Introduction to Software Testing
(*2nd edition*)
Chapter 4

Putting Testing First

Paul Ammann & Jeff Offutt

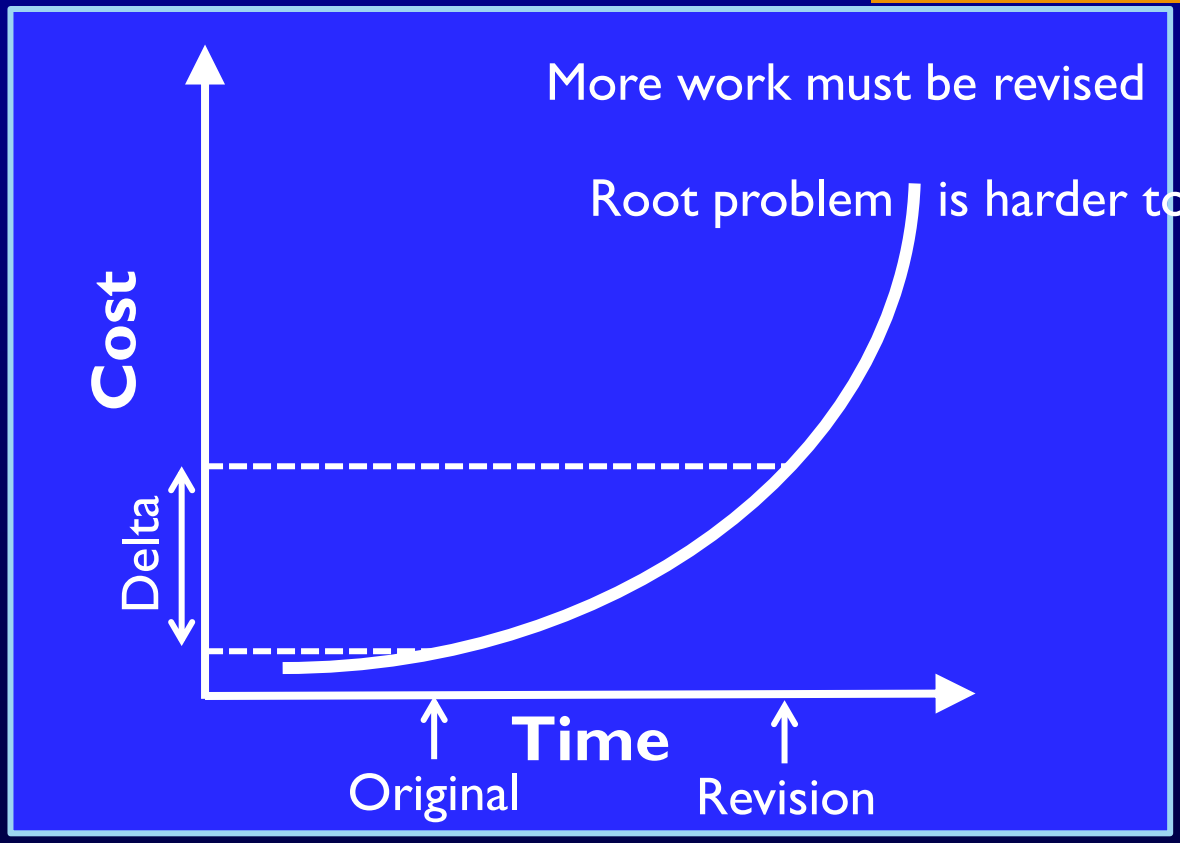
http://www.cs.gmu.edu/~offutt/softwaretest/

August 2014

The Increased Emphasis on Testing

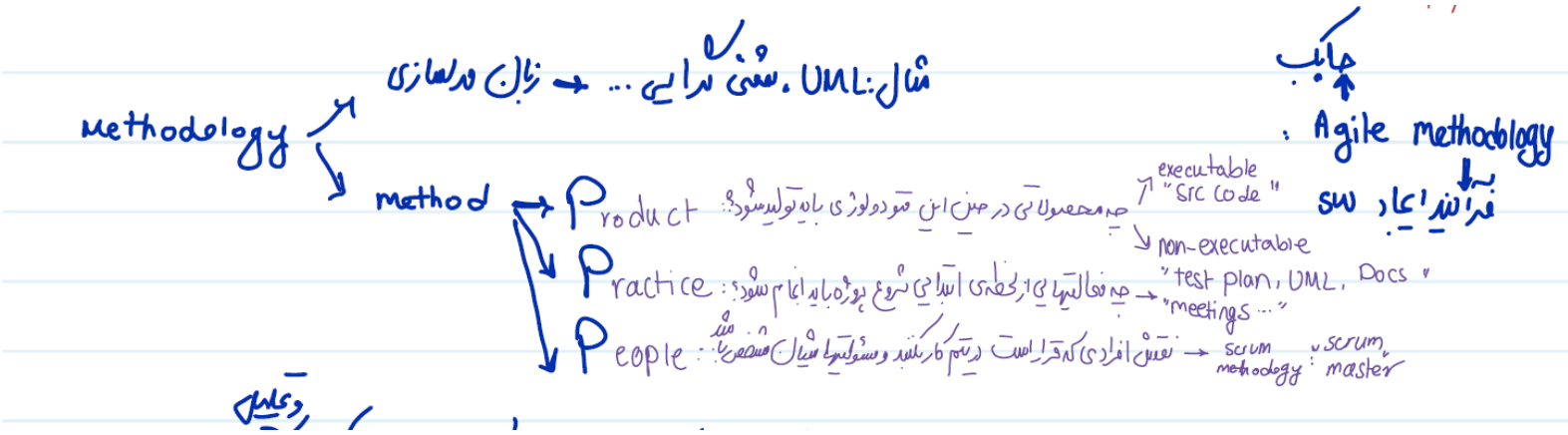
- Philosophy of traditional software development methods
  - Upfront analysis
  - Extensive modeling
  - Reveal problems as early as possible

نمودار زمان - هزینه:



-
متدلوژی: فرایند ایجاد نرم افزار هستش
agile: چابک
متدلوژی دو قسمت داره: 1- زبان مدل سازی 2- خود متد
زبان های مدل سازی: ینی توی طراحی قراره از چه زبانی استفاده بکنیم مثلا قراره از UML استفاده بکنیم برای فاز طراحی --> زبان برنامه نویسی هم جز زبان مدل سازی به حساب میاد
توی قسمت متد متدلوژی 3 تا p مهم است و باید مشخص بشه:
product: محصولاتمون هستش و همون software artifactهامون هستن و به دو دسته non executable , executable تقسیم بندی میشن
executable مثل سورس کد
non executable مثل تست پلن - نمودارهای دیزاین ...
practice: فعالیت ینی باید مشخص بکنیم که دقیقا از اون لحظه ابتدایی که پروژه به تیم ما سپرده میشه چه فعالیت هایی باید انجام بدیم تا اون مرحله اخر که کار تموم میشه - بعضی از فعالیت ها خیلی درشت هستن
people: ادم هایی هستن که درگیرن --> افرادی که قراره نقشی رو ایفا بکنند در تیم--> نقش ها مشخص بشن که چه افرادی چه مسئولیت هایی رو دارن --> پس باید توی متدلوژی اینا مشخص بشه

این نمودار ریشه طرز تفکر متدلوژی های سنتی رو مطرح میکنه:
توی متدلوژی های سنتی می گن قبل از اینکه کدنویسی رو شروع کنید بیان حسابی فکر بکنید ینی همه چیز رو شناسایی بکنید و تحلیل بکنید و.. --> چون می خوان اگر یک راه حلی وجود داره خیلی زود شناسایی بشه توسط این نمودار (این نمودار میشه نمودار زمان - هزینه) --> این نمودار نشون میده اگر توی یک مبدا یک مشکلی بروز پیدا بکنه فاصله زمانی بین اون لحظه ای که مشکل به وجود اومده تا اون لحظه ای که احساس نیاز برای رفع این مشکل هست اگر فاصله زمانی بیشتر باشه هزینه زیادی هم باید بدیم --> پس یک حالت تصاعدی اینجا داریم بین زمان و اون هزینه



فرق متدلوژی agile با متدلوژی سنتی در این است که: دیگه لازم نیست که خیلی مانور بدیم روی کارهای اولیه --> توی کارهای اولیه که توی متدلوژی سنتی وجود داشت تحلیل و مدل سازی زیاد بود و همه چی رو مستند می کنن ولی توی متدلوژی agile اینطوری نیست مثلا uml نمی کشن و اگر برای رفع ابهام خواستن uml بکشن بعدا میندازنش دور و داکیومنتش نمی کنن
نکته: برای پروژه های که کوچیک هستن و یا بحرانی نیستن از متدلوژی agile استفاده بکنیم ولی برای پروژه های بزرگ و بحرانی استفاده نکنیم --> سر این موضوع بحث است
مهمترین اصل در مهندسی نرم افزار این است که در موقعیت های بحرانی یک مهندسی نرم افزار باید از امتحان کردن روش های جدید خودداری کنه

دیزاین پترن: یک راه حل موفق است برای مشکلات رایجی که طراحان انجام میدادن
anti pattern یا ضد الگو: ینی این که راه حل هستن برای خرابکاری ها ولی خودشون باز خرابکاری بدتر هستن --> پس راه حل هستن ولی راه حل های اشتباه هستن
یکی از anti pattern هایی که داریم فلج تحلیل است دقیقا مثل overthinking میشه :)

Traditional Assumptions

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

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
- But those annoying customers keep changing their minds!
  - Humans are naturally good at approximating
  - But pretty bad at perfecting
- These two assumptions have made software engineering frustrating and difficult for decades

Thus, agile methods …

-
طرز تفکر متدلوژی های سنتی همیشه درست نیست چون ریکوارمنت هایی که ما توی مراحل اولیه تحلیل درشون میاریم همیشه کامل نیستن که بخوایم فقط توی فاز اول متمرکز بشیم روی ریکوارمنت دراوردن و بعد از روی اون ریکوارمنت ها بریم طراحی سطح بالا و ... --> پس اولین علت این میشه که ریکوارمنت ها کامل نیستن 2- هرچقدر هم ریکوارمنت ها کامل باشن مشتری ها نظرشون خیلی سریع عوض میشه
متدلوژی های سنتی --> مثل ابشاری و..

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
- 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
- Agile methods expect software to start small and evolve over time
  - Embraces software evolution instead of fighting it

-
متدلوژی های agile مثل اسکرام - XP - FDD - ... هستن
توی متدلوژی های Agile لازم نیست که همه چیز رو همون لحظه اول کار بیایم تحلیل بکنیم هی خرد خرد سریع خودمون رو برسونیم به سورس کد --> توی متدلوژی های Agile یکی از چیزایی که مهم است اینه که خیلی سریع به سورس کد برسیم
یکسری اصول برای agile تعریف کردن که هر متدلوژی Agile جدیدی که تعریف میشه اون اصول رو داشته باشه:
یکی از این اصل ها متمرکز هستش بر changes یا تغییرات ینی متدلوژی ما به شرطی agile است که با روی باز از تغییرات استقبال بکنیم حتی در اون فازهای نهایی تحویل
نکته: رضایت مشتری توی این متدلوژی برای ما خیلی مهم است
دومین اصل این است که روی مدل سازی و داکیومنت کردن نباید مانور بدیم و فقط برای رسیدن سریع به سورس کد باید مانور بدیم
اصل سوم--> ادم ها خیلی مهم هستن --> خیلی از متدلوژی های agile موقع تقسیم وظایف توی اعضای تیم میگه تسک ها رو مشخص بکنیم و بعد هر کس براساس علاقه خودش بیاد و بگه می خواد چه کاری رو انجام بده
اصل بعدی --> برنامه ریزی های دقیق هم نباید داشته باشیم و برنامه ریزی ها باید انعطاف پذیر باشن

یوزراستوری: کاربر از دید خودش با سیستم چه کاری رو میخواد انجام بده یا چه نیازی رو در حین کار با سیستم احساس میکنه مثلا احساس میکنیم این سرعتش بهتر بشه این میشه یک یوزر استوری وقتی چندتا از یوزراستوری ها رو نوشت توی متدلوژی xp میاد براساس اولویت مشتری اینا رو مرتب میکنه و اونایی ک از دیدی کاربر مهمه رو می ذاریم اول - توی بعضی از متدلوژی های دیگه براساس ریسک این یوزر استوری ها مرتب میشه
بعد وقتی که اولویت بندی شده منیجر میاد یوزر استوری رو برمیداره و به دولوپرها میگه این چقدر زمان می بره مثلا می گن یک ماه و میگه این زیاده و اینو می شکنه به تسک های کوچک پس هر یوزر استوری که زمان بر باشه شکسته میشه به تسک ها
دولوپرها میشن دوبه دو یکی ناظر و یک تایپ کننده
توی متدلوژی xp دولوپرها دوتا دوتا می شینن پشت کامپیوتر یکی کد می زنه و اون یکی نگاه می کنه و هر چند وقت یکبار هم جاشونو عوض میکنن
متدلوژی xp قدیمی است و نسبت به متدلوژی های Agile سنگین ترینشون است
توی xp هر چند وقت یکبار نقش ها باید بچرخه که متکی بر یک نفر خاص نباشه و از یک طرف دیگر مستندات هم نداریم اینجا
توی متدلوژی xp اینطوری است که صبح به صبح جلسه ایستاده دارن که سریع تموم بشه
اگر test driven development داشته باشیم که حتما اینو توی xp داریم باید قبل از اینکه کد رو بنویسیم باید تست هاشو نوشته باشیم ینی براساس خود تست ها کد رو جلو می برن و بعد که نوشتیم، کد ها رو ادغام میکنیم و این توی ریپوزیتوری است و بعد این کد توسط تمام دولوپرها دیده میشه و دولوپرها می تونن روش فیدبک بدن
متدلوژی xp همون لحظه که کار شروع میشه میخواد سریع یک محصولی رو تحویل یوزر بده --> اولین کدهایی که نوشته میشه توی متدلوژی xp سریع می ره قراره میگیره توی محیط واقعی (البته که این کد ناقص است و به تدریج کامل میشه ولی اسکرام صبر میکنه یکم و بعد تحویل میده)

planing هم به این صورت است که تخمین میزنیم و اگر نتونستیم تخمین بزنیم prototype ایجاد میکنیم

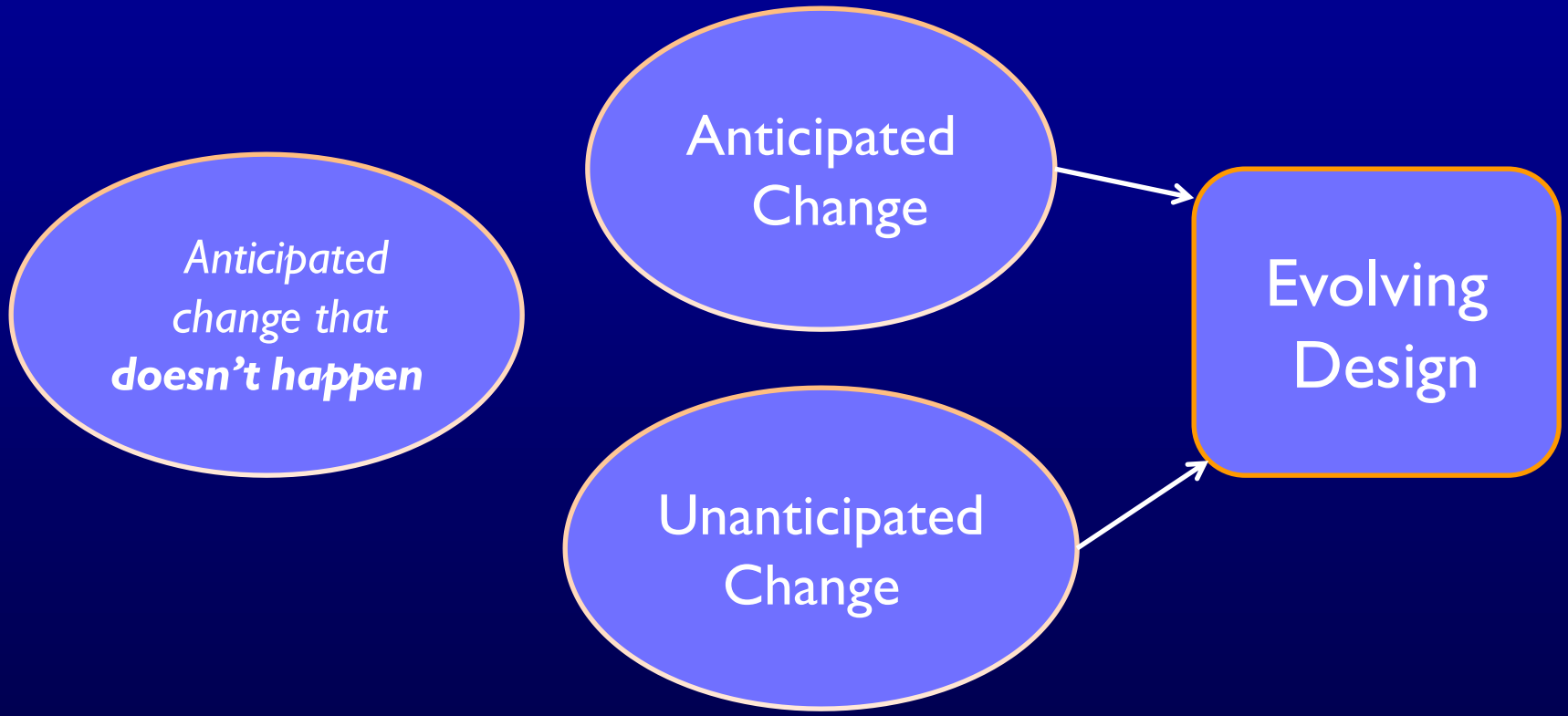
توی متدلوژی agile خطر out of date شدن داکیومنت ها وجود نداره
نکته: توی متدلوژی agile ایا ممکنه که کاربر هی نظرش عوض بشه؟ نه اینطور نیست چون نماینده کاربر عضوی از تیم ما است و هر هفته که یک ملاقات می ذاریم بهش نشون میدیم که اینطوری تا الان پیش رفته و ازش نظر می گیریم این موقع و بعد دوباره پیش می ریم

در متدلوژی های سنتی خیلی از برنامه ریزی ها بی استفاده و بی مورد هستن و خیلی از مشکل هایی که پیش بینی و بررسی میشوند اصلا اتفاق نمی افتند و وقت ما را هدر می دهند

Supporting Evolutionary Design

Traditional design advice says to anticipate changes

Designers often anticipate changes that don't happen

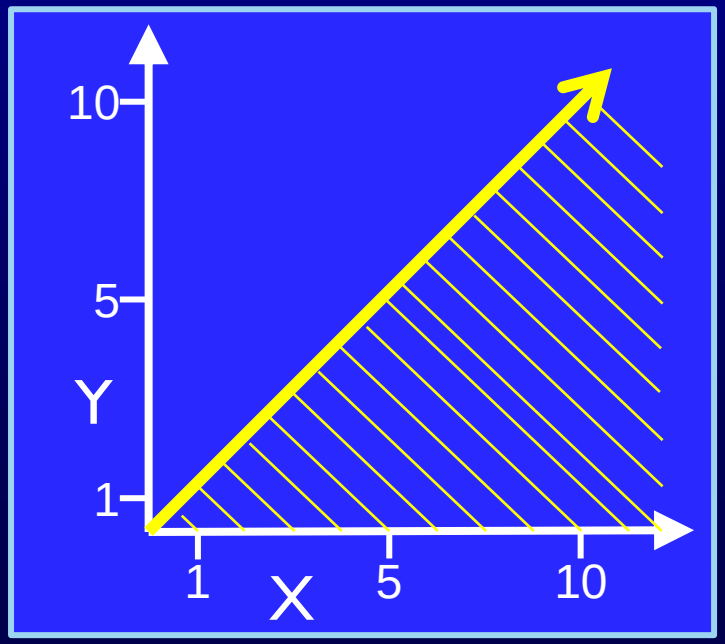


Both anticipated and unanticipated changes affect design

The Test Harness as Guardian (4.2)

What is Correctness ?

Traditional Correctness
(Universal)
∀ x,y, x ≥ y



Agile Correctness
(Existential)

{ (1, 1) → T
(1, 0) → T
(0, 1) → F
(10, 5) → T
(10, 12) → F }

-

توی متدلوژی های سنتی وقتی که ما می خوایم یک بخشی از x , y ها رو داشته باشیم که این قانون رو داشته باشه قبلش می ریم خیلی قشنگ فکر میکنیم و همه مسیرها رو بررسی میکنیم ولی توی متدلوژی agile می گه بیا با نمونه پیش برو مثلا اگر قرار باشه x>=y بیا اول تست ها رو توی test driven development طراحی بکن و ببین جواب درست چی هست ینی فقط روی یک بخشی از رفتار نرم افزار متمرکز میشه ولی توی سنتی اینطوری نیست و روی تمام رفتارها متمرکز میشه

A Limited View of Correctness

- In traditional methods, we try to define all correct behavior completely, at the beginning
  - What is correctness?
  - Does "correctness" mean anything in large engineering products?
  - People are VERY BAD at completely defining correctness
- In agile methods, we redefine correctness to be relative to a specific set of tests
  - 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

But software engineers ain't mathematicians!

Test Harnesses Verify Correctness

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

- 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
- The tests replace the requirements
- Tests must be high quality and must run quickly
- We run tests every time we make a change to the software

-
توی متدلوژی agile برامون خیلی مهم است که تست رو اتومات بکنیم
تست اتومات ینی توی اون فاز طراحی، دیزاینرتست هر تست کیسی که در اورد بدش به یک نرم افزاری که اتومات بیاد نرم افزار رو اجرا بکنه به ازای این تست کیس ها

توی متدلوژی هایی که test driven development هستن ریکوارمنت نداریم و یوزر استوری داریم و قبل از اینکه پیاده سازی رو انجام بدیم تست ها رو طراحی میکنیم

Continuous Integration

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

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

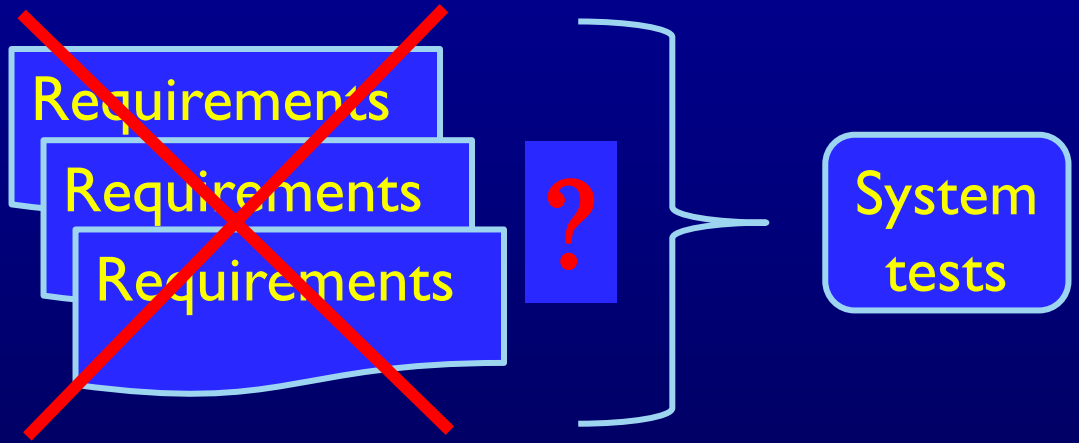
- Mistakes are caught earlier
- Other developers are aware of changes early
- The rebuild and reverify must happen as soon as possible
  - Thus, tests need to execute quickly

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

-
توی متدلوژی های agile یکی از کارهای مهم این هست که وقتی که ما به عنوان یک دولوپر کدی نوشتیم باید ادغام بشه با کد بقیه دولوپرها ؟؟
continuous integration serverها برنامه رو به ازای هر تغییری که اپلود میشه بازسازی میکنن و دوباره تایید می کنن ینی تست هایی که دیروز پاس شده بوده الان دوباره همون تست ها رو روی کد جدید اجرا می کنن که مطمئن بشن تغییر ما چیزی رو خراب نکرده باشه و بعد فیدبکش می ره برای همه
نکته: refactoring خیلی مهم است اینجا --> توی متدلوژی agile چون داکیومنت سازی وجود نداره خیلی refactoring با دقت انجام میشه
refactoring یکی از کارهای خیلی مهمی است که یک مهندس نرم افزار باید بلد باشه
refactoring ینی اصلاح (استراکچر) کد بدون اینکه رفتارش تغییر بکنه به منظور بهبود --> حالا اگر یک تیکه از کد رو عوض کردیم که رفتارش تغییر پیدا کرد این دیگه اسمش refactoring نیست - refactoring باعث میشود که قابلیت نگهداری یا maintainability کد افزایش پیدا کند

System Tests in Agile Methods

Traditional testers often design system tests from requirements



But … what if there are no traditional requirements documents ?

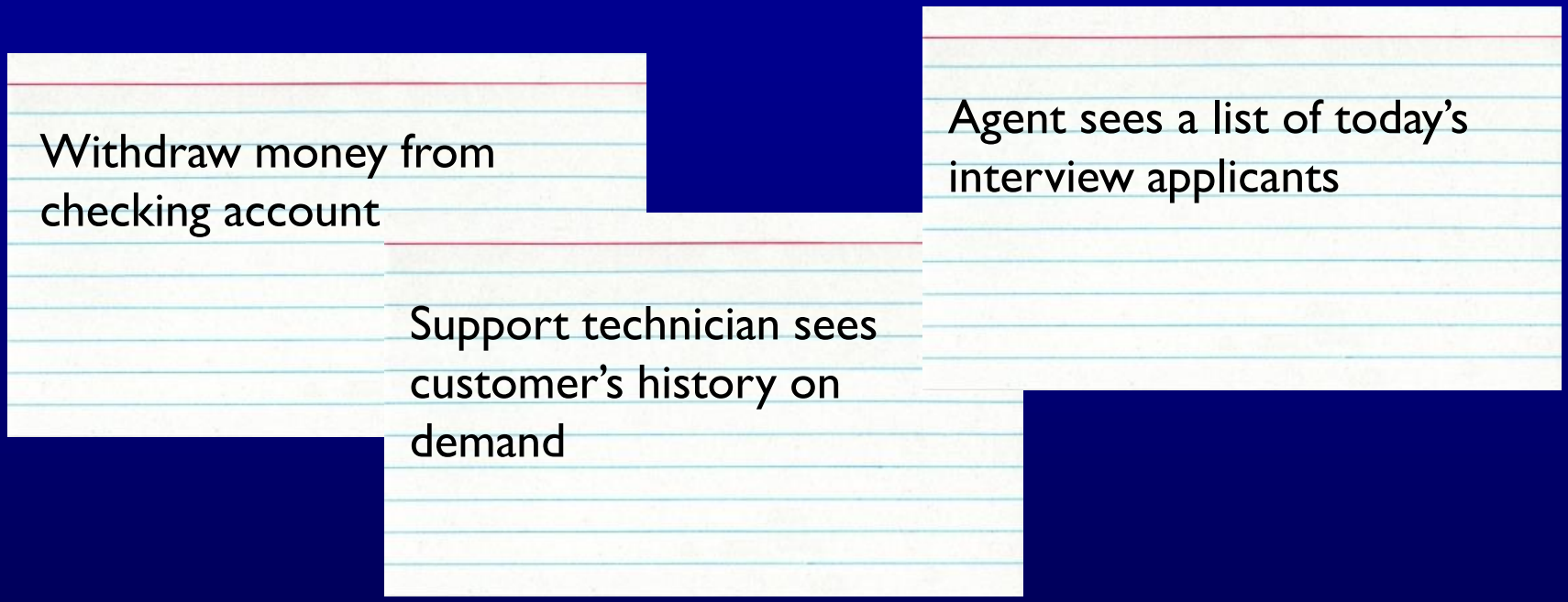
-
توی متدلوژی agile چطوری system tests انجام میشه؟ --> چون توی system testing ما از ریکوارمنت های شناخته شده استفاده می کردیم ولی اینجا دیگه سندی نداریم که داکیومنت کرده باشیم ریکوارمنت ها رو پس system testing رو چطوری طراحی میکنیم؟
system testing توی روش متدلوژی سنتی بنا به ریکوارمنت ها طراحی میشه ینی توی اون مدل v همون موقع که ریکوارمنت ها در می اوردیم یه سری کارها رو انجام میدادیم پس توی همون مراحل اولیه system testing هم طراحی میشه ولی انجام نمیشه فقط طراحی میشه حالا توی متدلوژی agile ما دیگه ریکوارمنت نداریم ولی یوزر استوری داریم اینجا پس براساس هر کدوم از یوزر استوری ها می رن system test رو همون لحظه طراحی می کنن و کد رو براساس اون جلو می برن
توی متدلوژی Agile هر کدوم از یوزر استوری ها که در اختیار دولوپرها قرار گرفت و اگر قراره وی متدلوژی agile از test driven development استفاده بکنیم ینی اول تست رو ایجاد بکنیم بعد کد رو پس از روی همین یوزر استوری یک acceptance test طراحی می کنیم مثلا الان یوزر استوری این است که از توی صفحه لاگین از طریق گزینه پسورد بیاد پسورد رو بازیابی کنه حالا برای acceptance test میتونیم اینو بگیم که "صفحه بازیابی پسورد درست کار بکنه" حالا براساس این که این جمله چقدر کلی باشه ما میایم هی تست میسازیم و بعد از روی این تست می ریم کد رو ایجاد میکنیم (بار اول که تست قطعا fail میشه چون کد رو هنوز ننوشتیم) --> چه خطری داره؟ happy path ینی این یوزر بیشتر روی happy pathها متمرکز هستش و با این روش ما فقط میایم اون رفتارهای سیستم در ازای ورودی های نرمال رو می سنجیم تا مثلا exception رو بسنجیم --> exception ها رو که بخوایم بسنجیم خیلی هاش وابسته است به اون مسائل تکنیکال یا اینکه یوزرمون با وجود اینکه قوی بوده ولی به ذهنش نرسیده -->چون این رفتارها رو چک نمی کنیم توی سیستم های agile به دردسر می خوریم و فقط متمرکز است روی happy pathها

نکته: deploy کردن ینی نرم افزار در محیط واقعیش بشینه مثلا اگر یک بانک به ما گفته کدمون رو بنویس تیم ما می شینه و اینو می نویسه و بعد که نوشت میره توی اون بانک و مثلا میگه سرور کجاست می خوایم این کد رو تحویل بدیم اینا که این کار میشه الان deploy کردن ینی نرم افزار رفت توی محیط واقعی خودش نشسته که اینجا میشه بانک الان پس deploy کردن ینی نرم افزار استقرار یافت در مکان واقعی خودش

نکته: refactoring علاوه بر اینکه داره نوشته ها رو و استراکچر رو عوض میکنه ممکنه روی پارامترهای مثل سرعت و مصرف حافظه هم تاثیر بذاره

User Stories

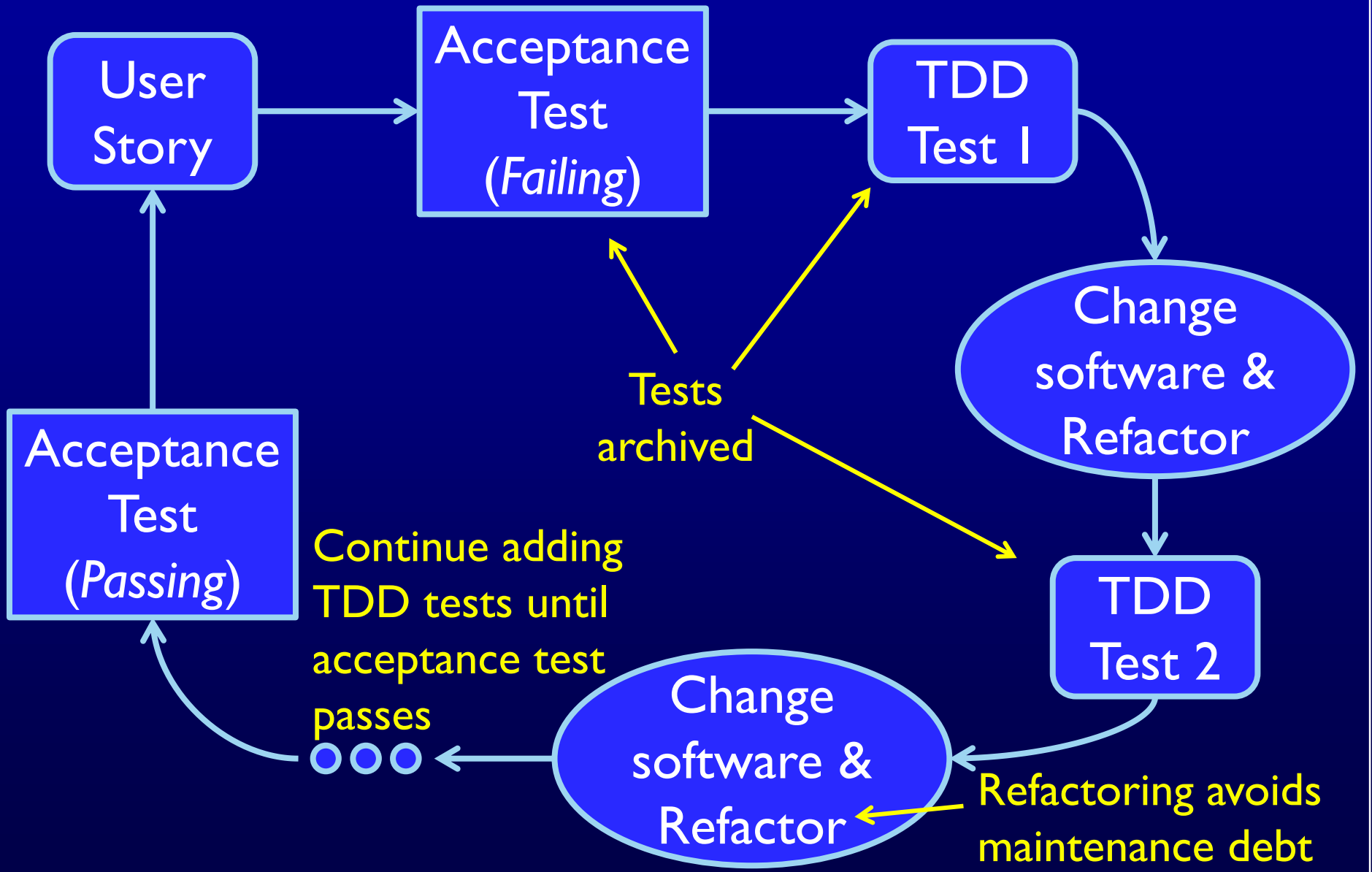
A *user story* is a few sentences that captures what a user will do with the software



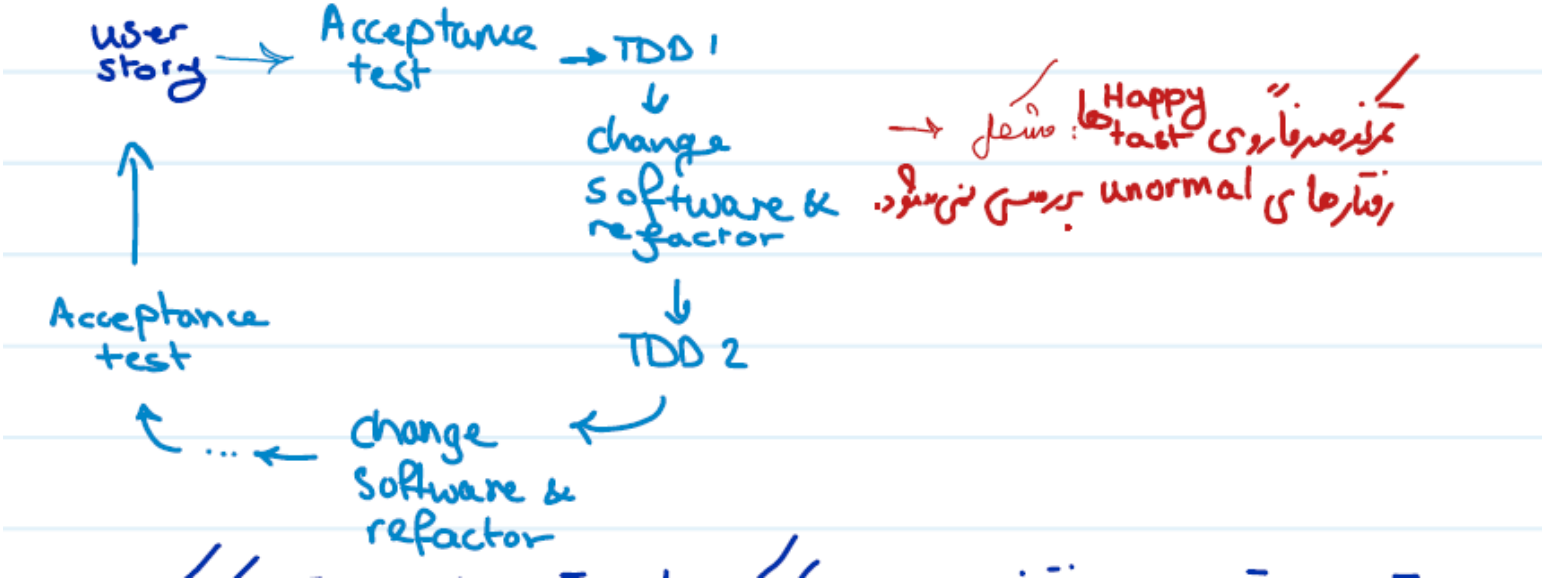
- In the language of the end user
- Usually small in scale with few details
- Not archived

-
یوزر استوری فاقد جزئیات فنی است و در حد یک یا دو جمله است (خلاصه است)

Acceptance Tests in Agile Methods



-
منظور از رفتارهای unnormal مثل همون مثال بانکی است که یک نفر مقصد و مبدا رو یکی بزنه



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

How to apply TDD to legacy software with no tests?

- Create an entire new test set? — too expensive!
- Give up? — a mixed project is unmanageable

-
legacy system: سیستم های قدیمی --> توی روش TDD ما نگاه میکنیم که تغییراتی که باید بدیم چیا هستن --> الان باید یک تغییر کوچیک انتخاب بکنیم که فقط ما به ازای همون تغییر کوچیک بیایم تست رو طراحی بکنیم و بعد بریم براساس اون چیزایی که از زبان یاد گرفتیم کد رو تغییر بدیم --> به ازای هر تغییر کوچیک تستش رو طراحی می کنیم و بعد می ریم کدش رو می نویسم و ادغام می کنیم با اون legacy system که داشتیم و بعد دوباره می ریم سراغ تغییر بعدی و به همین صورت می ریم جلو...

Incremental TDD

- When a change is made, add TDD tests for just that change
  - Refactor
- As the project proceeds, the collection of TDD tests continues to grow
- Eventually the software will have strong TDD tests

The Testing Shortfall

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

NO!

کمبود تست:

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

آیا تست ها پوشش خوبی روی کد دارند؟

آیا آزمایش ها بیشتر ایرادات را پیدا می کنند؟

اگر نرم افزار قبول شود، آیا مدیریت باید از قابل اعتماد بودن نرم افزار اطمینان داشته باشد؟

Refactor

legacy system : سیستم های قدیمی ← انتقاب تغییرات کوچک و طراحی تست برای همان تغییر کوچک

کم کم اعتماد، خود، مهندسان تغییرات را انجام داد و test کرد.

ولی کافی نیست : نیازه داشتن سنجت به Coverage criteria است.

Why Not?

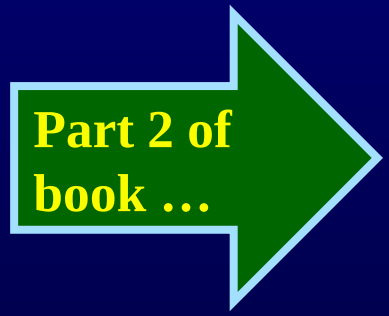
- Most agile tests focus on "*happy paths*"
  - What should happen under normal use

The agile methods literature does not give much guidance

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



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

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

--