...)

این بسیار سخت تر از آن چیزی است که ممکن است برای توسعه دهندگان به نظر برسد
نیاز به دانش:
– دامنه، تست و رابط کاربری
نحویاً به هیچ CS سنتی نیاز ندارد
– داشتن پیشینه در حوزه نرم افزار ضروری است
- پیشینه تجربی بسیار مفید است (زیست شناسی، روانشناسی، ...)
- پیشینه منطقی بسیار مفید است (حقوق، فلسفه، ریاضی، ...)
مقادیر آزمون را بر اساس دانش دامنه برنامه و دانش انسانی آزمایش طراحی کنید

1. Test Design—(b) Human-Based

- This is intellectually stimulating, rewarding, and challenging
 - But not to typical CS majors – they want to solve problems and build things
- Human-based test designers explicitly attempt to find stress tests,
 - Tests that stress the software by including very large or very small values, boundary values, invalid values, or other values that the software may not expect during typical behavior.
 - Criteria-based approaches can be blind to special situations

این از نظر فکری محرک، پاداش و چالش برانگیز است
- اما نه برای رشته های اصلی CS - آنها می خواهند مشکلات را حل کنند و چیز هایی بسازند
طراحان آزمون های مبتنی بر انسان به صراحت تلاش می کنند تا تست های استرس را بیابند،
- تست هایی که با گنجاندن مقادیر بسیار بزرگ یا بسیار کوچک، مقادیر مرزی، مقادیر نامعتبر یا سایر مقادیری
که نرم افزار ممکن است در رفتار معمولی انتظار نداشته باشد، به نرم افزار فشار وارد می کند.
- رویکردهای مبتنی بر معیار می توانند نسبت به موقعیت های خاص کور باشند

Comparison

- Many people think of **criteria-based test design** as being used for **unit testing** and **human-based test design** as being used for **system testing**.

بسیاری از مردم تصور می کنند که طراحی آزمون مبتنی بر معیار برای زمایش واحد و طراحی آزمایش مبتنی بر انسان برای آزمایش سیستم مورد استفاده قرار می گیرد. این یک تمایز مصنوعی است.

- This is an artificial distinction.
- When using **criteria**, a **graph** is just a graph and it does not matter if it started as a **control flow graph**, a **call graph**, or an **activity diagram**.
- Likewise, **human-based tests** can and should be used to **test individual methods** and **classes**.

به همین ترتیب، آزمونهای مبتنی بر انسان میتوانند و باید برای آزمایش متودها و کلاسها فردی مورد استفاده قرار گیرند.

هنگام استفاده از معیارها، یک نمودار فقط یک نمودار است و فرقی نمی کند که به عنوان یک نمودار جریان کنترلی، یک نمودار فراخوانی یا یک نمودار فعالیت شروع شده باشد.

The main point is that the approaches are complementary and we need both to fully test software.

نکته اصلی این است که رویکردها مکمل یکدیگر هستند و برای آزمایش کامل نرم افزار به هر دو نیاز داریم.

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
- Programming is out of reach for many domain experts

تست اتوماسیون
قدایر تست را در قالب اسکریپت های اجرایی بنویسیم و اجرا کنیم تا فعالیت
تست به شکل اتوماتیک انجام بشه.

بیشتر برنامه نویسی
میخاد.

چون کارشون خلاقانه
است.

برای متخصص های
اون حوزه هم ممکنه
خسته کننده باشه و
اینکه برنامه نویسی
بلد نیستند هم مشکله

این کمی کمتر فنی است
نیاز به دانش برنامه نویسی دارد
به نظریه بسیار کمی نیاز دارد
اغلب نیاز به راه حل برای مشکلات دشوار مربوط به قابلیت مشاهده و کنترل
می تواند برای طراحان آزمون خسته کننده باشد
برنامه نویسی برای بسیاری از کارشناسان حوزه دور از دسترس است

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**

تست ها را روی نرم افزار اجرا کنید و نتایج را ثبت کنید
که اگه روند تست کردن اتوماتیک باشه خیلی ساده است

اگر تست ها به خوبی خودکار شوند، این آسان و بی اهمیت است
به مهارت های اولیه کامپیوتر نیاز دارد
- کارآموزان

- کارکنان بدون سابقه فنی
درخواست از طراحان آزمون واجد شرایط برای اجرای تست ها راهی
مطمئن برای مقاعده کردن آنها به جستجوی شغل توسعه است

3. Test Execution

- 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

رایی مثل، اگر تستهای رابط کاربری گرافیکی به خوبی خودکار نباشند، این کار به کار دستی زیادی نیاز دارد

مجریان آزمون باید در حسابداری بسیار دقیق عمل کنند

4. Test Evaluation

Evaluate results of testing, report to developers

■ This is **much harder** than it may seem

نتایج آزمایش را ارزیابی کنید، به توسعه دهنگان گزارش دهید.

■ Requires **knowledge** of :

- Domain
- Testing
- User interfaces and psychology

دانش خود نرم افزار
نیازه چون باید بدانیم
چه انتظارهایی از
نرم افزار داریم.

سخت تر از چیزی که
تصور میشه چون
تحلیل میخاد

این بسیار سخت تر از آن چیزی است که
ممکن است به نظر بررسدا
یاز به دانش:

- دامنه
- آزمایش کردن
- رابط کاربری و روانشناسی
- معمولاً تقریباً به هیچ CS سنتی نیاز ندارد

■ 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

– داشتن پیشینه در حوزه نرم افزار ضروری است
– پیشینه تجربی بسیار مفید است (زیست شناسی، روانشناسی، ...)
– پیشینه منطقی بسیار مفید است (حقوق، فلسفه، ریاضی، ...)

Other Activities

- **Test management** : Sets policy, organizes team, interfaces with development, chooses criteria, decides how much automation is needed,

مدیریت تست: خطمشی را تنظیم میکند، تیم را سازماندهی میکند، با توسعه ارتباط برقرار میکند، معیارها را انتخاب میکند، تصمیم میگیرد که چقدر اتوماسیون مورد نیاز است، ...

- **Test maintenance** : Save tests for reuse as software evolves

چه زمانی تست را
متوقف کنیم؟

تست نگهداری

: تست ها را برای استفاده مجدد با تکامل نرم افزار ذخیره کنید
- نیاز به همکاری طراحان تست و اتوماسیون دارد

- 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

- تصمیم گیری در مورد زمان کوتاه کردن
مجموعه تست حدی سیاست و تا حدی فنی
است - و به طور کلی، بسیار سخت است!

- **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

مستندسازی تست
منطق پشت هر تست
باید مشخص شه

- تست ها باید در کنترل پیکربندی قرار گیرند
- مستندات آزمون: همه طرفین شرکت می کنند
- هر آزمون باید "چرا" را مستند کند - معیار و
ازامات آزمون برآورده شده است یا منطقی برای
تست های طراحی شده توسط انسان

هر خطابی در طول فرایند اطمینان حاصل کنید
از قابلیت ردیابی در تست های خودکار نگه دارید

کدام ماذول است?
برای اسناد را در تست های خودکار نگه دارید

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

اتوماسیون بهبود یافته تعداد مجریان آزمون را کاهش می دهد
از نظر تئوری به صفر ... اما در عمل نه

■ 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

یک طراح آزمون واحد شرایط از انجام وظایف دیگر خسته
می شود و به دنبال شغلی در حال توسعه می گردد

- A qualified test evaluator will not understand the benefits of test criteria

یک ارزیاب واجد شرایط آزمون مزایای معیارهای آزمون را درک
نخواهد کرد

■ 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

چهار فعالیت تست کاملاً متفاوت هستند

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

چون میخان نتایج را
تحلیل کنند.

بسیاری از تیم های تست از افراد یکسان برای هر چهار فعالیت استفاده می کنند!!

ارزیابهای آزمون دانش حوزه را دارند، بنابراین باید
آزاد باشند تا تستهایی را اضافه کنند که فرآیندهای
مهندسی «کور» به آن فکر نمیکنند.

Applying Test Activities

بکارگیری فعالیت های آزمایشی
برای استفاده مؤثر از افراد خود و آزمایش مؤثر، به فرآیندی نیاز داریم که به طراحان
آزمون اجازه دهد سطح انتزاع خود را بالا ببرند.

To use our people effectively

and to test efficiently

we need a process that

lets test designers

raise their level of abstraction

Using MDTD in Practice

این رویکرد به یک طراح آزمون اجازه می دهد تا حساب را انجام دهد

- 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

طراح تست، عملیات
منطقی و ریاضی را
انجام میده. تا تست
اینپوت ها تولید بشن

سپس تسترها و برنامه نویسان سنتی می توانند قسمت های خود را انجام دهند
یافتن مقادیر
- تست ها را خودکار کنید
- تست ها را اجرا کنید
- آزمون ها را ارزیابی کنید

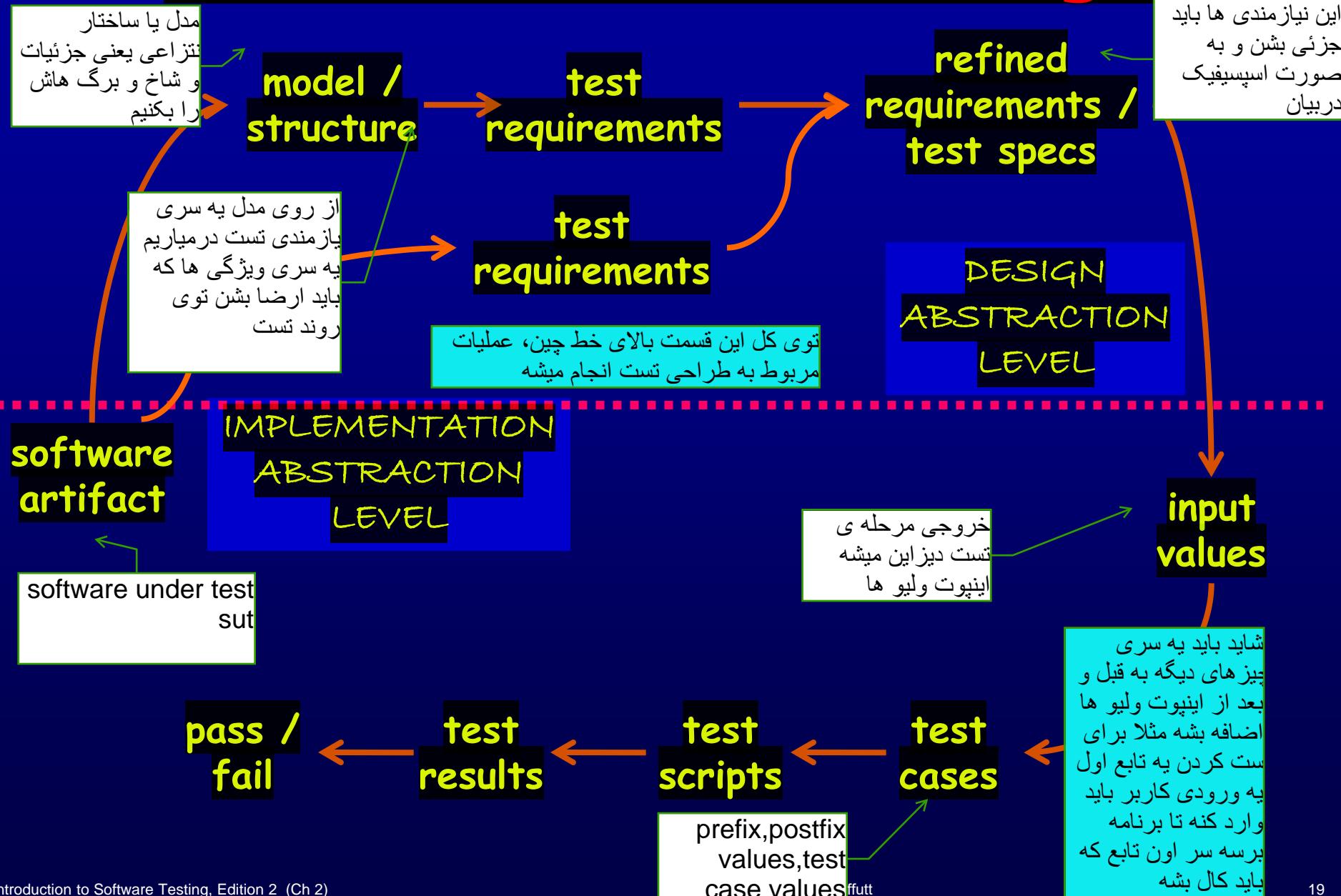
- Just like in traditional engineering ... an engineer constructs models with calculus, then gives direction to carpenters, electricians, technicians, ...

درست مانند مهندسی سنتی، یک مهندس با حساب دیفرانسیل و انتگرال مدل
ها را می سازد، سپس به نجار، برق، تکنسین ها جهت می دهد.

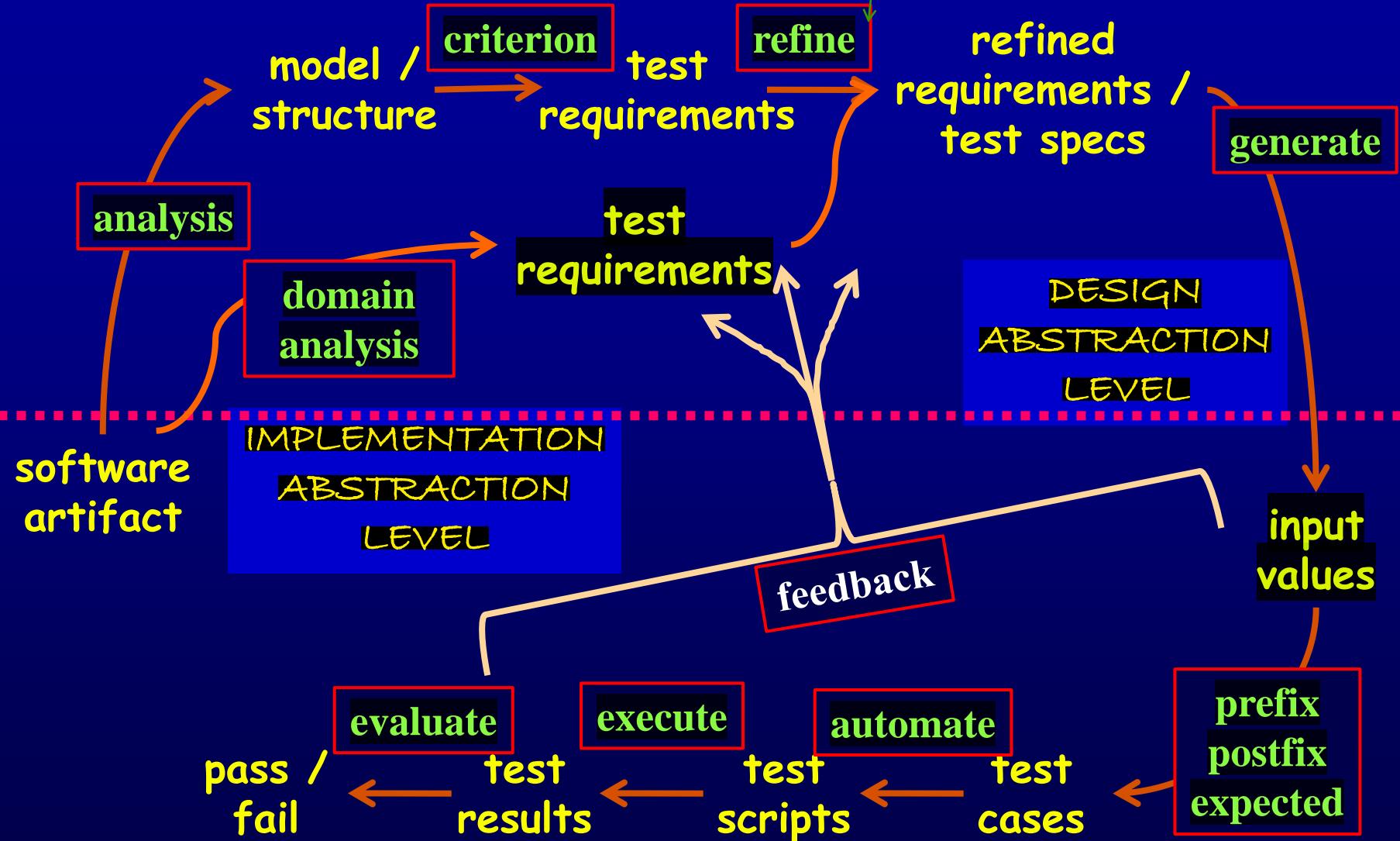
Test designers become technical experts

طراحان آزمون به کارشناسان فنی تبدیل می شوند

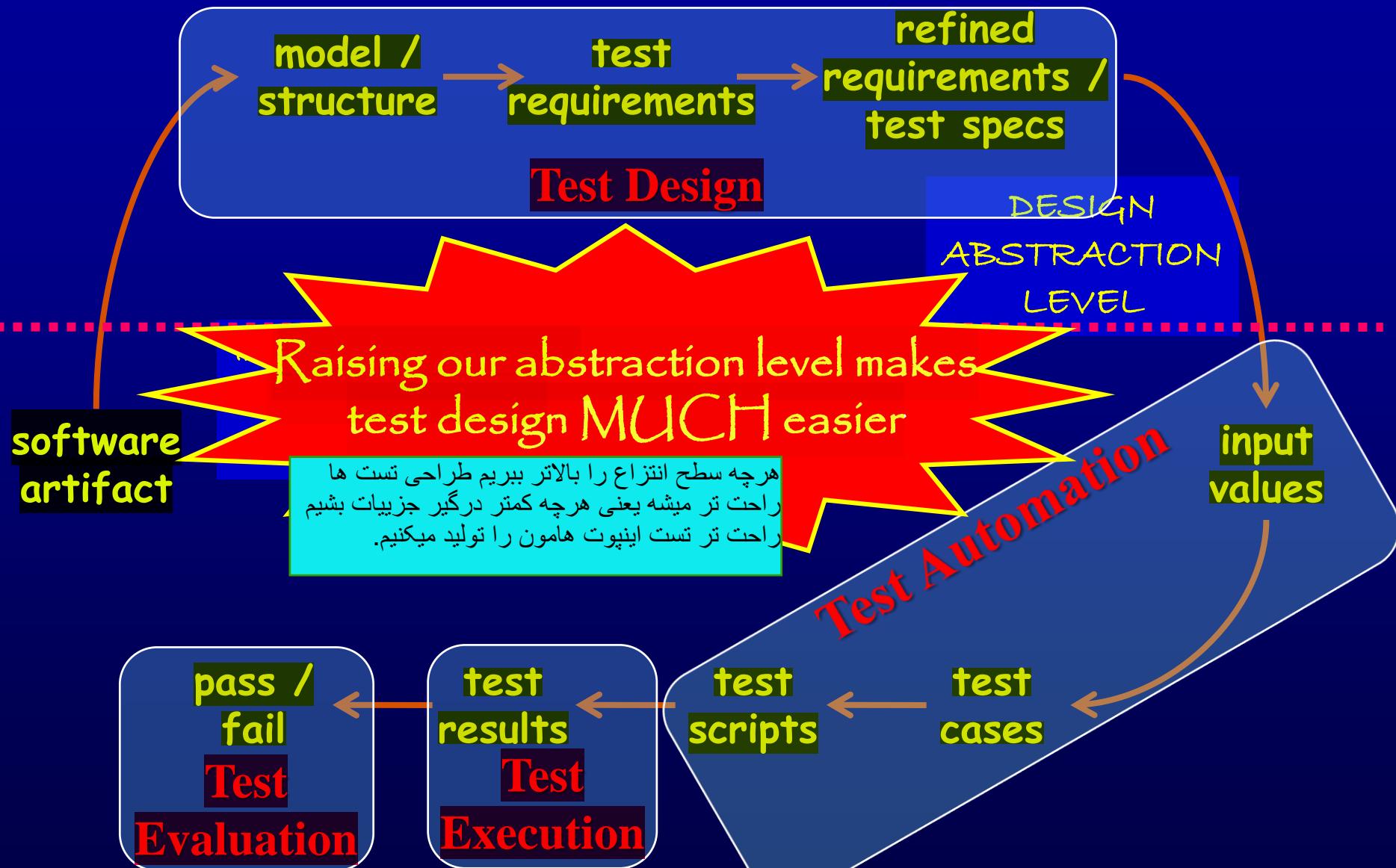
Model-Driven Test Design



Model-Driven Test Design – Steps



Model-Driven Test Design–Activities



Small Illustrative Example

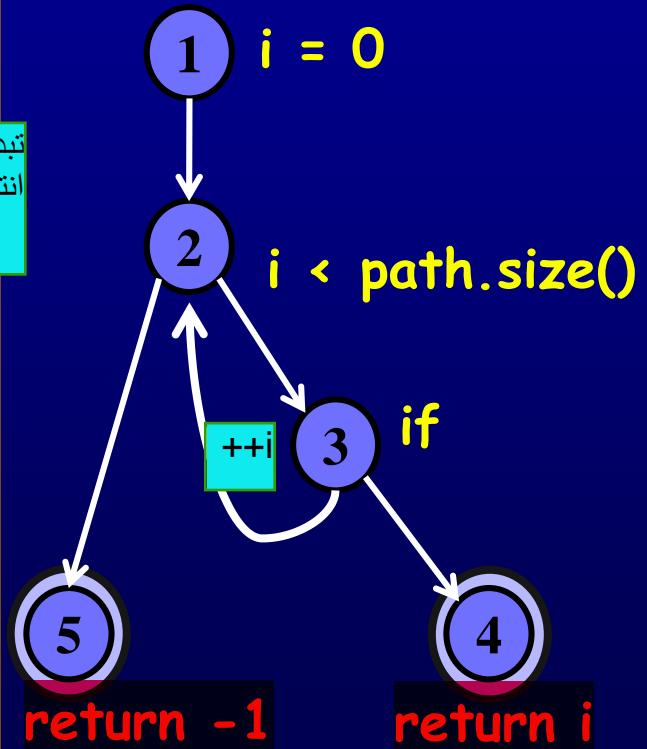
Software Artifact : Java Method

```
/**  
 * 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;  
}
```

تبدیل این متد به یک مدل
انتزاعی مثل گراف

گرافی که ترتیب اجرای
هر کدام از استیتمنت ها
را کنترل میکنه یک
گراف جهت دار است.

Control Flow Graph



با ریکوایرمنت ها و Edge pair coverage ای که داریم در نهایت یه مجموعه ای از تست اینپوت ها را تولید میکنیم. یا مسیر هایی را توانی گراف در میاریم که این TR ها را ارضاء کنند.

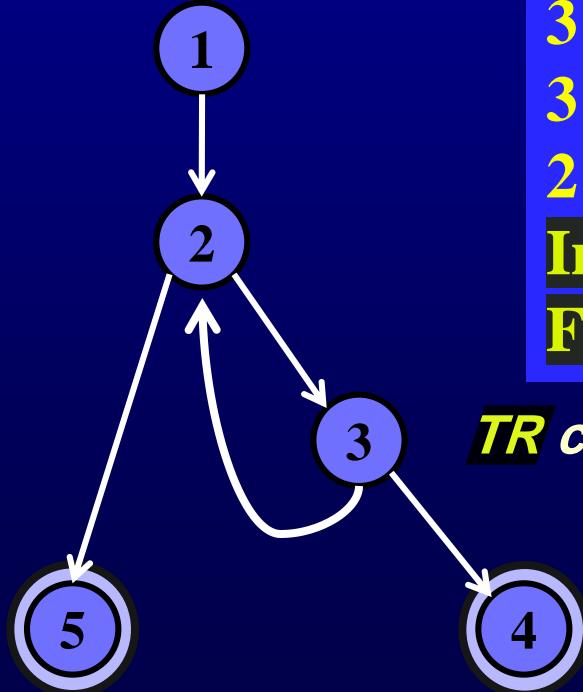
یه قانون است که میگه ریکوایرمنت های ما باید شامل همه مسیر هایی با حداقل طول ۲ باشند

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]

TR contains each reachable path of length up to 2 in graph G

Find values ...

Introduction to Software Testing (2nd edition)

Chapter 4

Putting Testing First

Paul Ammann & Jeff Offutt

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

August 2014

Testing has evolved from afterthought to a central activity in certain development methods

If high-quality testing is not centrally and deeply embedded in your development process, your project is at high risk for failure.

تست کردن از بعد فکری به یک فعالیت مرکزی در روشهای توسعه خاص تبدیل شده است
اگر تست با کیفیت بالا به طور مرکز و عمیق در فرآیند توسعه شما تعییه نشده باشد، پروژه شما در معرض خطر بالایی برای شکست قرار دارد.

The Increased Emphasis on Testing

■ Philosophy of traditional software development methods

- Upfront analysis
- Extensive modeling
- Reveal problems as early as possible

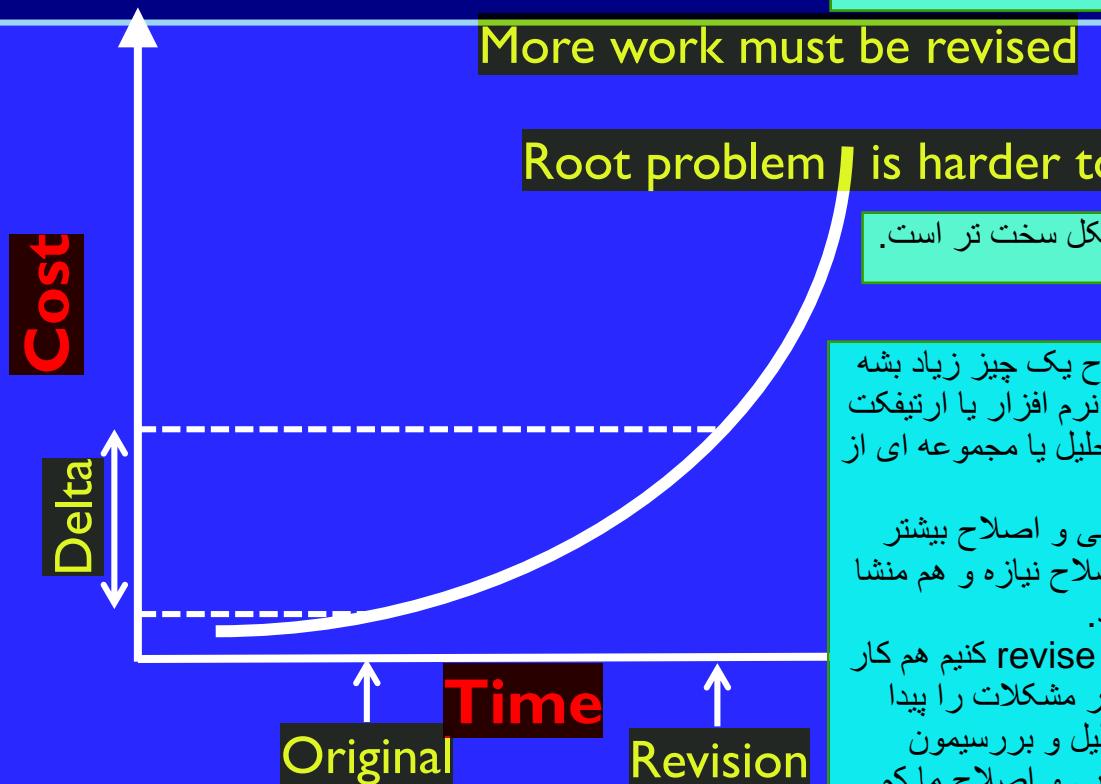
فلسفه روشهای سنتی توسعه نرم افزار

- تحلیل اولیه
- مدل سازی گستردگی
- مشکلات را در اسرع وقت آشکار کنید

اما این روند در روشهای سنتی نبوده چون
ونها revise س ماکس زمان بین تولید و
بین هر کدام از ارتیفیکت های نرم افزاری و
اون زمانی که محصول تحويل داده میشه

کارهای بیشتری باید اصلاح شود

فلسفه‌ی پشت این کارشون این
بوده که تحلیل‌ها مفصل‌انجام
پیشه و مدلینگ‌ها سنگین‌هستند
وسعی میکردن با تحلیل و مدل
سازی‌های اول‌کار، در حد
ممکن و هرچه سریع‌تر
مشکلات را شناسایی کنند.



هرچه زمان بین تولید و اصلاح یک چیز زیاد بشه
عنی دلتا زیاد بشه => مژول نرم افزار یا ارتیفیکت
میتواند مدل یا سورس کد یا تحلیل یا مجموعه ای از
تصمیم‌گیری‌ها باشه
هرچه زمان بین تولید و بررسی و اصلاح بیشتر
باشه هم کار بیشتری برای اصلاح نیازه و هم منشا
مشکلات سخت‌تر پیدا میشوند.
هرچه زودتر یک ارتیفیکتی را revise کنیم هم کار
کمتری انجام میدیم هم سریع‌تر مشکلات را پیدا
میکنیم و هم اینکه اسکوپ تحلیل و بررسی‌مون
کوچکتر است. درنتیجه خطایابی و اصلاح ما کم
هزینه‌تر میشه

The primary reason for the ever-increasing cost

دلیل اصلی افزایش روزافزون هزینه

- Additional work is invested that depends on the original decision, and this work must also be revised if the original decision is revised.
- A secondary problem is that as the software grows it gets harder to find the root cause of failures.

کار اضافی سرمایه‌گذاری می‌شود که بستگی به تصمیم اصلی دارد و در صورت تجدید نظر در تصمیم اولیه باید این کار نیز تجدید نظر شود.
یک مشکل ثانویه این است که با رشد نرم افزار، یافتن علت اصلی خرابی‌ها سخت‌تر می‌شود.

شتریان را از تغییر کردن نظراتشون نمیتوانید منع کنید پس همواره ممکن است اون نیازمندی‌ها تغییر کنند پس اینکه بباییم تغییر را حذف کنیم درست نیست چون شرایط طوری است که همواره امکان تغییر هست.

Traditional Assumptions

1. مدل سازی و تجزیه و تحلیل می تواند مشکلات بالقوه را در مراحل اولیه توسعه شناسایی کند.

I. Modeling and analysis can identify potential problems early in development

2. صرفه جویی در منحنی هزینه تغییر، هزینه مدل سازی و تجزیه و تحلیل را در طول عمر پروژه توجیه می کند

2. Savings implied by the cost-of-change curve justify the cost of modeling and analysis over the life of the project

■ These are true if requirements are always complete and current.

داریم هزینه ی تغییر را مینیمیم میکنیم یعنی نمیپذیریم که تغییری رخ بده

به جای پذیرفتن تغییر در طول فرایند، داریم روی مدل سازی سنگین وقت میگذاریم این فرضیات درست هستند به شرطی که نیازمندی ها واضح و کامل و شفاف باشند و ابهامی نداشته باشند

■ But those annoying customers keep changing their minds!

ادم ها در تقریب زدن کارها خوب هستند ولی در دقیق بیان کردن نه

– Humans are naturally good at approximating

– But pretty bad at perfecting

ادم ها بهتر میتوانند اول کار کلیات را بگن بعد هرچه کار جلو میره وارد جزئیات هم میشن

اگر الزامات همیشه کامل و جاری باشند، اینها درست هستند. اما آن مشتریان مزاحم مدام نظر خود را تغییر می دهند! انسان ها به طور طبیعی در تقریب خوب هستند. اما در کمال کردن بسیار بد است

■ These two assumptions have made software engineering frustrating and difficult for decades

این دو فرض، مهندسی نرم افزار را برای چندین دهه خسته کننده و دشوار کرده است

Thus, agile methods ...

Key end result of Agile methods

- Working software
- Responsiveness to change
- Effective development teams
- Happy customers.

تمرکز متد های اجایل

نتیجه نهایی کلیدی روش های چابک
نرم افزار کاری
پاسخگویی به تغییر
تیم های توسعه موثر
مشتریان خوشحال

در تعامل با مشتری هستند و مشتری ها را میتوانند جزو تیم
خودشون بیارن

چرا چاپک باشیم؟ روش های چاپک با تشخیص اینکه هیچ یک از این فرض ها برای بسیاری از پروژه های نرم افزاری فعلی معتبر نیستند شروع می شوند.

- مهندسان نرم افزار در توسعه نیازمندی ها خوب نیستند
- ما تغییرات زیادی را پیش بینی نمی کنیم.

Why Be Agile ?

■ Agile methods start by recognizing that neither assumption is valid for many current software projects

- Software engineers are not good at developing requirements
- We do not anticipate many changes
- Many of the changes we do anticipate are not needed

بسیاری از تغییراتی که ما پیش‌بینی می‌کنیم لازم نیست.

■ Requirements (and other “non-executable artifacts”) tend to go out of date very quickly

- We seldom take time to update them
- Many current software projects change continuously

نیازمندیها و سایر مصنوعات غیرقابل اجرا خیلی زود قدیمی می‌شوند.

- ما به ندرت برای به روز رسانی آنها وقت می گذاریم

■ Agile methods expect software to start small and evolve over time

- Embraces software evolution instead of fighting it

بسیاری از پروژه های نرم افزاری فعلی به طور مداوم تغییر می کنند

تکامل نرم افزار را به جای مبارزه با آن در آغوش می گیرد

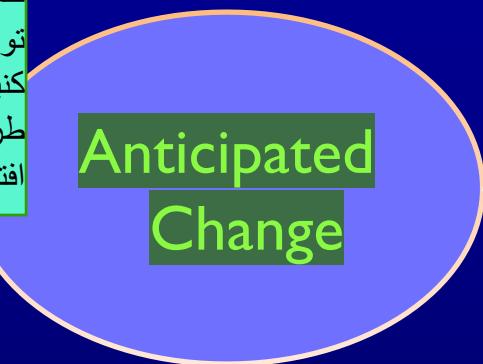
روشهای چاپک انتظار دارند که نرم افزار کوچک شروع شود و در طول زمان تکامل یابد

Supporting Evolutionary Design

Traditional design advice says to **anticipate changes**

Designers often anticipate changes that don't happen

حمایت از طراحی تکاملی
توصیه های طراحی سنتی می گوید که تغییرات را پیش بینی کنند
طراحان اغلب تغییراتی را پیش بینی می کنند که اتفاق نمی افتد



Both anticipated and unanticipated changes affect design

تغییرات پیش بینی شده و پیش بینی نشده هر دو بر طراحی تأثیر می گذارند

Agile methods

- An agile principle that goes directly to the heart of the both assumptions is “**You ain’t gonna need it!**”, or **YAGNI**.

روش های چابک
یک اصل چابک که مستقیماً به قلب هر دو فرض می رود این است که «به آن نیاز نخواهی داشت! یا YAGNI».
- The **YAGNI principle** states that **traditional planning** is fraught precisely because **predicting system evolution** is **fundamentally hard**, and hence expected savings from the **cost-of-change curve** do not materialize. تحقیق پاید
- Instead, agile methods **defer** many **design** and **analysis decisions** and **focus** on **creating a running system** that does “**something**” as early as possible.

در عوض، روش های چابک بسیاری از تصمیمگیری های طراحی و تحلیل را به تعویق میاندازند.
برایجاد یک سیستم در حال اجرا متمرکز میشوند که کاری را در اسرع وقت انجام دهد.

اصل YAGNI بیان میکند که برنامه ریزی سنتی دقیقاً به این دلیل که پیشビینی تکامل سیستم اساساً سخت است، پرمبالغه است و از این رو صرفه جوییهای مورد انتظار از منحنی هزینه تغییر محقق نمیشود.

The Test Harness as Guardian (4.2)

مهر تست به عنوان نگهبان

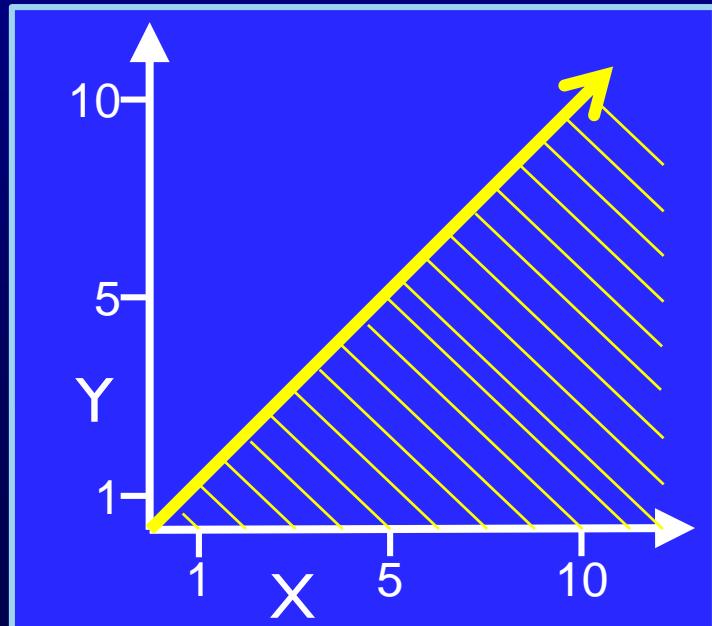
What is Correctness ?

Traditional Correctness

مفهوم جهانی

(Universal)

$$\forall x, y, x \geq y$$



Agile Correctness

(Existential)

درستی یا correctness یه مفهوم
غیرقابل دستیابی است

تعریف وجودی

{ $(1, 1) \rightarrow T$
 $(1, 0) \rightarrow T$
 $(0, 1) \rightarrow F$
 $(10, 5) \rightarrow T$
 $(10, 12) \rightarrow F$ }

Agile methods in general, and **test-driven development** in particular,
take a **novel**, and somewhat **more restricted**,
view of **correctness**.

روش های چابک به طور کلی، و توسعه مبتنی بر تست طور خاص
دیدگاهی بدیع و تا حدودی محدودتر از correctness دارند.

به جای درنظر گرفتن کل ورودی های ممکن یه
ای را درنظر میگیره بعد میگه اگه روی این
زیرمجموعه داره درست جواب میده برای من کافیه.

A Limited View of Correctness

- In traditional methods, we try to define all correct behavior completely, at the beginning

در روش‌های سنتی سعی می‌کنیم در ابتدا همه رفتارهای صحیح را به طور کامل تعریف کنیم
- صحت چیست؟

- What is correctness?
- Does “correctness” mean anything in large engineering products?
- People are VERY BAD at completely defining correctness

مردم در تعریف کامل درستی بسیار بد هستند

- In agile methods, we redefine correctness to be relative to a specific set of tests

در روش‌های چابک، ما صحت را نسبت به مجموعه‌های از آزمونها دوباره تعریف می‌کنیم

- If the software behaves correctly on the tests, it is “correct”
- Instead of defining all behaviors, we demonstrate some behaviors
- Mathematicians may be disappointed at the lack of completeness

- به جای تعریف همه رفتارها، برخی رفتارها را نشان می‌دهیم
- اگر نرم افزار در تست‌ها به درستی رفتار کند، "درست" است.

But software engineers ain't mathematicians!

- ممکن است ریاضیدانان از عدم کامل بودن نامید شوند
اما مهندسان نرم افزار ریاضیدان نیستند!

به صورت نسبی
تعریف می‌کنیم.

Test Harnesses Verify Correctness

control and make use of (natural resources)

A *test harness* runs all automated tests efficiently and reports results to the developers

یک مهار تست تمام تست های خودکار را به طور موثر اجرا می کند و نتایج را به توسعه دهنگان گزارش می دهد

■ Tests must be automated

– Test automation is a prerequisite to test driven development

■ Every test must include a test oracle that can evaluate whether that test executed correctly

آزمون ها باید خودکار باشند
اتوماسیون تست یک پیش نیاز برای توسعه آزمایش محور است

■ The tests replace the requirements

یه مژول نرم افزاری است

■ Tests must be high quality and must run quickly

■ We run tests every time we make a change to the software

هر آزمون باید شامل یک اوراکل آزمایشی باشد که بتواند آزمایی کند که آیا آن تست به درستی اجرا شده است یا خیر.
آزمون ها جایگزین الزامات می شوند.

ما وقتی میگیم برنامه ای میخاییم که یک تست را پاس کنے یعنی داریم میگیم فلان قابلیت را میخاییم داشته باشه نرم افزارمون

تست ها باید کیفیت بالایی داشته باشند و باید سریع اجرا شوند.
هر بار که تغییری در نرم افزار ایجاد می کنیم، تست هایی را اجرا می کنیم

Test driven development

مفهوم محوریت داشتن تست
ست یه مفهوم مرکزی و مهم در فرایند توسعه
نرم افزار میشه

TDD

- From the developer's perspective, testing is the central activity in development.

از دیدگاه توسعه‌دهنده، تست فعالیت اصلی در توسعه است.

- Good design still matters in TDD, it simply occupies a different, and niche in the development cycle.

طراحی خوب هنوز در TDD اهمیت دارد، به سادگی در چرخه توسعه جایگاه مقاوت و جایگاهی را اشغال می‌کند.

Consequence of the test-harness-

فلسفه اعتقادی است که در میان بسیاری از توسعه دهنگان
چاک مشترک است.

اسناد غیرقابل اجرا به طور بالقوه گمراه کننده هستند.

- A philosophy is a belief shared by many agile developers.
 - Non-executable documents are potentially misleading.
 - While everyone agrees that a non-executable document that correctly describes a software artifact is helpful, it is also true that a non-executable document that incorrectly describes a software artifact is a liability.

مسئولیت
obstacle
دردسر و مساله

در حالی که همه موافق هستند که یک سند غیرقابل اجرا که به درستی یک مصنوع نرم افزاری را توصیف می کند مفید است، این نیز درست است که یک سند غیرقابل اجرا که به اشتباه یک مصنوع نرم افزار را توصیف می کند یک تعهد و مسئولیت است.

یعنی یه working software ارائه بدیم که هم قابل ارزیابی است هم رفتار نرم افزار را نشان میده که قابل قبول است یا نه با رویکردهای TDD قابل ارزیابی است و میتوانیم بفهمیم که نیاز ها را میتواند ارضاء کنه یا نه؟

Agile methods attempt to make **executable artifacts** to **satisfy needs**, in traditional software engineering, were satisfied by **non-executable artifacts**.

روش های چاپک تلاش می کنند تا مصنوعات اجرایی برای برآوردن نیازها بسازند.
در مهندسی نرم افزار سنتی، با مصنوعات غیرقابل اجرا ارضاء شدند.

Definition of success

- Traditional development defines success as “On time and on budget,”

تعریف موفقیت توسعه سنتی موفقیت را اینگونه تعریف می کند: به موقع و با بودجه،
ولی اینکه خروجی کارچقدر از دید مشتری مناسب و خوب بوده قابل بحث نبوده

- Agile methods aim first for having **something executable** available from the **very beginning** of development and second producing a **different**, and presumably **better**, product than the one **originally envisioned**.

هدف روش‌های چابک او لا داشتن یک چیز قابل اجرا در دسترس از همان ابتدای توسعه و دوم تولید محصولی متفاوت و احتمالاً بهتر از آنچه در ابتدا تصور می‌شد.

How to make Agile work

- Test cases need to be of high quality.
- Test processes need to be efficient.
- Use of test automation is necessary, but not sufficient.

چون تست ها دارن جای requirements میگیرن.

باید بتوانیم با حداقل
تست ها بیشترین
Failure ها را دربیاریم.

چگونه Agile را عملی کنیم
موارد تست باید از کیفیت بالایی برخوردار باشند.
فرآیندهای تست باید کارآمد باشند.

استفاده از اتوماسیون تست ضروری است، اما کافی نیست.

Continuous Integration

- Agile methods work best when the current version of the software can be run against all tests at any time

روشهای چابک زمانی بهترین کار را دارند که نسخه فعلی نرمافزار را بتوان در هر زمان در مقابل همه آزمایشها اجرا کرد

A *continuous integration server* rebuilds the system, returns, and reverifies tests whenever *any update* is checked into the repository

- Mistakes are caught earlier

یک سرور یکپارچه سازی پیوسته سیستم را بازسازی می کند، هر زمان که هر به روز رسانی در مخزن بررسی شود، آزمایش ها را بر می گرداند و دوباره تأیید می کند.

- Other developers are aware of changes early

اشتباهات زودتر تشخیص داده شوند.

- The rebuild and reverify must happen *as soon as possible*

– is an important part of the test harness.

- بخش مهمی از مهار تست است.

سایر توسعه دهندگان زودتر از تغییرات آگاه هستند

بازسازی و تایید مجدد باید در اسرع وقت اتفاق بیفتد.

A *continuous integration server* doesn't just run tests, it decides if a modified system is still correct

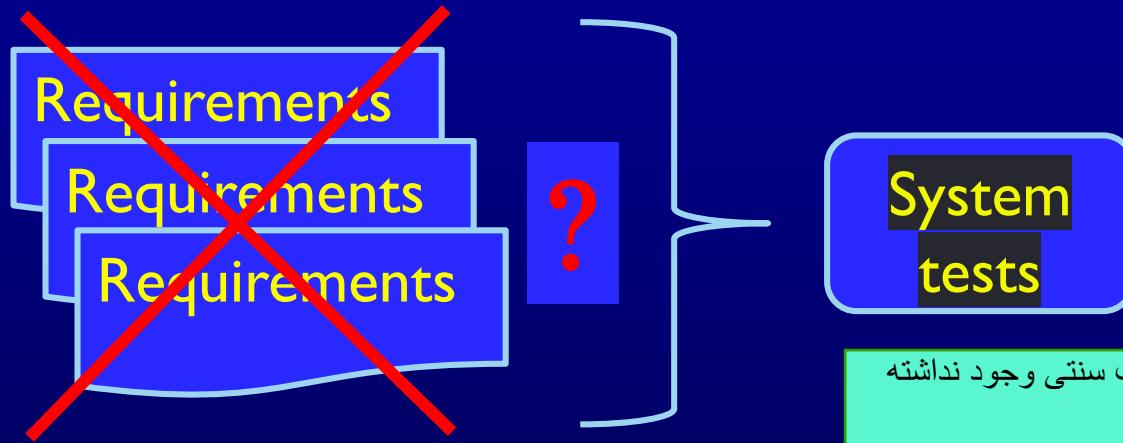
اصلاح خطاهای در اسرع وقت انجام می شده

سرور یکپارچه سازی پیوسته فقط آزمایش ها را اجرا نمی کند، بلکه تصمیم می گیرد که آیا سیستم اصلاح شده هنوز درست است یا خیر

System Tests in Agile Methods(I)

Traditional testers often design system tests **from requirements**

تسترهای سنتی اغلب تست های سیستم را
بر اساس الزامات طراحی می کنند



اما ... اگر اسناد الزامات سنتی وجود نداشته باشد چه؟

But ... what if there are **no traditional requirements** documents ?

User Stories

جای requirements
را میگیره.

A *user story* is a few sentences that captures what a user will do with the software

یک داستان کاربر چند جمله است که نشان می دهد کاربر با نرم افزار چه خواهد کرد

بوزر استوری میگه انتظار
کاربر از سیستم چیه؟

Withdraw money from
checking account

Support technician sees
customer's history on
demand

Agent sees a list of today's
interview applicants

- به زبان کاربر نهایی
- معمولاً در مقیاس کوچک با جزئیات کم
- باگانی نشده است

- In the **language of the end user**
- Usually **small** in scale with **few details**
- **Not archived**

System Tests in Agile Methods(II)

- Amount of **effort** required to **implement** the **functionality** captured by the system test might be quite large.

مقدار تلاش مورد نیاز برای اجرای عملکرد ثبت شده توسط تست سیستم ممکن است بسیار زیاد باشد.

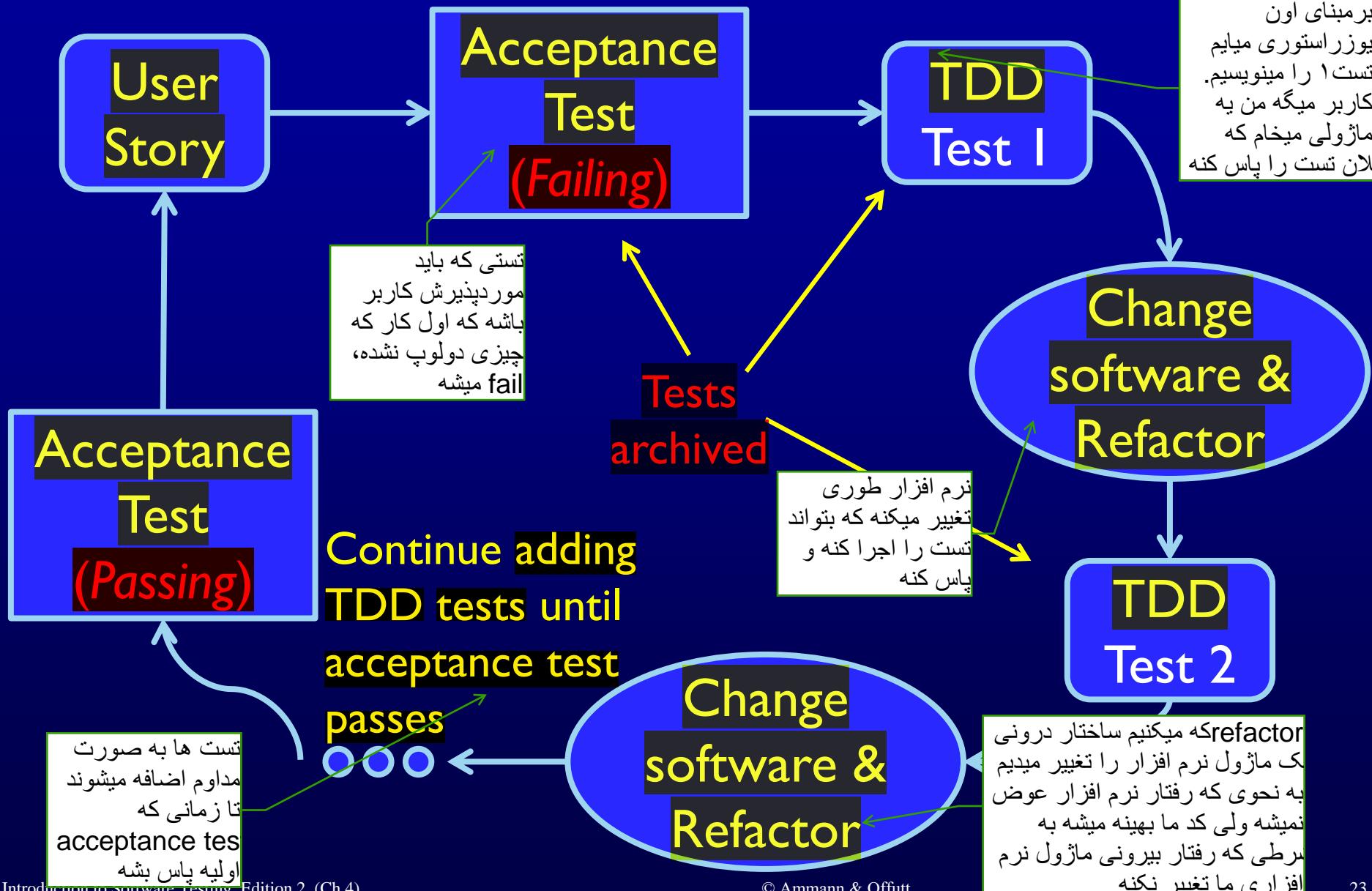
- How do you **run a test against a system** that not only isn't yet built, but cannot even be built in a **short time-frame**?

چگونه می توان آزمایشی را بر روی سیستمی اجرا کرد که نه تنها هنوز ساخته نشده است، بلکه حتی نمی تواند در یک بازه زمانی کوتاه ساخته شود؟

Agile methods place a premium on having a **test harness** continuously verify the system.

در اجایل به جای اینکه بگذاریم سیستم کامل دلولپ بشه و بعد تست را انجام بدیم، میایم گام به گام تست را انجام میدیم
چون هرچه که سیستم بزرگتر بشه، تست کردنش هم سخت تر و هزینه برتر میشه.
پس به صورت مداوم و **continuously** سیستم را Verify میکنیم.

Acceptance Tests in Agile Methods



An Example

- **User story:** “Support technician sees customer’s history on demand.”
داستان کاربر: “تکنسین پشتیبانی تاریخچه مشتری را در صورت تقاضا می بیند.”
- The story does not include implementation details, and is not specific enough to run as a test case.
داستان شامل جزئیات پیاده سازی نمی شود و به اندازه کافی مشخص نیست که به عنوان یک مورد آزمایشی اجرا شود.
- This user story might have a happy path test where a technician fields a call from a specific, existing user. The test passes if that specific user’s history is displayed on demand.
در صورتی که تاریخچه آن کاربر خاص در صورت تقاضا نمایش داده شود، آزمایش انجام می شود و تست pass می شود.
این یوزر استوری ممکن است یک تست مسیر خوشحال کننده داشته باشد که در آن یک تکنسین یک تماس از یک کاربر خاص و موجود را ارسال می کند.
- A different test might involve a new user; this test passes if the technician is informed that the user does not have a history.
یک آزمایش متفاوت ممکن است شامل یک کاربر جدید باشد. در صورتی که به تکنسین اطلاع داده شود که کاربر سابقه ندارد، این تست انجام می شود.
- These tests provide specific, concrete guidance to developers as to exactly what functionality needs to be implemented.

این تستها راهنمایی هایی مشخصی را برای توسعه دهنگان ارائه میکنند که دقیقاً چه عملکردی باید پیاده سازی شود.

تستهای با کیفیت بالا برای موفقیت پروژه‌های چابک اهمیت زیادی دارند.

High-quality tests
are of central importance to
whether agile projects succeed.

موفقیت پروژه‌ی اجایل به کیفیت تست‌ها بستگی دارد چون انتظارات یا
همان requirement‌ها در قالب تست‌ها تعریف می‌شوند.
پس هرچه کیفیت تست‌ها بره بالا پروژه موفق‌تر می‌شوند.

Adding Tests to Existing Systems

■ Most of today's software is legacy

- No legacy tests
- Legacy requirements hopelessly outdated
- Designs, if they were ever written down, lost

بیشتر نرم افزارهای امروزی قدیمی هستند
- بدون تست میراث
- الزامات میراث به طرز نامیدکننده ای منسخ شده است
- طرح ها، اگر زمانی نوشته شده باشند، گم شده اند

■ Companies sometimes choose not to change software out of fear of failure

شرکت ها گاهی اوقات از ترس شکست نرم افزار را تغییر نمی دهند

How to apply TDD to legacy software with no tests?

چگونه می توان TDD را در نرم افزارهای قدیمی بدون تست اعمال کرد؟

■ Create an entire new test set? — too expensive!

■ Give up? — a mixed project is unmanageable

نه میشه پروژه جدیدی تعریف کرد و نه میشه کل سیستم را تست کنیم و
نه میشه از اون پروژه legacy منصرف شد چون ممکنه به پروژه ای
داشته باشیم که داره از کدهای legacy استفاده میکنه.

یک مجموعه آزمایشی جدید ایجاد کنید?
- بیش از حد گران!
تسليم شدن?
- یک پروژه مختلط غیر قابل مدیریت است

Incremental TDD

Needs to **incrementally** introduce test cases,
so that over time a system **safely** moves towards
both **new functionality**
and **new test cases** that **verify that functionality.**

TDD افزایشی

نیاز به معرفی تدریجی موارد آزمایشی یا همان تست کیس ها را دارد، به طوری که در طول زمان یک سیستم با خیال راحت به سمت عملکرد جدید و تست کیس های جدید که آن عملکرد را تأیید می کند حرکت کند.

تست کیس ها را مرحله به مرحله و
کوچیک کوچیک تولید میکنیم.
و همین طور که جلو میریم و
فانکشنالیتی های جدید به سیستم
اضافه میکنیم، برای اونها تست کیس
های جدید تعریف میکنیم

پس درین بهبود سیستم براش تست مینویسیم.

Incremental TDD

- When a **change is made**, add TDD tests for **just that change**
 - **Refactor:** is a way to **modify** (hopefully improving) the **structure of existing code without changing its behavior.**

: راهی برای اصلاح (امیدوارانه بهبود) ساختار کد موجود بدون تغییر رفتار آن است.
- As the project proceeds, the **collection of TDD tests continues to grow**

با ادامه پروژه، مجموعه تست های TDD به رشد خود ادامه می دهد
- Eventually the software will have **strong TDD tests**

در نهایت یه مجموعه تست خوبی داریم

The Testing Shortfall

■ Do TDD tests (acceptance or otherwise) test the software well?

آیا تست های TDD (قبولی یا غیر آن) نرم افزار را به خوبی تست می کنند؟

- Do the tests achieve good coverage on the code?
- Do the tests find most of the faults?
- If the software passes, should management feel confident the software is reliable?

ایا میتوانیم بگیم این تست های TDD از همه
تنبیه ای کدهامون را بررسی کردن؟ ایا میتوانیم
بگیم بیشترین فلت ها را برآمون در اورده؟
اگه نرم افزارمون با اون تست هایی که
جایگزین نیازمندی ها بودن و برمنای اونها ما
پیاده سازی را انجام دادیم، پاس شد ایا میتوانیم
بگیم نرم افزارمون reliable است؟
نهنهنه!!

NO!

- آیا تست ها پوشش خوبی روی کد دارند?
- آیا تست ها بیشتر ایرادات را پیدا می کنند؟
اگر نرم افزار قبول شود یعنی تست هاش پاس شود، آیا مدیریت
باید از قابل اعتماد بودن نرم افزار اطمینان داشته باشد؟
نهنهنه!!



Why Not?

■ Most agile tests focus on “*happy paths*”

– What should happen **under normal use**

■ They often **miss** things like

– Confused-user paths

– Creative-user paths

– Malicious-user paths

مسیرهایی که ما قبلاً
پیشبینی نکردیم

آنها اغلب چیزهایی مانند
- مسیرهای کاربر سردرگم
- مسیرهای کاربر خلاق
- مسیرهای کاربر مخرب
را از دست میدهند.

چرا که نه؟
اکثر تست های چاپک بر روی «مسیرهای شاد»
تمرکز دارند.

- در استفاده معمولی چه اتفاقی باید بیفتد

مسیرهای نامشخص

سناریو اصلی و
موفقیت امیز نرم
افزار

سناریوهای شکست هم طبق انتظارات
مشتری میریم جلو

The agile methods literature
does not give much guidance

ادبیات روش‌های چاپک راهنمایی زیادی نمیکند

ایا پس بررسی happy path ها کافی است یا نه؟

Design Good Tests

1. Use a human-based approach

- Create additional user stories that describe non-happy paths
- How do you know when you're finished?
- Some people are very good at this, some are bad, and it's hard to teach

رویکرد تجربی است و اموزشی نیست.

Part 2 of book ...

مدل سازی انجام می‌دیم.

از رویکرد انسان محور استفاده کنید

• داستان های کاربر اضافی ایجاد کنید که مسیرهای غیر شاد را توصیف می کند

• چگونه متوجه می شوید که کارتنان تمام شده است؟

• برخی از مردم در این کار بسیار خوب هستند، برخی بد هستند، و آموزش دادن آن دشوار است

از مدل سازی و معیارها استفاده کنید

• دامنه ورودی را برای طراحی تست ها مدل کنید رفتار نرم افزار را با نمودارها، منطق یا گرامرها مدل کنید

• حس کامل بودن

• آموزش بسیار ساده تر-مهندسی
• به داشت ریاضی گستته نیاز دارد

2. Use modeling and criteria

- Model the input domain to design tests
- Model software behavior with graphs, logic, or grammars
- A built-in sense of completion
- Much easier to teach—engineering
- Requires discrete math knowledge

شرط توقف معنا پیدا
میکنے یعنی معلومه
که کامل تست کردن
به چه معناست.

Summary

- More companies are **putting testing first**
- This can dramatically **decrease cost** and **increase quality**
- A **different view of “correctness”**
 - **Restricted but practical**
- Embraces **evolutionary design**
- TDD is definitely **not test automation**
 - Test automation is a **prerequisite** to TDD
- **Agile tests aren't enough**

در اجایل، نوع نگاه به درستی یا correctness را عوض میکنیم.

طراحی به صورت تکاملی جلو میره.

چون میتوانند happy path ها و برخی از سناریوهای شکست را تعریف کنند و نمیتوانیم مطمئن شیم که نرم افزار ما به طور کامل reliable است.

شرکت های بیشتری آزمایش را در اولویت قرار می دهند
این می تواند به طور چشمگیری هزینه را کاهش دهد و کیفیت
را افزایش دهد
دیدگاه منقاوت از صحت
- محدود اما کاربردی
از طراحی تکاملی استقبال می کند
قطعاً اتوماسیون تست نیست
- اتوماسیون تست پیش نیاز TDD است
تست های چابک کافی نیستند

Introduction to Software Testing (2nd edition) Chapter 5

Criteria-Based Test Design

Paul Ammann & Jeff Offutt

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

Abstraction

*should be used to handle complexity,
not to ignore it.*

انتزاع باید برای رسیدگی به پیچیدگی استفاده شود، نه برای
نادیده گرفتن آن.

Introduction

- The number of potential inputs for most programs is so large as to be effectively infinite and cannot be explicitly enumerated.

تعداد ورودی های بالقوه برای اکثر برنامه ها به قدری زیاد است که به طور موثر بی نهایت است و نمی توان به طور صریح آنها را برشمرد.

- This is where formal coverage criteria come in.

انجاست که معیار های پوشش رسمی وارد می شود.

- Since we cannot test with all inputs, coverage criteria are used to decide which test inputs to use.

از آنجایی که نمیتوانیم با همه ورودیها آزمایش کنیم، معیار های پوشش برای تصمیم گیری برای استفاده از ورودی های تست استفاده میشود.

Introduction(Cnt'd)

- The rationale behind coverage criteria is that they divide up the input space to maximize the number of faults found per test case.
- From a practical perspective, coverage criteria also provide useful rules for when to stop testing.

منطق پشت معیار های پوشش این است که آنها فضای ورودی را برای به حداقل رساندن تعداد خطاهای یافت شده در هر مورد آزمایشی تقسیم می کنند.

از منظر عملی، معیار های پوشش نیز قوانین مفیدی را برای زمان توقف آزمایش ارائه می کنند.

Changing Notions of Testing

- Old view focused on testing at each software development phase as being very different from other phases
 - Unit, module, integration, system .

تغییر مفاهیم تست
دیدگاه قدیمی بر آزمایش در هر مرحله توسعه نرم افزار متمرکز بود زیرا بسیار متفاوت از سایر مراحل است
– واحد، مازول، یکپارچه سازی، سیستم ...

- New view is in terms of structures and criteria
 - input space, graphs, logical expressions, syntax
- Test design is largely the same at each phase
 - Creating the model is different
 - Choosing values and automating the tests is different

دیدگاه جدید از نظر ساختارها و معیارها است
- فضای ورودی، نمودارها، عبارات منطقی، نحو طراحی آزمون تا حد زیادی در هر مرحله یکسان است
- ایجاد مدل متفاوت است
- انتخاب مقادیر و خودکار کردن تست ها متفاوت است

New : Test Coverage Criteria

A tester's job is simple :

کار تستر ساده است: یک مدل از نرم افزار را تعریف کنید، سپس راه هایی برای پوشش آن پیدا کنید

Define a model of the software, then find ways to cover it

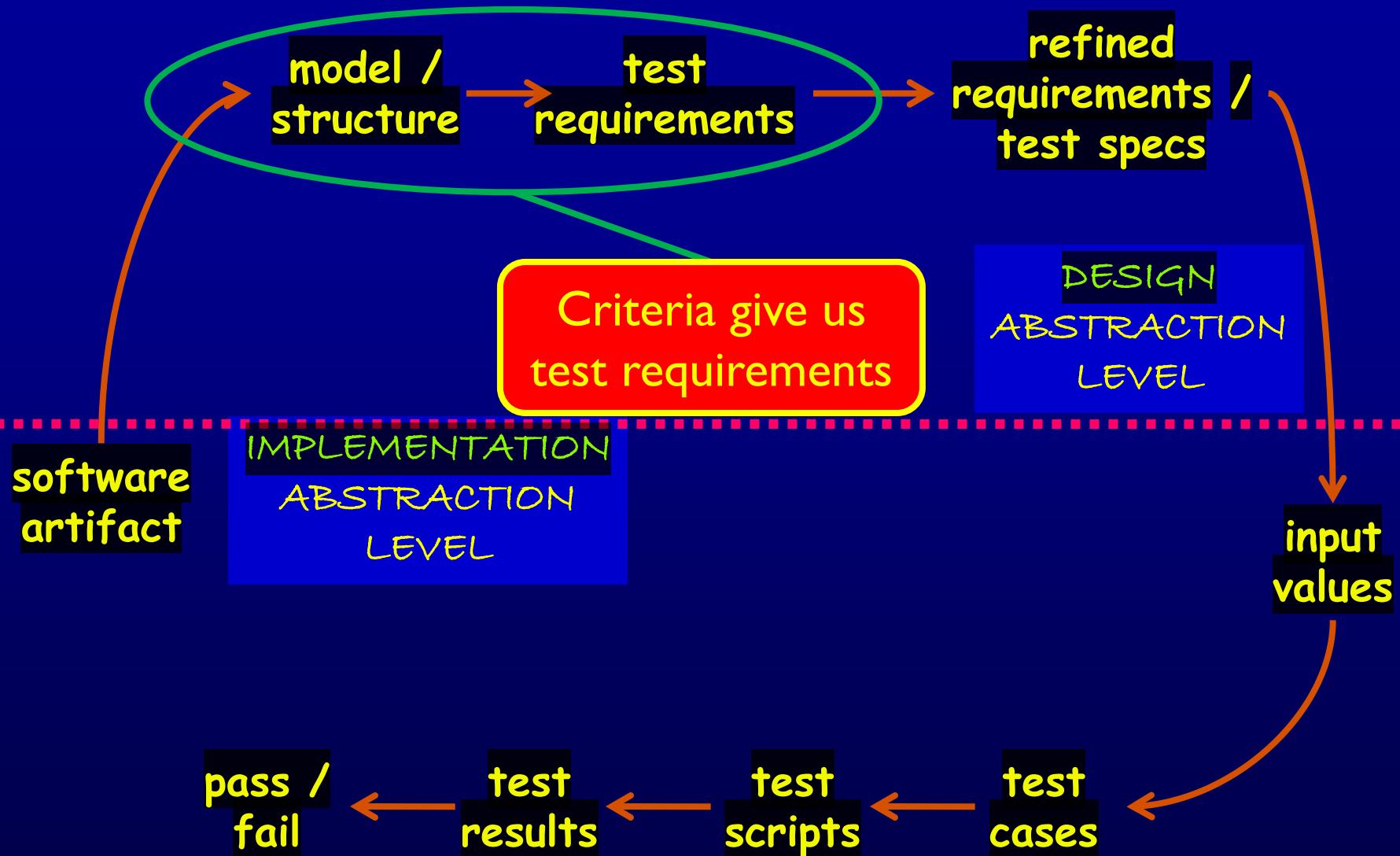
■ **Test Requirements** : A specific element of a software artifact that a test case must satisfy or cover

الزامات تست: یک عنصر خاص از یک مصنوع نرم افزاری که یک مورد آزمایشی باید آن را برآورده یا پوشش دهد
معیار پوشش: قاعده یا مجموعه ای از قوانینی که الزامات آزمون را بر یک مجموعه آزمایشی تحمیل می کند.

■ **Coverage Criterion** : A rule or collection of rules that impose test requirements on a test set

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

Model-Driven Test Design



Source of Structures

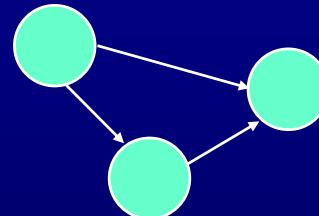
- These **structures** can be **extracted** from lots of software artifacts
 - Graphs can be extracted from UML use cases, finite state machines, source code, ...
 - Logical expressions can be extracted from decisions in program source, guards on transitions, conditionals in use cases, ...
- This is **not** the same as “*model-based testing*,” which derives tests from a model that describes some aspects of the system under test
 - The **model** usually describes part of the behavior
 - The **source** is explicitly not considered a model

Criteria Based on Structures

Structures : Four ways to model software

1. Input Domain
Characterization
(sets)

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}



2. Graphs

3. Logical Expressions

(not X or not Y) and A and B

4. Syntactic Structures
(grammars)

```
if (x > y)  
    z = x - y;  
else  
    z = 2 * x;
```

Example : Jelly Bean Coverage

Flavors :

1. Lemon
2. Pistachio
3. Cantaloupe
4. Pear
5. Tangerine
6. Apricot



Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

■ Possible coverage criteria :

1. Taste one jelly bean of each flavor
 - Set of Test Requirements: $TR=\{\text{Lemon, Pistachio, Cantaloupe, Pear, Tangerine, Apricot}\}$
2. Taste one jelly bean of each color
 - Set of Test Requirements: $TR=\{\text{Yellow, Green, Orange, White}\}$

Coverage

Given a set of test requirements TR for coverage criterion C , a test set T satisfies C coverage if and only if for every test requirement tr in TR , there is at least one test t in T such that t satisfies tr

با توجه به مجموعه‌ای از الزامات آزمون TR برای معیار پوشش C ، یک مجموعه آزمایشی T پوشش C را آورده می‌کند اگر و تنها اگر برای هر شرط آزمون tr در TR ، حداقل یک آزمون t در T وجود داشته باشد به طوری که tr را ارضاء نکند.

More Jelly Beans

T1 = { three Lemons, one Pistachio, two Cantaloupes, one Pear, one Tangerine, four Apricots }

■ Does **test set T1** satisfy the **flavor criterion** ?

- Set of Test Requirements: **TR={Lemon, Pistachio, Cantaloupe, Pear, Tangerine, Apricot}**

T2 = { One Lemon, two Pistachios, one Pear, three Tangerines }

■ Does **test set T2** satisfy the **flavor criterion** ?

■ Does **test set T2** satisfy the **color criterion** ?

- Set of Test Requirements: **TR={Yellow, Green, Orange, White}**

Minimal Test Set

Given a set of test requirements TR and a test set T that satisfies all test requirements, T is minimal if removing any single test from T will cause T to no longer satisfy all test requirements.

$T_1 = \{ \text{three Lemons, one Pistachio, two Cantaloupes, one Pear, one Tangerine, four Apricots} \}$

- Checking to see if a test set is minimal is fairly easy, and deleting tests to make the set minimal is straightforward.
- We can delete two Lemon, one Cantaloupe, and three Apricot jelly beans to make the above set minimal.

Minimum Test Set

Given a set of test requirements TR and a test set T that satisfies all test requirements, T is minimum if there is no smaller set of tests that also satisfies all test requirements.

Coverage Level

Given a set of test requirements TR and a test set T , The ratio of the number of test requirements satisfied by T to the size of TR

$T_2 = \{ \text{One Lemon, two Pistachios, one Pear, three Tangerines} \}$

- T_2 satisfies 4 of 6 test requirements.

Infeasible test requirements

■ Test requirements that cannot be satisfied

- No test case values exist that meet the test requirements
- Example: Dead code
- Detection of infeasible test requirements is formally undecidable for most test criteria.

■ Thus, 100% coverage is impossible in practice

الزمات آزمایشی که نمی توان آنها را برآورده کرد
- هیچ مقدار مورد آزمایشی وجود ندارد که شرایط آزمون را برآورده کند
- مثال: کد مرده
- تشخیص الزمات آزمایش غیرممکن برای اکثر معیارهای آزمون به طور رسمی غیرقابل تصمیم گیری است.
بنابراین پوشش 100% در عمل غیرممکن است

Two Ways to Use Test Criteria

I. Directly generate test values to satisfy the criterion

- Often assumed by the research community
- Most obvious way to use criteria
- Very hard without automated tools

2. Generate test values externally and measure against the criterion

- Usually favored by industry
- Sometimes misleading
- If tests do not reach 100% coverage, what does that mean?

به طور مستقیم مقدیر تست را برای برآورده کردن معیار ایجاد کنید

- اغلب توسط جامعه پژوهشی فرض می شود
- واضح ترین راه برای استفاده از معیارها
- بدون ابزار خودکار بسیار سخت است

مقدیر تست را به صورت خارجی تولید کنید و بر اساس معیار اندازه گیری کنید
- معمولاً مورد علاقه صنعت است
- گاهی اوقات گمراه کننده

**Test criteria are sometimes called
metrics**

- اگر تست ها به پوشش 100% نرسند، به چه معناست؟

Generators and Recognizers

- Generator : A procedure that automatically generates values to satisfy a criterion
- Recognizer : A procedure that decides whether a given set of test values satisfies a criterion

Generator: رویه‌ای که به طور خودکار مقادیری را برای برآورده کردن یک معیار تولید می‌کند
شناساگر: رویه‌ای که تصمیم می‌گیرد آیا مجموعه داده‌ای از مقادیر آزمون یک معیار را برآورده می‌کند یا خیر.
- Both problems are provably undecidable for most criteria
- It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion

تشخیص اینکه آیا موارد آزمایشی یک معیار را برآورده می‌کنند
بسیار بیشتر از تولید تست‌هایی که معیار را برآورده می‌کنند ممکن است.
- Coverage analysis tools are quite plentiful

ابزارهای تحلیل پوشش بسیار زیاد هستند

Comparing Criteria with Subsumption (5.2)

- Criteria Subsumption : A test criterion C_1 **subsumes** C_2 if and only if every set of test cases that satisfies criterion C_1 also satisfies C_2

باید برای هر مجموعه ای از موارد آزمایش درست باشد

- Must be true for every set of test cases

- Examples :

- The flavor criterion on jelly beans subsumes the color criterion ... if we taste every flavor we taste one of every color.
- A many-to-one mapping exists between the requirements for the flavor criterion and the requirements for the color criterion.
- If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement.

زیرمجموعه معیارها: یک معیار آزمایشی C_2 ، C_1 را زیرمجموعه می‌گیرد اگر و فقط اگر هر مجموعه ای از موارد آزمایشی که معیار C_1 را برآورده می‌کند C_2 را نیز برآورده کند.

معیار طعم روی لوپیاهای ژله ای، معیار رنگ را در بر می‌گیرد ...
اگر هر طعمی را بچشیم، یکی از هر رنگی را می‌چشیم.

Advantages of Criteria-Based Test Design (5.3)

- Criteria **maximize** the “bang for the buck”
 - Fewer tests that are **more effective** at finding faults
- **Comprehensive** test set with **minimal overlap**
- **Traceability** from software artifacts to tests
 - The “**why**” for each test is answered
 - **Built-in support** for **regression testing**
- A “**stopping rule**” for testing—**advance knowledge of how many tests are needed**
- **Natural to automate**

ضوابط باعث به حداقل رساندن "بنگ برای دلار" می شوند

- تست های کمتری که در یافتن عیوب موثرتر هستند

مجموعه تست جامع با حداقل همپوشانی

قابلیت ردیابی از مصنوعات نرم افزاری تا تست ها

- "چرا" برای هر آزمون پاسخ داده می شود

- پشتیبانی داخلی برای تست رگرسیون

یک "قانون توقف" برای آزمایش - اگاهی از تعداد آزمایشات مورد نیاز

طبیعی برای خودکار

Characteristics of a Good Coverage Criterion

1. It should be fairly easy to **compute** test requirements **automatically**
 2. It should be **efficient** to **generate test values**
 3. The resulting tests should reveal **as many faults as possible**
- Subsumption is only a **rough approximation** of fault revealing capability
 - Researchers still need to give us more data on how to compare coverage criteria

محاسبه خودکار الزامات آزمون باید نسبتاً آسان باشد

2. برای تولید مقادیر تست باید کارآمد باشد

3. آزمایشها بدهستامده باید تا حد امکان عیوب را آشکار کنند
نتها یک تقریب تقریبی از قابلیت آشکارسازی خطأ است

حقوقان هنوز باید اطلاعات بیشتری در مورد
نحوه مقایسه معیارهای پوشش به ما بدهند

Test Coverage Criteria

- Traditional software testing is **expensive** and **labor-intensive**
- Formal coverage criteria are used to decide **which test inputs to use**
- More likely that the tester will **find problems**
- Greater **assurance** that the software is of **high quality** and **reliability**
- A goal or **stopping rule for testing**
- Criteria makes testing more **efficient** and **effective**

How do we start applying these ideas in practice?

How to Improve Testing ?

- Testers need more and better software tools
- Testers need to adopt practices and techniques that lead to more efficient and effective testing
 - More education
 - Different management organizational strategies
- Testing & QA teams need more technical expertise
 - Developer expertise has been increasing dramatically
- Testing & QA teams need to specialize more

Criteria Summary

- Many companies still use “monkey testing”
 - A **human** sits at the keyboard, **wiggles** the mouse and **bangs** the keyboard
 - **No automation**
 - **Minimal training required**
- Some companies **automate** human-designed tests
- But companies that use both **automation** and **criteria-based testing**

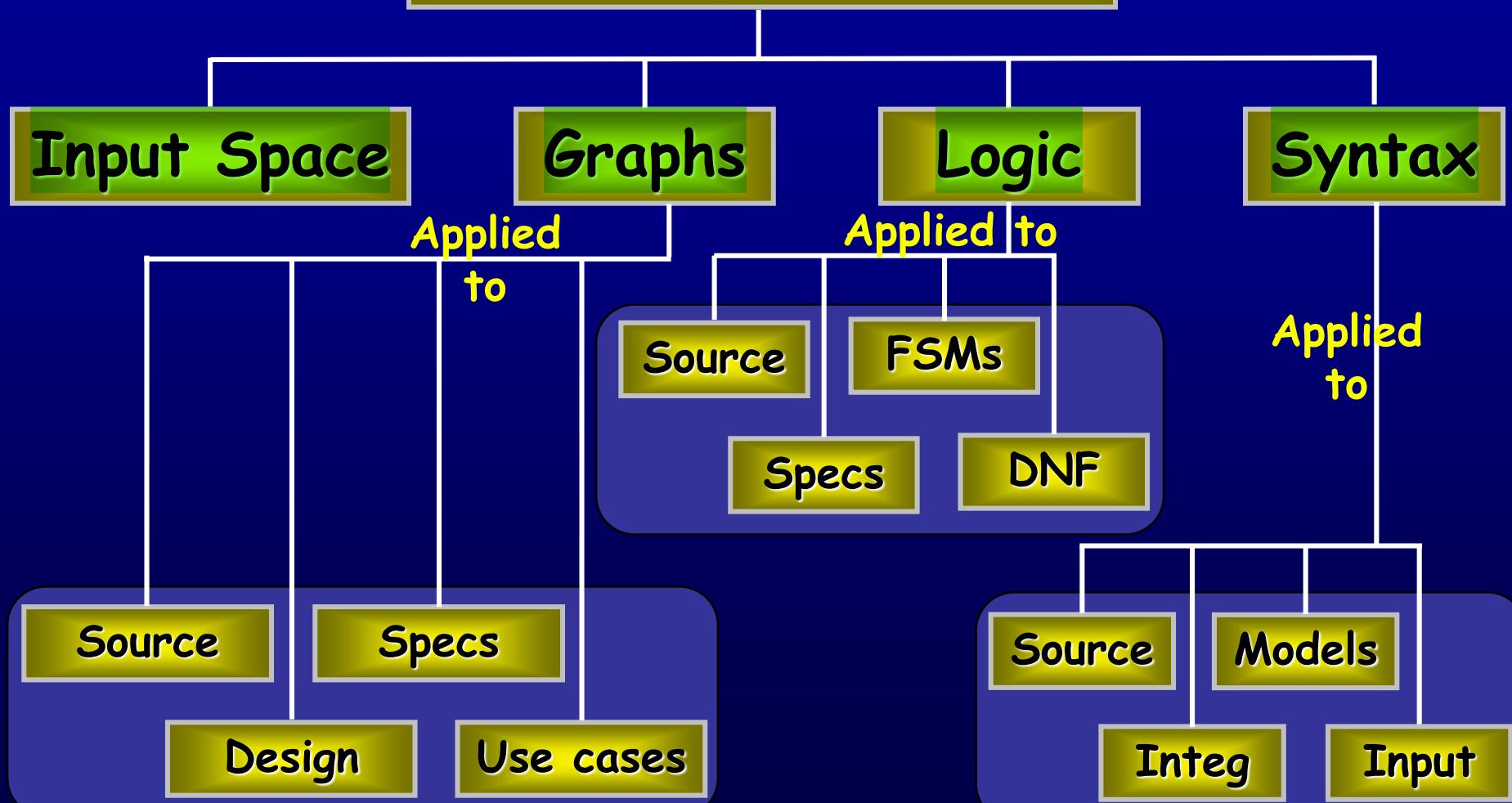
Save money

Find more faults

Build better software

Structures for Criteria-Based Testing

Four Structures for Modeling Software



Summary of Part 1's New Ideas

1. Why do we test – to reduce the risk of using software
 - Faults, failures, the RIPR model
 - Test process maturity levels – level 4 is a mental discipline that improves the quality of the software
2. Model-Driven Test Design
 - Four types of test activities – test design, automation, execution and evaluation
3. Test Automation
 - Testability, observability and controllability
4. Test Driven Development
5. Criteria-based test design
 - Four structures – test requirements and criteria

Earlier and better testing empowers test managers

Introduction to Software Testing (2nd edition)

Chapter 3

Test Automation

Paul Ammann & Jeff Offutt

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

Updated October 2018

What is Test Automation?

The **use of software** to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other **test control** and **test reporting** functions

استفاده از نرم افزار برای کنترل اجرای آزمونها، مقایسه نتایج واقعی با نتایج پیش بینی شده، تنظیم پیش شرط های آزمون، و سایر عملکردهای کنترل و گزارش تست

- Reduces cost
- Reduces human error
- Reduces variance in test quality from different individuals
- Significantly reduces the cost of regression testing by allowing a test to be run **repeatedly**.

هزینه را کاهش می دهد
خطای انسانی را کاهش می دهد
واریانس کیفیت تست را از افراد مختلف کاهش می دهد
به طور قابل توجهی هزینه تست رگرسیون را با اجازه دادن به یک تست برای اجرای مکرر کاهش می دهد.

Software testing can be **expensive** and **labor intensive**, so an important **goal of software testing** is
to **automate as much as possible**.

تست نرم افزار می تواند پر هزینه و کار فشرده باشد، بنابراین هدف مهم تست نرم افزار خودکارسازی تا حد امکان است.

Types of tasks

■ **Revenue tasks:** contribute directly to the solution of a problem.

- Example: determining which methods are appropriate to define a data abstraction in a Java class.

■ **Excise tasks:** do not.

- Example: compiling a Java class, because contributes **nothing** to the behavior of that class.

کمک مستقیم به حل یک مشکل.
کارهایی که نیاز به خلاقیت ذهن انسان داره مثلًا میخاهیم
کلاس دیاگرام های یک سیستم را طراحی کنیم . یا مثلًا یه
کلاس داریم میخاییم تصمیم بگیریم چه متدهایی براش بنویسیم؟
یعنی مسائلی که با توجه به تحلیل های مختلف میتواند جواب
های متفاوتی داشته باشد. که کاملا با دخالت ذهن و خلاقیت
انسان انجام میشه

نیاز به خلاقیت و مشارکت خاصی ندارند مثل کامپایل کردن
کلاس ها فقط یه مجموعه قانون باید بررسی شه که اون کلاس
طبق قوانین نوشته شده یا نه؟

Excise tasks are candidates for automation;

تسک هایی که به
صورت اتوماتیک
بایل انجام هستند و به
دخالت انسان ها نیاز
ندارند.

Types of tasks

- Software testing probably has more excise tasks than any other aspect of software development.
- Maintaining test scripts, rerunning tests, and comparing expected results with actual results are all common excise tasks that routinely use large amounts of test engineers' time.

کارهایی که به صورت تکراری و اتوماتیک میتوانند انجام شوند و به خلاقیت انسان نیاز ندارند.

گهداری از اسکریپتهای تست، اجرای مجدد تستها و مقایسه نتایج مورد انتظار با نتایج واقعی، همگی از وظایف متداول هستند که به طور معمول زمان زیادی از مهندسان تست استفاده میکنند.

Benefits of automating excise tasks

سختی کار را کمتر میکنند و باعث لذت بخش تر شدن
شغل مهندس تست میشوند.

- Eliminating excise tasks **eliminates drudgery**, thereby making the test engineer's job **more satisfying**.
- Automation **frees up time** to **focus on the fun** and **challenging parts** of testing, such as **test design**, a revenue task.

تمرکز مهندس تست بر سمت جاهایی که نیاز به خلاقیت دارد مثل طراحی تست ها و تست های revenue
- Automation **allows the same test to be run thousands** of times **without extra effort** in environments where tests are run daily or even hourly.

Benefits of automating excise tasks (Cnt'd)

- Automation can help eliminate errors of omission, such as failing to update all the relevant files with the new set of expected results.
- Automation eliminates some of the variance in test quality caused by differences in individual's abilities.

کاهش تنوع کیفیت
تست یعنی دیگه به
توانایی انسان ها
وابسته نمیشه

ممکنه فراموش کنیم همه فایل ها را پیدا
کنیم ولی اگه اتومات باشه دیگه فراموش
نمیشن یک بار فقط کدش را مینویسیم و
دفعات بعد خودش پروسه را انجام میده

Software Testability (3.1)

- Estimates how likely testing will reveal a fault if one exists.
- Plainly speaking – how easy or hard it is for faults to escape detection in the software

اگه یه fault‌ای وجود داره اون تست ما با چه درجه ای میتواند اشکارش کنه؟

تخمین میزند که در صورت وجود خطا، انجام تست چقدر احتمال دارد که خطا را آشکار کند.
به زبان ساده - چقدر آسان یا سخت است که خطاها در نرم افزار شناسایی نشوند

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met

Software Testability (3.1)

- Testability is dominated by two practical problems
 - How to provide the test values to the software
 - How to observe the results of test execution

دو مشکل عملی بر آزمون پذیری غالب است

- نحوه ارائه مقادیر تست به نرم افزار
- نحوه مشاهده نتایج اجرای آزمون

Observability and Controllability

قابلیت مشاهده

مشاهده رفتار یک برنامه از نظر خروجی‌ها، اثرات آن بر محیط و سایر اجزای سخت افزاری و نرم افزاری چقدر آسان است.

■ Observability

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

- Software that affects hardware devices, databases, or remote files have low observability

نرم افزارهایی که بر دستگاه‌های سخت افزاری، پایگاه‌های داده یا فایل‌های راه دور تأثیر می‌گذارند، قابلیت مشاهده کمی دارند

■ Controllability

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

- Easy to control software with inputs from keyboards
- Inputs from hardware sensors or distributed software is harder

قابلیت کنترل

ارائه یک برنامه با ورودی‌های مورد نیاز، از نظر مقادیر، عملیات و رفتار چقدر آسان است
– کنترل آسان نرم افزار با ورودی از صفحه کلید
– ورودی از سنسورهای سخت افزاری یا نرم افزارهای توزیع شده سخت تر است

Observability and Controllability

- Data abstraction **reduces** controllability and observability
- Many observability and controllability **problems** can be addressed with *simulation*,
 - By extra **software** built to “**bypass**” the **hardware** or **software** components that interfere with testing

انتزاع داده ها قابلیت کنترل و مشاهده را کاهش می دهد
بسیاری از مشکلات قابل مشاهده و کنترل را می توان با شبیه سازی حل کرد،
- توسط نرم افزار اضافی ساخته شده برای "دور زدن" قطعات سخت افزاری یا نرم افزاری که در تست تداخل دارند

فرایند تست دخالت
میکنیم تا اون
, observability
controllability
بهش دست پیدا کنیم تا
حدی

Types of software with low observability and controllability

- Embedded software,
- Component-based software,
- Distributed software,
- Web applications.

تست کردنشون سخت تره
وروودی ها رو مسقیم نمیشه بهشون
داد.

اجرای یک نرم افزار به
اجرای نرم افزار دیگه ای
وابسته است.

یعنی فالت ها
در حدممکن باید
بتوانند خودشون را
نشون بند.

Testability is **crucial** to **test automation**
because **test scripts** need to **control the execution** of the
component under test
and to **observe the results** of the test.

تستپذیری برای تست اتوماسیون بسیار مهم است زیرا اسکریپتهای تست
نیاز به کنترل اجرای کامپوننت تحت آزمایش و مشاهده نتایج تست دارند.

هرچه **testability** پایین باشه نمیتوانیم به طور شفاف فرایند تست را انجام بدیم
و نتایج ای که میگیریم نمیتوانند معیار مناسبی برای مقایسه و کیفیت باشند.

Components of a Test Case (3.2)

- A test case is a multipart artifact with a definite structure

یک نمونه آزمایشی یک مصنوع چند بخشی با ساختار مشخص است

- Test case values

The input values needed to complete an execution of the software under test

grant truth
خروجی حقیقی

- Expected results

مقدیر ورودی مورد نیاز برای تکمیل اجرای نرم افزار تحت آزمایش

نتیجه ای که در صورتی که نرم افزار مطابق انتظار عمل کند، توسط تست حاصل می شود

The result that will be produced by the test if the software behaves as expected

passed or failed.

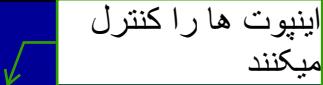
a test oracle uses expected results to decide whether a test passed or failed.

Test case value

- Are inputs to the program that test designers use to directly satisfy the test requirements.
- They determine the quality of the testing.
- Definition of test case values is quite broad.
- Test case values are not enough. In addition to test case values, other inputs are often needed to run a test.

ورودی هایی برای برنامه هستند که طراحان آزمون برای برآورده کردن مستقیم نیازهای آزمون از آنها استفاده می کنند.
کیفیت تست را تعیین می کنند.
تعريف مقادیر Test case کاملاً گسترده است.
مقادیر مورد آزمایش کافی نیست. علاوه بر مقادیر مورد آزمایش، ورودی های دیگری نیز برای اجرای یک آزمون مورد نیاز است.

Affecting Controllability and Observability



اینپوت ها را کنترل
میکنند

■ Prefix values

ورودی های لازم برای قرار دادن نرم افزار در وضعیت مناسب برای دریافت مقادیر مورد آزمایش

Inputs necessary to put the software into the appropriate state to receive the test case values

■ Postfix values

روی
observability
تاثیر میگذاره

مثلث برای رزرو کتاب در یک سایت اول باید یوزرنیم و پسورد را بزنیم و باید اول عضو شده باشیم تا بتوانیم رزرو کنیم.

Any inputs that need to be sent to the software after the test case values are sent

چه اینپوت هایی را نیاز داریم تا به استیت ای بررسیم که بتوانیم نتایج را مشاهده کنیم.

مقادیری که نیازه به نرم افزار بدیم بعد از اینکه اون تست کیس ولیو ها را به نرم افزار دادیم.

1. *Verification Values* : Values needed to see the results of the test case values

هر ورودی که باید پس از ارسال مقادیر مورد آزمایش به نرم افزار ارسال شود

2. *Exit Values* : Values or commands needed to terminate the program or otherwise return it to a stable state

Verification Values: مقادیر مورد نیاز برای مشاهده نتایج مقادیر مورد آزمایشی

Exit Values: مقادیر یا دستورات مورد نیاز برای خاتمه دادن به برنامه یا بازگرداندن آن به حالت پایدار

Putting Tests Together

■ Test case

The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test

مقادیر مورد آزمایش، مقادیر پیشوند، مقادیر پسوند و نتایج
موردنظر لازم برای اجرا و ارزیابی کامل نرم افزار
تحت آزمایش

■ Test set

A set of test cases

■ Executable test script

A test case that is prepared in a form to be executed automatically on the test software and produce a report

یک تست کیس که در فرمی آمده می شود تا به صورت خودکار روی
نرم افزار تست اجرا شود و گزارش تهیه شود

Example

- Consider the function `estimateShipping()` that estimates shipping charges for preferred customers in an automated shopping cart application.
- Suppose we are writing tests to check whether the estimated shipping charges match the actual shipping charges.

Example (Cnt'd)

- Prefix values, designed to reach (R) the estimateShipping() function in an appropriate state.

- Involve creating a shopping cart, adding various items to it, and obtaining a preferred customer object with an appropriate address.

مقادیر پیشوند، طراحی شده برای رسیدن به (R) عملکرد برآورده شنیدن (Shipping()) در یک حالت مناسب.

- شامل ایجاد یک سبد خرید، افزودن اقلام مختلف به آن و به دست آوردن یک شی مشتری ترجیحی با آدرس مناسب است.

- Test case values, designed to achieve infection (I), might be the type of shipping desired: overnight vs. regular.

مقادیر مورد آزمایش، طراحی شده برای رسیدن به عفونت! (Infection)،
ممکن است نوع حمل و نقل مورد نظر باشد: یک شبه در مقابل معمولی
خود فانکشن اصلی چک میشه!

ample (Cnt'd)

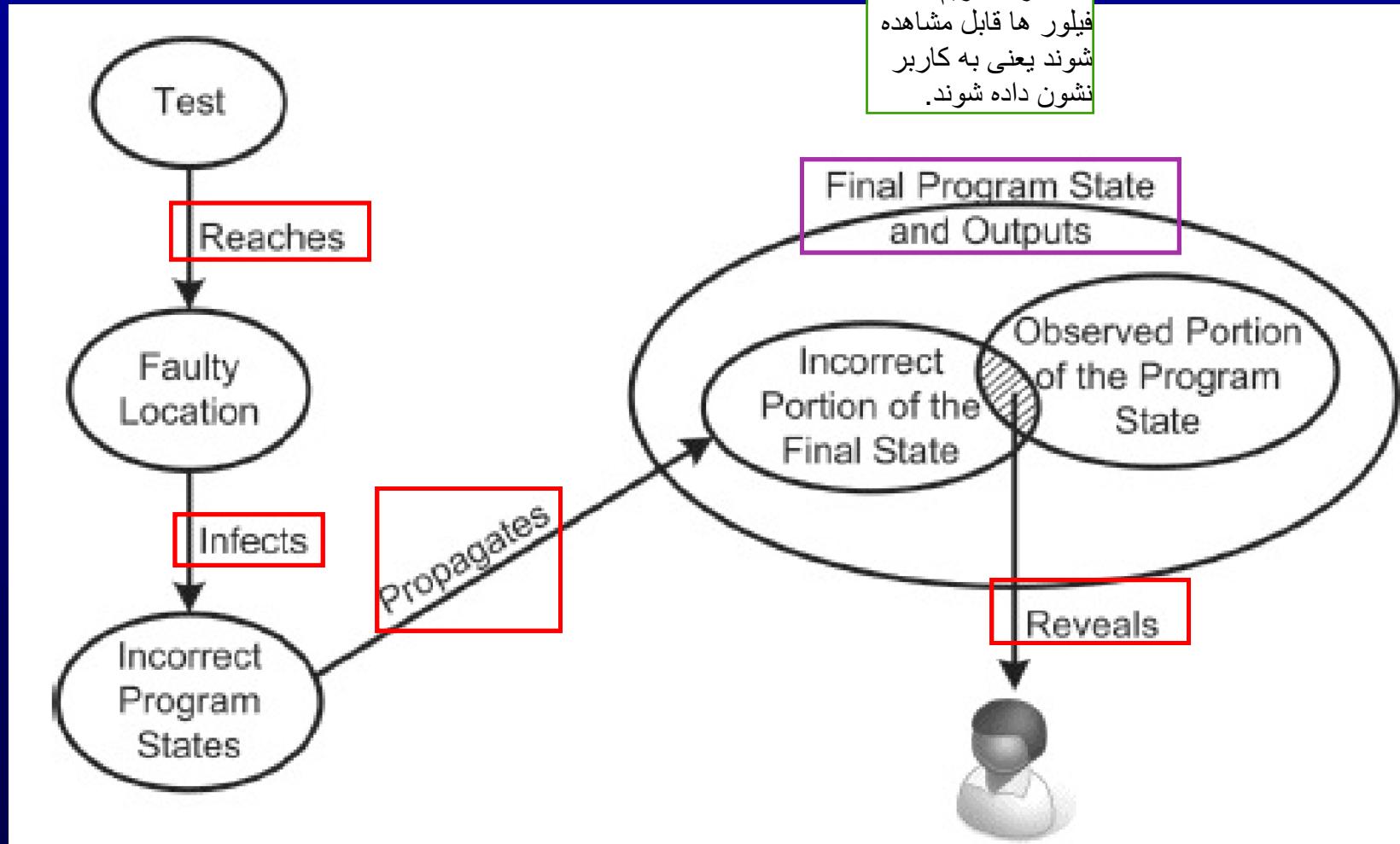
- Postfix values, designed to achieve propagation (P) and make an infection result in an observable failure,
 - might involve completing the order, so that actual shipping charges are computed.
- the revealing part (R) of the final order is probably implemented by extracting the actual shipping charge, although there are many other parts of the final order that could also be incorrect.

قسمت آشکار کننده (R) سفارش نهایی احتمالاً با استخراج هزینه حمل و نقل واقعی انجام می شود، اگرچه بسیاری از بخش های دیگر سفارش نهایی نیز می توانند نادرست باشند.

قادیر Postfix، طراحی شده برای رسیدن به انتشار (P) و ایجاد عفونت منجر به شکست قابل مشاهده می شود.
- ممکن است شامل تکمیل سفارش باشد، به طوری که هزینه های حمل و نقل واقعی محاسبه شود.

Test case and RIPR model

- The components in a test case are concrete realizations of the RIPR model.



Test case and RIPR model

A test can be thought of as being designed to look for a fault in a particular location in the program.

یک تست را می‌توان به عنوان طراحی شده برای جستجوی عیب در یک مکان خاص در برنامه در نظر گرفت.

معمولًا نمی‌تواند مقادیری را برای کل وضعیت

خروجی برنامه در اختصار یافتن عیوب این دوره

آزمایشی کنیم که بتواند همه عیوب را پیدا کند و در نتیجه از

وضعیت خروجی را که با مقادیر ورودی و هدف

آنچه می‌تواند اتفاق بیند را پیدا کند.

■ Prefix values are included to achieve reachability (R),

■ Test case values to achieve infection (I),

■ Postfix values to achieve propagation (P),

■ Expected results to reveal the failures (R).

– Usually cannot include values for the entire output state of the program, so a well designed test case should check the portion of the output state that is relevant to the input values and the purpose of the test.

معمولًا نمی‌تواند مقادیری را برای کل وضعیت خروجی برنامه در بر بگیرد، بنابراین یک مورد آزمایشی که به خوبی طراحی شده است باید بخشی از وضعیت خروجی را که با مقادیر ورودی و هدف آزمایش مرتبط است بررسی کند.

Ideally, the automation should be complete in the sense of running the software with the test case values, getting the results, comparing the results with the expected results, and preparing a clear report for the test engineer.

در حالت ایدهآل، اتوماسیون باید کامل باشد به این معنا که نرمافزار را با مقادیر مورد آزمایش، دریافت نتایج، مقایسه نتایج با نتایج موردنظر و تهییه گزارش شفاف برای مهندس آزمون، کامل باشد.

- The **only time** a test engineer would **not want to automate** is if the **cost of automation outweighs the benefits**.
 - Happen if we are **sure the test will only be used once**.
 - If the automation **requires knowledge or skills** that the test engineer does not have.

تنها زمانی که یک مهندس آزمایشی مایل به خودکارسازی نیست این است که هزینه اتوماسیون بیشتر از مزایای آن باشد.

- در صورتی اتفاق میافتد که مطمئن باشیم از تست فقط یک بار استفاده میشود.
- اگر اتوماسیون به دانش یا مهارت هایی نیاز دارد که مهندس آزمون آن را ندارد. نمیصرفه که هزینه اموزش را برای اعضای تیم بدھیم تا اون روش خاص اتوماسیون را یادبگیرند و به کار بگیرن